

# Développement du code métier

**Christophe BLAESS**

christophe.blaess@logilin.fr

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

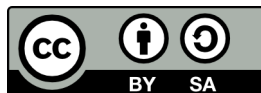
twitter: @chrisblaess



Ingénierie et formations sur Linux et les logiciels libres  
<https://www.logilin.fr>

<b>Intégration de scripts personnalisés.....</b>	<b>3</b>
Travaux pratiques.....	5
<b>Workflow du développement embarqué.....</b>	<b>6</b>
<b>Cross compilation du code métier.....</b>	<b>7</b>
La chaîne de compilation Gnu : GCC (Gnu Compiler Collection).....	7
Travaux pratiques : extraction de la <i>toolchain</i> .....	8
<b>Débogage distant.....</b>	<b>13</b>
Travaux pratiques : débogage avec Gdbserver.....	14
Outils d'aide au débogage.....	15
<b>Intégration du code métier.....</b>	<b>16</b>
Principes des recettes.....	16
Nom du fichier recette.....	17
Contenu d'une recette.....	18
Travaux pratiques : développer une recette pour son code métier.....	21
<b>Lancement d'une application au démarrage.....</b>	<b>24</b>
Scripts de démarrage.....	24
Travaux pratiques : intégration d'une application au démarrage.....	25

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
- *Excalidraw* (en ligne) pour certains schémas

ILY v. 8.7

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

<https://www.logilin.fr>

# Intégration de scripts personnalisés

Notre système de fichiers est en lecture-seule, néanmoins il peut être nécessaire de le remonter temporairement en lecture-écriture pour faire de brèves modifications.

Créons l'arborescence suivante : **meta-my-layer / recipes-custom / rw-ro /**

Puis la recette **rw-ro\_1.0.bb** :

```
SUMMARY = "Custom rw and ro scripts."
SECTION = "custom"

LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8
506ecda2f7b4f302"

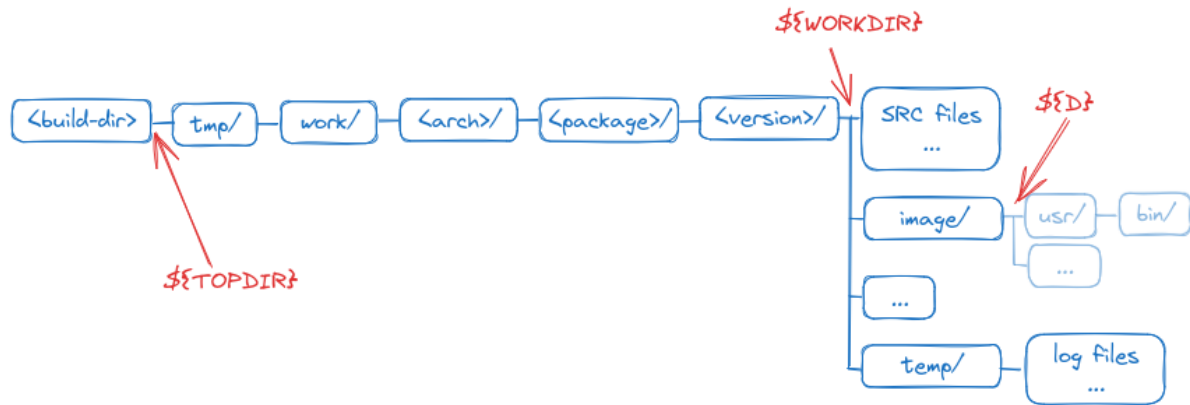
SRC_URI += "file://ro"
SRC_URI += "file://rw"

do_install() {
    install -d ${D}${sbindir}
    install -m 0755 ${WORKDIR}/ro ${D}${sbindir}
    install -m 0755 ${WORKDIR}/rw ${D}${sbindir}
}
```

Pour optimiser le traitement par Bitbake, on peut ajouter la ligne :

```
inherit allarch
```

si tous les fichiers installés sont indépendants de l'architecture (scripts, fichiers de données, etc.).



