

Aspects avancés des recettes

Christophe BLAESS

christophe.blaess@logilin.fr

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



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

Classes et héritages.....	3
Travaux pratiques : création d'une classe personnalisée.....	5
Principe des distros.....	7
Choix de la libC.....	10
Choix du système d'initialisation.....	10
Travaux pratiques : création d'une distro personnalisée.....	11
Options et conditions.....	12
PACKAGECONFIG.....	12
Travaux pratiques : utilisation de PACKAGECONFIG.....	13
Conditions.....	14
OVERRIDES.....	16
Exécution au démarrage.....	18
Lancement d'un service avec SysVinit.....	18
Lancement d'un service avec Systemd.....	19
Travaux pratiques : Lancement d'une application au démarrage.....	20

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
- *Gimp* pour les images bitmap
- *Excalidraw* (en ligne) pour les schémas vectoriels.

Yocto Project avancé

YPA v. 1.0

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

<https://www.logilin.fr>

Classes et héritages

Une **classe** est implémentée par un fichier **.bbclass** dans le sous-répertoire ``classes/`` d'un layer.

La classe contient des affectations de variables (généralement des affectations par défaut avec l'opérateur ``?='``) et des définitions de fonctions.

Une recette peut **hériter** d'une classe en utilisant la syntaxe

```
|| inherit class-name
```

Les classes peuvent ainsi définir des données ou des fonctionnalités communes à plusieurs recettes.

En utilisant ``${PN}`` et ``${PV}`` une classe peut faire référence au nom et au numéro de version de la recette qui en hérite.

Quelques classes intéressantes de ``poky/meta/classes/`` :

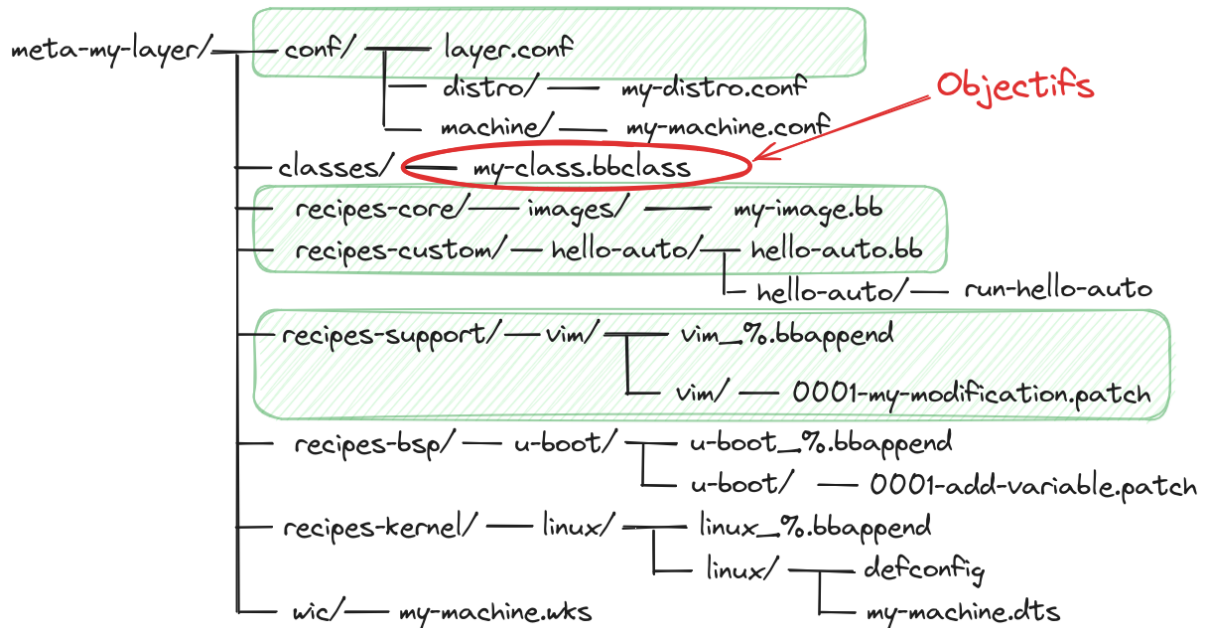
- ``base.bbclass`` : une classe automatiquement héritée par toutes les recettes ;
- ``core-image.bbclass`` : la classe qui implémente les fonctionnalités habituelles de ``IMAGE_FEATURES`` ;
- ``kernel.bbclass`` : la classe qui gère le noyau Linux ;
- ``autotools.bbclass``, ``cmake.bbclass`` : les classes contenant les méthodes pour gérer les packages utilisant les Autotools ou Cmake, que l'on pouvait voir dans les recettes créées précédemment par Devtool.

