

Développement du code métier

Christophe BLAESS

christophe.blaess@logilin.fr

<https://www.linkedin.com/in/christophe-blaess/>

<https://www.blaess.fr/christophe/>

twitter: @chrisblaess



Ingénierie et formations sur Linux et les logiciels libres

<https://www.logilin.fr>

Workflow du développement embarqué.....	3
Outils de développement libres.....	4
Chaîne de compilation.....	5
La chaîne de compilation Gnu : GCC (Gnu Compiler Collection).....	6
Développement et débogage distants.....	7
Chaîne de compilation de Buildroot.....	7
Cross-compilation.....	8
Travaux pratiques : compiler du code personnel avec la toolchain.....	9
Débogage distant – principes.....	10
Travaux pratiques : utilisation du débogueur GDB à distance.....	11
Outils d'aide au débogage.....	12
Ltrace & Strace.....	12
Valgrind.....	13
Travaux pratiques : utilisation de Valgrind.....	14

Ce support de formation est distribué sous licence **Creative Commons 4.0**



(Attribution - Partage dans les mêmes conditions).

Vous êtes libres de copier et partager ce document, en mentionnant son origine. Si vous l'intégrez dans un contenu plus vaste, ce dernier devra être distribué avec les mêmes droits.

Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap

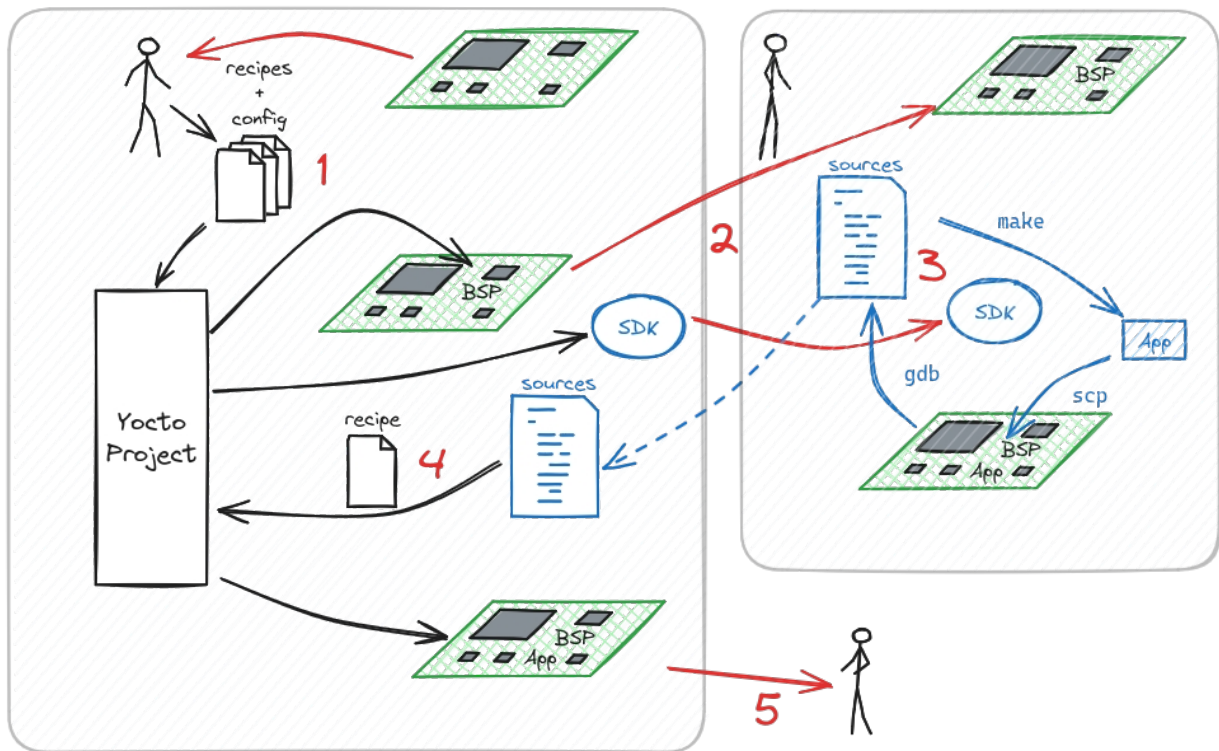
Linux embarqué

ILE v.5.4

<https://www.blaess.fr/christophe/>

<https://www.logilin.fr>

Workflow du développement embarqué



1. L'équipe de développement BSP prépare le support pour une nouvelle carte embarquée.
2. L'équipe BSP fournit la carte et le SDK aux développeurs applicatifs.
3. Les développeurs applicatifs mettent au point le code métier en utilisant le SDK.
4. Les développeurs BSP intègrent les applicatifs métiers dans le *rootfs overlay*.
5. La carte comportant le BSP et le code métier est livrée aux clients. Hourra !

Outils de développement libres

Les distributions Linux fournissent un ensemble complet d'outils de développement libres et gratuits provenant pour la plupart du projet Gnu.

Le développement de modules du noyau (drivers, protocoles réseau, systèmes de fichiers, etc.) se fait uniquement en langage C.

