

## ENSEIRB-MATMECA



# MISE EN ŒUVRE DE LINUX EMBARQUE : $\mu$ CLINUX

**Patrice KADIONIK**  
[www.enseirb.fr/~kadionik](http://www.enseirb.fr/~kadionik)

## TABLE DES MATIERES

1.	<i>But des travaux pratiques</i> .....	3
2.	<i>TP 0 : prise en main</i> .....	3
3.	<i>TP 1 : téléchargement d'un noyau <math>\mu</math>Clinux dans la cible</i> .....	4
4.	<i>TP 2 : génération d'un noyau <math>\mu</math>Clinux et tests</i> .....	4
5.	<i>TP 3 : mise en œuvre de NFS sous <math>\mu</math>Clinux</i> .....	8
6.	<i>TP 4 : création d'une application userland sous <math>\mu</math>Clinux</i> .....	9
7.	<i>TP 5 : développement d'une application serveur embarquée sous <math>\mu</math>Clinux et d'une application cliente Linux pour le contrôle de la carte cible</i> .....	10
8.	<i>TP 6 : mise en œuvre d'un client SMTP sous linux</i> .....	11
9.	<i>TP 7 : mise en œuvre d'un client SMTP embarqué sous <math>\mu</math>Clinux</i> .....	12
10.	<i>TP 8 : mise en œuvre d'un serveur web embarqué sous <math>\mu</math>Clinux</i> .....	13
11.	<i>TP 9 : mise en œuvre de SNMP sous Linux</i> .....	14
12.	<i>TP 10 : mise en œuvre d'un agent SNMP embarqué sous <math>\mu</math>Clinux</i> .....	15
13.	<i>TP 11 : système de fichiers JFFS2 pour mémoire FLASH</i> .....	18
14.	<i>TP 12 : debug d'applications <math>\mu</math>Clinux avec GDBSERVER</i> .....	22
15.	<i>TP 13 : debug d'applications <math>\mu</math>Clinux par le BDM</i> .....	23
15.1.	Debugger le noyau $\mu$ Clinux .....	24
15.2.	Debugger une application sous $\mu$ Clinux .....	25
16.	<i>Références</i> .....	26

## 1. BUT DES TRAVAUX PRATIQUES

Le but de ces Travaux Pratiques est d'étudier la mise en œuvre de Linux embarqué sur une carte d'évaluation Motorola ColdFire EVB5407C3.

On verra les points suivants :

- Mise en œuvre de  $\mu$ Clinux.
- Développement d'applications sous  $\mu$ Clinux (Hello World, application Client/Serveur).
- Mise en œuvre d'un client SMTP embarqué sous  $\mu$ Clinux. Application au pilotage par SMTP de systèmes embarqués.
- Mise en œuvre d'un serveur WWW embarqué sous  $\mu$ Clinux. Application au pilotage par WWW de systèmes embarqués.
- Mise en œuvre de SNMP sous Linux et sous  $\mu$ Clinux. Application au pilotage par SNMP de systèmes embarqués.
- Mise en œuvre du système de fichiers JFFS2 pour les mémoires FLASH.
- Debug d'applications  $\mu$ Clinux et du noyau par GDBSERVER et par le BDM.

A tout moment, on se référera à l'aide contenue en annexe dans ce manuel ou bien à l'aide en ligne :

```
% man ...
```

## 2. TP 0 : PRISE EN MAIN

- Démarrer le PC sous Linux. Se connecter sous le nom **uclinux**, mot de passe : **uclinux** ☺.
- Se créer un répertoire de travail à son nom et s'y placer :

```
% cd
% mkdir mon_nom
% cd mon_nom
```
- Etablir le schéma de l'environnement de développement :
  - Matériels.
  - Liaisons : série, réseau...
  - Logiciels et OS utilisés.
  - Adresses IP du PC de développement (hôte ou *host*) et de la plateforme de test (cible ou *target*).
- Se connecter à la carte d'évaluation ColdFire EVB5407C3 (cible) en utilisant l'outil minicom.
- Retrouver les commandes du moniteur de la cible pour :
  - Télécharger par le réseau Ethernet un fichier binaire.
  - Télécharger par le réseau Ethernet un fichier S-Record.
  - Lancer et exécuter le fichier précédemment chargé en mémoire.

- Quelle est l'adresse de point d'entrée de l'exécutable précédemment téléchargé ?
- A quoi correspond le format Motorola S-Record ?

### 3. TP 1 : TELECHARGEMENT D'UN NOYAU $\mu$ CLINUX DANS LA CIBLE

- Télécharger par le réseau dans la cible le fichier de image `image0.bin`.
- Lancer le noyau  $\mu$ Clinux ainsi téléchargé. Observer les traces sur la console au boot. Quelle est la puissance de la cible en bogoMips ? Quelle est la taille du noyau  $\mu$ Clinux ?
- Sous quel utilisateur est-on connecté par défaut ?
- Quels sont les services  $\mu$ Clinux actifs sur la cible ?
- Comment retrouver des informations sur le taille mémoire et son utilisation sur la cible ?
- Configurer l'interface Ethernet de la cible et tester la connectivité IP avec la commande `ping` (entre la cible et l'hôte).
- Tester l'accès au serveur `www` embarqué sur la cible depuis l'hôte avec le navigateur Netscape. Peut-on utiliser les scripts CGI ? Quel est le nom du package WWW mis en œuvre ?

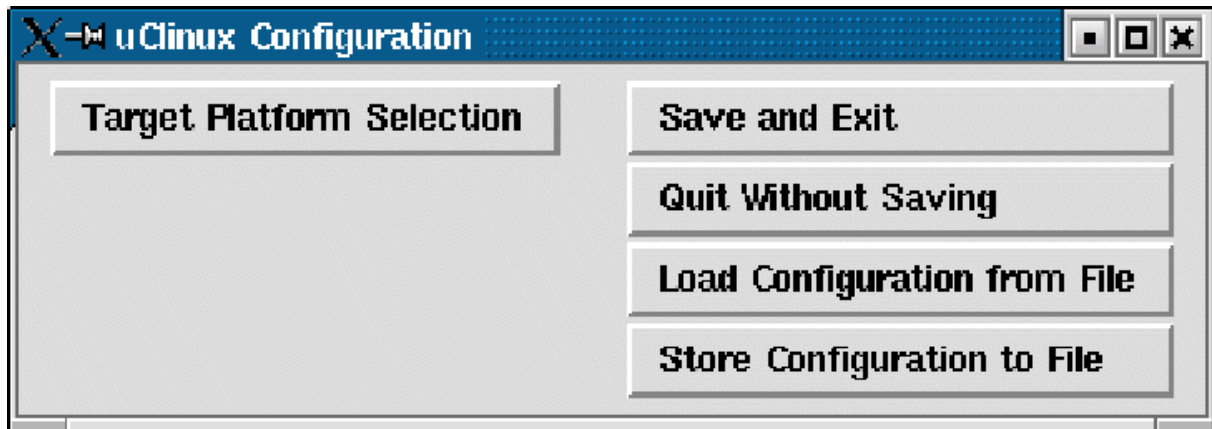
### 4. TP 2 : GENERATION D'UN NOYAU $\mu$ CLINUX ET TESTS

- Dans son répertoire à son nom, recopier tous les fichiers contenus sous `~kadionik` :  