Dans un sous-répertoire **files/** les deux scripts suivants :

**rw :**

```
#!/bin/sh
mount / -o rw,remount
```

**ro :**

```
#!/bin/sh
mount / -o ro,remount
```

Ne pas oublier de rajouter la recette rw-ro dans IMAGE\_INSTALL.

## Travaux pratiques

Créez un script Python de type « *Hello World* ».

Créez une recette pour installer ce script sur la cible.

La recette devra s'assurer de l'installation du package « python3-core ».

### Solution

```
[build-qemu]$ mkdir ../../layers/meta-my-layer/recipes-custom/hello-python

[build-qemu]$ mkdir ../../layers/meta-my-layer/recipes-custom/hello-python/
hello-python

[build-qemu]$ nano ../../layers/meta-my-layer/recipes-custom/hello-python/
hello-python/hello-python
|#!/usr/bin/python3
|print("Hello from my Python script")

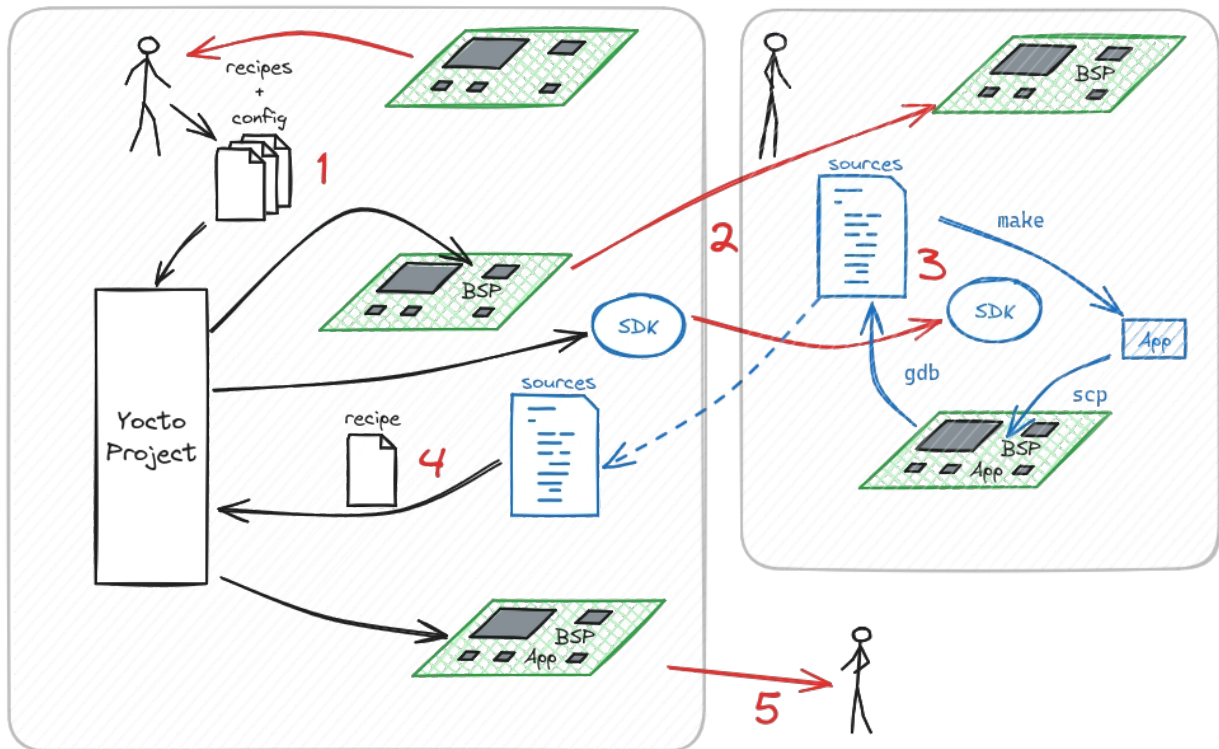
[build-qemu]$ nano ../../layers/meta-my-layer/recipes-custom/hello-python/
hello-python_1.0.bb
|SUMMARY = "Python script to display Hello World"
|LICENSE = "MIT"
|LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8
506ecda2f7b4f302"

|SRC_URI = "file://hello-python"

|RDEPENDS:${PN} = "python3-core"

|do_install() {
|    install -d ${D}${bindir}
|    install -m 0755 ${WORKDIR}/hello-python ${D}${bindir}
|}
```