Des recettes d'applications métiers peuvent ainsi hériter d'une classe personnalisée pour partager un numéro de version global du projet, une licence spécifique, une adresse de téléchargement des codes sources applicatifs, une méthodes d'envoi des packages compilés vers un système d'intégration continue, etc.

Travaux pratiques : création d'une classe personnalisée

- Définissez une classe `my-class` qui crée dans le répertoire `/etc` de la cible un fichier du nom de la recette héritant de la classe.

Dans ce fichier on trouvera le numéro de version de la recette.



- ➔ Créez une recette ``my-version`` avec un numéro (ex: ``1.0``) qui hérite simplement de cette classe. N'oubliez pas d'ajouter la recette dans l'image.
- ➔ Modifiez votre recette ``hello-auto`` pour qu'elle hérite aussi de la classe précédente.
- ➔ Créez une classe ``gplv2.bbclass`` (par exemple) qui définit les champs ``LICENSE`` et ``LIC_FILES_CHKSUM``.
- ➔ Faites hériter vos recettes de cette classe.

Solution :

```
$ mkdir ../../layers/meta-my-layer/classes

$ nano ../../layers/meta-my-layer/classes/my-class.bbclass
|LICENSE = "MIT"
|
|do_install() {
|    install -d ${D}${sysconfdir}
|    echo "my version: ${PV}" > ${D}${sysconfdir}/${PN}
|}
|
|FILES:${PN} = "${sysconfdir}/${PN}"

$ mkdir ../../layers/meta-my-layer/recipes-custom/my-version

$ nano ../../layers/meta-my-layer/recipes-custom/my-version/my-
version_1.0.bb
|inherit my-class

$ nano ../../layers/meta-my-layer/classes/gplv2.bbclass
|LICENSE = "GPL-2.0-only"
|
|LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-2.0-
|only;md5=801f80980d171dd6425610833a22dbe6"
```

Principe des distros

Le choix de la **distribution** se fait dans la variable `DISTRO`, généralement renseignée dans `conf/local.conf`.

Par défaut on trouve :

```
DISTRO ?= "poky"
```

La définition du contenu de cette distribution est contenue dans un fichier `poky/meta-poky/conf/distro/poky.conf`.

Elle contient surtout des préférences de comportement et des valeurs par défaut :

```
DISTRO = "poky"
DISTRO_NAME = "Poky (Yocto Project Reference Distro)"
[...]
POKY_DEFAULT_DISTRO_FEATURES = "largefile opengl ptest multiarch wayland
vulkan"
POKY_DEFAULT_EXTRA_RDEPENDS = "packagegroup-core-boot"
[...]
DISTRO_FEATURES ?= "${DISTRO_FEATURES_DEFAULT} ${
POKY_DEFAULT_DISTRO_FEATURES}"
[...]
PREFERRED_VERSION_linux-yocto ?= "5.15%"
PREFERRED_VERSION_linux-yocto-rt ?= "5.15%"
[...]
```

La distro vient en complément de la recette d'image :

- l'image sélectionne les packages (`IMAGE_INSTALL``) et les fonctionnalités (`IMAGE_FEATURES``)
- la distro définit les versions des packages ou l'implémentation des fonctionnalités.

NB : On peut souvent enregistrer dans une distro d'un layer personnalisé ce qu'on aurait tendance spontanément à stocker dans le fichier ``local.conf``.

La variable ``DISTRO_FEATURES`` décrit des fonctionnalités (``x11``, ``wayland``, ``systemd``, ``ipv6``, ``bluetooth``...) qui peuvent être testées lors de la configuration des recettes.

```
poky/meta/recipes-core/busybox/busybox.inc:
[...]
```

```
def features_to_busybox_settings(d):
    [...]

    busybox_cfg(bb.utils.contains('DISTRO_FEATURES', 'ipv6', True, False,
d), 'CONFIG_FEATURE_IPV6', cnf, rem)

    busybox_cfg(bb.utils.contains('DISTRO_FEATURES', 'ipv6', True, False,
d), 'CONFIG_FEATURE_IFUPDOWN_IPV6', cnf, rem)
[...]
```