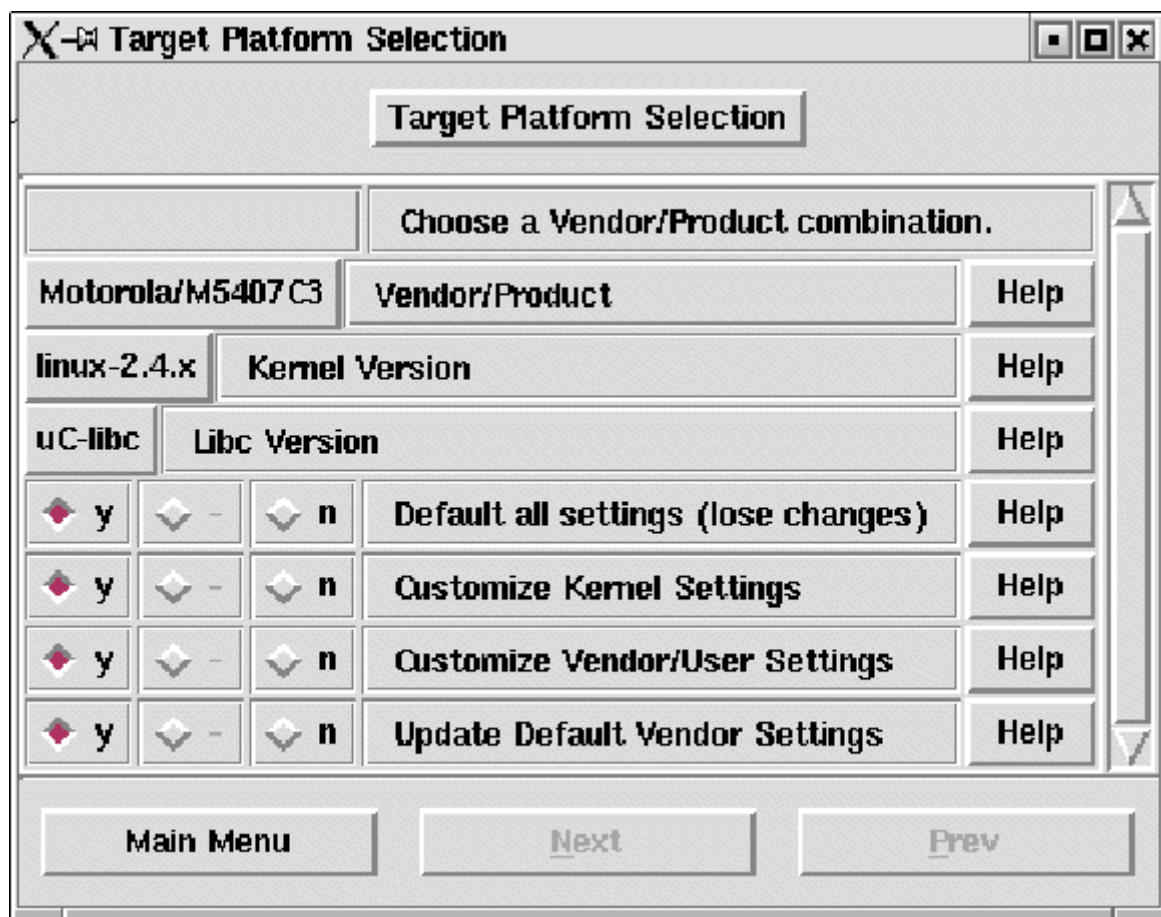
```
% cp -r ~kadionik/* .
```
- Décompresser et installer le package  $\mu$ Clinux pour ColdFire `uClinux-dist.xxxxxxxx.tar.gz` :  

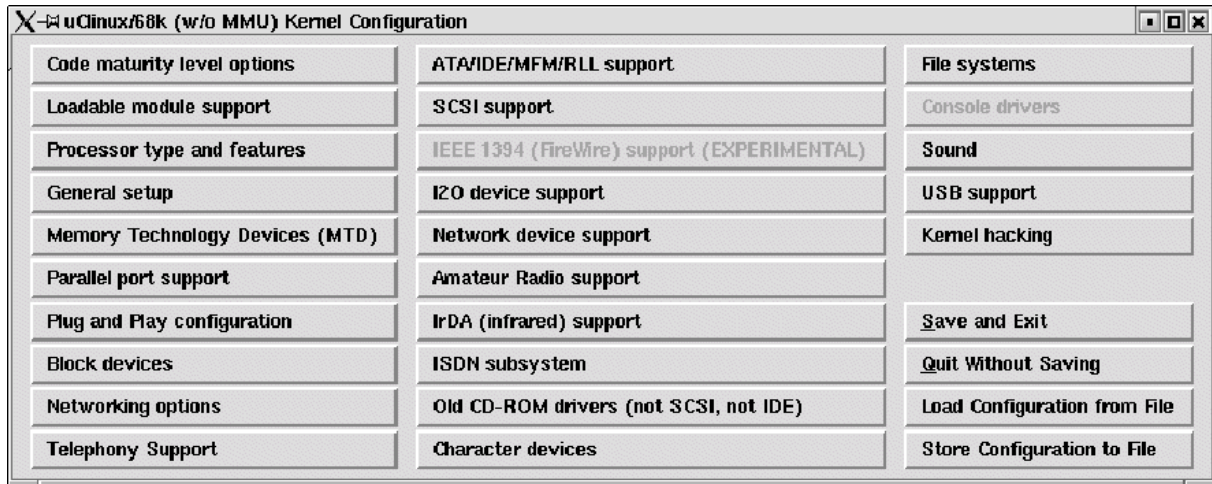
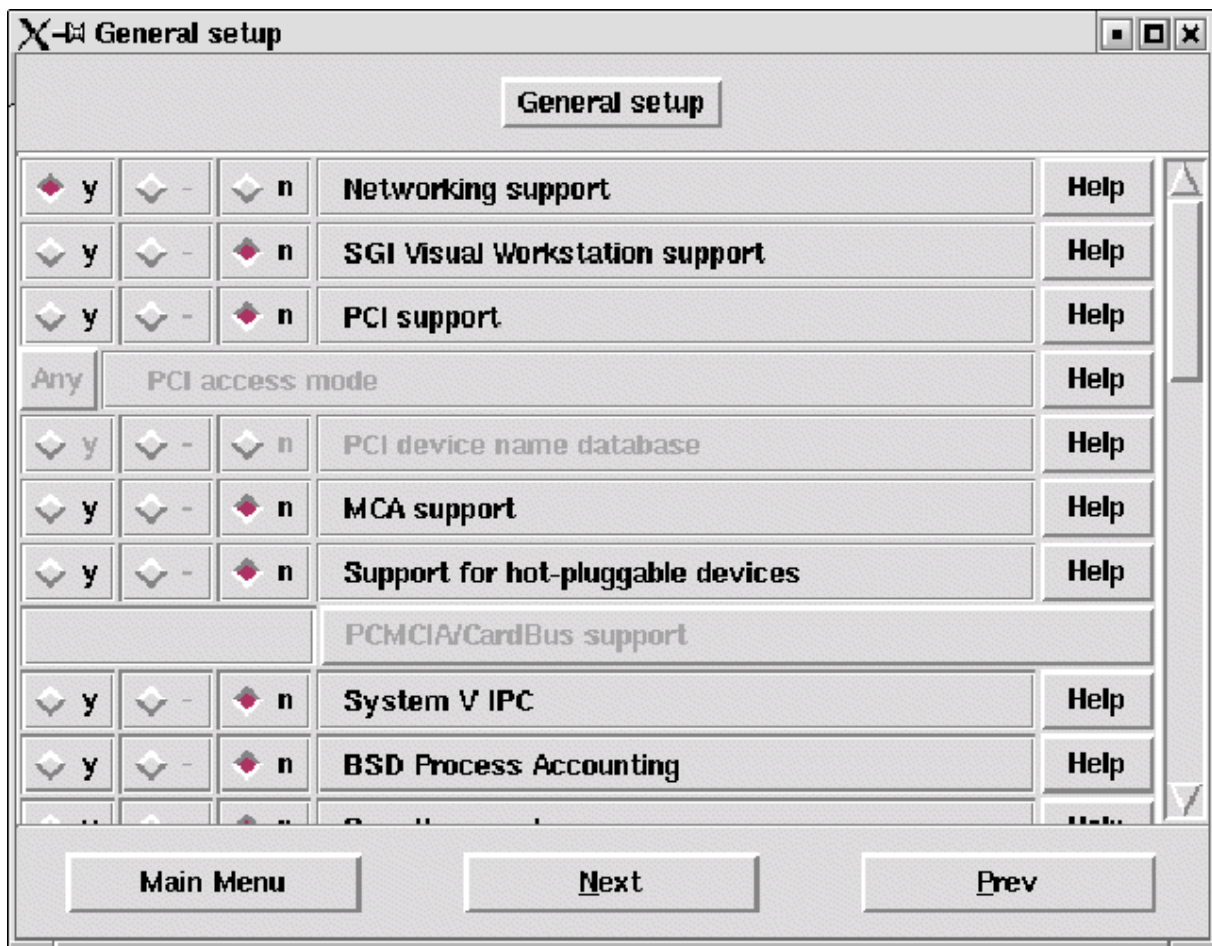
```
% tar -xvzf uClinux-dist.xxxxxxxx.tar.gz
```
- Se placer ensuite dans le répertoire `uClinux-dist` ainsi créé. **L'ensemble du travail sera réalisé à partir de ce répertoire !**
- Configurer le package  $\mu$ Clinux (**noyau 2.4.x**) pour la carte Motorola M5407C3 en utilisant la commande (voir les figures suivantes pour les options à cocher) :  

```
% make xconfig
```



