Buildroot: Ajouter de nouveaux packages

Table of Contents

Cette section explique comment de nouveaux packages (bibliothèques ou applications de l'espace utilisateur) peuvent être intégrés dans **Buildroot**. Il montre également comment les packages

existants sont intégrés, ce qui est nécessaire pour résoudre les problèmes ou ajuster leur configuration.

Lorsqu'on ajoute un nouveau paquet, il faut le tester dans diverses conditions (voir Section « Comment tester un paquet ») et vérifier également son style de codage (voir Section « Comment vérifier le codage style“).

Répertoire des packages

Tout d'abord, créer un répertoire sous le répertoire du package pour le logiciel, par exemple libfoo.

Certains packages ont été regroupés par thème dans un sous-répertoire : **x11r7**, **qt5** et **gstreamer**. Si le package appartient à l'une de ces catégories, créer le répertoire de packages dans celles-ci. Les nouveaux sous-répertoires sont toutefois déconseillés.

Fichiers de configuration

Pour que le package s'affiche dans l'outil de configuration, créer un fichier de configuration dans le répertoire de package. Il existe deux types : `Config.in` et `Config.in.host`.

Fichier config.in

Pour les packages utilisés sur la cible, créer un fichier nommé `Config.in`. Ce fichier contiendra les descriptions des options liées à notre logiciel libfoo qui seront utilisées et affichées dans l'outil de configuration. Il doit essentiellement contenir :

```
config BR2_PACKAGE_LIBFOO
        bool "libfoo"
        help
          This is a comment that explains what libfoo is. The help text
          should be wrapped.

          http://foosoftware.org/libfoo/
```

La ligne **bool**, la ligne **help** et les autres informations de métadonnées sur l'option de configuration doivent être mises en retrait avec un onglet. Le texte d'aide lui-même doit être en retrait avec une tabulation et deux espaces, les lignes doivent être enveloppées pour contenir 72 colonnes, où la tabulation compte pour 8, donc 62 caractères dans le texte lui-même. Le texte d'aide doit mentionner l'URL amont du projet après une ligne vide.

En tant que convention spécifique à Buildroot, l'ordre des attributs est le suivant :

- Le type d'option : **bool**, **string**... avec le prompt
- Si nécessaire, la ou les valeurs par défaut
- Toute dépendance sur la cible
- Toutes les dépendances sur la chaîne d'outils

- Toute dépendance à d'autres packages
- Toute dépendance du formulaire de sélection
- Le mot-clé **help** et le texte d'aide.

on peut ajouter d'autres sous-options dans une instruction `if BR2_PACKAGE_LIBFOO…endif` pour configurer des éléments particuliers dans le logiciel. on peut consulter des exemples dans d'autres packages. La syntaxe du fichier Config.in est la même que celle du fichier **Kconfig** du noyau. La documentation de cette syntaxe est disponible sur http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt

Enfin, ajouter le nouveau `libfoo/Config.in` à `package/Config.in` (ou dans un sous-répertoire de catégorie si on a décidé de mettre le paquet dans l'une des catégories existantes). Les fichiers qui y sont inclus sont triés par ordre alphabétique par catégorie et ne sont censés contenir que le nom du package.

```
source "package/libfoo/Config.in"
```

Fichier config.in.host

Certains packages doivent également être compilés pour le système hôte. Il y a deux options ici:

- Le package hôte n'est requis que pour satisfaire les dépendances au moment de la construction d'un ou plusieurs packages cibles. Dans ce cas, ajouter host-foo à la variable **BAR_DEPENDENCIES** du package cible. Aucun fichier `Config.in.host` ne doit être créé.
- Le package hôte doit être explicitement sélectionnable par l'utilisateur à partir du menu de configuration. Dans ce cas, créer un fichier `Config.in.host` pour ce package hôte :

```
config BR2_PACKAGE_HOST_FOO
        bool "host foo"
        help
          This is a comment that explains what foo for the host is.

          http://foosoftware.org/foo/
```

- Le même style de codage et les mêmes options que pour le fichier Config.in sont valides.
- Enfin, ajouter le nouveau `libfoo/Config.in.host` à `package/Config.in.host`. Les fichiers qui y sont inclus sont triés par ordre alphabétique et ne sont censés contenir que le nom du package.

```
    source "paquet/foo/Config.in.host"
```

- Le package hôte sera alors disponible dans le menu Utilitaires de l'hôte.

Le choix dépend de ou sélectionnez

Le fichier `Config.in` du package doit également garantir que les dépendances sont activées. Généralement, Buildroot utilise les règles suivantes :

- Utiliser un type de dépendance sélectionné pour les dépendances sur les bibliothèques. Ces dépendances ne sont généralement pas évidentes et il est donc logique que le système kconfig s'assure que les dépendances sont sélectionnées. Par exemple, le package libgtk2 utilise select **BR2_PACKAGE_LIBGLIB2** pour s'assurer que cette bibliothèque est également activée. Le mot-clé select exprime la dépendance avec une sémantique inversée.
- Utiliser un dépend du type de dépendance lorsque l'utilisateur a vraiment besoin d'être conscient de la dépendance. Typiquement, Buildroot utilise ce type de dépendance pour les dépendances sur l'architecture cible, le support MMU et les options de la chaîne d'outils (voir Section « Dépendances sur les options de la cible et de la chaîne d'outils »), ou pour les dépendances sur les « grandes » choses, comme le X. orgsystème. Le mot clé depend on exprime la dépendance avec une sémantique directe.

Le problème actuel avec le langage kconfig est que ces deux sémantiques de dépendance ne sont pas liées en interne. Par conséquent, il peut être possible de sélectionner un package dont l'une de ses dépendances/exigences n'est pas satisfaite.

Un exemple illustre à la fois l'utilisation de select et de depend on.

```
config BR2_PACKAGE_RRDTOOL
        bool "rrdtool"
        depends on BR2_USE_WCHAR
        select BR2_PACKAGE_FREETYPE
        select BR2_PACKAGE_LIBART
        select BR2_PACKAGE_LIBPNG
        select BR2_PACKAGE_ZLIB
        help
          RRDtool is the OpenSource industry standard, high performance
          data logging and graphing system for time series data.

          http://oss.oetiker.ch/rrdtool/

comment "rrdtool needs a toolchain w/ wchar"
        depends on !BR2_USE_WCHAR
```

Cces deux types de dépendances ne sont transitifs qu'avec les dépendances du même type.

Cela signifie, dans l'exemple suivant :

```
config BR2_PACKAGE_A
        bool "Package A"

config BR2_PACKAGE_B
        bool "Package B"
```

```
        depends on BR2_PACKAGE_A

config BR2_PACKAGE_C
        bool "Package C"
        depends on BR2_PACKAGE_B

config BR2_PACKAGE_D
        bool "Package D"
        select BR2_PACKAGE_B

config BR2_PACKAGE_E
        bool "Package E"
        select BR2_PACKAGE_D
```

- La sélection du package C sera visible si le package B a été sélectionné, qui à son tour n'est visible que si le package A a été sélectionné.
- La sélection du package E sélectionnera le package D, qui sélectionnera le package B, il ne vérifiera pas les dépendances du package B, il ne sélectionnera donc pas le package A.
- Puisque le package B est sélectionné mais que le package A ne l'est pas, cela viole la dépendance du package B sur le package A. Par conséquent, dans une telle situation, la dépendance transitive doit être ajoutée explicitement :

```
config BR2_PACKAGE_D
        bool "Package D"
        select BR2_PACKAGE_B
        depends on BR2_PACKAGE_A

config BR2_PACKAGE_E
        bool "Package E"
        select BR2_PACKAGE_D
        depends on BR2_PACKAGE_A
```

Dans l'ensemble, pour les dépendances de la bibliothèque de packages, select doit être préféré.

De telles dépendances garantiront que l'option de dépendance est également activée, mais pas nécessairement construite avant le package. Pour ce faire, la dépendance doit également être exprimée dans le fichier .mk du package.

Dépendances sur les options de cible et de chaîne d'outils

De nombreux packages dépendent de certaines options de la chaîne d'outils : le choix de la bibliothèque C, la prise en charge de C++, la prise en charge des threads, la prise en charge RPC, la prise en charge de wchar ou la prise en charge de la bibliothèque dynamique. Certains packages ne peuvent être construits que sur certaines architectures cibles, ou si une MMU est disponible dans le processeur.

Ces dépendances doivent être exprimées avec les instructions depend on appropriées dans le fichier Config.in. De plus, pour les dépendances sur les options de la chaîne d'outils, un commentaire doit être affiché lorsque l'option n'est pas activée, afin que l'utilisateur sache pourquoi le package n'est pas disponible. Les dépendances sur l'architecture cible ou le support MMU ne doivent pas être rendues visibles dans un commentaire : puisqu'il est peu probable que l'utilisateur puisse choisir librement une autre cible, cela n'a aucun sens de montrer ces dépendances explicitement.

Le commentaire ne doit être visible que si l'option de configuration elle-même est visible lorsque les dépendances de l'option de la chaîne d'outils sont remplies. Cela signifie que toutes les autres dépendances du package (y compris les dépendances sur l'architecture cible et la prise en charge de la MMU) doivent être répétées dans la définition de commentaire. Pour que ce soit clair, l'instruction depend de pour ces options non liées à la chaîne d'outils doit être séparée de l'instruction depend de pour les options de la chaîne d'outils. S'il existe une dépendance sur une option de configuration dans ce même fichier (généralement le package principal), il est préférable d'avoir une construction globale `if … endif` plutôt que de répéter l'instruction depend on sur le commentaire et d'autres options de configuration.

Le format général d'un commentaire de dépendance pour le paquet foo est :

```
foo a besoin d'une chaîne d'outils avec featA, featB, featC
```

par exemple:

```
mpd a besoin d'une chaîne d'outils avec C++, threads, wchar
```

ou

```
crda a besoin d'une chaîne d'outils avec des threads
```

Ce texte est délibérément bref, afin qu'il puisse tenir sur un terminal de 80 caractères.

Le reste de cette section énumère les différentes options de cible et de chaîne d'outils, les symboles de configuration correspondants dont il faut dépendre et le texte à utiliser dans le commentaire.

Options	**Symbole de dépendance**	**Chaîne de commentaire**
Architecture cible	**BR2_powerpc**, **BR2_mips**, ... (voir arch/Config.in)	aucun commentaire à ajouter
Prise en charge MMU	**BR2_USE_MMU**	aucun commentaire à ajouter
Gcc _sync[1)]	**BR2_TOOLCHAIN_HAS_SYNC_1** pour 1 octet **BR2_TOOLCHAIN_HAS_SYNC_2** pour 2 octets **BR2_TOOLCHAIN_HAS_SYNC_4** pour 4 octets **BR2_TOOLCHAIN_HAS_SYNC_8** pour 8 octets.	aucun commentaire à ajouter

Options	Symbole de dépendance	Chaîne de commentaire
Eléments intégrés Gcc _atomic*	**BR2_TOOLCHAIN_HAS_ATOMIC.**	aucun commentaire à ajouter
En-têtes du noyau	**BR2_TOOLCHAIN_HEADERS_AT_LEAST_X_Y**[2)]	en-têtes >= X.Y et/ou en-têtes ⇐ X.Y
Version CCG	**BR2_TOOLCHAIN_GCC_AT_LEAST_X_Y**[3)]	gcc >= X.Y et/ou gcc ⇐ X.Y
Version hôte de GCC	**BR2_HOST_GCC_AT_LEAST_X_Y**[4)]	aucun commentaire à ajouter ce n'est généralement pas le paquet lui-même qui a une version minimale de GCC hôte, mais plutôt un paquet hôte dont il dépend.
Bibliothèque C	**BR2_TOOLCHAIN_USES_GLIBC** **BR2_TOOLCHAIN_USES_MUSL** **BR2_TOOLCHAIN_USES_UCLIBC**	pour la bibliothèque C, un texte de commentaire légèrement différent est utilisé : foo a besoin d'une chaîne d'outils glibc, ou foo a besoin d'une chaîne d'outils glibc avec C++
Prise en charge de C++	**BR2_INSTALL_LIBSTDCPP**	C++
Prise en charge D	**BR2_TOOLCHAIN_HAS_DLANG**	Dlang
Prise en charge Fortran	**BR2_TOOLCHAIN_HAS_FORTRAN**	fortran
Prise en charge des threads	**BR2_TOOLCHAIN_HAS_THREADS**	threads (sauf si BR2_TOOLCHAIN_HAS_THREADS_NPTL est également nécessaire, auquel cas, spécifier uniquement NPTL est suffisant)
Prise en charge des filetages NPTL	**BR2_TOOLCHAIN_HAS_THREADS_NPTL**	NPTL
Prise en charge RPC	**BR2_TOOLCHAIN_HAS_NATIVE_RPC**	RPC
Prise en charge wchar	**BR2_USE_WCHAR**	wchar
bibliothèque dynamique	**!BR2_STATIC_LIBS**	bibliothèque dynamique

Dépendances sur un noyau Linux construit par buildroot

Certains packages ont besoin d'un noyau Linux pour être construits par buildroot. Il s'agit généralement de modules de noyau ou de micrologiciels. Un commentaire doit être ajouté dans le fichier Config.in pour exprimer cette dépendance, similaire aux dépendances sur les options de la chaîne d'outils. Le format général est :

```
foo needs a Linux kernel to be built
```

S'il existe une dépendance à la fois sur les options de la chaîne d'outils et sur le noyau Linux, utiliser ce format :

```
foo needs a toolchain w/ featA, featB, featC and a Linux kernel to be built
```

Dépendances sur la gestion udev /dev

Si un paquet nécessite une gestion udev /dev, il doit dépendre du symbole **BR2_PACKAGE_HAS_UDEV**, et le commentaire suivant doit être ajouté :

```
foo needs udev /dev management
```

S'il existe une dépendance à la fois sur les options de la chaîne d'outils et sur la gestion udev /dev, utiliser ce format :

```
foo needs udev /dev management and a toolchain w/ featA, featB, featC
```

Dépendances sur les fonctionnalités fournies par les packages virtuels

Certaines fonctionnalités peuvent être fournies par plusieurs packages, comme les bibliothèques openGL.