# 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 son SDK aux développeurs applicatifs.
3. Les développeurs applicatifs mettent au point le code métier en utilisant le SDK.
4. Les développeurs BSP intègrent le code métier en écrivant les recettes nécessaires.
5. La carte comportant le BSP et le code métier est livrée aux clients.

# Cross compilation du code métier

## 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 (depuis GCC 4) ;
- gcj – Langage Java compilation en pseudo-code ou en code natif ;
- gnat – Langage Ada.

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...

Pour compiler un noyau Linux, il est parfois nécessaire de vérifier le numéro de version du compilateur gcc. Pour compiler des applications classiques, il y a une compatibilité ascendante suffisante.

Le projet Gnu propose aussi d'autres compilateurs :

gc1 (Common Lisp), gprolog (Prolog), gst (Small Talk), gforth (Forth),

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

Perl, Python, Tcl/Tk, Ruby, Scheme, Smart Eiffel, Pascal, Java, Objective Caml, REXX, Logo...

## Voir aussi

### Web :

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

## Travaux pratiques : extraction de la *toolchain*

*Préparation du SDK*

```
[build-qemu]$ bitbake -c populate_sdk my-image
```

*Scripts résultats*

```
[build-qemu]$ ls tmp/deploy/sdk
[...]
poky-glibc-x86_64-my-image-cortexa15t2hf-neon-qemuarm-toolchain-5.0.8.host.manifest
poky-glibc-x86_64-my-image-cortexa15t2hf-neon-qemuarm-toolchain-5.0.8.sh
poky-glibc-x86_64-my-image-cortexa15t2hf-neon-qemuarm-toolchain-5.0.8.target.manifest
poky-glibc-x86_64-my-image-cortexa15t2hf-neon-qemuarm-toolchain-5.0.8.testdata.json
```

*La toolchain est prévue pour s'exécuter sur une architecture compatible avec celle mentionnée dans la variable **SDKMACHINE**.*

```
[build-qemu]$ bitbake -e my-image | grep ^SDKMACHINE
SDKMACHINE="x86_64"
```

*Le code produit par cette toolchain est configuré pour fonctionner sur l'architecture **MACHINE\_ARCH** :*

```
[build-qemu]$ bitbake -e my-image | grep ^MACHINE_ARCH
MACHINE_ARCH="qemuarm"
```