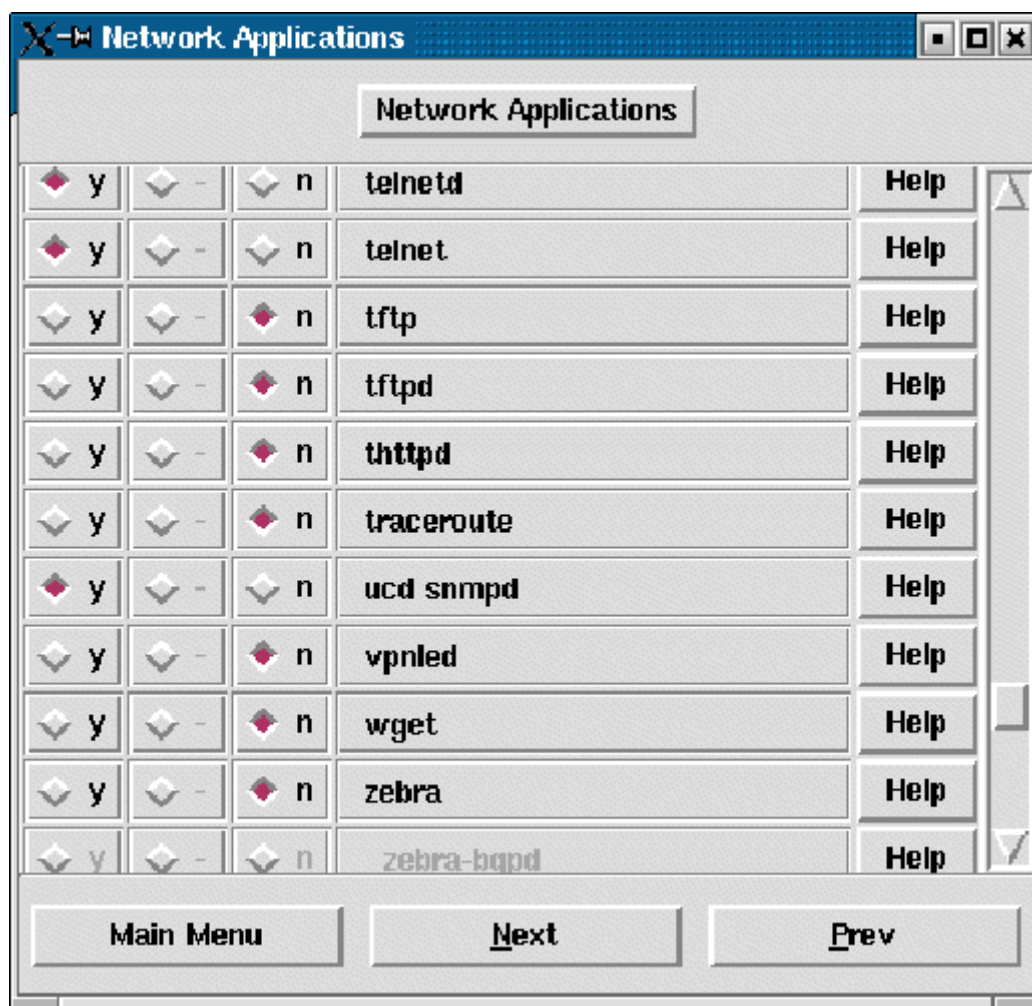
Menu 1 : Menu général

Menu 2 : Choix du BSP (à configurer comme ci-dessus). Validation de la configuration du noyau  $\mu$ Clinux et des applications userland

Menu 3 : Configuration du noyau  $\mu$ ClinuxMenu 4 : Exemple de configuration du noyau  $\mu$ Clinux



Menu 5 : Configuration des applications userland



Menu 6 : Exemple de configuration des applications userland

- Dans le menu Games, choisir le package dungeon.
- Après sauvegarde, générer les dépendances :  
% make dep

- Compiler pour générer le fichier image `image.bin` du noyau  $\mu$ Clinux :  

```
% make
```
- Où se trouve installé le fichier image ainsi créé sur l'hôte ?
- Comment générer le fichier S-Record du noyau ? Le faire.
- Télécharger par le réseau l'image  $\mu$ Clinux sur la cible par le réseau. Lancer le noyau  $\mu$ Clinux. Vérifier que l'application jeu `dungeon` est disponible. Que conclure sur la configurabilité de  $\mu$ Clinux.
- Télécharger le fichier S-Record précédemment généré dans la cible par la liaison série depuis `minicom`. Aller prendre un café... Conclusion par rapport à la première méthode. En quoi cette dernière méthode est-elle quand même intéressante ?  
Côté carte cible : `dBUG> dl 20000`  
Côté PC hôte : depuis `minicom`, taper sur les touches `CTRL+A`, puis `Z`, puis `S` (*Send*) puis choisir le fichier S-Record.

## 5. TP 3 : MISE EN ŒUVRE DE NFS SOUS $\mu$ CLINUX

Cet exercice permet de mettre en œuvre NFS côté client sur la cible. On en verra l'intérêt au fil du TP...