Voir Section « Infrastructure pour les packages virtuels » pour plus d'informations sur les packages virtuels.

Le fichier .mk

Enfin, voici la partie la plus difficile. Créer un fichier nommé libfoo.mk. Il décrit comment le paquet doit être téléchargé, configuré, construit, installé, etc.

Selon le type de package, le fichier .mk doit être écrit de manière différente, en utilisant différentes infrastructures :

- **Makefiles pour les packages génériques (n'utilisant pas les autotools ou CMake)** : ils sont basés sur une infrastructure similaire à celle utilisée pour les packages basés sur les autotools, mais nécessitent un peu plus de travail de la part du développeur. Ils spécifient ce qui doit être fait pour la configuration, la compilation et l'installation du paquet. Cette infrastructure doit être utilisée pour tous les packages qui n'utilisent pas les autotools comme système de construction. À l'avenir, d'autres infrastructures spécialisées pourraient être écrites pour d'autres systèmes de construction. buildroot les couvrons dans un tutoriel et une référence.
- **Makefiles pour les logiciels basés sur les autotools (autoconf, automake, etc.)** : buildroot fournit une infrastructure dédiée pour de tels packages, car les autotools sont un système de construction très courant. Cette infrastructure doit être utilisée pour les nouveaux

packages qui s'appuient sur les autotools comme système de construction. buildroot les couvrons à travers un tutoriel et une référence.
- **Makefiles pour les logiciels basés sur cmake**: buildroot fournit une infrastructure dédiée pour de tels packages, car CMake est un système de construction de plus en plus utilisé et a un comportement standardisé. Cette infrastructure doit être utilisée pour les nouveaux packages qui reposent sur CMake. buildroot les couvrons à travers un tutoriel et une référence.
- **Makefiles pour les modules Python**: buildroot a une infrastructure dédiée pour les modules Python qui utilisent soit le distutils ou le mécanisme setuptools. buildroot les couvrons à travers un tutoriel et une référence.
- **Makefiles pour les modules Lua** : buildroot a une infrastructure dédiée pour les modules Lua disponible sur le site Web LuaRocks. buildroot les couvrons à travers un tutoriel et une référence.

Plus de précisions sur la mise en forme : voir les règles d'écriture.

Le fichier .hash

Lorsque cela est possible, ajouter un troisième fichier, nommé libfoo.hash, qui contient les hachages des fichiers téléchargés pour le paquet libfoo. La seule raison de ne pas ajouter de fichier .hash est lorsque la vérification du hachage n'est pas possible en raison de la façon dont le package est téléchargé.

Lorsqu'un paquet a un choix de sélection de version, le fichier de hachage peut être stocké dans un sous-répertoire nommé d'après la version, par ex. package/libfoo/1.2.3/libfoo.hash. Ceci est particulièrement important si les différentes versions ont des termes de licence différents, mais qu'elles sont stockées dans le même fichier. Sinon, le fichier de hachage doit rester dans le répertoire du package.

Les hachages stockés dans ce fichier sont utilisés pour valider l'intégrité des fichiers téléchargés et des fichiers de licence.

Le format de ce fichier est d'une ligne pour chaque fichier pour lequel vérifier le hachage, chaque ligne avec les trois champs suivants séparés par deux espaces :

- le type de hachage, l'un des suivants :
  - md5, sha1, sha224, sha256, sha384, sha512, aucun
- le hash du fichier :
  - pour aucun, un ou plusieurs caractères sans espace, généralement juste la chaîne xxx
  - pour md5, 32 caractères hexadécimaux
  - pour sha1, 40 caractères hexadécimaux
  - pour sha224, 56 caractères hexadécimaux
  - pour sha256, 64 caractères hexadécimaux
  - pour sha384, 96 caractères hexadécimaux
  - pour sha512, 128 caractères hexadécimaux
- le nom du fichier :
  - pour une archive source : le nom de base du fichier, sans élément de répertoire,
  - pour un fichier de licence : le chemin tel qu'il apparaît dans **FOO_LICENSE_FILES**.

Les lignes commençant par un signe # sont considérées comme des commentaires et ignorées. Les lignes vides sont ignorées.

Il peut y avoir plusieurs hachages pour un même fichier, chacun sur sa propre ligne. Dans ce cas, tous les hachages doivent correspondre.

Idéalement, les hachages stockés dans ce fichier doivent correspondre aux hachages publiés par l'amont, par ex. sur leur site Web, dans l'annonce par e-mail... Si l'amont fournit plus d'un type de hachage (par exemple, sha1 et sha512), il est préférable d'ajouter tous ces hachages dans le fichier .hash. Si l'amont ne fournit aucun hachage, ou ne fournit qu'un hachage md5, calculer au moins un hachage fort (de préférence sha256, mais pas md5), et mentionnez-le dans une ligne de commentaire au-dessus des hachages.

Les hachages des fichiers de licence sont utilisés pour détecter un changement de licence lorsqu'une version de package est modifiée. Les hachages sont vérifiés lors de l'exécution de la cible make legal-info. Pour un paquet avec plusieurs versions (comme Qt5), créer le fichier de hachage dans un sous-répertoire <packageversion> de ce paquet (voir aussi Section 19.2, « Comment les correctifs sont appliqués »).

Le type de hachage none est réservé aux archives téléchargées depuis un référentiel, comme un clone git, un checkout subversion...

L'exemple ci-dessous définit un sha1 et un sha256 publiés par l'amont pour l'archive principale libfoo-1.2.3.tar.bz2, un md5 de l'amont et un hachage sha256 calculé localement pour un blob binaire, un sha256 pour un patch téléchargé, et une archive sans hachage :

```
# Hashes from:
http://www.foosoftware.org/download/libfoo-1.2.3.tar.bz2.{sha1,sha256}:
sha1  486fb55c3efa71148fe07895fd713ea3a5ae343a  libfoo-1.2.3.tar.bz2
sha256  efc8103cc3bcb06bda6a781532d12701eb081ad83e8f90004b39ab81b65d4369
libfoo-1.2.3.tar.bz2

# md5 from: http://www.foosoftware.org/download/libfoo-1.2.3.tar.bz2.md5,
sha256 locally computed:
md5  2d608f3c318c6b7557d551a5a09314f03452f1a1  libfoo-data.bin
sha256  01ba4719c80b6fe911b091a7c05124b64eeece964e09c058ef8f9805daca546b
libfoo-data.bin

# Locally computed:
sha256  ff52101fb90bbfc3fe9475e425688c660f46216d7e751c4bbdb1dc85cdccacb9
libfoo-fix-blabla.patch

# No hash for 1234:
none  xxx  libfoo-1234.tar.gz

# Hash for license files:
sha256  a45a845012742796534f7e91fe623262ccfb99460a2bd04015bd28d66fba95b8
```

```
COPYING
sha256  01b1f9f2c8ee648a7a596a1abe8aa4ed7899b1c9e5551bda06da6e422b04aa55
doc/COPYING.LGPL
```

Si le fichier .hash est présent et qu'il contient un ou plusieurs hachages pour un fichier téléchargé, le ou les hachages calculés par Buildroot (après le téléchargement) doivent correspondre au(x) hachage(s) stocké(s) dans le fichier .hash. Si un ou plusieurs hachages ne correspondent pas, Buildroot considère qu'il s'agit d'une erreur, supprime le fichier téléchargé et abandonne.

Si le fichier .hash est présent, mais qu'il ne contient pas de hachage pour un fichier téléchargé, Buildroot considère qu'il s'agit d'une erreur et abandonne. Cependant, le fichier téléchargé est laissé dans le répertoire de téléchargement car cela indique généralement que le fichier .hash est erroné mais que le fichier téléchargé est probablement correct.

Les hachages sont actuellement vérifiés pour les fichiers extraits des serveurs http/ftp, des référentiels Git, des fichiers copiés à l'aide de scp et des fichiers locaux. Les hachages ne sont pas vérifiés pour les autres systèmes de contrôle de version (tels que Subversion, CVS, etc.) car Buildroot ne génère actuellement pas de fichiers reproductibles.tarballs lorsque le code source est extrait de ces systèmes de contrôle de version.

Les hachages ne doivent être ajoutés dans les fichiers .hash que pour les fichiers dont la stabilité est garantie. Par exemple, les correctifs générés automatiquement par Github ne sont pas garantis stables et, par conséquent, leurs hachages peuvent changer au fil du temps. Ces correctifs ne doivent pas être téléchargés, mais plutôt ajoutés localement au dossier du package.

Si le fichier .hash est manquant, aucune vérification n'est effectuée.

Infrastructure pour les packages Génériques

Tous les packages dont le système de construction n'est pas l'un des systèmes standard, tels que autotools ou CMake, peuvent être inclus avec l'infrastructure pour les packages Génériques. Cela inclut généralement les packages dont le système de construction est basé sur des Makefiles écrits à la main ou des scripts shell.

didacticiel sur les packages génériques

```
01:
################################################################################
####
02: #
03: # libfoo
04: #
05:
################################################################################
####
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
```

```
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_LICENSE = GPL-3.0+
11: LIBFOO_LICENSE_FILES = COPYING
12: LIBFOO_INSTALL_STAGING = YES
13: LIBFOO_CONFIG_SCRIPTS = libfoo-config
14: LIBFOO_DEPENDENCIES = host-libaaa libbbb
15:
16: define LIBFOO_BUILD_CMDS
17:     $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D) all
18: endef
19:
20: define LIBFOO_INSTALL_STAGING_CMDS
21:     $(INSTALL) -D -m 0755 $(@D)/libfoo.a $(STAGING_DIR)/usr/lib/libfoo.a
22:     $(INSTALL) -D -m 0644 $(@D)/foo.h $(STAGING_DIR)/usr/include/foo.h
23:     $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(STAGING_DIR)/usr/lib
24: endef
25:
26: define LIBFOO_INSTALL_TARGET_CMDS
27:     $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(TARGET_DIR)/usr/lib
28:     $(INSTALL) -d -m 0755 $(TARGET_DIR)/etc/foo.d
29: endef
30:
31: define LIBFOO_USERS
32:     foo -1 libfoo -1 * - - - LibFoo daemon
33: endef
34:
35: define LIBFOO_DEVICES
36:     /dev/foo c 666 0 0 42 0 - - -
37: endef
38:
39: define LIBFOO_PERMISSIONS
40:     /bin/foo f 4755 foo libfoo - - - - -
41: endef
42:
43: $(eval $(generic-package))
```

Le Makefile commence:

- Aux lignes 7 à 11 avec des informations de métadonnées : la version du paquet (**LIBFOO_VERSION)**, le nom de l'archive contenant le paquet (**LIBFOO_SOURCE**) (archive xz-ed recommandée) l'emplacement Internet à partir duquel l'archive peut être téléchargée à partir de (LIBFOO_SITE), la licence (**LIBFOO_LICENSE**) et le fichier avec le texte de la licence (**LIBFOO_LICENSE_FILES**). Toutes les variables doivent commencer par le même préfixe**, LIBFOO_** dans ce cas. Ce préfixe est toujours la version majuscule du nom du package (voir ci-dessous pour comprendre où le nom du package est défini).
- À la ligne 12, on spécifie que ce paquet veut installer quelque chose dans l'espace de staging. Ceci est souvent nécessaire pour les bibliothèques, car elles doivent installer des fichiers d'en-tête et d'autres fichiers de développement dans l'espace de staging. Cela garantira que les commandes répertoriées dans la variable **LIBFOO_INSTALL_STAGING_CMDS** seront exécutées.
- À la ligne 13, on spécifie qu'il y a des corrections à apporter à certains des fichiers libfoo-config

qui ont été installés pendant la phase **LIBFOO_INSTALL_STAGING_CMDS**. Ces fichiers *-config sont des fichiers de script shell exécutables situés dans le répertoire **$(STAGING_DIR)/usr/bin** et sont exécutés par d'autres packages tiers pour connaître l'emplacement et les indicateurs de liaison de ce package particulier.