Autre exemple, l'application ``ofono`` (outils pour la communication GSM/UMTS) est activée si la fonctionnalité ``3g`` est inscrite dans ``DISTRO_FEATURES``.

```
poky/meta/recipes-core/packagegroups/packagegroup-base.bb:

[...]
```

```
PACKAGES = ' \\\n[...]\n${@bb.utils.contains("DISTRO_FEATURES", "3g", "packagegroup-base-3g",\n"" , d)} \\\n[...]
```

```
SUMMARY:packagegroup-base-3g = "Cellular data support"\nRDEPENDS:packagegroup-base-3g = "\\ofono"
```


Une recette peut **réclamer** que certaines fonctionnalités de distro soit définies ainsi :

```
|| inherit features_check  
|| REQUIRED_DISTRO_FEATURES = "x11"
```

En remplissant plutôt la variable `ANY_OF_DISTRO_FEATURES` la recette réclame qu'au moins une fonctionnalité de la liste soit définie.

Une recette peut **refuser** qu'une fonctionnalité de distro soit définie ainsi :

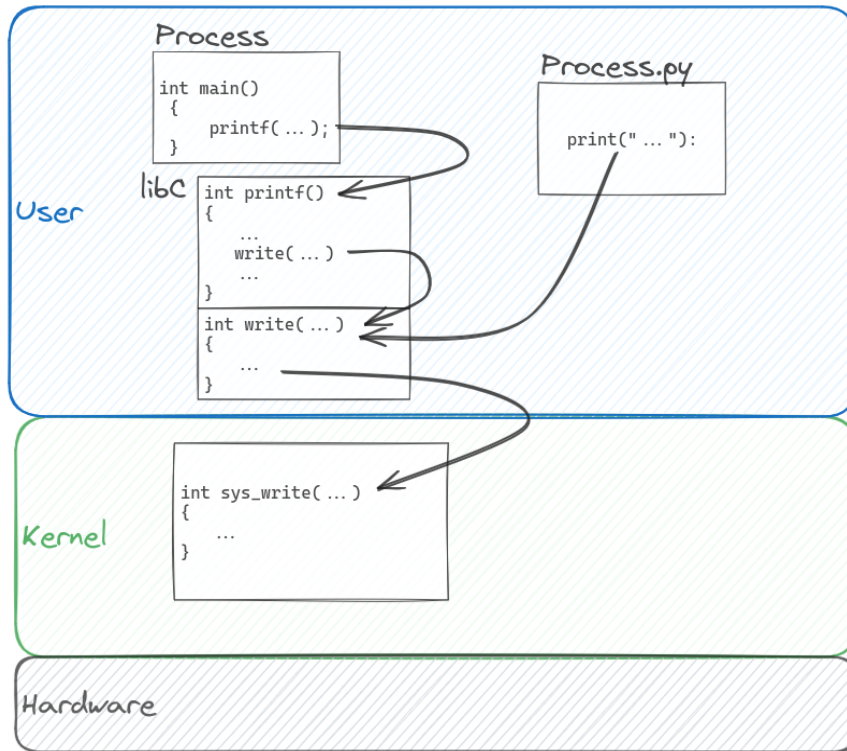
```
|| inherit features_check  
|| CONFLICT_DISTRO_FEATURES = "systemd"
```

Travaux pratiques : options de distro

- ➔ Vérifiez la présence des utilitaires ``rfkill`` et ``bluetoothctl`` sur la cible.
 - ➔ Supprimez (dans ``local.conf``) les options ``wifi`` et ``bluetooth`` de la variable ``DISTRO_FEATURES``
 - ➔ Régénérez l'image et vérifiez si les utilitaires mentionnés plus haut sont toujours présents
-
- ➔ Vérifiez la présence de ``ping6`` sur la cible.
 - ➔ Supprimez à présent l'option ``ipv6`` de ``DISTRO_FEATURES``
 - ➔ Vérifiez après re-compilation la présence de l'utilitaire ``ping6`` sur la cible.