- Générer un noyau  $\mu$ Clinux avec la validation des commandes `mount/umount` pour le montage NFS. On choisira le noyau **linux 2.0.x** pour travailler avec NFS version 2 :  

```
% make xconfig
```

et  

```
uClinux      application      Configuration>      Filesystem
Applications> valider mount/umount
```
- Par analyse du fichier `/etc/exports` sur le PC hôte, retrouver le point de montage NFS depuis la cible.
- TRUC : On pourra recopier le fichier `/etc/hosts` du PC hôte dans le répertoire `uClinux-dist/romfs/etc` pour travailler avec les noms symboliques des machines et non leur adresse IP décimale. Il est nécessaire de refaire un `make` afin de recréer le `romfs`.  

```
% make
% cp /etc/hosts uClinux-dist/romfs/etc
% make
```
- Télécharger puis lancer le noyau  $\mu$ Clinux. Configurer l'interface réseau de la cible puis effectuer le montage NFS depuis la cible en utilisant comme répertoire de montage `/mnt`.
- Tester le système de fichiers NFS : création d'un fichier, exécution d'un fichier.

- Quel est l'intérêt d'un montage NFS client sur la cible dans le processus de développement d'une application ?

## 6. TP 4 : CREATION D'UNE APPLICATION USERLAND SOUS $\mu$ CLINUX

On désire maintenant développer une application  $\mu$ Clinux exécutée par la cible. Il s'agit de la célèbre application « Hello World ! » de K&R.

On suivra le HOWTO Adding-User-Apps-HOWTO donné en annexe.

- Création du répertoire hello dans le userland sous `uClinux-dist/user` à partir du répertoire `ledcon` s'y trouvant :

```
% cd uClinux-dist/user
% cp -r ledcon hello
```
- Modification du fichier `uClinux-dist/user/hello/Makefile` et création du fichier source C `uClinux-dist/user/hello/hello.c`.
- Modification du fichier `uClinux-dist/user/Makefile` par ajout de la directive :

```
dir_$(CONFIG_USER_HELLO_HELLO) += hello
```
- Modification du fichier `uClinux-dist/config/config.in` par ajout de la directive dans le menu Miscellaneous Applications :

```
bool 'hello' CONFIG_USER_HELLO_HELLO
```
- Edition du fichier `uClinux-dist/user/hello/hello.c`.
- On régénérera le noyau  $\mu$ Clinux avec le support NFS (noyau 2.0.x) et la validation de l'application `hello` :

```
% make xconfig
```
- Générer les dépendances, compiler et télécharger le noyau dans la cible puis le lancer.
- Tester la commande locale `hello` sous `/bin` de la cible.
- Faire de même par NFS avec la commande `hello` du PC hôte.

## 7. TP 5 : DEVELOPPEMENT D'UNE APPLICATION SERVEUR EMBARQUEE SOUS $\mu$ CLINUX ET D'UNE APPLICATION CLIENTE LINUX POUR LE CONTROLE DE LA CARTE CIBLE

On désire mettre en œuvre une application Client/Serveur TCP/IP pour le contrôle des leds de la carte cible. Ce TP met ici en valeur la connectivité IP à l'aide d'une interface de contrôle propriétaire (la sienne) d'un système embarqué.

- On appliquera les acquis du TP précédent.
- Créer une application userland myserver que l'on intégrera sous  $\mu$ Clinux. On développera un serveur TCP qui acceptera les requêtes envoyées par un client TCP dans la socket de connexion. La requête envoyée par le client comportera 2 octets :

Octet 1 : numéro de la led à contrôler (entre 0 et 7)

Octet 2 : valeur de la led (0 : led éteinte, 1 : led allumée)

Le serveur enverra en retour un octet comme code de retour :

0 : OK

1 : ERREUR

```
% cd uClinux-dist/user
```

```
% cp -r ~kadionik/myserver .
```

- On pourra utiliser le fichier squelette myserver0.c à renommer myserver.c comme base de travail (structure de données, enchaînement des appels systèmes).

La carte cible possède 8 leds (en haut à droite) dont 4 seulement sont réellement contrôlables :

LED 0 : correspond à la led 8 de la carte cible (la plus à gauche).

LED 1 : correspond à la led 7 de la carte cible.

LED 2 : correspond à la led 6 de la carte cible.

LED 3 : correspond à la led 5 de la carte cible.

Pour piloter ces 4 leds, on utilise un registre interne 16 bits du processeur ColdFire : le registre PADAT. Seuls les bits 7, 6, 5 et 4 sont utilisés dans ce registre. Il convient de ne pas changer la valeur courante des autres bits sous peine de planter la carte cible...

short \*PADAT = (short \*) 0x10000244 : est le registre 16 bits pour piloter les leds de la carte cible.

```
LED 0 ON : bit7 = 0 *PADAT &= 0xFF7F; led 8 carte
LED 0 OFF : bit7 = 1 *PADAT |= 0x0080; led 8 carte
```

```
LED 1 ON : bit6 = 0 *PADAT &= 0xFFBF; led 7 carte
LED 1 OFF : bit6 = 1 *PADAT |= 0x0040; led 7 carte
```

```
LED 2 ON : bit5 = 0 *PADAT &= 0xFFDF; led 6 carte
LED 2 OFF : bit5 = 1 *PADAT |= 0x0020; led 6 carte
```

```
LED 3 ON : bit4 = 0 *PADAT &= 0xFFEF; led 5 carte
```



```
LED 3   OFF   : bit4   = 1           *PADAT |= 0x0010; led 5 carte
```

- Configurer le noyau  $\mu$ Clinux (**noyau 2.4.x**) pour utiliser l'application userland myserver :  
    % make xconfig
- Créer les dépendances :  
    % make dep
- Compiler le noyau  $\mu$ Clinux :  
    % make
- Après compilation, téléchargement et lancement du noyau  $\mu$ Clinux dans la cible, on lancera l'application myserver sur le numéro de port 2000 :  
    % myserver 2000
- Créer l'application cliente de test myclient.c. On pourra utiliser le fichier squelette myclient0.c sous ~kadionik comme base de travail (structure de données, enchaînement des appels systèmes).  
    % cp ~kadionik/myclient0.c .
- Tester le tout :  
    % myclient @IP\_cible 2000
- Que faut-il penser de cette méthode de connectivité IP ?

## 8. TP 6 : MISE EN ŒUVRE D'UN CLIENT SMTP SOUS LINUX

On va dans un premier temps tester le serveur de mails SMTP du PC hôte sous Linux et comprendre le dialogue entre un client et un serveur SMTP et constater que :

- Il est du type commande/réponse.
- Il est orienté flux d'octets structurés sous forme de chaînes de caractères ASCII.
- Se connecter par telnet au service mail de la machine. Les commandes envoyées au serveur de mail (protocole SMTP *Simple Mail Transfer Protocol*, RFC821) sont structurées sous forme de lignes de commandes ASCII. Un exemple d'échanges de commandes SMTP est proposé ci-après (C: commande à taper, R: réponse du serveur SMTP) :

```
R: 220 kiwil ESMTP Sendmail 8.12.8/8.12.8; Wed, 4 Feb 2004
```

```
C: HELO enseirb.fr
```

```
R: 250 kiwil Hello localhost [127.0.0.1], pleased to meet you
```

```
C: MAIL FROM:<root@localhost>
```



```
R: 250 2.1.0 <root@localhost>... Sender ok
C: RCPT TO:<root@localhost>
R: 250 2.1.5 <root@localhost>... Recipient ok
C: DATA
R: 354 Enter mail, end with "." on a line by itself
un test
.
R: 250 2.0.0 i14Lo6ND001260 Message accepted for delivery
C: QUIT
R: 221 2.0.0 kiwi1 closing connection
```

- En s'aidant de l'exemple, envoyer un mail à l'utilisateur `uclinux`.
- Vérifier sa bonne réception en utilisant la commande de lecture de mail `mail`.