Le problème est que tous ces fichiers *-config donnent par défaut des drapeaux de liaison au système hôte erronés qui ne conviennent pas à la compilation croisée.

Par exemple : `-I/usr/include instead of -I$(STAGING_DIR)/usr/include` ou: `-L/usr/lib instead of -L$(STAGING_DIR)/usr/lib`

Donc, un peu de magie sed est faite à ces scripts pour leur faire donner des drapeaux corrects. L'argument à donner à **LIBFOO_CONFIG_SCRIPTS** est le ou les noms de fichier du ou des scripts shell à corriger. Tous ces noms sont relatifs à **$(STAGING_DIR)/usr/bin** et si nécessaire plusieurs noms peuvent être donnés.

De plus, les scripts répertoriés dans **LIBFOO_CONFIG_SCRIPTS** sont supprimés de **$(TARGET_DIR)/usr/bin**, car ils ne sont pas nécessaires sur la cible.

Exemple 1 Script de configuration : package divine

Le package divine installe le script shell **$(STAGING_DIR)/usr/bin/divine-config**.

Donc, sa correction serait:

```
DIVINE_CONFIG_SCRIPTS = divine-config
```

Exemple 2 Script de configuration : package imagemagick :

Le package imagemagick installe les scripts suivants :
$(STAGING_DIR)/usr/bin/{Magick,Magick++,MagickCore,MagickWand,Wand}-config

Donc, sa correction serait:

```
IMAGEMAGICK_CONFIG_SCRIPTS = \
   Magick-config Magick++-config \
   MagickCore-config MagickWand-config Wand-config
```

- À la ligne 14, buildroot spécifie la liste des dépendances sur lesquelles repose ce paquet. Ces dépendances sont répertoriées en termes de noms de packages en minuscules, qui peuvent être des packages pour la cible (sans le préfixe host-) ou des packages pour l'hôte (avec le préfixe host-). Buildroot s'assurera que tous ces packages sont construits et installés avant que le package actuel ne démarre sa configuration.

Le reste du Makefile, lignes 16..29, définit ce qui doit être fait aux différentes étapes de la configuration, de la compilation et de l'installation du paquet. **LIBFOO_BUILD_CMDS** indique les étapes à suivre pour construire le package. **LIBFOO_INSTALL_STAGING_CMDS** indique les étapes à suivre pour installer le package dans l'espace de staging. **LIBFOO_INSTALL_TARGET_CMDS** indique

les étapes à suivre pour installer le package dans l'espace cible.

Toutes ces étapes reposent sur la variable $(@D), qui contient le répertoire où le code source du package a été extrait.

- Aux lignes 31..33, buildroot définit un utilisateur qui est utilisé par ce paquet (par exemple pour exécuter un démon en tant que non root) (**LIBFOO_USERS**).
- Aux lignes 35..37, buildroot définit un fichier de nœud de périphérique utilisé par ce paquet (**LIBFOO_DEVICES**).
- À la ligne 39..41, buildroot définit les autorisations à définir sur des fichiers spécifiques installés par ce package (**LIBFOO_PERMISSIONS**).
- A la ligne 43, buildroot appelons la fonction generic-package, qui génère, selon les variables définies précédemment, tout le code Makefile nécessaire au fonctionnement du package.

référence de package générique

Il existe deux variantes de la cible générique. La macro generic-package est utilisée pour les packages à compiler de manière croisée pour la cible. La macro host-generic-package est utilisée pour les packages hôtes, compilés nativement pour l'hôte. Il est possible d'appeler les deux dans un seul fichier .mk : une fois pour créer les règles pour générer un package cible et une fois pour créer les règles pour générer un package hôte :

```
$(eval $(generic-package))
$(eval $(host-generic-package))
```

Cela peut être utile si la compilation du package cible nécessite l'installation de certains outils sur l'hôte. Si le nom du package est libfoo, alors le nom du package pour la cible est également libfoo, tandis que le nom du package pour l'hôte est host-libfoo. Ces noms doivent être utilisés dans les variables DEPENDENCIES des autres packages, s'ils dépendent de libfoo ou host-libfoo.

L'appel à la macro generic-package et/ou host-generic-package doit se trouver à la fin du fichier .mk, après toutes les définitions de variables. L'appel à host-generic-package doit être après l'appel à generic-package, le cas échéant.

Pour le package cible, le package-générique utilise les variables définies par le fichier .mk et préfixées par le nom du package en majuscule : **LIBFOO_**. host-generic-package utilise les variables **HOST_LIBFOO_**. Pour certaines variables, si la variable préfixée **HOST_LIBFOO_** n'existe pas, l'infrastructure du package utilise la variable correspondante préfixée par **LIBFOO_**. Ceci est fait pour les variables susceptibles d'avoir la même valeur pour les packages cible et hôte. Voir ci-dessous pour plus de détails.

La liste des variables pouvant être définies dans un fichier .mk pour fournir des informations sur les métadonnées est (en supposant que le nom du package est libfoo) :

LIBFOO_VERSION	obligatoire, doit contenir la version du package (si **HOST_LIBFOO_VERSION** n'existe pas, il est supposé être le même que **LIBFOO_VERSION**. Il peut également s'agir d'un numéro de révision ou d'une balise pour les packages extraits directement de leur système de contrôle de version). Exemples: - une version pour une version tarball : `LIBFOO_VERSION = 0.1.2` - un sha1 pour un arbre git : `LIBFOO_VERSION = cb9d6aa9429e838f0e54faa3d455bcbab5eef057` - une balise pour un arbre git `LIBFOO_VERSION = v0.1.2` L'utilisation d'un nom de branche comme **FOO_VERSION** n'est pas prise en charge, car cela ne fonctionne pas: - en raison de la mise en cache locale, Buildroot ne récupèrera pas le référentiel; - parce que deux builds ne peuvent jamais être parfaitement simultanés, et parce que le référentiel distant peut obtenir de nouveaux commits sur la branche à tout moment, deux utilisateurs, utilisant le même arbre Buildroot et construisant la même configuration, peuvent obtenir une source différente, rendant ainsi la construction non reproductible.
LIBFOO_SOURCE	peut contenir le nom de l'archive tar du paquet, que Buildroot utilisera pour télécharger l'archive tar depuis **LIBFOO_SITE**. Si **HOST_LIBFOO_SOURCE** n'est pas spécifié, la valeur par défaut est **LIBFOO_SOURCE**. Si aucune n'est spécifiée, alors la valeur est supposée être **libfoo-$(LIBFOO_VERSION).tar.gz**. Exemple : `LIBFOO_SOURCE = foobar-$(LIBFOO_VERSION).tar.bz2`
LIBFOO_PATCH	peut contenir une liste de noms de fichiers de correctif séparés par des espaces, que Buildroot téléchargera et appliquera au code source du package. Si une entrée contient `://`, alors Buildroot supposera qu'il s'agit d'une URL complète et téléchargera le correctif à partir de cet emplacement. Sinon, Buildroot supposera que le correctif doit être téléchargé depuis **LIBFOO_SITE**. Si **HOST_LIBFOO_PATCH** n'est pas spécifié, la valeur par défaut est **LIBFOO_PATCH**. Les correctifs inclus dans Buildroot lui-même utilisent un mécanisme différent : tous les fichiers de la forme.patch présents dans le répertoire du package à l'intérieur de Buildroot seront appliqués au package après extraction (voir patcher un package). Enfin, les correctifs listés dans la variable **LIBFOO_PATCH** sont appliqués avant les correctifs stockés dans le Répertoire du package Buildroot.

LIBFOO_SITE	fournit l'emplacement du package, qui peut être une URL ou un chemin de système de fichiers local. HTTP, FTP et SCP sont des types d'URL pris en charge pour la récupération des archives de packages. Dans ces cas, ne pas mettre de barre oblique finale : elle sera ajoutée par Buildroot entre le répertoire et le nom de fichier, le cas échéant. Git, Subversion, Mercurial et Bazaar sont des types d'URL pris en charge pour récupérer des packages directement à partir de systèmes de gestion de code source. Il existe une fonction d'assistance pour faciliter le téléchargement des archives sources depuis GitHub (se reporter à la Section « Comment ajouter un paquet depuis GitHub » pour plus de détails). Un chemin de système de fichiers peut être utilisé pour spécifier soit une archive tar, soit un répertoire contenant le code source du paquet. Voir **LIBFOO_SITE_METHOD** ci-dessous pour plus de détails sur le fonctionnement de la récupération. Les URL SCP doivent être au format `scp://[user@]host:filepath`, et que filepath est relatif au répertoire personnel de l'utilisateur, on peut donc ajouter une barre oblique au début du chemin pour les chemins absolus : `scp://[utilisateur@]hôte :/cheminabsolu`. Si **HOST_LIBFOO_SITE** n'est pas spécifié, la valeur par défaut est **LIBFOO_SITE**. Exemples :
	`LIBFOO_SITE=http://www.libfoosoftware.org/libfoo`
	`LIBFOO_SITE=http://svn.xiph.org/trunk/Tremor`
	`LIBFOO_SITE=/opt/software/libfoo.tar.gz`
	`LIBFOO_SITE=$(TOPDIR)/. ./src/libfoo`

LIBFOO_DL_OPTS	est une liste séparée par des espaces d'options supplémentaires à transmettre au téléchargeur. Utile pour récupérer des documents avec vérification côté serveur des identifiants et mots de passe des utilisateurs, ou pour utiliser un proxy. Toutes les méthodes de téléchargement valides pour **LIBFOO_SITE_METHOD** sont prises en charge ; les options valides dépendent de la méthode de téléchargement.
LIBFOO_EXTRA_DOWNLOADS	est une liste séparée par des espaces de fichiers supplémentaires que Buildroot doit télécharger. Si une entrée contient `://`, Buildroot supposera qu'il s'agit d'une URL complète et téléchargera le fichier à l'aide de cette URL. Sinon, Buildroot supposera que le fichier à télécharger se trouve dans **LIBFOO_SITE**. Buildroot ne fera rien avec ces fichiers supplémentaires, sauf les télécharger : ce sera à la recette du paquet de les utiliser depuis **$(LIBFOO_DL_DIR)**.

LIBFOO_SITE_METHOD	détermine la méthode utilisée pour récupérer ou copier le code source du package. Dans de nombreux cas, Buildroot devine la méthode à partir du contenu de **LIBFOO_SITE** et la définition de **LIBFOO_SITE_METHOD** n'est pas nécessaire. Lorsque **HOST_LIBFOO_SITE_METHOD** n'est pas spécifié, la valeur par défaut est **LIBFOO_SITE_METHOD**. Les valeurs possibles de **LIBFOO_SITE_METHOD** sont: - **wget** pour les téléchargements FTP/HTTP normaux d'archives. Utilisé par défaut lorsque **LIBFOO_SITE** commence par `http://`, `https://` ou `ftp://`. - **scp** pour les téléchargements d'archives via SSH avec scp. Utilisé par défaut lorsque **LIBFOO_SITE** commence par `scp://`. - **svn** pour récupérer le code source d'un dépôt Subversion. Utilisé par défaut lorsque **LIBFOO_SITE** commence par `svn://`. Lorsqu'une URL de référentiel Subversion `http://` est spécifiée dans **LIBFOO_SITE**, il faut spécifier **LIBFOO_SITE_METHOD=svn**. Buildroot effectue une extraction qui est conservée sous forme d'archive dans le cache de téléchargement ; les versions suivantes utilisent l'archive tar au lieu d'effectuer une autre extraction. - **cvs** pour récupérer le code source d'un référentiel CVS. Utilisé par défaut lorsque **LIBFOO_SITE** commence par `cvs://`. Le code source téléchargé est mis en cache comme avec la méthode svn. Le mode pserver anonyme est supposé autrement explicitement défini sur **LIBFOO_SITE**. - - `LIBFOO\_SITE=cvs://libfoo.net:/cvsroot/libfoo` et `LIBFOO\_SITE=cvs://:ext:libfoo.net:/cvsroot/libfoo sont acceptés`, sur l'ancien mode d'accès pserver anonyme est supposé. **LIBFOO_SITE** doit contenir l'URL source ainsi que le répertoire du référentiel distant. Le module est le nom du package. **LIBFOO_VERSION** est obligatoire et doit être une balise, une branche ou une date (par exemple “2014-10-20”, “2014-10-20 13:45”, “2014-10-20 13:45+01” voir ” man cvs“ pour plus de détails). - **git** pour récupérer le code source d'un référentiel Git. Utilisé par défaut lorsque **LIBFOO_SITE** commence par `git://`. Le code source téléchargé est mis en cache comme avec la méthode **svn**. - **hg** pour récupérer le code source d'un référentiel Mercurial. Il faut spécifier **LIBFOO_SITE_METHOD=hg** lorsque **LIBFOO_SITE** contient une URL de référentiel Mercurial. Le code source téléchargé est mis en cache comme avec la méthode **svn**. - **bzr** pour récupérer le code source d'un référentiel Bazaar. Utilisé par défaut lorsque **LIBFOO_SITE** commence par `bzr://`. Le code source téléchargé est mis en cache comme avec la méthode **svn**. - **file** pour une archive locale. On devrait l'utiliser lorsque **LIBFOO_SITE** spécifie une archive tar de package comme nom de fichier local. Utile pour les logiciels qui ne sont pas disponibles publiquement ou dans le contrôle de version. - **local** pour un répertoire de code source local. Il convient de l'utiliser lorsque **LIBFOO_SITE** spécifie un chemin de répertoire local contenant le code source du package. Buildroot copie le contenu du répertoire source dans le répertoire de construction du package. Pour les packages locaux, aucun correctif n'est appliqué. Si on a encore besoin de patcher le code source, utiliser **LIBFOO_POST_RSYNC_HOOKS**, voir Section « Utiliser le **POST_HOOKS RSYNC**».