La plupart des applications libres développées pour Linux sont écrites en C et C++.

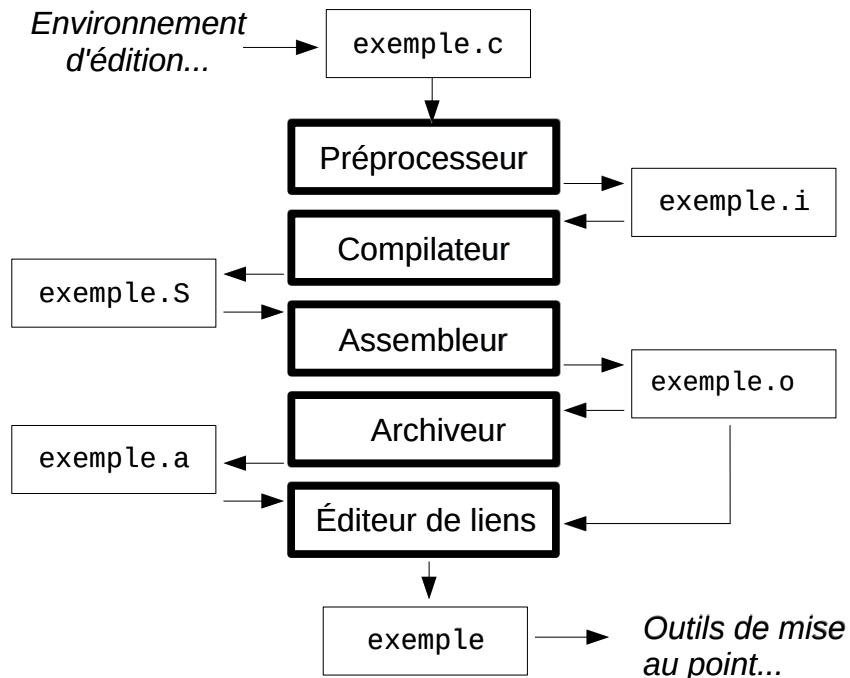
On utilise beaucoup de scripts pour l'administration et la configuration du système (Shell, Sed, Awk) ou pour réaliser des applications interactives simples (Python, Perl, etc.)

Le langage Java est également employé pour certaines applications portables sur différents systèmes d'exploitation.

Il existe également des environnements de développement et des chaînes de compilation propriétaires.

Chaîne de compilation

La chaîne de compilation est l'ensemble des outils qui vont permettre de transformer un code source en fichier exécutable.



La chaîne de compilation Gnu fonctionne sur la plupart des systèmes Unix et compatibles et peut produire du code pour de nombreux processeurs.

La chaîne de compilation *native* permet de créer un fichier exécutable pour le processeur sur lequel elle s'exécute.

Une chaîne de compilation *croisée* permet de produire du code s'exécutant sur un processeur différent.

La chaîne de compilation Gnu : GCC (Gnu Compiler Collection)

Le projet GCC fournit des compilateurs pour différents langages, utilisables sur de très nombreuses plates-formes, et produisant du code pour divers processeurs :

- gcc – (*Gnu C Compiler*) Langage C et Objective-C ;
- g++ – Langage C++
- gobjc – Objective-C++ ;
- g77 – Langage fortran 77 (avant GCC 4) ;
- gfortran – Langage fortran 95 (depuis GCC 4) ;
- gcj – Langage Java compilation en pseudo-code ou en code natif ;
- gnat – Langage Ada.
- go – Langage Go

D'autres langages sont supportés par GCC mais les modules correspondants ne sont pas maintenus aussi régulièrement que ceux indiqués ci-dessus : Modula, Pascal, PL/I...

Le projet Gnu propose aussi d'autres compilateurs :

gc l (Common Lisp), gprolog (Prolog), gst (Small Talk), gforth (Forth),

D'autres compilateurs ou interpréteurs libres sont disponibles hors du projet Gnu :

Python, Perl, Tcl/Tk, Ruby, Rust, Scheme, Smart Eiffel, Pascal, Java, O-Caml, Rexx, Logo...

Voir aussi

Web :

- *The Gnu Compiler Collection* : <http://gcc.gnu.org/>

Développement et débogage distants

Chaîne de compilation de Buildroot

Avec notre configuration, la *toolchain* produite par Buildroot se trouve dans le répertoire :

```
~/build-qemuarm/host/usr/bin/
```

Vous pouvez ajouter ce chemin dans votre variable d'environnement PATH dynamiquement :

```
$ PATH=$PATH:~/build-qemuarm/host/usr/bin/
```

ou ajouter cette ligne à la fin de votre fichier `~/ .bashrc` et relancer votre shell.

Pour rendre la chaîne de compilation indépendante de son emplacement dans l'arborescence :

```
$ make sdk
```

Extraire (sur un autre système éventuellement) l'archive trouvée dans `image/` puis lancer le script

```
$ ./relocate-sdk.sh
```

qui se trouve dans le nouveau répertoire.