Pour développer une application cliente SMTP sous Linux, il convient donc de développer une application réseau cliente comme dans le TP précédent en formattant le flux de données échangées sous forme de chaînes de caractères ASCII respectant le protocole SMTP.

Pour cela, on va se simplifier la vie et partir d'un client SMTP existant issu des sources du livre de T. Jones cité en référence. On consultera les sources donnés en annexe.

- Se placer dans le répertoire `mon_nom` et recopier le répertoire `mysmtp` :

```
% cd
% cd mon_nom
% cp -r ~kadionik/mysmtp .
% cd mysmtp
```
- Ce répertoire contient un fichier C principal `main.c` et un fichier bibliothèque de fonctions `emstp.c` et son fichier `.h` associé ainsi qu'un fichier `Makefile`. Par analyse du code source, modifier les fichiers C pour envoyer un mail de format texte et de format HTML à l'utilisateur `uclinux` du PC hôte (`kiwi???`).
- Compiler et tester le client SMTP

## 9. TP 7 : MISE EN ŒUVRE D'UN CLIENT SMTP EMBARQUE SOUS $\mu$ CLINUX

Il convient maintenant de transformer l'exemple précédent en application userland  $\mu$ Clinux.

- Créer une application userland `smtp` que l'on intégrera sous  $\mu$ Clinux :

```
% cd uClinux-dist/user
% mkdir smtp
```

```
% cp ~/kadionik/mysmtp/* smtp
```

- Comme pour le TP 4, intégrer l'application `smtp` dans la distribution  $\mu$ Clinux. Modifier les programmes sources pour que le client SMTP émette un email à destination de l'utilisateur `uclinux` du PC hôte toutes les 10 secondes.
- On régénérera le noyau  $\mu$ Clinux avec la validation de l'application `smtp` :

```
% make xconfig
```
- Générer les dépendances, compiler et télécharger le noyau dans la cible puis le lancer :

```
% make
% cp /etc/hosts uClinux-dist/romfs/etc
% make
```
- Tester la commande locale `smtp` sous `/bin` de la cible.
- Que faut-il penser de cette méthode de connectivité IP ?

## 10. TP 8 : MISE EN ŒUVRE D'UN SERVEUR WEB EMBARQUE SOUS $\mu$ CLINUX

Ce TP se propose de mettre en œuvre un serveur WWW sous Linux embarqué. C'est généralement cette solution qui est mise en œuvre sur un système embarqué pour assurer une connectivité IP. Le but est de piloter par le WWW les leds de la carte cible...

- Configurer le noyau  $\mu$ Clinux (**noyau 2.4.x**) pour utiliser le package `boa` avec le support des scripts CGI génériques (application `userland cgi_generic` commun à tous les serveurs WWW portés sous  $\mu$ Clinux) :

```
% make xconfig
```

et

```
uClinux application Configuration> Network Applications>
valider boa
uClinux application Configuration> Miscellaneous
Configuration> valider generic cgi
```
- Modification du fichier `uClinux-dist/user/Makefile` par ajout de la directive :

```
dir_$(CONFIG_USER_CGI_GENERIC) += cgi_generic
```
- Générer les dépendances :

```
% make dep
```
- Se placer dans le répertoire `uClinux-dist` et recopier le patch `mywwwpatch` pour la configuration du serveur `boa` avec support des CGI et la compilation du CGI (exécutable) `enseirb` pour le pilotage des leds de la cible :

```
% cd uClinux-dist
% cp -r ~/kadionik/mywwwpatch .
```

- Se placer dans `uClinux-dist/mywwwpatch` et exécuter le shell script de patch :  
% `cd uClinux-dist/mywwwpatch`  
% `./mywwwpatch`
- Le fichier source CGI pour le pilotage des leds de la cible est le fichier C `enseirb.c` situé sous dans `uClinux-dist/user/cgi_generic`. Modifier ce fichier C pour traiter les commandes CGI envoyées par le navigateur WWW. Dans un CGI, les E/S sont redirigées. Il convient simplement d'utiliser des appels systèmes `printf()` pour générer l'entête HTTP et les données de la réponse renvoyée vers le navigateur. Une requête CGI est de la forme `http://@IP:port/cgi-bin/mon_cgi?param1`. Le CGI a accès a des variables d'environnement CGI et la variable d'environnement `QUERY_STRING` contient les paramètres de la requête envoyée par le navigateur. Modifier le fichier C `enseirb.c` pour contrôler l'allumage (`param1=1`) ou l'extinction (`param1=0`) de la led 1.
- Après compilation, téléchargement et lancement du noyau  $\mu$ Clinux dans la cible, on testera le serveur WWW embarqué. On le lancera à la main à l'aide de la commande :  
% `boa -c /home/httpd`
- Il est à noter que le répertoire racine du serveur `boa` sur la cible est `/home/httpd`. Le fichier de configuration de `boa` est le fichier `/home/httpd/boa.conf`. Le fichier de d'accueil de `boa` est le fichier `/home/httpd/index.html`. Le répertoire des scripts CGI est le répertoire `/home/httpd/cgi-bin`.
- A l'aide du navigateur Netscape, contrôler la led 1 de la carte cible.
- Que faut-il penser de cette méthode de connectivité IP ?

## 11. TP 9 : MISE EN ŒUVRE DE SNMP SOUS LINUX

Cet exercice traite de la mise en œuvre de SNMP sous Linux. La finalité est d'illustrer la connectivité IP par SNMP d'un équipement électronique (module connecté sur le port parallèle du PC).

- Lancer l'agent SNMP du package NET-SNMP sur le PC hôte :  
% `snmpd`
- Quel est le port d'écoute de l'agent SNMP ? On utilisera la commande `netstat`.
- A l'aide du document donné en annexe, retrouver les commandes SNMP pour accéder aux objets `sysUpTime` et `sysDescr` de la MIB gérée par l'agent. La communauté à utiliser pour les opérations SNMP GET et SET est `tsst`.
- A l'aide d'une commande SNMP, parcourir la branche `system` de la MIB.
- A l'aide d'une commande SNMP, parcourir toute la MIB.

- L'agent SNMP du PC gère 8 leds D0 à D7 connectées sur le port parallèle. A l'aide d'une commande SNMP, retrouver l'état courant des 8 leds.
- Lancer dans une fenêtre l'outil `tcpdump` pour l'analyse du trafic :  