LIBFOO_GIT_SUBMODULES	peut être défini sur YES pour créer une archive avec les sous-modules git dans le référentiel. Ceci n'est disponible que pour les packages téléchargés avec git (c'est-à-dire lorsque **LIBFOO_SITE_METHOD=git**). Buildroot essaye de ne pas utiliser ces sous-modules git lorsqu'ils contiennent des bibliothèques groupées, auquel cas buildroot préférons utiliser ces bibliothèques à partir de leur propre package.
LIBFOO_STRIP_COMPONENTS	est le nombre de composants principaux (répertoires) que tar doit supprimer des noms de fichiers lors de l'extraction. L'archive tar de la plupart des packages a un composant principal nommé ”<pkg-name>-<pkg-version>“, ainsi Buildroot passe –strip-components=1 à tar pour le supprimer. Pour les packages non standard qui n'ont pas ce composant, ou qui ont plus d'un composant principal à supprimer, définir cette variable avec la valeur à transmettre à tar. Par défaut : 1.
LIBFOO_EXCLUDES	est une liste de modèles séparés par des espaces à exclure lors de l'extraction de l'archive. Chaque élément de cette liste est passé en tant qu'option –exclude de tar. Par défaut, vide.
LIBFOO_DEPENDENCIES	répertorie les dépendances (en termes de nom de package) nécessaires à la compilation du package cible actuel. Ces dépendances sont garanties d'être compilées et installées avant le démarrage de la configuration du package actuel. Cependant, les modifications apportées à la configuration de ces dépendances ne forceront pas une reconstruction du package actuel. De la même manière, HOST_LIBFOO_DEPENDENCIES répertorie les dépendances du package hôte actuel.
LIBFOO_EXTRACT_DEPENDENCIES	répertorie les dépendances (en termes de nom de package) requises pour que le package cible actuel soit extrait. Ces dépendances sont garanties d'être compilées et installées avant le démarrage de l'étape d'extraction du package actuel. Ceci n'est utilisé qu'en interne par l'infrastructure de packages et ne doit généralement pas être utilisé directement par les packages.
LIBFOO_PATCH_DEPENDENCIES	répertorie les dépendances (en termes de nom de package) requises pour que le package actuel soit corrigé. Il est garanti que ces dépendances seront extraites et corrigées (mais pas nécessairement construites) avant que le package actuel ne soit corrigé. De la même manière, **HOST_LIBFOO_PATCH_DEPENDENCIES** répertorie les dépendances du package hôte actuel. Ceci est rarement utilisé; généralement**,LIBFOO_DEPENDENCIES** est ce qu'on veutvraiment utiliser.
LIBFOO_PROVIDES	répertorie tous les packages virtuels dont libfoo est une implémentation. Voir Section « Infrastructure pour les packages virtuels ».
LIBFOO_INSTALL_STAGING	peut être défini sur **YES** ou **NO** (par défaut). Si défini sur **YES**, les commandes des variables **LIBFOO_INSTALL_STAGING_CMDS** sont exécutées pour installer le package dans le répertoire intermédiaire.
LIBFOO_INSTALL_TARGET	peut être défini sur **YES** (par défaut) ou **NO**. Si défini sur **YES**, les commandes des variables **LIBFOO_INSTALL_TARGET_CMDS** sont exécutées pour installer le package dans le répertoire cible.
LIBFOO_INSTALL_IMAGES	peut être défini sur **YES** ou **NO** (par défaut). Si défini sur **YES**, les commandes de la variable **LIBFOO_INSTALL_IMAGES_CMDS** sont exécutées pour installer le package dans le répertoire des images.

LIBFOO_CONFIG_SCRIPTS	répertorie les noms des fichiers dans **$(STAGING_DIR)/usr/bin** qui nécessitent une correction spéciale pour les rendre compatibles avec la compilation croisée. Plusieurs noms de fichiers séparés par un espace peuvent être donnés et tous sont relatifs à **$(STAGING_DIR)/usr/bin**. Les fichiers répertoriés dans **LIBFOO_CONFIG_SCRIPTS** sont également supprimés de **$(TARGET_DIR)/usr/bin** car ils ne sont pas nécessaires sur la cible.
LIBFOO_DEVICES	répertorie les fichiers de périphérique à créer par Buildroot lors de l'utilisation de la table de périphérique statique. La syntaxe à utiliser est celle de makedevs. on peut trouver de la documentation pour cette syntaxe dans le Chapitre 25, Documentation de la syntaxe Makedev. Cette variable est facultative.
LIBFOO_PERMISSIONS	liste les changements de permissions à faire à la fin du processus de construction. La syntaxe est encore une fois celle de makedevs. on peut trouver de la documentation pour cette syntaxe dans le Chapitre 25, Documentation de la syntaxe Makedev. Cette variable est facultative.
LIBFOO_USERS	répertorie les utilisateurs à créer pour ce package, s'il installe un programme qu'on souhaite exécuter en tant qu'utilisateur spécifique (par exemple, en tant que démon ou en tant que tâche cron). La syntaxe est similaire dans l'esprit à celle de makedevs, et est décrite dans le Chapitre 26, Documentation de la syntaxe de Makeusers. Cette variable est facultative.
LIBFOO_LICENSE	définit la licence (ou les licences) sous lesquelles le package est publié. Ce nom apparaîtra dans le fichier manifeste produit par make legal-info. Si la licence apparaît dans la liste des licences **SPDX**, utiliser l'identificateur court **SPDX** pour uniformiser le fichier manifeste. Sinon, décrire la licence de manière précise et concise, en évitant les noms ambigus comme BSD qui désignent en fait une famille de licences. Cette variable est facultative. S'il n'est pas défini, inconnu apparaîtra dans le champ de licence du fichier manifeste de ce package. Le format attendu pour cette variable doit respecter les règles suivantes : - Si différentes parties du package sont publiées sous différentes licences, séparer les licences par des virgules es (par exemple `LIBFOO\_LICENSE = GPL-2.0+ LGPL-2.1+`). S'il existe une distinction claire entre quel composant est sous licence sous quelle licence, alors annoter la licence avec ce composant, entre parenthèses (par exemple, `LIBFOO\_LICENSE = GPL-2.0+ (programmes), LGPL-2.1+ (bibliothèques)`). - Si certaines licences sont conditionnées à l'activation d'une sous-option, ajouter une virgule aux licences conditionnelles (par exemple : `FOO\_LICENSE += , GPL-2.0+ (programmes))`; l'infrastructure supprimera en interne l'espace avant la virgule. - Si le package est à double licence, séparer les licences avec le mot-clé ou (par exemple, `LIBFOO\_LICENSE = AFL-2.1 ou GPL-2.0+`).
LIBFOO_LICENSE_FILES	est une liste de fichiers séparés par des espaces dans l'archive tar du package qui contient la ou les licences sous lesquelles le package est publié. make legal-info copie tous ces fichiers dans le répertoire legal-info. Voir Chapitre 13, Mentions légales et licences pour plus d'informations. Cette variable est facultative. S'il n'est pas défini, un avertissement sera généré pour informer et non enregistré apparaîtra dans le champ des fichiers de licence du fichier manifeste de ce package.

LIBFOO_ACTUAL_SOURCE_TARBALL	ne s'applique qu'aux packages dont le couple **LIBFOO_SITE / LIBFOO_SOURCE** pointe vers une archive qui ne contient pas réellement de code source, mais du code binaire. Il s'agit d'un cas très rare, connu uniquement pour s'appliquer aux chaînes d'outils externes déjà compilées, bien qu'en théorie, il puisse s'appliquer à d'autres packages. Dans de tels cas, une archive tar séparée est généralement disponible avec le code source réel. Définir **LIBFOO_ACTUAL_SOURCE_TARBALL** sur le nom de l'archive de code source réelle et Buildroot la téléchargera et l'utilisera lorsqu'on exécutere **make legal-info** pour collecter du matériel juridiquement pertinent. Ce fichier ne sera pas téléchargé lors des constructions régulières ni par make source.
LIBFOO_ACTUAL_SOURCE_SITE	fournit l'emplacement de l'archive source réelle. La valeur par défaut est **LIBFOO_SITE**, il n'est donc pas besoin de définir cette variable si les archives binaire et source sont hébergées sur le même répertoire. Si **LIBFOO_ACTUAL_SOURCE_TARBALL** n'est pas défini, cela n'a pas de sens de définir **LIBFOO_ACTUAL_SOURCE_SITE**.
LIBFOO_REDISTRIBUTE	peut être défini sur **YES** (par défaut) ou **NO** pour indiquer si le code source du package est autorisé à être redistribué. Définissez-le sur **NO** pour les packages non open source : Buildroot n'enregistrera pas le code source de ce package lors de la collecte des informations légales.
LIBFOO_FLAT_STACKSIZE	définit la taille de la pile d'une application intégrée au format binaire FLAT. La taille de la pile d'applications sur les processeurs d'architecture NOMMU ne peut pas être agrandie au moment de l'exécution. La taille de pile par défaut pour le format binaire FLAT n'est que de 4 ko. Si l'application consomme plus de pile, ajouter le nombre requis ici.
LIBFOO_BIN_ARCH_EXCLUDE	est une liste de chemins séparés par des espaces (relatifs au répertoire cible) à ignorer lors de la vérification que le paquet installe correctement les binaires compilés de manière croisée. On a rarement besoin de définir cette variable, sauf si le paquet installe des blobs binaires en dehors des emplacements par défaut, /lib/firmware, `/usr/lib/firmware`, `/lib/modules`, `/usr/lib/modules` et `/usr/share`, qui sont automatiquement exclus.

LIBFOO_IGNORE_CVES	est une liste de CVE séparés par des espaces qui indique aux outils de suivi Buildroot CVE quels CVE doivent être ignorés pour ce package. Ceci est généralement utilisé lorsque le CVE est corrigé par un correctif dans le package, ou lorsque le CVE, pour une raison quelconque, n'affecte pas le package Buildroot. Un commentaire Makefile doit toujours précéder l'ajout d'un CVE à cette variable. Exemple:
	`# 0001-fix-cve-2020-12345.patch`
	`LIBFOO_IGNORE_CVES += CVE-2020-12345`
	`# only when built with libbaz, which Buildroot doesn't support`
	`LIBFOO_IGNORE_CVES += CVE-2020-54321`

Les variables **LIBFOO_CPE_ID_** sont un ensemble de variables permettant au package de définir son identifiant CPE. Les variables disponibles sont :

LIBFOO_CPE_ID_PREFIX	spécifie le préfixe de l'identifiant CPE, c'est-à-dire les trois premiers champs. Lorsqu'elle n'est pas définie, la valeur par défaut est cpe:2.3:a.
LIBFOO_CPE_ID_VENDOR	spécifie la partie fournisseur de l'identifiant CPE. Lorsqu'elle n'est pas définie, la valeur par défaut est <pkgname>_project.

LIBFOO_CPE_ID_PRODUCT	spécifie la partie produit de l'identifiant CPE. Lorsqu'elle n'est pas définie, la valeur par défaut est <pkgname>.
LIBFOO_CPE_ID_VERSION	spécifie la partie version de l'identifiant CPE. Lorsqu'elle n'est pas définie, la valeur par défaut est **$(LIBFOO_VERSION)**.
LIBFOO_CPE_ID_UPDATE	spécifie la partie mise à jour de l'identifiant CPE. Lorsqu'elle n'est pas définie, la valeur par défaut est *.

- Si l'une de ces variables est définie, l'infrastructure de package générique suppose que le package fournit des informations CPE valides. Dans ce cas, l'infrastructure de package générique définira **LIBFOO_CPE_ID**.
- Pour un package hôte, si ses variables **LIBFOO_CPE_ID_*** ne sont pas définies, il hérite de la valeur de ces variables du package cible correspondant.