## **Installation de la toolchain**

*On peut transférer l'ensemble du répertoire sdk/ vers un autre poste sur lequel on fait le développement du code métier.*

```
[build-qemu]$ cd tmp/deploy/sdk/
[ sdk]$ ./poky-glibc-x86_64-my-image-cortexa15t2hf-neon-qemuarm-toolchain-
5.0.8.sh
Poky (Yocto Project Reference Distro) SDK installer version 5.0.8
=====
Enter target directory for SDK (default: /opt/poky/5.0.8): ~/cross/
You are about to install the SDK to "/home/USER/cross". Proceed[Y/n]? Y
Extracting SDK.....done
Setting it up...done
SDK has been successfully set up and is ready to be used.
[...]

[ sdk]$ ls ~/cross/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux/
arm-poky-linux-addr2line  arm-poky-linux-gcc-ar      arm-poky-linux-ld.gold
arm-poky-linux-ar         arm-poky-linux-gcc-nm      arm-poky-linux-nm
arm-poky-linux-as         arm-poky-linux-gcc-ranlib  arm-poky-linux-objcopy
arm-poky-linux-c++filt    arm-poky-linux-gcov        arm-poky-linux-objdump
arm-poky-linux-cpp        arm-poky-linux-gcov-tool   arm-poky-linux-ranlib
arm-poky-linux-dwp        arm-poky-linux-gdb         arm-poky-linux-readelf
arm-poky-linux-elfedit    arm-poky-linux-gprof       arm-poky-linux-size
arm-poky-linux-g++        arm-poky-linux-ld          arm-poky-linux-strings
arm-poky-linux-gcc        arm-poky-linux-ld.bfd      arm-poky-linux-strip
```

### **Utilisation de la toolchain de Yocto**

Plutôt que d'appeler directement le *cross-compiler*, il est préférable de *sourcer* le script `environment-setup-*` situé dans le répertoire d'installation, et d'utiliser les macros :

Macro	Défaut	Signification
CC	gcc	Compilateur C
CXX	g++	Compilateur C++
CPP	gcc	Préprocesseur C/C++
AS	as	Assembleur
LD	ld	Éditeur de liens
AR	ar	Archiveur

On peut utiliser (et compléter) les macros `CFLAGS`, `CXXFLAGS`, `CPPFLAGS`, `LDFLAGS` pour configurer l'action des outils ci-dessus.

Les variables `ARCH` et `CROSS_COMPILE` sont renseignées ; elles servent lors de la compilation du kernel ou de modules externes.

On se reportera directement au script pour voir l'ensemble des variables définies.

*Exemple de **Makefile** :*

```
CFLAGS += -Wall -g
CC ?= gcc

all: hello

hello: hello.c
    $(CC) $(CFLAGS) $(LDFLAGS) hello.c -o hello

clean:
    rm -f hello
```

*Avec un fichier **hello.c** :*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    char buffer[128];

    if (gethostname(buffer, 128) < 0)
        return EXIT_FAILURE;
    printf("Hello from %s\n", buffer);
    return EXIT_SUCCESS;
}
```

*Testez une compilation native :*

```
[custom-code]$ make  
cc -Wall -g hello.c -o hello  
[custom-code]$ ./hello  
Hello from TR-B-01  
[custom-code]$ make clean  
rm -f hello
```

*Lancez une cross-compilation :*

```
[custom-code]$ source ~/cross/environment-setup-cortexa15t2hf-neon-qemuarm-  
poky-linux-gnueabi  
[custom-code]$ make  
cortexa15t2hf-neon-qemuarm-poky-linux-gcc --sysroot=/home/USER/yocto-  
training/cross/cortexa15t2hf-neon-qemuarm-poky-linux -O2 -pipe -Wall -g  
hello.c -o hello  
[custom-code]$ ./hello  
-bash: ./hello : impossible d'exécuter le fichier binaire : Erreur de  
format pour exec()  
[custom-code]$ scp hello root@192.168.7.2:~/  
root@192.168.7.2's password:
```

*Sur la cible lancez l'exécutable :*

```
root@qemuarm:~# ./hello  
Hello from qemuarm
```

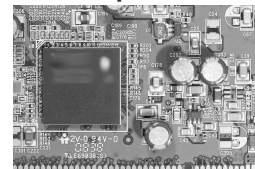
# Débogage distant



192.168.7.1

```
$ arm-poky-linux-gnueabi-gdb ./hello
[...]
(gdb) target remote 192.168.7.2:1234
Remote debugging using 192.168.7.2:1234
0x40000a70 in ??()
(gdb) break main
Breakpoint 1 at 0x8081491 : file hello.c line 12
(gdb) continue
...
```

192.168.7.2