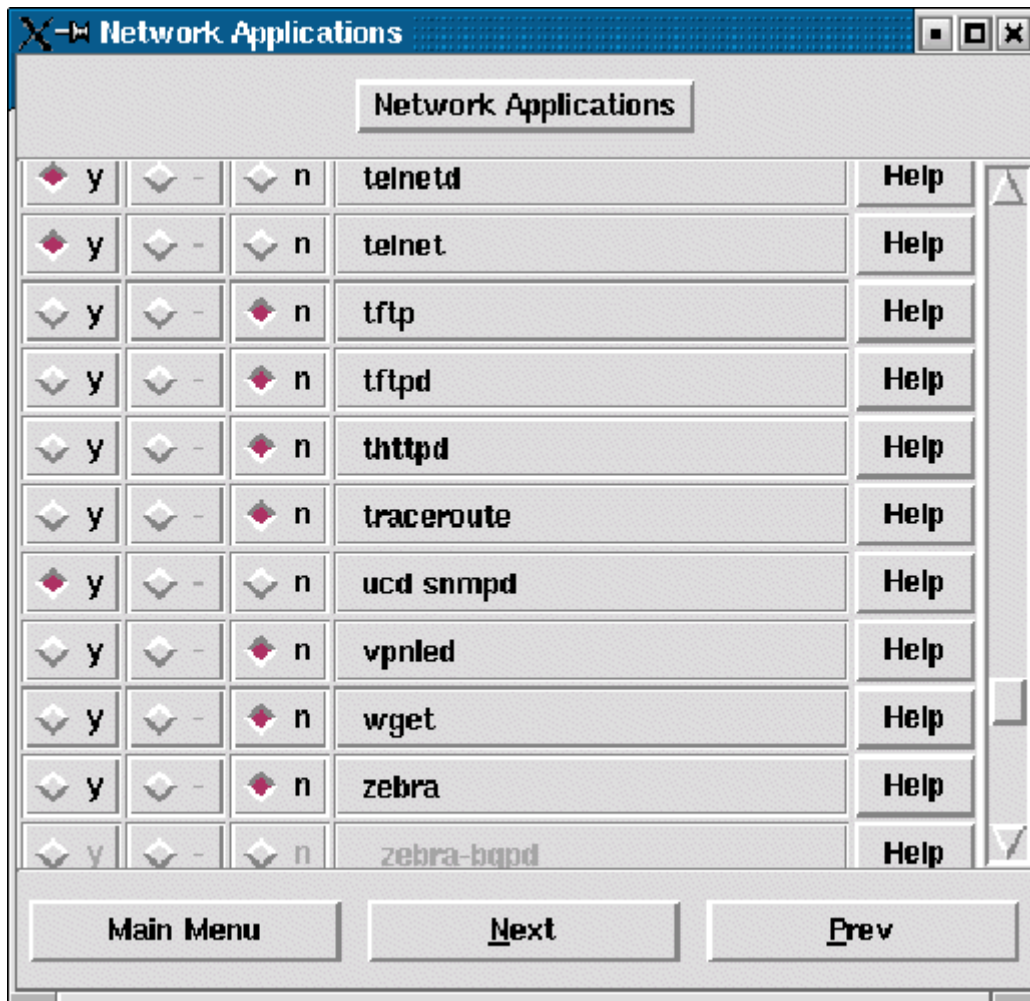
```
% tcpdump -X -s 0 -vv -i lo
```
- Lancer dans une fenêtre la commande `snmptrapd` pour la collecte des traps SNMP générés par l'agent SNMP.
- Faire de même pour les leds 2 et 3 en mode ligne de commande.
- Ecrire un shell script réalisant un chenillard avec les leds en utilisant SNMP.

## 12. TP 10 : MISE EN ŒUVRE D'UN AGENT SNMP EMBARQUE SOUS $\mu$ CLINUX

Ce TP se propose de mettre en œuvre SNMP sous Linux embarqué. Intégrer un agent SNMP sur un système embarqué est une des solutions mises en œuvre pour assurer une connectivité IP.

- Configurer le noyau  $\mu$ Clinux (**noyau 2.4.x**) pour utiliser le package SNMP (application `userland ucd snmpd`):  

```
% make xconfig
```



Menu : validation de l'application userland ucd snmpd

- Générer les dépendances :  

```
% make dep
```
- Se placer dans le répertoire `uClinux-dist/user/ucdsnmp` et recopier le patch `mysnmppatch` pour étendre l'agent SNMP :  

```
% cd uClinux-dist/user/ucdsnmp  
% cp -r ~kadionik/mysnmppatch .
```
- Le patch étend donc l'agent SNMP pour le contrôle des leds de la carte cible en utilisant la même MIB que sous Linux .
- Se placer dans `uClinux-dist/user/ucdsnmp/mysnmppatch` et exécuter le shell script de patch :  

```
% cd uClinux-dist/user/ucdsnmp/mysnmppatch  
% ./mysnmppatch
```
- L'extension de l'agent SNMP avec la MIB ENSEIRB apparaît dans le répertoire `uClinux-dist/user/ucdsnmp/agent/mibgroup` sous la forme de 2 fichiers C `enseirb.c` et `enseirb.h`. Le fichier `enseirb.c` contient le code C pour le contrôle des leds de la carte cible.

- Il convient de le compléter avec le code C nécessaire pour piloter chaque led suivant les consignes suivantes (voir les consignes sur le TP 7 sur le client SMTP) :

```
LED 0   ON    : bit7      = 0
LED 0   OFF   : bit7      = 1

LED 1   ON    : bit6      = 0
LED 1   OFF   : bit6      = 1

LED 2   ON    : bit5      = 0
LED 2   OFF   : bit5      = 1

LED 3   ON    : bit4      = 0
LED 3   OFF   : bit4      = 1

LED 4   ON    : envoi d'un TRAP SNMP (fonction send_easy_trap() )
LED 4   OFF   : envoi d'un TRAP SNMP (fonction send_easy_trap() )

LED 5   ON    : envoi d'un TRAP SNMP (fonction send_easy_trap() )
LED 5   OFF   : envoi d'un TRAP SNMP (fonction send_easy_trap() )

LED 6   ON    : envoi d'un TRAP SNMP (fonction send_easy_trap() )
LED 6   OFF   : envoi d'un TRAP SNMP (fonction send_easy_trap() )

LED 7   ON    : envoi d'un TRAP SNMP (fonction send_easy_trap() )
LED 7   OFF   : envoi d'un TRAP SNMP (fonction send_easy_trap() )
```

On trouvera des informations sur la fonction `send_easy_trap()` dans le manuel en ligne :

```
% man snmp_trap_api
```

- On ajustera le fichier de configuration de l'agent SNMP `uclinux-dist/user/ucdsnmp/mysnmpd.conf` pour coller aux adresses IP de la cible et de l'hôte.
- Après compilation, téléchargement et lancement du noyau  $\mu$ Clinux sur la cible, on testera l'agent SNMP embarqué. On lancera dans une fenêtre `tcpdump` et `snmptrapd` pour l'analyse du trafic SNMP.
- Allumer la led 0 de la cible en utilisant SNMP depuis le PC hôte.
- Eteindre la led 0 de la cible en utilisant SNMP depuis le PC hôte.
- Allumer la led 7 de la cible en utilisant SNMP depuis le PC hôte.
- Eteindre la led 7 de la cible en utilisant SNMP depuis le PC hôte.
- Vérifier la cohérence du trafic SNMP généré.
- Que faut-il penser de cette méthode de connectivité IP ?

## 13. TP 11 : SYSTEME DE FICHIERS JFFS2 POUR MEMOIRE FLASH

Le système de fichiers JFFS2 (*Journaling Flash File System 2*) est un système de fichiers Linux développé spécialement pour être utilisé sur une mémoire de type FLASH que l'on retrouve fréquemment dans un système embarqué. Cela peut être un circuit intégré de mémoire FLASH, une carte Compact Flash, une clé USB. La particularité d'une mémoire FLASH est sa programmation par secteur (512 octets à quelques Ko) qui nécessite d'effacer complètement le secteur pour y programmer ne serait-ce qu'un seul octet. De plus, il convient de supporter les pannes d'alimentation durant la phase de reprogrammation. JFFS2 est prévu pour cela : c'est un système de fichier journalisé qui stocke toutes les versions d'un fichier modifié, ce qui permet d'accélérer la phase de boot d'un système Linux (pas de fsck intempestif) et d'assurer l'intégrité des fichiers stockés. Il intègre un processus de ramasse miettes (*Garbage Collector*) quand le taux d'occupation du système de fichiers est proche de 80-90 % afin de libérer de la place par suppression des trop anciennes versions d'un fichier mais cela a un coût : le temps d'exécution du GC.