La méthode recommandée pour définir ces variables consiste à utiliser la syntaxe suivante :

```
LIBFOO_VERSION = 2.32
```

Maintenant, les variables qui définissent ce qui devrait être performé aux différentes étapes du processus de construction.

LIBFOO_EXTRACT_CMDS	liste les actions à effectuer pour extraire le package. Ce n'est généralement pas nécessaire car les archives tar sont automatiquement gérées par Buildroot. Cependant, si le package utilise un format d'archive non standard, tel qu'un fichier ZIP ou RAR, ou a une archive tar avec une organisation non standard, cette variable permet de remplacer le comportement par défaut de l'infrastructure du package.
LIBFOO_CONFIGURE_CMDS	liste les actions à effectuer pour configurer le package avant sa compilation.
LIBFOO_BUILD_CMDS	liste les actions à effectuer pour compiler le package.
HOST_LIBFOO_INSTALL_CMDS	répertorie les actions à effectuer pour installer le package, lorsque le package est un package hôte. Le paquet doit installer ses fichiers dans le répertoire donné par **$(HOST_DIR)**. Tous les fichiers, y compris les fichiers de développement tels que les en-têtes, doivent être installés, car d'autres packages peuvent être compilés par-dessus ce package.
LIBFOO_INSTALL_TARGET_CMDS	répertorie les actions à effectuer pour installer le package dans le répertoire cible, lorsque le package est un package cible. Le package doit installer ses fichiers dans le répertoire indiqué par **$(TARGET_DIR)**. Seuls les fichiers nécessaires à l'exécution du package doivent être installés. Les fichiers d'en-tête, les bibliothèques statiques et la documentation seront à nouveau supprimés lorsque le système de fichiers cible sera finalisé.
LIBFOO_INSTALL_STAGING_CMDS	répertorie les actions à effectuer pour installer le package dans le répertoire intermédiaire, lorsque le package est un package cible. Le package doit installer ses fichiers dans le répertoire indiqué par **$(STAGING_DIR)**. Tous les fichiers de développement doivent être installés, car ils peuvent être nécessaires pour compiler d'autres packages.

LIBFOO_INSTALL_IMAGES_CMDS	répertorie les actions à effectuer pour installer le package dans le répertoire des images, lorsque le package est un package cible. Le paquet doit installer ses fichiers dans le répertoire donné par **$(BINARIES_DIR)**. Seuls les fichiers qui sont des images binaires (c'est-à-dire des images) qui n'appartiennent pas au **TARGET_DIR** mais qui sont nécessaires au démarrage de la carte doivent être placés ici. Par exemple, un package doit utiliser cette étape s'il contient des fichiers binaires similaires à l'image du noyau, au chargeur de démarrage ou aux images du système de fichiers racine.
LIBFOO_INSTALL_INIT_SYSV, LIBFOO_INSTALL_INIT_OPENRC LIBFOO_INSTALL_INIT_SYSTEMD	listent les actions pour installer les scripts init soit pour les systèmes init de type systemV (busybox, sysvinit, etc.), openrc ou pour les unités systemd. Ces commandes seront exécutées uniquement lorsque le système d'initialisation approprié est installé (c'est-à-dire que si systemd est sélectionné comme système d'initialisation dans la configuration, seul **LIBFOO_INSTALL_INIT_SYSTEMD** sera exécuté). La seule exception est lorsque openrc est choisi comme système d'initialisation et que **LIBFOO_INSTALL_INIT_OPENRC** n'a pas été défini, dans une telle situation, **LIBFOO_INSTALL_INIT_SYSV** sera appelé, car openrc prend en charge les scripts d'initialisation sysv. Lorsque systemd est utilisé comme système d'initialisation, buildroot activera automatiquement tous les services à l'aide de la commande systemctl preset-all dans la phase finale de la création d'image. on peut ajouter des fichiers prédéfinis pour empêcher qu'une unité particulière soit automatiquement activée par buildroot.
LIBFOO_HELP_CMDS	répertorie les actions pour imprimer l'aide du package, qui est incluse dans la sortie principale de make help. Ces commandes peuvent imprimer n'importe quoi dans n'importe quel format. Ceci est rarement utilisé, car les packages ont rarement des règles personnalisées. Ne pas utiliser cette variable, sauf si on sait vraiment qu'on a besoin d'imprimer de l'aide.
LIBFOO_LINUX_CONFIG_FIXUPS	répertorie les options de configuration du noyau Linux qui sont nécessaires pour construire et utiliser ce paquet, et sans lesquelles le paquet est fondamentalement cassé. Il s'agit d'un ensemble d'appels à l'une des options de réglage de kconfig : **KCONFIG_ENABLE_OPT**, **KCONFIG_DISABLE_OPT** ou **KCONFIG_SET_OPT**. Ceci est rarement utilisé, car le package n'a généralement pas d'exigences strictes sur les options du noyau.

La meilleure façon de définir ces variables est :

```
define LIBFOO_CONFIGURE_CMDS
        action 1
        action 2
        action 3
endef
```

Dans les définitions d'action, on peut utiliser les variables suivantes :

- **$(LIBFOO_PKGDIR)** contient le chemin d'accès au répertoire contenant les fichiers libfoo.mk et

Config.in. Cette variable est utile lorsqu'il est nécessaire d'installer un fichier fourni dans Buildroot, comme un fichier de configuration d'exécution, une image de splashscreen...
- **$(@D)**, qui contient le répertoire dans lequel le code source du package a été décompressé.
- **$(LIBFOO_DL_DIR)** contient le chemin d'accès au répertoire dans lequel tous les téléchargements effectués par Buildroot pour libfoo sont stockés.
- **$(TARGET_CC)**, **$(TARGET_LD)**, etc. pour obtenir les utilitaires de compilation croisée cibles
- **$(TARGET_CROSS)** pour obtenir le préfixe de la chaîne d'outils de compilation croisée
- Bien sûr les variables **$(HOST_DIR)**, **$(STAGING_DIR)** et **$(TARGET_DIR)** pour installer correctement les packages. Ces variables pointent vers les répertoires global host, staging et target, sauf si la prise en charge des répertoires par package est utilisée, auquel cas elles pointent vers les répertoires hôte, staging et target du package actuel. Dans les deux cas, cela ne fait aucune différence de tdu point de vue du package : il doit simplement utiliser **HOST_DIR**, **STAGING_DIR** et **TARGET_DIR**.

Enfin, on peut également utiliser des hooks. Voir Section « hooks disponibles dans les différentes étapes de construction » pour plus d'informations.

Infrastructure pour les packages basés sur les outils automatiques

tutoriel autotools-package

Voyons d'abord comment écrire un fichier .mk pour un package basé sur autotools, avec un exemple :

```
01:
################################################################################
####
02: #
03: # libfoo
04: #
05:
################################################################################
####
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_INSTALL_STAGING = YES
11: LIBFOO_INSTALL_TARGET = NO
12: LIBFOO_CONF_OPTS = --disable-shared
13: LIBFOO_DEPENDENCIES = libglib2 host-pkgconf
14:
15: $(eval $(autotools-package))
```

- A la ligne 7, on déclare la version du package.
- Aux lignes 8 et 9, on déclare le nom du tarball (xz-ed tarball recommandé) et l'emplacement du tarball sur le Web. Buildroot téléchargera automatiquement l'archive tar depuis cet

emplacement.
- À la ligne 10, on dit à Buildroot d'installer le package dans le répertoire intermédiaire. Le répertoire staging, situé dans output/staging/ est le répertoire où tous les packages sont installés, y compris leurs fichiers de développement, etc. Par défaut, les packages ne sont pas installés dans le répertoire staging, car généralement, seules les bibliothèques doivent être installées dans le répertoire intermédiaire : leurs fichiers de développement sont nécessaires pour compiler d'autres bibliothèques ou applications qui en dépendent. De plus, par défaut, lorsque l'installation intermédiaire est activée, les packages sont installés à cet emplacement à l'aide de la commande make install.
- À la ligne 11, on dit à Buildroot de ne pas installer le package dans le répertoire cible. Ce répertoire contient ce qui deviendra le système de fichiers racine exécuté sur la cible. Pour les bibliothèques purement statiques, il n'est pas nécessaire de les installer dans le répertoire cible car elles ne seront pas utilisées à l'exécution. Par défaut, l'installation cible est activée ; définir cette variable sur NO n'est presque jamais nécessaire. Par défaut également, les packages sont installés à cet emplacement à l'aide de la commande make install.
- À la ligne 12, on dit à Buildroot de transmettre une option de configuration personnalisée, qui sera transmise au script ./configure avant de configurer et de construire le package.
- À la ligne 13, on déclare les dépendances, afin qu'elles soient construites avant le démarrage du processus de construction de notre paquet.
- A la ligne ligne 15, on invoque la macro autotools-package qui génère toutes les règles Makefile qui permettent réellement de construire le package.

référence du package autotools

La macro principale de l'infrastructure du package autotools est autotools-package. Elle est similaire à la macro de package générique. La possibilité d'avoir des packages cible et hôte est également disponible, avec la macro host-autotools-package.

Tout comme l'infrastructure générique, l'infrastructure autotools fonctionne en définissant un certain nombre de variables avant d'appeler la macro autotools-package.

Tout d'abord, toutes les variables d'informations de métadonnées de package qui existent dans l'infrastructure générique existent également dans l'infrastructure des autotools : **LIBFOO_VERSION**, **LIBFOO_SOURCE**, **LIBFOO_PATCH**, **LIBFOO_SITE**, **LIBFOO_SUBDIR**, **LIBFOO_DEPENDENCIES**, **LIBFOO_INSTALL_STAGING**, **LIBFOO_INSTALL_TARGET**.

Quelques variables supplémentaires, spécifiques à l'infrastructure des autotools, peuvent également être définies. Beaucoup d'entre eux ne sont utiles que dans des cas très spécifiques, les packages typiques n'en utiliseront donc que quelques-uns.

LIBFOO_SUBDIR	peut contenir le nom d'un sous-répertoire à l'intérieur du package qui contient le script de configuration. Ceci est utile si, par exemple, le script de configuration principal n'est pas à la racine de l'arborescence extraite par l'archive. Si **HOST_LIBFOO_SUBDIR** n'est pas spécifié, la valeur par défaut est **LIBFOO_SUBDIR**.
LIBFOO_CONF_ENV	pour spécifier des variables d'environnement supplémentaires à passer au script de configuration. Par défaut, vide.
LIBFOO_CONF_OPTS	pour spécifier des options de configuration supplémentaires à passer au script de configuration. Par défaut, vide.

LIBFOO_MAKE	pour spécifier une autre commande make. Ceci est généralement utile lorsque la création parallèle est activée dans la configuration (à l'aide de **BR2_JLEVEL**) mais que cette fonctionnalité doit être désactivée pour le package donné, pour une raison ou une autre. Par défaut, défini sur $(MAKE). Si la construction parallèle n'est pas prise en charge par le package, il doit être défini sur **LIBFOO_MAKE=$(MAKE1)**.
LIBFOO_MAKE_ENV	pour spécifier des variables d'environnement supplémentaires à passer à make dans l'étape de construction. Ceux-ci sont passés avant la commande make. Par défaut, vide.
LIBFOO_MAKE_OPTS	pour spécifier des variables supplémentaires à passer à make dans l'étape de construction. Ceux-ci sont passés après la commande make. Par défaut, vide.
LIBFOO_AUTORECONF	indique si le paquet doit être reconfiguré automatiquement ou non (c'est-à-dire si le script de configuration et les fichiers Makefile.in doivent être regénérésed en réexécutant autoconf, automake, libtool, etc.). Les valeurs valides sont **YES** et **NO**. Par défaut, la valeur est NON
LIBFOO_AUTORECONF_ENV	pour spécifier des variables d'environnement supplémentaires à passer au programme autoreconf si **LIBFOO_AUTORECONF=YES**. Ceux-ci sont passés dans l'environnement de la commande autoreconf. Par défaut, vide.
LIBFOO_AUTORECONF_OPTS	pour spécifier les options supplémentaires passées au programme autoreconf si **LIBFOO_AUTORECONF=YES**. Par défaut, vide.
LIBFOO_GETTEXTIZE	indique si le package doit être gettextisé ou non (c'est-à-dire si le package utilise une version de gettext différente de celle fournie par Buildroot et qu'elle est nécessaire pour exécuter gettextize.) Uniquement valide lorsque LIBFOO_AUTORECONF=YES. Les valeurs valides sont **YES** et NON. La valeur par défaut est **NO**.
LIBFOO_GETTEXTIZE_OPTS	pour spécifier les options supplémentaires passées au programme gettextize, si **LIBFOO_GETTEXTIZE=YES**. on peut l'utiliser si, par exemple, les fichiers .po ne se trouvent pas à l'emplacement standard (c'est-à-dire dans po/ à la racine du paquet). Par défaut, -f.
LIBFOO_LIBTOOL_PATCH	indique si le correctif Buildroot pour résoudre les problèmes de compilation croisée de libtool doit être appliqué ou non. Les valeurs valides sont **YES** et **NO**. Par défaut, la valeur est **YES**
LIBFOO_INSTALL_STAGING_OPTS	contient les options make utilisées pour installer le package dans le répertoire intermédiaire. Par défaut, la valeur est **DESTDIR=$(STAGING_DIR)** install, ce qui est correct pour la plupart des packages d'autotools. Il est encore possible de le contourner.
LIBFOO_INSTALL_TARGET_OPTS	contient les options make utilisées pour installer le package dans le répertoire cible. Par défaut, la valeur est **DESTDIR=$(TARGET_DIR)** install. La valeur par défaut est correcte pour la plupart des packages d'autotools, mais il est toujours possible de la remplacer si nécessaire.