Choix de la libC

Une variable configurable par la distro est `TCLIBC` (au choix parmi `glibc`, `musl`, `newlib`, `baremetal`) qui choisit l'implémentation de la bibliothèque C utilisée

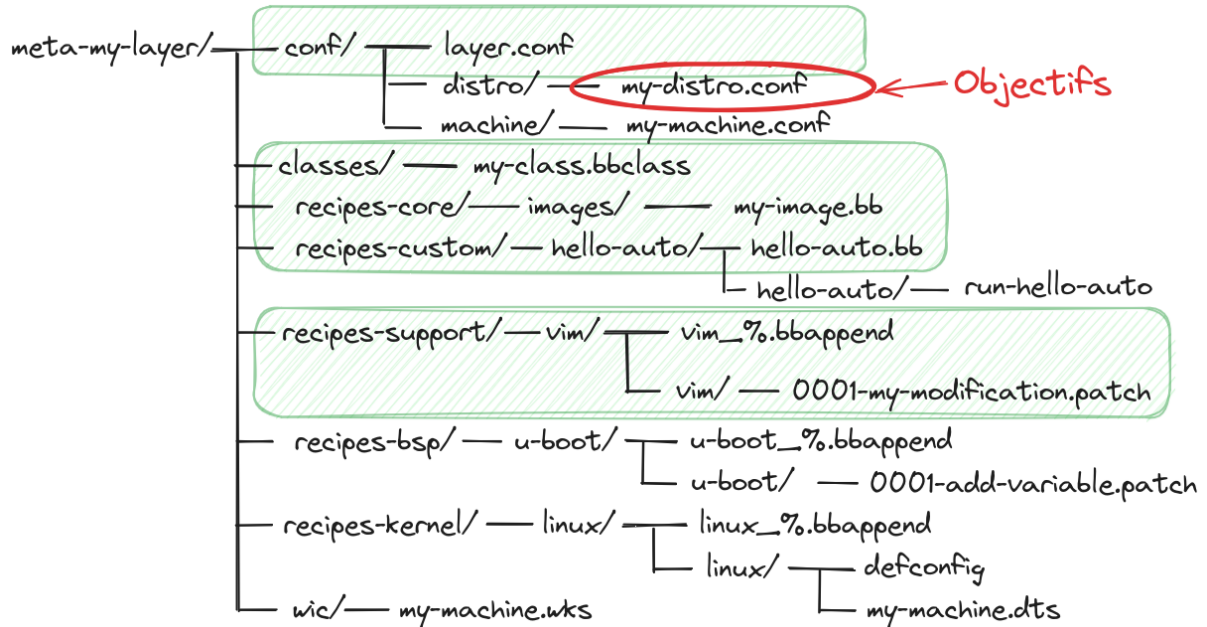


Outre sa fonction de bibliothèque de fonctions pour les applications écrites en langage C, la libC assure l'interface entre l'espace utilisateur et le *kernel*.

Travaux pratiques : création d'une distro personnalisée

- Créez un fichier de configuration `my-distro.conf` qui commence par récupérer le contenu de `poky.conf` (directive `require`) puis surcharge quelques variables comme :

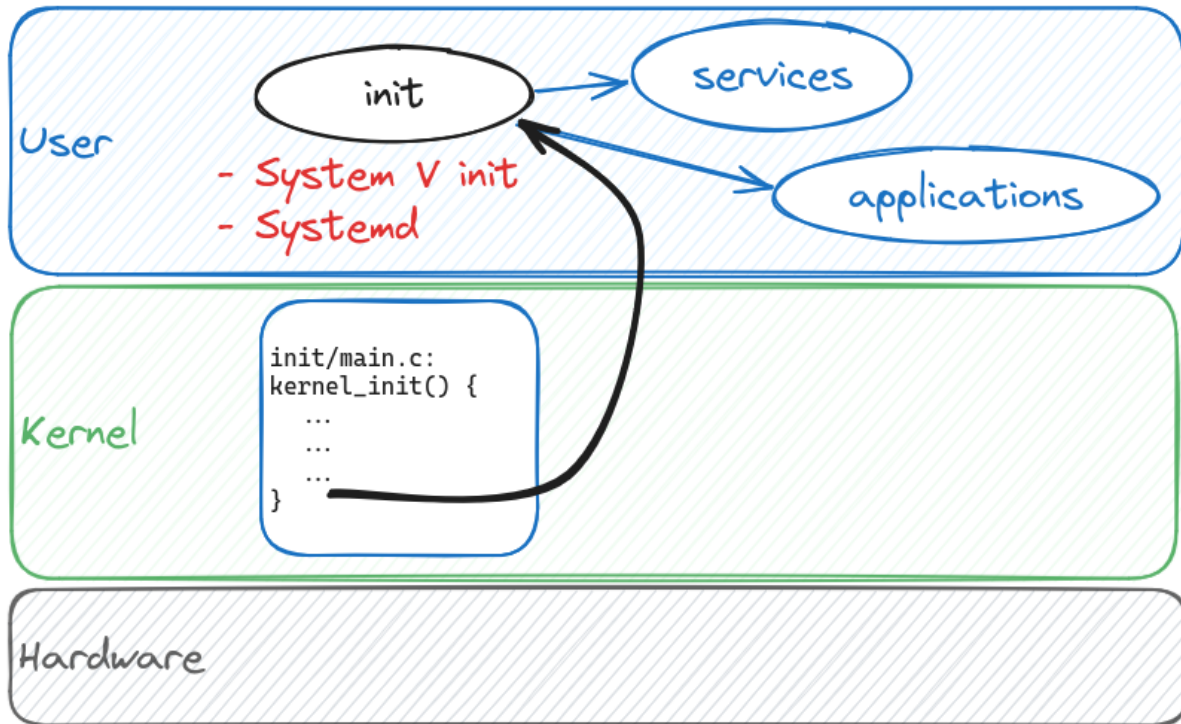
```
DISTRO = "my-distro"
DISTRO_NAME = "My Own Distro"
DISTRO_VERSION = "1.0.1"
```