On va regarder dans ce TP la mise en œuvre de JFFS2. On se reportera à l'annexe pour une présentation détaillée de la mise en œuvre de JFFS2 sous  $\mu$ Clinux. On va stocker des fichiers dans le système de fichiers JFFS2 et vérifier leur existence même après coupure de l'alimentation de la carte cible.

La carte cible possède une mémoire FLASH 16 bits (AMD Am29PL160C) mappée dans l'espace d'adressage de \$7FE00000 à \$7FFFFFFF.

Les explications en anglais du HOWTO donné en annexe précise les points importants suivants :

You have to read carefully the technical data on your FLASH memory chip and find its sector address table :

Sector	A19	A18	A17	A16	A15	A14	A13	A12	Sector Size (Kbytes/ Kwords)	Address Range (in hexadecimal)	
										Byte Mode (x8)	Word Mode (x16)
SA0	0	0	0	0	0	0	0	X	16/8	000000-003FFF	00000-01FFF
SA1	0	0	0	0	0	0	1	0	8/4	004000-005FFF	02000-02FFF
SA2	0	0	0	0	0	0	1	1	8/4	006000-007FFF	03000-03FFF
SA3	0	0	0		01000-11111				224/112	008000-03FFFF	04000-1FFFF
SA4	0	0	1	X	X	X	X	X	256/128	040000-07FFFF	20000-3FFFF
SA5	0	1	0	X	X	X	X	X	256/128	080000-0BFFFF	40000-5FFFF
SA6	0	1	1	X	X	X	X	X	256/128	0C0000-0FFFFF	60000-7FFFF
SA7	1	0	0	X	X	X	X	X	256/128	100000-13FFFF	80000-9FFFF
SA8	1	0	1	X	X	X	X	X	256/128	140000-17FFFF	A0000-BFFFF
SA9	1	1	0	X	X	X	X	X	256/128	180000-1BFFFF	C0000-DFFFF
SA10	1	1	1	X	X	X	X	X	256/128	1C0000-1FFFFFF	E0000-FFFFFF

The FLASH memory has 11 sectors with different sizes. Its contains the dBUG Motorola monitor in the first 256 Kbytes (SA0 to SA3). The rest of the memory is available to the user (SA4 to SA10, 256 Kbytes each). 2 MTD partitions are defined:

- The dBUG partition: \$7FE00000 to \$7FE3FFFF (256 Kbyte size). Read Only. Don't erase!
- The user partition: \$7FE40000 to \$7FFFFFFF (1792 Kbyte size).

**It's important to notice (read in the  $\mu$ Clinux archive) that you have to create each MTD partition with at least 6 contiguous sectors in order to use JFFS2 (for Garbage Collection).**

- Se placer dans le répertoire `uClinux-dist` et recopier le patch `myjffs2patch` pour la configuration de la distribution  $\mu$ Clinux pour le support de JFFS2 :

```
% cd uClinux-dist
% cp -r ~kadionik/myjffs2patch .
```
- Se placer dans `uClinux-dist/myjffs2patch` et exécuter le shell script de patch :

```
% cd uClinux-dist/myjffs2patch
% ./myjffs2patch
```
- Vérifier la configuration correcte du noyau  $\mu$ Clinux (noyau 2.4.x, pour la cible M5407C3) et notamment les points suivants :

```
% make xconfig
```

et

```
uClinux/68k (w/o MMU) Kernel Configuration>File
systems>Journalling Flash File System v2 (JFFS2) support>
valider y
```

et

```
uClinux application Configuration> Flash Tools> mtd-utils>
valider y, erase, mkfs.jff2
```

et

```
uClinux application Configuration> BusyBox> valider y, dd,
mount, mount: loop devices, umount
```



- Créer les dépendances :  
% make dep
- Compiler le noyau  $\mu$ Clinux :  
% make
- Après compilation, téléchargement et lancement du noyau  $\mu$ Clinux dans la cible, on testera le système de fichiers JFFS2. On peut voir que 2 partitions MTD pour être utilisées avec JFFS2 ont été créées (/dev/mtd0 et /dev/mtd1). On le vérifiera aussi dans les traces de boot du noyau :

```
/> cd /proc
```

```
/proc> cat mtd
```

```
dev: size erasesize name
```

```
mtd0: 00040000 00038000 "dBUG (256K)"
```

```
mtd1: 001c0000 00040000 "user (1792K)"
```

- Construire un système de fichiers JFFS2 sommaire à installer dans la partition MTD numéro 1 (/dev/mtd1):

```
/proc> cd /tmp
/var/tmp> mkdir jffs2
/var/tmp> mkdir jffs2/bin
/var/tmp> cd jffs2
/var/tmp/jffs2> vi file1
/var/tmp/jffs2> cat file1
coucou
/var/tmp/jffs2> cd ..
/var/tmp> mkfs.jffs2 -d jffs2 -o jffs2.img
/var/tmp> erase /dev/mtd1
/var/tmp> cp jffs2.img /dev/mtd1
MTD_open
MTD_write
MTD_close
```

- Monter la partition JFFS2 sur /mnt :

```
/var/tmp> mount -t jffs2 /dev/mtdblock1 /mnt
mtdblock_open
ok
/var/tmp> cd /mnt
/mnt> ls
bin
file1
toto
/mnt> cd /proc
/proc> cat mounts
rootfs / rootfs rw 0 0
```



```
/dev/root / romfs ro 0 0
/proc /proc proc rw 0 0
/dev/ram1 /var ext2 rw 0 0
/dev/mtdblock1 /mnt jffs2 rw 0 0
```

- Démonter la partition JFFS2 :

```
/proc> umount /mnt
mtdblock_release
ok
```

- Arrêter la cible, couper l'alimentation, rebooter le noyau  $\mu$ Clinux et vérifier de la présence intacte du système de fichiers JFFS2.

## 14. TP 12 : DEBUG D'APPLICATIONS $\mu$ CLINUX AVEC GDBSERVER

On désire maintenant étudier le debug d'une application  $\mu$ Clinux. On a à disposition 2 méthodes :

- Utilisation du port de debug Motorola BDM couplé au debugger GNU GDB. Cela nécessite une sonde connectée entre le port BDM de la carte cible et le port parallèle du PC. Cette méthode permet un debug en Temps Réel.
- Utilisation de l'accès réseau Ethernet de la carte cible. On utilise alors une application client/serveur GDB sur l'hôte et `gdbserver` sur la cible. Cette méthode est à utiliser pour déverminer une application « marchant à peu près ».

En cas de gros plantage, on peut revenir à la première méthode plus lourde ou utiliser aussi un émulateur ! On mettra en œuvre la méthode de debug par le réseau plus simple.