Avec l'infrastructure autotools, toutes les étapes requises pour créer et installer les packages sont déjà définies, et elles fonctionnent généralement bien pour la plupart des packages basés sur autotools. Cependant, si nécessaire, il est toujours possible de personnaliser ce qui est fait à une étape particulière :

- En ajoutant un hook post-opération (après extraction, patch, configuration, compilation ou installation). Voir Section « hooks disponibles dans les différentes étapes de construction » pour plus de détails.
- En remplaçant l'une des étapes. Par exemple, même si l'infrastructure autotools est utilisée, si le fichier .mk du package définit sa propre variable **LIBFOO_CONFIGURE_CMDS**, elle sera utilisée à la place de celle par défaut des autotools. Cependant, l'utilisation de cette méthode doit être limitée à des cas très spécifiques. Ne pas l'utiliser dans le cas général.

Infrastructure pour les packages basés sur CMake

tutoriel cmake-package

Voyons d'abord comment écrire un fichier .mk pour un package basé sur CMake, avec un exemple :

```
01: ################################################################################
02: #
03: # libfoo
04: #
05: ################################################################################
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_INSTALL_STAGING = YES
11: LIBFOO_INSTALL_TARGET = NO
12: LIBFOO_CONF_OPTS = -DBUILD_DEMOS=ON
13: LIBFOO_DEPENDENCIES = libglib2 host-pkgconf
14:
15: $(eval $(cmake-package))
```

- A la ligne 7, on déclare la version du package.
- Aux lignes 8 et 9, on déclare le nom du tarball (xz-ed tarball recommandé) et l'emplacement du tarball sur le Web. Buildroot téléchargera automatiquement l'archive tar depuis cet emplacement.
- À la ligne 10, on dit à Buildroot d'installer le package dans le répertoire intermédiaire. Le répertoire staging, situé dans `output/staging/` est le répertoire où tous les packages sont installés, y compris leurs fichiers de développement, etc. Par défaut, les packages ne sont pas installés dans le répertoire staging, car généralement, seules les bibliothèques doivent être installées dans le répertoire intermédiaire : leurs fichiers de développement sont nécessaires pour compiler d'autres bibliothèques ou applications qui en dépendent. De plus, par défaut, lorsque l'installation intermédiaire est activée, les packages sont installés à cet emplacement à l'aide de la commande make install.
- À la ligne 11, on dit à Buildroot de ne pas installer le package dans le répertoire cible. Ce répertoire contient ce qui deviendra le système de fichiers racine exécuté sur la cible. Pour les

bibliothèques purement statiques, il n'est pas nécessaire de les installer dans le répertoire cible car elles ne seront pas utilisées à l'exécution. Par défaut, l'installation cible est activée ; définir cette variable sur NO n'est presque jamais nécessaire. Par défaut également, les packages sont installés à cet emplacement à l'aide de la commande make install.
- À la ligne 12, on dit à Buildroot de transmettre des options personnalisées à CMake lors de la configuration du package.
- À la ligne 13, on déclare les dépendances, afin qu'elles soient construites avant le démarrage du processus de construction de notre paquet.
- A la ligne ligne 15, on invoque la macro **cmake-package** qui génère toutes les règles Makefile qui permettent réellement de construire le package.

cmake-package référence

La macro principale de l'infrastructure de package **CMake** est **cmake-package**. Elle est similaire à la macro de package générique. La capacité à avoir des packages cible et hôte est également disponible, avec la macro host-cmake-package.

Tout comme l'infrastructure générique, l'infrastructure **CMake** fonctionne en définissant un certain nombre de variables avant d'appeler la macro **cmake-package**.

Tout d'abord, toutes les variables d'informations de métadonnées de package qui existent dans l'infrastructure générique existent également dans l'infrastructure CMake : **LIBFOO_VERSION**, **LIBFOO_SOURCE**, **LIBFOO_PATCH**, **LIBFOO_SITE**, **LIBFOO_SUBDIR**, **LIBFOO_DEPENDENCIES**, **LIBFOO_INSTALL_STAGING**, **LIBFOO_INSTALL_TARGET**.

Quelques variables supplémentaires, spécifiques à l'infrastructure **CMake**, peuvent également être définies. Beaucoup d'entre eux ne sont utiles que dans des cas très spécifiques, les packages typiques n'en utiliseront donc que quelques-uns.

LIBFOO_SUBDIR	peut contenir le nom d'un sous-répertoire à l'intérieur du package qui contient le fichier principal CMakeLists.txt. Ceci est utile, si par exemple, le fichier principal CMakeLists.txt n'est pas à la racine de l'arborescence extraite par le tarball. Si **HOST_LIBFOO_SUBDIR** n'est pas spécifié, la valeur par défaut est **LIBFOO_SUBDIR**.
LIBFOO_CONF_ENV	pour spécifier des variables d'environnement supplémentaires à passer à CMake. Par défaut, vide.

LIBFOO_CONF_OPTS	pour spécifier des options de configuration supplémentaires à transmettre à CMake. Par défaut, vide. Un certain nombre d'options courantes de CMake sont définies par l'infrastructure cmake-package ; il n'est donc normalement pas nécessaire de les définir dans le fichier *.mk du package, sauf si on souhaite les remplacer : * **CMAKE_BUILD_TYPE** est piloté par **BR2_ENABLE_RUNTIME_DEBUG** ; * **CMAKE_INSTALL_PREFIX** ; * **BUILD_SHARED_LIBS** est piloté par **BR2_STATIC_LIBS** ; * **BUILD_DOC**, **BUILD_DOCS** sont désactivés ; * **BUILD_EXAMPLE**, **BUILD_EXAMPLES** sont désactivés ; * **BUILD_TEST**, **BUILD_TESTS**, **BUILD_TESTING** sont désactivés.
LIBFOO_SUPPORTS_IN_SOURCE_BUILD = NO	doit être défini lorsque le paquet ne peut pas être construit à l'intérieur de l'arborescence source mais a besoin d'un répertoire de construction séparé.
LIBFOO_MAKE	pour spécifier une autre commande make. Ceci est généralement utile lorsque la création parallèle est activée dans la configuration (à l'aide de BR2_JLEVEL) mais que cette fonctionnalité doit être désactivée pour le package donné, pour une raison ou une autre. Par défaut, défini sur $(MAKE). Si la construction parallèle n'est pas prise en charge par le package, il doit être défini sur LIBFOO_MAKE=$(MAKE1).
LIBFOO_MAKE_ENV	pour spécifier des variables d'environnement supplémentaires à passer à make dans l'étape de construction. Ceux-ci sont passés avant la commande make. Par défaut, vide.
LIBFOO_MAKE_OPTS	pour spécifier des variables supplémentaires à passer à make dans l'étape de construction. Ceux-ci sont passés après la commande make. Par défaut, vide.
LIBFOO_INSTALL_OPTS	contient les options make utilisées pour installer le paquet dans le répertoire hôte. Par défaut, la valeur est install, ce qui est correct pour la plupart des packages CMake. Il est encore possible de le contourner.
LIBFOO_INSTALL_STAGING_OPTS	contient les options make utilisées pour installer le package dans le répertoire intermédiaire. Par défaut, la valeur est **DESTDIR=$(STAGING_DIR)** install/fast, ce qui est correct pour la plupart des packages CMake. Il est encore possible de le contourner.
LIBFOO_INSTALL_TARGET_OPTS	contient les options make utilisées pour installer le package dans le répertoire cible. Par défaut, la valeur est **DESTDIR=$(TARGET_DIR)** install/fast. La valeur par défaut est correcte pour la plupart des packages CMake, mais il est toujours possible de la remplacer si nécessaire.

Avec l'infrastructure CMake, toutes les étapes nécessaires pour créer et installer les packages sont déjà définies et fonctionnent généralement bien pour la plupart des packages basés sur CMake. Cependant, si nécessaire, il est toujours possible de personnaliser ce qui est fait à une étape

particulière :

- En ajoutant un hook post-opération (après extraction, patch, configuration, compilation ou installation). Voir Section « hooks disponibles dans les différentes étapes de construction » pour plus de détails.
- En remplaçant l'une des étapes. Par exemple, même si l'infrastructure CMake est utilisée, si le fichier .mk du package définit sa propre variable **LIBFOO_CONFIGURE_CMDS**, elle sera utilisée à la place de celle par défaut de CMake. Cependant, l'utilisation de cette méthode doit être limitée à des cas très spécifiques. Ne pas l'utiliser dans le cas général.

Infrastructure pour les packages Python

Cette infrastructure s'applique aux packages Python qui utilisent le mécanisme standard Python setuptools comme système de construction, généralement reconnaissable à l'utilisation d'un script setup.py.

tutoriel sur le package python

Voyons d'abord comment écrire un fichier .mk pour un package Python, avec un exemple :

```
01:
################################################################################
####
02: #
03: # python-foo
04: #
05:
################################################################################
####
06:
07: PYTHON_FOO_VERSION = 1.0
08: PYTHON_FOO_SOURCE = python-foo-$(PYTHON_FOO_VERSION).tar.xz
09: PYTHON_FOO_SITE = http://www.foosoftware.org/download
10: PYTHON_FOO_LICENSE = BSD-3-Clause
11: PYTHON_FOO_LICENSE_FILES = LICENSE
12: PYTHON_FOO_ENV = SOME_VAR=1
13: PYTHON_FOO_DEPENDENCIES = libmad
14: PYTHON_FOO_SETUP_TYPE = distutils
15:
16: $(eval $(python-package))
```

- A la ligne 7, buildroot déclarer la version du package.
- Sur la ligne 8 et 9, on déclare le nom du tarball (xz-ed tarball recommandé) et l'emplacement du tarball sur le Web. Buildroot téléchargera automatiquement l'archive tar depuis cet emplacement.
- Aux lignes 10 et 11, on donne des détails de licence sur le paquet (sa licence à la ligne 10 et le fichier contenant le texte de la licence à la ligne 11).

- À la ligne 12, on dit à Buildroot de transmettre des options personnalisées au script Python setup.py lors de la configuration du package.
- À la ligne 13, on déclare les dépendances, afin qu'elles soient construites avant le démarrage du processus de construction de notre paquet.
- À la ligne 14, on déclare le système de construction Python spécifique utilisé. Dans ce cas, le système de construction Python distutils est utilisé. Les deux pris en charge sont distutils et setuptools.
- A la ligne 16, on invoque la macro python-package qui génère toutes les règles Makefile qui permettent réellement de construire le package.

référence du package python

En règle générale, les packages qui fournissent simplement des modules Python doivent tous être nommés python-<something> dans Buildroot. D'autres packages qui utilisent le système de construction Python, mais ne sont pas des modules Python, peuvent choisir librement leur nom (les exemples existants dans Buildroot sont scons et superviseur).

Les packages qui ne sont compatibles qu'avec une seule version de Python (comme dans : Python 2 ou Python 3) doivent dépendre explicitement de cette version dans leur fichier Config.in (**BR2_PACKAGE_PYTHON** pour Python 2, **BR2_PACKAGE_PYTHON3** pour Python 3). Les packages compatibles avec les deux versions ne doivent pas en dépendre explicitement dans leur fichier Config.in, puisque cette condition est déjà exprimée pour tout le menu “External python modules”.

La macro principale de l'infrastructure de packages Python est python-package. Elle est similaire à la macro de package générique. Il est également possible de créer des packages hôtes Python avec la macro host-python-package.

Tout comme l'infrastructure générique, l'infrastructure Python fonctionne en définissant un certain nombre de variables avant d'appeler les macros python-package ou host-python-package.

Toutes les variables d'informations de métadonnées de package qui existent dans l'infrastructure de package générique existent également dans l'infrastructure Python : **PYTHON_FOO_VERSION**, **PYTHON_FOO_SOURCE**, **PYTHON_FOO_PATCH**, **PYTHON_FOO_SITE**, **PYTHON_FOO_SUBDIR**, **PYTHON_FOO_DEPENDENCIES**, **PYTHON_FOO_LICENSE**, **PYTHON_FOO_LICENSE_FILES**, **PYTHON_FOO_INSTALL**, etc.