- Sélectionnez votre distro dans `local.conf`.

Choix du système d'initialisation

Le type de système d'initialisation influe sur le démarrage des services une fois le boot du noyau achevé.



Par défaut, Yocto choisit une initialisation à la manière « System V »

On peut choisir d'utiliser `systemd` en remplacement de `sysVinit` avec :

```
|| Distro_FEATURES += "systemd"  
|| VIRTUAL-RUNTIME_init_manager = "systemd"
```

Travaux pratiques : installation de systemd

- ➔ Dans votre fichier distro, ajoutez les lignes pour utiliser Systemd comme environnement d'initialisation.

Options et conditions

PACKAGECONFIG

Si une recette hérite de la classe ``pkgconfig``, sa variable ``PACKAGECONFIG`` permet de préciser des options de compilation ou d'installation propres à cette recette.

On déclare une option ainsi :

```
PACKAGECONFIG[my-option] = "str1,str2,str3,str4,str5,str6"
```

Les composants de la chaîne sont :

- ``str1`` : chaîne ajoutée dans ``EXTRA_OECONF`` si ``my-option`` est présente dans ``PACKAGECONFIG``. Cette variable est utilisée lors de la configuration du package.
- ``str2`` : chaîne ajoutée dans ``EXTRA_OECONF`` si ``my-option`` est absente de ``PACKAGECONFIG``.
- ``str3`` : ajoutée dans ``DEPENDS`` si ``my-option`` est présente dans ``PACKAGECONFIG``.
- ``str4`` : ajoutée dans ``RDEPENDS`` si ``my-option`` est présente dans ``PACKAGECONFIG``.
- ``str5`` : ajoutée dans ``RRECOMMENDS`` si ``my-option`` est présente dans ``PACKAGECONFIG``.
- ``str6`` : liste des options de ``PACKAGECONFIG`` incompatibles avec ``my-option``.

Le contenu par défaut de ``PACKAGECONFIG`` est défini dans la recette d'image et peut être surchargé dans un fichier ``.bbappend``.

Par exemple dans la recette ``nano_%.bb`` on trouve la ligne suivante :

```
PACKAGECONFIG[tiny] = "--enable-tiny,"
```

Lorsque ``nano`` est compilée avec cette option, le message « *Welcome to nano* » n'est pas affiché au démarrage de l'éditeur.

Travaux pratiques : utilisation de PACKAGECONFIG

- ➔ Créez une extension de recette pour ``nano`` qui ajoute ``tiny`` dans ``PACKAGECONFIG``.
- ➔ Vérifier après le *build* si le message « *Welcome to nano* » a bien disparu.

Conditions

Plusieurs syntaxes permettent d'implémenter des traitements conditionnels en fonction du contenu de certaines variables.

Des opérateurs de test sont implémentés dans les méthodes ``${@bb.utils...()}``. Ces fonctions sont écrites en Python dans `poky/bitbake/lib/bb/utils.py`.

Renvoyer une valeur si une variable contient une certaine chaîne, et une autre valeur sinon :

```
`${@bb.utils.contains('VARIABLE', 'string', '<content if present>',  
'<content if absent>', d)}
```

``d`` est le “*data store*”, une représentation de l'environnement de Bitbake.