- Créer une application `userland bug` que l'on intégrera sous  $\mu$ Clinux :

```
% cd uClinux-dist/user
% mkdir bug
% cp ~kadionik/mybug/* bug
```
- Configurer le noyau  $\mu$ Clinux (noyau 2.4.x) pour utiliser l'application `userland bug` et `gdbserver` :

```
% make xconfig
```

- Créer les dépendances :  
% make dep
- Compiler le noyau  $\mu$ Clinux :  
% make
- Après compilation, téléchargement et lancement du noyau  $\mu$ Clinux dans la cible, on lancera l'application bug :  
% /bin/bug
- Que se passe-t-il ?
- Relancer le noyau  $\mu$ Clinux.
- Lancer d'abord l'application serveur gdbserver sur la cible :  
% gdbserver :3000 /bin/bug
- Le debug se fera sur le port socket 3000. Lancer sur l'hôte l'application cliente GDB depuis le répertoire de compilation de bug.c pour bénéficier du debug au niveau symbolique :  
% cd uClinux-dist/user/bug  
% m68k-bdm-elf-gdb bug.gdb
- Au prompt GDB, se connecter au serveur gdbserver :  
(gdb) target remote @IP\_cible:3000
- Utiliser alors les commandes GDB pour localiser le bug et la cause du plantage (commandes s, c, print...).
- Les commandes en ligne de GDB étant assez étonnantes, on peut utiliser un debugger comme DDD qui possède une interface graphique conviviale. Il faut voir DDD comme un « front end » à GDB. Comme précédemment, après avoir lancé gdbserver :  
% cd uClinux-dist/user/bug  
% ddd --debugger m68k-bdm-elf-gdb bug.gdb
- Dans la fenêtre de commandes GDB de DDD, se connecter au serveur gdbserver :  
(gdb) target remote @IP\_cible:3000
- Debugger maintenant de façon graphique avec DDD. Conclusion sur les 2 méthodes.

## 15. TP 13 : DEBUG D'APPLICATIONS $\mu$ CLINUX PAR LE BDM

On se reportera à l'annexe pour avoir une vue d'ensemble du procédé de debug par le *BDM* (*BackGround Debug Monitor*).

Le BDM est un OCD (*OnChip Debugger*). Il permet de debugger :

- Une application en Temps Réel seule (pas de système d'exploitation). Ce cas ne sera pas traité ici (voir en annexe pour la méthodologie).
  - Le noyau  $\mu$ Clinux.
  - Le noyau  $\mu$ Clinux et les applications userland globalement.
- 
- On vérifiera aussi que le câble BDM est bien branché sur la cible (cible éteinte !!!).
  - Il convient d'abord de charger le module Linux bdm sur le PC hôte :  
    % gobdm
  - On vérifiera que le module bdm est bien chargé.
  - Lors de la configuration du noyau  $\mu$ Clinux (choisir le noyau version 2.4), vérifier que les options suivantes sont validées pour pouvoir debugger au niveau symbolique :  
  
    % make xconfig  
et  
     $\mu$ Clinux/68k (w/o MMU) Kernel) Configuration> Kernel  
hacking> valider Full Symbolic/Source Debugging support  
et  
     $\mu$ Clinux application Configuration> Debug build> valider  
build debugable libraries et build debugable applications

## 15.1. Debugger le noyau $\mu$ Clinux

- Créer les dépendances :  
    % make dep
- Compiler le noyau  $\mu$ Clinux :  
    % make
- Se placer dans le répertoire uClinux-dist/images.  
    % cd uClinux-dist/images
- On utilisera DDD pour debugger. Le fichier d'initialisation 5407.gdb permet d'initialiser les ressources de la carte cible utilisée.  
    % ddd --debugger m68k-bdm-elf-gdb --command  
/usr/local/bdm/5407.gdb -n image.elf
- Dans la fenêtre ligne de commandes de DDD, on tapera la commande GDB cont pour lancer le programme en mémoire FLASH de la carte cible, c'est-à-dire ici le moniteur dBUG puis on téléchargera normalement le noyau et le lancera comme avant :  
    (gdb) cont (si besoin, taper avant target bdm  
/dev/bdm)  
    dBUG> dn  
    dBUG> go 20000

- Pour reprendre la main, il suffit de taper  $\wedge$ C au niveau de DDD. Il est alors possible de consulter la mémoire, l'état de variables du noyau. Cette manipulation est en fait intéressante quand on porte  $\mu$ Clinux sur une nouvelle cible et analyser le portage en cas de crash précoce du noyau.
- Pour prendre la main à un endroit particulier (avant le lancement du noyau, fonction `start_kernel()` ou ailleurs, il suffit d'insérer dans son code source l'instruction assembleur HALT dans un fichier assembleur ou l'instruction d'assemblage en ligne suivante dans le fichier source C :  

```
asm(«      HALT ») ;
```
- On modifiera le fichier source C du noyau `uClinux-dist/linux-2.4.x/init/main.c` pour prendre la main avant le montage de la partition root (avant l'appel de la fonction `mount_root()`). On recompilera le noyau, l'exécutera puis on vérifiera que l'on reprend la main depuis DDD, ce qui permettra de debugger ensuite en pas à pas le noyau...

## 15.2. Debugger une application sous $\mu$ Clinux

La méthode est similaire au debug du noyau. On reprend la main avec DDD préférentiellement :

1. Au crash de l'application userland.
  2. En insérant en ligne l'instruction assembleur HALT dans le source de l'application userland.
  3. En tapant  $\wedge$ C depuis DDD lors de l'exécution de l'application userland, il n'est pas sûr d'interrompre ce processus particulier à cause du scheduling.
- On peut avoir des infos supplémentaires en lançant la commande GDB de trace :  

```
(gdb) bt
```
  - Pour avoir le nom symbolique du processus interrompu pour le noyau 2.4 :  

```
(gdb) print _current_task->comm
```
  - Pour charger la table des symboles, il faut ensuite charger le fichier `mon_application.gdb` par la commande GDB :  

```
(gdb) add-symbol-file uClinux-dist/user/mon_application/mon_application.gdb _current_task->mm->start_code
```
  - En relançant la commande GDB de trace, on récupère alors des informations au niveau symbolique cette fois-ci et ainsi debugger en pas à pas :  

```
(gdb) bt
```

On regardera les notes en annexe pour plus d'informations sur ce mode de debug...

## 16. REFERENCES

- Cours ENSEIRB de Systèmes embarqués, Linux embarqué. P. Kadionik.  
<http://www.enseirb.fr/~kadionik/formation/embeddedlinux/introduction.html>
- TCP/IP Application Layer Protocols for Embedded Systems. M. Tim Jones. Editions Charles River Media. CDROM inclus avec sources utilisé pour le TP sur SMTP.
- Linux embarqué. P. ficheux. Editions Eyrolles.

**THAT'S ALL FOLKS !**

**A venir :**

- Bootloader Linux (colilo)



## **ANNEXE 1 : LINUX EMBARQUE**



## **ANNEXE 2 : PRESENTATION DE $\mu$ Clinux**

**ANNEXE 3 :**  
**MISE EN ŒUVRE DE  $\mu$ Clinux SUR PROCESSEUR**  
**COLDFIRE**



**ANNEXE 4 :**  
**MISE EN ŒUVRE DE SMTP SOUS Linux ET  $\mu$ Clinux**



## **ANNEXE 5 : INTERFACE CGI POUR SERVEUR WWW**

**ANNEXE 6 :**  
**MISE EN ŒUVRE DE SNMP SOUS Linux ET  $\mu$ Clinux**

**ANNEXE 7 :**  
**MISE EN ŒUVRE DE JFFS2 sous  $\mu$ Clinux**



**ANNEXE 8 :**  
**MISE EN ŒUVRE DU BDM  $\mu$ Clinux**



**ANNEXE 9 :**  
**DOCUMENTATION SUR LE PROCESSEUR**  
**COLDFIRE ET LA CARTE EVB5407C3**