Pour vérifier si le PATH est correct, essayez :

```
$ arm-linux-gcc -v
```

Cross-compilation

Le développement en C/C++ se fait simplement en ajoutant le préfixe d'accès à la chaîne de compilation croisée. Dans un *Makefile*, on initialise la variable `CROSS_COMPILE` :

```
CROSS_COMPILE ?=  
CC = $(CROSS_COMPILE)gcc  
  
all: hello  
  
hello: hello.c  
      $(CC) hello.c -o hello -Wall  
  
clean:  
      rm -f hello
```

On invoque

make

ou

make CROSS_COMPILE=arm-linux-

Pour avoir l'exécutable pour la cible ou pour tester sur la plate-forme de développement

Travaux pratiques : compiler du code personnel avec la toolchain

Lancez une compilation native du fichier d'exemple hello.c :

```
$ cd ~/ile/exemples/chapitre-03/
```

```
$ make
```

```
gcc -Wall -g hello.c -o hello
```

Testez le résultat :

```
$ ./hello
```

```
Hello World!
```

Effacez l'exécutable

```
$ make clean
```

```
rm -f hello
```

Lancez une cross-compilation du même fichier :

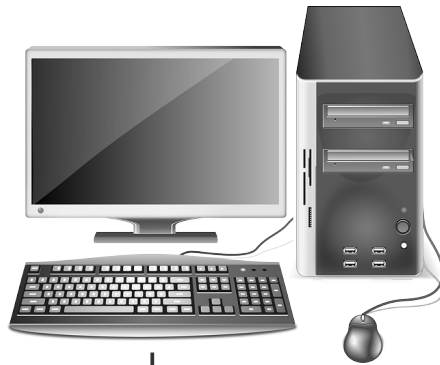
```
$ make CROSS_COMPILE=arm-linux-
```

Transférez le fichier sur la cible et vérifiez son fonctionnement.

```
$ scp hello root@192.168.1.200:/tmp/
```

Vous pouvez aussi copier le fichier dans l'overlay et relancer un build

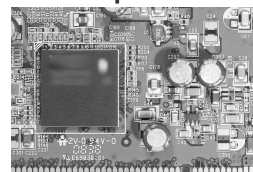
Débogage distant – principes



192.168.1.10

```
$ arm-linux-gdb ./hello
[...]
(gdb) target remote 192.168.1.200:1234
Remote debugging using 192.168.1.200:1234
0x40000a70 in ??()
(gdb) break main
Breakpoint 1 at 0x8081491 : file hello.c line 12
(gdb) continue
...
```

192.168.1.200



```
$ gdbserver 192.168.1.10:1234 ./hello
Process hello created ; pid=779
Remote debugging from host 192.168.1.10
...
```

Sources graphiques : images 158675 et 3262915 sur Pixabay.

`gdbserver` est compilé et installé par Buildroot.

Travaux pratiques : utilisation du débogueur GDB à distance

Sur la cible :

```
(cible)# gdbserver 192.168.1.10:4567 hello
Process hello created; pid = 67
Listening on port 4567
```

Sur la plate-forme de développement :

```
$ arm-linux-gdb hello
GNU gdb
[...]
(gdb) target remote 192.168.1.200:4567
Remote debugging using 192.168.1.200:4567
```

Observez sur la cible :

Remote debugging from host 192.168.1.10

```
[...]
(gdb) break main
Breakpoint 1 at 0x83a0: file hello.c, line 8.
(gdb) continue
Continuing.
Breakpoint 1, main () at hello.c:8
8   fprintf(stdout, "Hello World!\n");
(gdb) next
```

Observez sur la cible :

Hello World!

Outils d'aide au débogage

Ltrace & Strace

Ltrace enregistre la liste des fonctions de bibliothèque appelées par un programme :

```
$ ltrace -o trace.txt ./command
```

Strace enregistre la liste des appels-système invoqués par une application

```
$ strace -o trace.txt ./command
```

Ltrace et Strace se trouvent dans les packages proposés par Buildroot.

Valgrind

Valgrind fait tourner une application en vérifiant ses interactions avec le système.

Il est composé de différents modules, le plus utilisé est « memcheck » qui vérifie :

- la validité des pointeurs auxquels on accède ;
- la validité des pointeurs que l'on libère ;
- la libération de toute la mémoire allouée.

Il existe d'autres modules :

- massif pour connaître l'utilisation du tas,
- cachegrind pour aider à l'optimisation d'un programme,
- helgrind pour vérifier les accès concurrents dans les processus multithreads,
- ...

Travaux pratiques : utilisation de Valgrind

Cross-compilez les exemples pour Valgrind 1 et 2

```
$ cd ~/ile/exemples/chapitre-03
```

```
$ arm-linux-gcc -g exemple-valgrind-01.c -o exemple-valgrind-01
```

```
$ arm-linux-gcc -g exemple-valgrind-02.c -o exemple-valgrind-02
```

```
$ scp exemple-valgrind-01 root@192.168.1.200:/tmp/
```

```
$ scp exemple-valgrind-02 root@192.168.1.200:/tmp/
```

Sur la cible :

```
# valgrind --tool=memcheck exemple-valgrind-01
```

Utilisez les messages de diagnostic de Valgrind pour trouver les erreurs dans les codes des exemples