```
$ gdbserver 192.168.7.1:1234 ./hello
Process hello created ; pid=779
Remote debugging from host 192.168.7.1
...
```

Sources graphiques : images 158675 et 3262915 sur Pixabay.

## Travaux pratiques : débogage avec Gdbserver

*Lancez gdbserver sur la cible :*

```
root@qemuarm:~# gdbserver 192.168.7.1:4567 ./hello  
Process /tmp/hello created; pid = 67  
Listening on port 4567
```

*Sur la plate-forme de développement :*

```
[custom-code]$ ${GDB} hello  
GNU gdb  
[...]  
(gdb) target remote 192.168.7.2:4567  
Remote debugging using 192.168.7.2:4567
```

*Observez sur la cible :*

Remote debugging from host 192.168.7.1

```
[...]  
(gdb) break main  
Breakpoint 1 at 0x4004c4: file hello.c, line 9.  
(gdb) cont  
Continuing.  
Breakpoint 1, main () at hello.c:9  
9      if (gethostname(buffer, 128) < 0)  
(gdb) next
```

*Observez la progression sur la cible...*

<code>break [function, line, address]</code>	Pose d'un point d'arrêt
<code>info breakpoint</code>	Affichage des points d'arrêts
<code>delete &lt;number&gt;</code>	Effacement du point d'arrêt indiqué
<code>next</code>	Avancement d'une ligne sans entrer dans les fonction
<code>step</code>	Avancement d'une instruction en entrant dans les fonctions
<code>finish</code>	Poursuite jusqu'à la fin de la fonction courante
<code>continue</code>	Reprise de l'exécution normale
<code>print &lt;variable&gt;</code>	Affichage du contenu d'une variable
<code>list</code>	Listing du code
<code>x &lt;address&gt;</code>	Affichage du contenu de la mémoire
<code>disassemble [function, address]</code>	Désassemblage du code en mémoire
<code>backtrace</code>	Affichage du contenu de la pile
<code>thread &lt;number&gt;</code>	Basculement sur le thread indiqué
<code>set var &lt;variable&gt; = &lt;value&gt;</code>	Affectation d'une variable

## Outils d'aide au débogage

### *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 quantité de mémoire allouée et libérée.

Il existe d'autres modules :

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

*Sur le PC :*

```
[custom-code]$ ${CC} -g example.c -o example  
[custom-code]$ scp example root@192.168.7.2:~/
```

*Sur la cible :*

```
root@qemuarm:~# valgrind --tool=memcheck ./example
```

# Intégration du code métier

## Principes des recettes

Une recette est un fichier de texte décrivant comment produire un package en partant de ses sources.

La syntaxe des recettes est un mélange de scripts Shell et Python.

Les recettes sont analysées et traitées par Bitbake, il s'agit de fichiers avec l'extension `.bb` ou `.bbappend`.

Un fichier `.bbappend` permet de surcharger le contenu d'une recette standard. Il est également possible d'inclure des fichiers externes.

Il peut y avoir des dépendances entre recettes (présence d'une bibliothèque par exemple).



## Nom du fichier recette

Le fichier est nommée à partir du nom du package et de son numéro de version séparés par un caractère *underscore* : *nom-package\_numéro-version.bb*.

Par exemple : poky/meta/recipes-core/busybox/busybox\_1.36.1.bb.

Lors de l'interprétation d'une recette les variables suivantes sont automatiquement définies :

- BPN (*Base Package Name*) composé de la première partie du nom du fichier (ex. dhcp)
- PV (*Package Version*) extrait du nom de fichier (ex. 1.36.1)
- BP équivaut à \${PN}-\${PV}
- PR (*Package Release*) par défaut r0.

Plusieurs recettes aux fonctionnalités connexes peuvent inclure un fichier commun. Par exemple `busybox\_1.36.1.bb` commence par la ligne :

```
|| require busybox.inc
```

Le fichier `busybox.inc` est dans le répertoire `poky/meta/recipes-core/busybox/`

Les directives **require** et **include** recherchent le fichier .inc dans les répertoires contenus dans la variable **BBPATH**. La directive **include** n'échoue pas si le fichier n'est pas trouvé.

## Contenu d'une recette

### Informations sur l'application

La variable **SECTION** indique la catégorie d'application.