- Il n'est pas nécessaire d'ajouter python ou host-python dans la variable **PYTHON_FOO_DEPENDENCIES** d'un package, puisque ces dépendances de base sont automatiquement ajoutées selon les besoins par l'infrastructure du package Python. De même, il n'est pas nécessaire d'ajouter host-setuptools à **PYTHON_FOO_DEPENDENCIES** pour les packages basés sur setuptools, car il est automatiquement ajouté par l'infrastructure Python selon les besoins.

Une variable spécifique à l'infrastructure Python est obligatoire :

PYTHON_FOO_SETUP_TYPE	pour définir quel système de construction Python est utilisé par le package. Les deux valeurs prises en charge sont distutils et setuptools. Si on ne sait pas lequel est utilisé dans le package, regarder le fichier setup.py dans le code source du package et voir s'il importe des éléments du module distutils ou du module setuptools.

Quelques variables supplémentaires, spécifiques à l'infrastructure Python, peuvent éventuellement être définies, en fonction des besoins du package. Beaucoup d'entre eux ne sont utiles que dans des cas très spécifiques, les packages typiques n'en utiliseront donc que quelques-uns, voire aucun.

PYTHON_FOO_SUBDIR	peut contenir le nom d'un sous-répertoire à l'intérieur du package qui contient le fichier setup.py principal. Ceci est utile, si par exemple, le fichier setup.py principal n'est pas à la racine de l'arborescence extraite par l'archive. Si **HOST_PYTHON_FOO_SUBDIR** n'est pas spécifié, la valeur par défaut est **PYTHON_FOO_SUBDIR**.
PYTHON_FOO_ENV	pour spécifier des variables d'environnement supplémentaires à transmettre au script Python setup.py (pour les étapes de construction et d'installation). L'infrastructure transmet automatiquement plusieurs variables standard, définies dans: * **PKG_PYTHON_DISTUTILS_ENV** (pour les packages cibles distutils), * **HOST_PKG_PYTHON_DISTUTILS_ENV**, (pour les packages hôtes distutils), * **PKG_PYTHON_SETUPTOOLS_ENV** (pour les packages cibles setuptools), * **HOST_PKG_PYTHON_SETUPTOOLS_ENV** (pour les packages hôtes setuptools).
PYTHON_FOO_BUILD_OPTS	pour spécifier des options supplémentaires à transmettre au script Python setup.py lors de l'étape de construction. Pour les packages distutils cibles, les options **PKG_PYTHON_DISTUTILS_BUILD_OPTS** sont déjà transmises automatiquement par l'infrastructure.
PYTHON_FOO_INSTALL_TARGET_OPTS **PYTHON_FOO_INSTALL_STAGING_OPTS** **HOST_PYTHON_FOO_INSTALL_OPTS**	pour spécifier des options supplémentaires à transmettre au script Python setup.py lors de l'étape d'installation cible, de l'étape d'installation intermédiaire ou de l'installation hôte, respectivement. L'infrastructure passe automatiquement certaines options, définies dans: * **PKG_PYTHON_DISTUTILS_INSTALL_TARGET_OPTS** * **PKG_PYTHON_DISTUTILS_INSTALL_STAGING_OPTS** (pour les packages distutils cibles), * **HOST_PKG_PYTHON_DISTUTILS_INSTALL_OPTS** (pour les packages distutils hôtes), * **PKG_PYTHON_SETUPTOOLS_INSTALL_TARGET_OPTS** * **PKG_PYTHON_SETUPTOOLS_INSTALL_STAGING_OPTS** (pour les packages setuptools cibles) * **HOST_PKG_PYTHON_SETUPTOOLS_INSTALL_OPTS** (pour les packages setuptools hôte).
HOST_PYTHON_FOO_NEEDS_HOST_PYTHON	pour définir l'interpréteur python hôte. L'utilisation de cette variable est limitée aux packages hôtes. Les deux valeurs prises en charge sont python2 et python3. Il s'assurera que le bon package python hôte est disponible et l'invoquera pour la construction. Si certaines étapes de construction sont surchargées, le bon interpréteur Python doit être explicitement appelé dans les commandes.

Avec l'infrastructure Python, toutes les étapes requises pour créer et installer les packages sont déjà définies et fonctionnent généralement bien pour la plupart des packages basés sur Python. Cependant, si nécessaire, il est toujours possible de personnaliser ce qui est fait à une étape particulière :

- En ajoutant un hook post-opération (après extraction, patch, configuration, compilation ou installation). Voir Section « hooks disponibles dans les différentes étapes de construction » pour plus de détails.
- En remplaçant l'une des étapes. Par exemple, même si l'infrastructure Python est utilisée, si le fichier .mk du package définit sa propre variable **PYTHON_FOO_BUILD_CMDS**, elle sera utilisée à la place de celle Python par défaut. Cependant, l'utilisation de cette méthode doit être limitée à des cas très spécifiques. Ne pas l'utiliser dans le cas général.

Génération d'un package python à partir d'un référentiel PyPI

Si le package Python pour lequel on souhaite créer un package Buildroot est disponible sur PyPI, on peut utiliser l'outil scanpypi situé dans utils/ pour automatiser le processus.

on peut trouver la liste des packages PyPI existants ici.

scanpypi nécessite que le package setuptools de Python soit installé sur l'hôte.

Lorsqu'on est à la racine du répertoire buildroot, faire simplement :

```
utils/scanpypi foo bar -o paquet
```

Cela générera les packages python-foo et python-bar dans le dossier du package s'ils existent sur https://pypi.python.org.

Trouver le menu des modules python externes et insérer le package à l'intérieur (les éléments d'un menu doivent être classés par ordre alphabétique).

Il faudra très probablement vérifier manuellement le paquet pour toute erreur car il y a des choses qui ne peuvent pas être devinées par le générateur (par exemple, des dépendances sur l'un des modules de base python tels que **BR2_PACKAGE_PYTHON_ZLIB**). La licence et les fichiers de licence sont devinés et doivent être vérifiés. Il faut également ajouter manuellement le package au fichier package/Config.in.

Si le package Buildroot n'est pas dans l'arborescence officielle Buildroot mais dans une arborescence externe br2, utiliser l'indicateur -o comme suit :

```
utils/scanpypi foo bar -o other_package_dir
```

Cela générera des packages python-foo et python-bar dans `other_package_directory` au lieu de package.

L'option -h listera les options disponibles :

```
utils/scanpypi -h
```

python-package CFFI backend

C Foreign **F**unction **I**nterface for Python (CFFI) fournit un moyen pratique et fiable d'appeler du code C compilé à partir de Python à l'aide de déclarations d'interface écrites en C. Les packages Python reposant sur ce backend peuvent être identifiés par l'apparition d'une dépendance cffi dans le champ install_requires de leur fichier setup.py.

Un tel paquet nécessite :

- ajout **python-cffi** en tant que dépendance d'exécution afin d'installer le wrapper de la bibliothèque C compilée sur la cible. Ceci est réalisé en ajoutant select **BR2_PACKAGE_PYTHON_CFFI** au package Config.in.

```
config BR2_PACKAGE_PYTHON_FOO
        bool "python-foo"
        select BR2_PACKAGE_PYTHON_CFFI # runtime
```

- ajout **host-python-cffi** en tant que dépendance au moment de la construction afin de cross-compiler le wrapper C. Ceci est réalisé en ajoutant host-python-cffi à la variable **PYTHON_FOO_DEPENDENCIES**.

```
################################################################################
####
#
# python-foo
#
################################################################################
####

...

PYTHON_FOO_DEPENDENCIES = host-python-cffi

$(eval $(python-package))
```

Infrastructure pour les packages basés sur LuaRocks

tutoriel sur le paquet luarocks

Voyons d'abord comment écrire un fichier .mk pour un package basé sur LuaRocks, avec un exemple :

```
01:
################################################################################
####
02: #
03: # lua-foo
04: #
05:
```

```
################################################################################
####
06:
07: LUA_FOO_VERSION = 1.0.2-1
08: LUA_FOO_NAME_UPSTREAM = foo
09: LUA_FOO_DEPENDENCIES = bar
10:
11: LUA_FOO_BUILD_OPTS += BAR_INCDIR=$(STAGING_DIR)/usr/include
12: LUA_FOO_BUILD_OPTS += BAR_LIBDIR=$(STAGING_DIR)/usr/lib
13: LUA_FOO_LICENSE = luaFoo license
14: LUA_FOO_LICENSE_FILES = $(LUA_FOO_SUBDIR)/COPYING
15:
16: $(eval $(luarocks-package))
```

- A la ligne 7, on déclare la version du paquet (la même que dans la rockspec, qui est la concaténation de la version amont et de la révision rockspec, séparées par un trait d'union -).
- A la ligne 8, on déclare que le paquet s'appelle “foo” sur LuaRocks. Dans Buildroot, buildroot donnons aux packages liés à Lua un nom qui commence par “lua”, de sorte que le nom Buildroot est différent du nom en amont. **LUA_FOO_NAME_UPSTREAM** fait le lien entre les deux noms.
- À la ligne 9, on déclare les dépendances vis-à-vis des bibliothèques natives, afin qu'elles soient construites avant le démarrage du processus de construction de notre paquet.
- Aux lignes 11-12, on dit à Buildroot de transmettre les options personnalisées à LuaRocks lors de la construction du package.
- Aux lignes 13 et 14, on spécifie les conditions de licence du package.
- A la ligne 16, on invoque la macro luarocks-package qui génère toutes les règles Makefile qui permettent réellement de construire le package.

La plupart de ces détails peuvent être récupérés à partir de la roche et de la roche. Ainsi, ce fichier et le fichier Config.in peuvent être générés en exécutant la commande luarocks buildroot foo lua-foo dans le répertoire Buildroot. Cette commande exécute un addon Buildroot spécifique de luarocks qui générera automatiquement un package Buildroot. Le résultat doit encore être inspecté manuellement et éventuellement modifié.

- Le fichier package/Config.in doit être mis à jour manuellement pour inclure les fichiers Config.in générés.

référence du package luarocks

LuaRocks est un système de déploiement et de gestion pour les modules Lua, et prend en charge divers build.type : builtin, make et cmake. Dans le contexte de Buildroot, l'infrastructure **luarocks-package** ne prend en charge que le mode intégré. Les packages **LuaRocks** qui utilisent les mécanismes de construction make ou cmake doivent plutôt être empaquetés à l'aide des infrastructures generic-package et cmake-package dans Buildroot, respectivement.

La macro principale de l'infrastructure de paquets **LuaRocks** est **luarocks-package** : comme generic-package, elle fonctionne en définissant un certain nombre de variables fournissant des informations de métadonnées sur le paquet, puis en appelant **luarocks-package**.

Tout comme l'infrastructure générique, l'infrastructure **LuaRocks** fonctionne en définissant un certain nombre de variables avant d'appeler la macro luarocks-package.

Tout d'abord, toutes les variables d'informations de métadonnées de package qui existent dans l'infrastructure générique existent également dans l'infrastructure LuaRocks : **LUA_FOO_VERSION**, **LUA_FOO_SOURCE**, **LUA_FOO_SITE**, **LUA_FOO_DEPENDENCIES**, **LUA_FOO_LICENSE**, **LUA_FOO_LICENSE_FILES**.

Deux d'entre eux sont peuplés par l'infrastructure LuaRocks (pour l'étape de téléchargement). Si le package n'est pas hébergé sur le miroir LuaRocks $(BR2_LUAROCKS_MIRROR), on peut les remplacer :

LUA_FOO_SITE	dont la valeur par défaut est **$(BR2_LUAROCKS_MIRROR)**
LUA_FOO_SOURCE	qui par défaut est **$(minuscule LUA_FOO_NAME_UPSTREAM)-$(LUA_FOO_VERSION).src.rock**

Quelques variables supplémentaires, spécifiques à l'infrastructure LuaRocks, sont également définies. Ils peuvent être annulés dans des cas spécifiques.

LUA_FOO_NAME_UPSTREAM	qui par défaut est **lua-foo**, c'est-à-dire le nom du package Buildroot
LUA_FOO_ROCKSPEC	dont la valeur par défaut est **$(minuscule LUA_FOO_NAME_UPSTREAM)-$(LUA_FOO_VERSION).rockspec**
LUA_FOO_SUBDIR	qui par défaut est **$(LUA_FOO_NAME_UPSTREAM)-$(LUA_FOO_VERSION_WITHOUT_ROCKSPEC_REVISION)**
LUA_FOO_BUILD_OPTS	contient des options de build supplémentaires pour l'appel de build luarocks.