Exemple :

```
FILESEXTRAPATHS:prepend := "${@bb.utils.contains('DISTRO_FEATURES',  
'debugmode', '${THISDIR}/${PN}:', '', d)}
```

Renvoyer une valeur si une variable contient une chaîne appartenant à une liste (et une autre valeur sinon) :

```
|| ${@bb.utils.contains_any('VARIABLE', 'string1 string2 string3', '<content  
if present>', '<content if absent>', d)}
```

Exemple :

```
|| SRC_URI += "${@bb.utils.contains_any('DISTRO_FEATURES', 'x11 wayland',  
'file://graphic-app', '', d)}"
```

Renvoyer les éléments d'une variable contenus dans une liste :

```
|| ${@bb.utils.filter('VARIABLE', 'string1 string2 string3...', d)}
```

Exemple :

```
|| if [ "${@bb.utils.filter('DISTRO_FEATURES', 'x11', d)}" ]; then  
||   ...  
|| fi
```

OVERRIDES

Dans l'environnement de Yocto, les *overrides* sont des extensions qui permettent de préciser le champ d'application d'une variable.

La variable ``OVERRIDES`` contient une liste d'extensions séparées par des deux-points ``:``.

Cette variable est construite à partir de ``DistroOverrides``, de ``ClassOverrides`` et de ``MachineOverrides``.

Par exemple :

```
|| DistroOverrides:append = ":preempt-rt:upstream"
```

On peut ensuite rendre certaines actions et lignes conditionnelles :

```
|| PreferredProviderVirtual/kernel = "linux-5.15"
|| PreferredProviderVirtual/kernel:preempt-rt = "linux-5.15-rt"
```

```
|| do_install:append:upstream {
||     ...
|| }
```

On peut imaginer le scénario suivant :

- Les variables suivies d'une extension comme ``MY_VARIABLE:my_ext`` sont mises en mémoire comme toutes les autres variables
- Une fois le *parsing* terminé, les extensions présentes dans ``OVERRIDES`` sont supprimées, par exemple ``my_ext``
- La variable ``MY_VARIABLE:my_ext`` devient alors ``MY_VARIABLE`` remplaçant la précédente variable le cas échéant.

On peut voir le contenu de la variable ``OVERRIDES`` à la fin du *parsing* en utilisant la commande :

```
$ bitbake -e <image-name>
```

recipe.bb

```
VARIABLE = "value"  
VARIABLE:ext1 = "value 1"  
VARIABLE:ext2 = "value2"
```

parsing...

```
OVERRIDES = "linux-gnueabi:arm:...:raspberrypi4:..."
```

```
VARIABLE = "value"
```

recipe.bb

```
VARIABLE = "value"  
VARIABLE:ext1 = "value 1"  
VARIABLE:ext2 = "value2"
```

local.conf

```
...  
OVERRIDES:append = ":ext1"  
...
```

parsing...

```
OVERRIDES = "linux-gnueabi:arm:...:raspberrypi4:...:ext1"
```

```
VARIABLE: ext1
```

VARIABLE

```
VARIABLE = "value 1"
```

L'ordre des extensions dans la liste `OVERRIDES` est donc important.

Les extensions finales peuvent surcharger les variables déjà modifiées par les extensions précédentes.

Exécution au démarrage

Suivant le type de système d'initialisation (*Systemd* et *SysVinit* principalement), le lancement automatique d'une tâche se fera différemment.

Lancement d'un service avec SysVinit

La recette concernée doit hériter de la classe `update-rc.d` et installer un script de démarrage :

```
[...]
SRC_URI += "file://run-${BPN}"

inherit update-rc.d
INITSCRIPT_NAME = "run-${BPN}"
INITSCRIPT_PARAMS = "start 99 5 . stop 1 0 ."
# 99 = start order, 5 = start runlevel (default)
# 1 = stop order, 0 = stop runlevel (halt).

do_install:append() {
    install -d ${D}${sysconfdir}/init.d
    install -m 0755 ${WORKDIR}/run-${BPN} ${D}${sysconfdir}/init.d/
}
```

Le script `run-<nom-du-package>` :

```
#!/bin/sh
/usr/bin/<application-to-run> &
exit 0
```