```
SECTION = "base"
```

Les variables **SUMMARY** et **DESCRIPTION** contiennent des descriptions

```
SUMMARY = "Tiny versions of many common UNIX utilities in a single[...]"
DESCRIPTION = "\"BusyBox combines tiny versions of many common UNIX
[...]\nfor any small or embedded system."
```

La variable **HOMEPAGE** renvoie vers la page principale du projet.

```
HOMEPAGE = "https://www.busybox.net"
```

La variable **LICENSE** indique la licence du logiciel et **LIC\_FILES\_CHKSUM** indique le(s) fichier(s) consulté(s) pour déterminer le(s) type(s) de licence.

```
LICENSE = "GPL-2.0-only & bzip2-1.0.4"
LIC_FILES_CHKSUM = "file://LICENSE;md5=de10de48642ab74318e893a61105afbb \
file://archival/libarchive/bz/
LICENSE;md5=28e3301eae987e8cfe19988e98383dae"
```

Les variables **DEPENDS** et **RDEPENDS** indiquent les noms des packages indispensables pour la compilation et le fonctionnement de l'application.

```
DEPENDS += "kern-tools-native virtual/crypt"
```

RDEPENDS signifie « *Runtime Dependencies* ».

## Accès aux sources

La variable **SRC\_URI** contient la liste des fichiers sources pour le package.

```
SRC_URI = "https://busybox.net/downloads/busybox-${PV}.tar.bz2;name=tarball \  
file://0001-depmod-Ignore-.debug-directories.patch \  
file://busybox-udhcp-no_deconfig.patch \  
"
```

Les URI (*Uniform Resource Identifier*) ont le format suivant :

method://url;parameters

Méthodes :

- locale : file://name : fichiers recherchés dans les répertoires de FILESPATH
- distantes : http:// https:// ftp:// git:// ...

Pour la méthode git:// on ajoute souvent un paramètre branch=name et une variable **SRCREV** contenant l'identifiant de commit.

Pour les méthodes distantes sans contrôle d'intégrité (http, https, ftp, etc.) on ajoute une somme de contrôle dans la variable **SRC\_URI[md5sum]** ou **SRC\_URI[sha256sum]**.

Les archives tar sont décompressées et doivent donner un répertoire correspondant au motif \${PN}-\${PV}. Sinon il faut remplir explicitement la **variable S** avec le nom du répertoire.

## *Tâches de construction*

Les tâches successives de Bitbake pour une recette sont décrites par la commande `listtasks` :

```
$ bitbake -c listtasks busybox
```

Les tâches les plus courantes sont :

`do_fetch`, `do_checkuri`, `do_unpack`, `do_patch`, `do_configure`, `do_compile`, `do_install`,  
`do_package`, `do_populate_sysroot`

Elles sont implémentées sous forme de fonctions shell avec un éventuel suffixe `:append` ou `:prepend` :

```
do_install() {  
    [...]   
    install -d ${D}${sysconfdir}/init.d  
    [...]   
    install -d ${D}${base_bindir}  
    [...]   
    install -m 4755 ${B}/busybox.suid ${D}${base_bindir}  
    [...]   
}
```

La **variable D** représente la racine du système de fichiers temporaire avant création de l'image. La variable **WORKDIR** contient le répertoire de travail pour cette recette

## Travaux pratiques : développer une recette pour son code métier

*Créez d'abord une petite application et son Makefile dans un répertoire dédié :*