Lancement d'un service avec Systemd

Suivant un principe similaire, la recette hérite de la classe `systemd` et installe un fichier de service :

```
[[...]]
SRC_URI += "file://${BPN}.service"

inherit systemd
SYSTEMD_SERVICE:${PN} += "${BPN}.service"

do_install:append() {
    [[...]]
    install -d ${D}/${systemd_system_unitdir}
    install -m 0644 ${WORKDIR}/${BPN}.service ${D}/${systemd_system_unit
dir}/
}

FILES:${PN} += "${systemd_system_unitdir}/${BPN}.service"

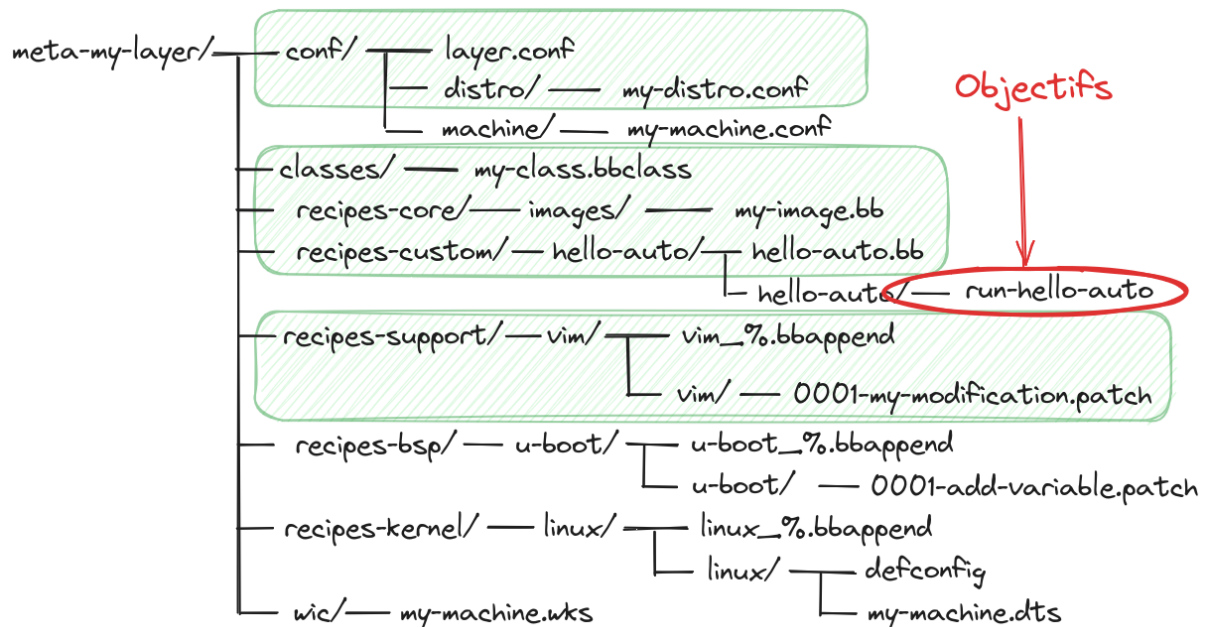
[[Unit]]
Description=Custom Application Starter

[[Service]]
ExecStart=/usr/bin/<application-to-run>

[[Install]]
WantedBy=multi-user.target
```

Travaux pratiques : Lancement d'une application au démarrage

Créez le script et éditez la recette du package `hello-auto` installé précédemment pour qu'il démarre automatiquement au boot. Attention, même si nous avons nommé le package `hello-auto`, l'exécutable installé s'appelle `hello-autotools`.



Solution

```
$ mkdir ../../layers/meta-my-layer/recipes-custom/hello-auto/hello-auto
```

```
$ nano ../../layers/meta-my-layer/recipes-custom/hello-auto/hello-auto/run-hello-auto
```

```
#!/bin/sh
```

```
/usr/bin/hello-autotools &
exit 0
```

```
$ nano ../../layers/meta-my-layer/recipes-custom/hello-auto/hello-auto_git.bb
```

```
SRC_URI += "file://${BPN}"
```

```
inherit update-rc.d
```

```
INITSCRIPT_NAME = "run-${BPN}"
```

```
INITSCRIPT_PARAMS = "start 99 5 . stop 1 0 ."
```

```
do_install:append() {
```

```
    install -d ${D}${sysconfdir}/init.d
```

```
    install -m 0755 ${WORKDIR}/run-${BPN} ${D}${sysconfdir}/init.d/
```

```
}
```