```
[training]$ mkdir my-app
```

**my-app/my-app.c :**

```
#include <stdio.h>

int main(void)
{
    printf("Hello from my recipe!\n");
    return 0;
}
```

**my-app/Makefile :**

```
DESTDIR ?= /usr/local/bin

all: my-app

my-app: my-app.c
    $(CC) $(CFLAGS) $(LDFLAGS) -o my-app my-app.c

install: my-app
    cp my-app $(DESTDIR)/

clean:
    rm -f *.o my-app
```

*Créez une archive de votre application :*

```
[training]$ tar cjf my-app-1.0.tar.bz2 my-app/
```

*Créez une arborescence pour votre recette :*

```
[training]$ mkdir -p layers/meta-my-layer/recipes-custom/my-app/
```

*Créez un répertoire pour votre package :*

```
[training]$ mkdir layers/meta-my-layer/recipes-custom/my-app/files/
```

*Copiez l'archive dans le répertoire :*

```
[training]$ cp my-app-1.0.tar.bz2 layers/meta-my-layer/recipes-custom/my-app/files/
```

*Créez une recette dans votre layer personnel :*

**layers/meta-my-layer/recipes-custom/my-app/my-app\_1.0.bb**

```
SUMMARY = "My Application"
LICENSE = "CLOSED"
LIC_FILES_CHKSUM = ""
SRC_URI += "file://my-app-${PV}.tar.bz2"

S = "${WORKDIR}/${BPN}"

do_compile() {
    oe_runmake
}

do_install() {
    install -d ${D}${bindir}
    oe_runmake DESTDIR=${D}${bindir} install
}
```

*Ajoutez votre recette dans l'image :*

**IMAGE\_INSTALL:append = " my-app"**

*Puis demandez à **bitbake** de cuisiner votre recette ! 😊*

# Lancement d'une application au démarrage

## Scripts de démarrage

Le processus `/sbin/init` lit le fichier `/etc/inittab` et agit en conséquence. Ce fichier contient une action à réaliser par ligne.

*Chaque ligne contient plusieurs champs :*

`<Terminal>:<Niveau d'exécution>:<Type d'action>:<Commande>`

*Le type d'action est généralement `sysinit` ou `respawn`. La ligne `id:5:initdefault:` indique que le niveau d'exécution au boot est 5.*

*Observez les scripts exécutés au démarrage*

```
root@beaglebone:/# ls /etc/rcS.d/
S00psplash.sh          S06devpts.sh
S01keymap.sh           S07bootlogd
S02banner.sh           S29read-only-rootfs-hook.sh
S02sysfs.sh            S36udev-cache
S03mountall.sh          S37populate-volatile.sh
S04udev                S38dmesg.sh
S05modutils.sh         S38urandom
S06alignment.sh        S39hostname.sh
S06checkroot.sh        S55bootmisc.sh
```

```
root@beaglebone:/# ls /etc/rc5.d/
S01networking    S15mountnfs.sh    S99rmnologin.sh
S10dropbear      S20syslog         S99stop-bootlogd
```

Le script `rc` lance successivement tous les scripts se trouvant dans `/etc/rc<niveau>.d/` dont le nom commence par un `s` suivi d'au-moins deux caractères en leur fournissant l'argument `start`.



## Travaux pratiques : intégration d'une application au démarrage

*Éditez la recette `my-app_1.0.bb` et ajoutez les lignes suivantes :*

```
SRC_URI += "file://run-my-app"

inherit update-rc.d
INITSCRIPT_NAME = "run-my-app"
INITSCRIPT_PARAMS = "start 80 5 ."

do_install() {
    [...]
    install -d ${D}${sysconfdir}/init.d
    install -m 0755 ${WORKDIR}/run-my-app ${D}${sysconfdir}/init.d/
}
```

*Et ajoutez le script suivant dans le répertoire `my-app/files/` :*

```
run-my-app:
#!/bin/sh

/usr/bin/my-app &

exit 0
```