Infrastructure pour les packages Perl/CPAN

tutoriel perl-package

Voyons d'abord comment écrire un fichier .mk pour un package Perl/CPAN, avec un exemple :

```
01:
################################################################################
####
02: #
03: # perl-foo-bar
04: #
05:
################################################################################
####
06:
07: PERL_FOO_BAR_VERSION = 0.02
08: PERL_FOO_BAR_SOURCE = Foo-Bar-$(PERL_FOO_BAR_VERSION).tar.gz
09: PERL_FOO_BAR_SITE = $(BR2_CPAN_MIRROR)/authors/id/M/MO/MONGER
10: PERL_FOO_BAR_DEPENDENCIES = perl-strictures
11: PERL_FOO_BAR_LICENSE = Artistic or GPL-1.0+
12: PERL_FOO_BAR_LICENSE_FILES = LICENSE
13: PERL_FOO_BAR_DISTNAME = Foo-Bar
14:
15: $(eval $(perl-package))
```

- A la ligne 7, on déclare la version du package.
- Aux lignes 8 et 9, on déclare le nom du tarball et l'emplacement du tarball sur un serveur CPAN.

Buildroot téléchargera automatiquement l'archive tar depuis cet emplacement.
- À la ligne 10, on déclare les dépendances, afin qu'elles soient construites avant le démarrage du processus de construction de notre paquet.
- Aux lignes 11 et 12, on donne des détails de licence sur le paquet (sa licence à la ligne 11 et le fichier contenant le texte de la licence à la ligne 12).
- A la ligne 13, le nom de la distribution tel que requis par le script utils/scancpan (afin de régénérer/mettre à jour ces fichiers de package).
- A la ligne 15, on invoque la macro perl-package qui génère toutes les règles Makefile qui permettent réellement de construire le paquet.

La plupart de ces données peuvent être récupérées sur https://metacpan.org/. Ainsi, ce fichier et le Config.in peuvent être générés en exécutant le script utils/scancpan Foo-Bar dans le répertoire Buildroot (ou dans une arborescence externe br2). Ce script crée un fichier Config.in et un fichier foo-bar.mk pour le package demandé, ainsi que de manière récursive pour toutes les dépendances spécifiées par CPAN. Il faut toujours modifier manuellement le résultat. En particulier, les éléments suivants doivent être vérifiés.

- Si le module perl est lié à une bibliothèque partagée fournie par un autre package (non-perl), cette dépendance n'est pas ajoutée automatiquement. Il a à ajouter manuellement à **PERL_FOO_BAR_DEPENDENCIES**.
- Le fichier package/Config.in doit être mis à jour manuellement pour inclure les fichiers Config.in générés. À titre indicatif, le script scancpan imprime les instructions source “…” requises, triées par ordre alphabétique.

référence du paquet perl

En règle générale, les packages qui fournissent des modules Perl/CPAN doivent tous être nommés perl-<quelque chose> dans Buildroot.

Cette infrastructure gère différents systèmes de build Perl : ExtUtils-MakeMaker (EUMM), Module-Build (MB) et Module-Build-Tiny. Build.PL est préféré par défaut lorsqu'un package fournit un Makefile.PL et un Build.PL.

La macro principale de l'infrastructure de paquets Perl/CPAN est perl-package. Elle est similaire à la macro de package générique. La possibilité d'avoir des packages cible et hôte est également disponible, avec la macro host-perl-package.

Tout comme l'infrastructure générique, l'infrastructure Perl/CPAN fonctionne en définissant un certain nombre de variables avant d'appeler la macro perl-package.

Tout d'abord, toutes les variables d'informations de métadonnées de package qui existent dans l'infrastructure générique existent également dans l'infrastructure Perl/CPAN : **PERL_FOO_VERSION**, **PERL_FOO_SOURCE**, **PERL_FOO_PATCH**, **PERL_FOO_SITE**, **PERL_FOO_SUBDIR**, **PERL_FOO_DEPENDENCIES**, **PERL_FOO_INSTALL_TARGET**.

définir **PERL_FOO_INSTALL_STAGING** sur YES n'a aucun effet à moins qu'une variable **PERL_FOO_INSTALL_STAGING_CMDS** ne soit définie. L'infrastructure Perl ne définit pas ces commandes puisque les modules Perl n'ont généralement pas besoin d'être installés dans le répertoire intermédiaire.

Quelques variables supplémentaires, spécifiques à l'infrastructure Perl/CPAN, peuvent également être définies. Beaucoup d'entre eux ne sont utiles que dans des cas très spécifiques, les packages typiques n'en utiliseront donc que quelques-uns.

PERL_FOO_PREFER_INSTALLER HOST_PERL_FOO_PREFER_INSTALLER	spécifie la méthode d'installation préférée. Les valeurs possibles sont EUMM (pour une installation basée sur Makefile.PL à l'aide d'ExtUtils-MakeMaker) et MB (pour une installation basée sur Build.PL à l'aide de Module-Build). Cette variable n'est utilisée que lorsque le package fournit les deux méthodes d'installation.
PERL_FOO_CONF_ENV HOST_PERL_FOO_CONF_ENV	pour spécifier des variables d'environnement supplémentaires à passer au perl Makefile.PL ou perl Build.PL. Par défaut, vide.
PERL_FOO_CONF_OPTS HOST_PERL_FOO_CONF_OPTS	pour spécifier des options de configuration supplémentaires à passer au perl Makefile.PL ou perl Build.PL. Par défaut, vide.
PERL_FOO_BUILD_OPTS HOST_PERL_FOO_BUILD_OPTS	pour spécifier des options supplémentaires à passer pour faire construire pure_all ou perl Build dans l'étape de construction. Par défaut, vide.
PERL_FOO_INSTALL_TARGET_OPTS	pour spécifier des options supplémentaires à transmettre pour effectuer l'installation de pure_install ou perl Build à l'étape d'installation. Par défaut, vide.
HOST_PERL_FOO_INSTALL_OPTS	

Infrastructure pour packages virtuels

Dans Buildroot, un package virtuel est un package dont les fonctionnalités sont fournies par un ou plusieurs packages, appelés fournisseurs. La gestion virtuelle des paquets est un mécanisme extensible permettant à l'utilisateur de choisir le fournisseur utilisé dans le rootfs.

Par exemple, OpenGL ES est une API pour les graphiques 2D et 3D sur les systèmes embarqués. L'implémentation de cette API est différente pour les plates-formes Allwinner Tech Sunxi et Texas Instruments OMAP35xx. Ainsi libgles sera un package virtuel et sunxi-mali et ti-gfx seront les fournisseurs.

didacticiel sur les packages virtuels

Dans l'exemple suivant, buildroot expliquerons comment ajouter un nouveau paquet virtuel (something-virtual) et un fournisseur pour celui-ci (some-provider).

Commençons par créer le package virtuel.

Fichier Config.in du package virtuel

Le fichier Config.in du package virtuel Something-Virtual doit contenir :

```
01: config BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
```

```
02:     bool
03:
04: config BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL
05:     depends on BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
06:     string
```

Dans ce fichier, on déclare deux options, **BR2_PACKAGE_HAS_SOMETHING_VIRTUAL** et **BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL**, dont les valeurs seront utilisées par les fournisseurs.

Fichier .mk du package virtuel

Le .mk du package virtuel doit juste évaluer la macro du package virtuel :

```
01:
################################################################################
####
02: #
03: # something-virtual
04: #
05:
################################################################################
####
06:
07: $(eval $(virtual-package))
```

La possibilité d'avoir des packages cible et hôte est également disponible, avec la macro host-virtual-package.

Fichier Config.in du fournisseur

Lors de l'ajout d'un package en tant que fournisseur, seul le fichier Config.in nécessite quelques modifications.

Le fichier Config.in du package some-provider, qui fournit les fonctionnalités de Something-Virtual, doit contenir :

```
01: config BR2_PACKAGE_SOME_PROVIDER
02:     bool "some-provider"
03:     select BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
04:     help
05:       This is a comment that explains what some-provider is.
06:
07:       http://foosoftware.org/some-provider/
08:
09: if BR2_PACKAGE_SOME_PROVIDER
10: config BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL
```

```
11:     default "some-provider"
12: endif
```

- À la ligne 3, buildroot sélectionnons **BR2_PACKAGE_HAS_SOMETHING_VIRTUAL**, et * à la ligne 11, buildroot définit la valeur de **BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL** sur le nom du fournisseur, mais uniquement s'il est sélectionné.

Fichier .mk du fournisseur

Le fichier .mk doit également déclarer une variable supplémentaire SOME_PROVIDER_PROVIDES pour contenir les noms de tous les packages virtuels dont il est une implémentation :

```
01: SOME_PROVIDER_PROVIDES = something-virtual
```

Bien sûr, il faut d'ajouter les dépendances de construction et d'exécution appropriées pour ce paquet !

Remarques sur la dépendance à un package virtuel

Lors de l'ajout d'un package qui nécessite une certaine FEATURE fournie par un package virtuel, utiliser depend on **BR2_PACKAGE_HAS_FEATURE**, comme ceci :

```
config BR2_PACKAGE_HAS_FEATURE
    bool

config BR2_PACKAGE_FOO
    bool "foo"
    depends on BR2_PACKAGE_HAS_FEATURE
```

Remarques sur la dépendance à un fournisseur spécifique

Si le paquet nécessite vraiment un fournisseur spécifique, alors il faut faire en sorte que le paquet dépende de ce fournisseur ; on ne peut pas sélectionner un fournisseur.

Prenons un exemple avec deux fournisseurs pour une FEATURE :

```
config BR2_PACKAGE_HAS_FEATURE
    bool

config BR2_PACKAGE_FOO
    bool "foo"
    select BR2_PACKAGE_HAS_FEATURE

config BR2_PACKAGE_BAR
    bool "bar"
```

```
    select BR2_PACKAGE_HAS_FEATURE
```

Et on ajoute un paquet qui a besoin de FEATURE comme fourni par foo, mais pas comme fourni par bar.

Si on devait utiliser select **BR2_PACKAGE_FOO**, l'utilisateur pourrait toujours sélectionner **BR2_PACKAGE_BAR** dans le menuconfig. Cela créerait une incohérence de configuration, dans laquelle deux fournisseurs de la même FEATURE seraient activés à la fois, l'un défini explicitement par l'utilisateur, l'autre implicitement par la sélection.

Au lieu de cela, utiliser `depends on BR2_PACKAGE_FOO`, ce qui évite toute incohérence de configuration implicite.

Infrastructure pour les packages utilisant kconfig pour les fichiers de configuration

Un moyen populaire pour un progiciel de gérer la configuration spécifiée par l'utilisateur est kconfig. Entre autres, il est utilisé par le noyau Linux, Busybox et Buildroot lui-même. La présence d'un fichier .config et d'une cible menuconfig sont deux symptômes bien connus de l'utilisation de kconfig.

Buildroot propose une infrastructure pour les packages qui utilisent kconfig pour leur configuration. Cette infrastructure fournit la logique nécessaire pour exposer la cible menuconfig du package en tant que foo-menuconfig dans Buildroot et pour gérer correctement la copie dans les deux sens du fichier de configuration.

L'infrastructure du package kconfig est basée sur l'infrastructure du package générique. Toutes les variables prises en charge par generic-package sont également disponibles dans kconfig-package. Voir Section « référence de paquetage générique » pour plus de détails.

Afin d'utiliser l'infrastructure kconfig-package pour un package Buildroot, les lignes minimales requises dans le fichier .mk, en plus des variables requises par l'infrastructure de package générique, sont :

```
FOO_KCONFIG_FILE = reference-to-source-configuration-file

$(eval $(kconfig-package))
```

Cet extrait crée les cibles make suivantes :

- **foo-menuconfig**, qui appelle la cible menuconfig du paquet
- **foo-update-config**, qui copie la configuration dans le fichier de configuration source. Il n'est pas possible d'utiliser cette cible lorsque des fichiers fragmentés sont définis.
- **foo-update-defconfig**, qui copie la configuration dans le fichier de configuration source. Le fichier de configuration répertorie uniquement les options qui diffèrent des valeurs par défaut. Il n'est pas possible d'utiliser cette cible lorsque des fichiers fragmentés sont définis.
- **foo-diff-config**, qui affiche les différences entre la configuration actuelle et celle définie dans la configuration Buildroot pour ce package kconfig. La sortie est utile pour identifier les changements de configuration qui peuvent devoir être propagés aux fragments de configuration par exemple.

et s'assure que le fichier de configuration source est copié dans le répertoire de construction au bon moment.

Il existe deux options pour spécifier un fichier de configuration à utiliser, soit **FOO_KCONFIG_FILE** (comme dans l'exemple ci-dessus) ou **FOO_KCONFIG_DEFCONFIG**. Il est obligatoire de fournir soit, mais pas les deux :

- **FOO_KCONFIG_FILE** spécifie le chemin d'accès à un fichier defconfig ou full-config à utiliser pour configurer le package.
- **FOO_KCONFIG_DEFCONFIG** spécifie la règle de defconfig make à appeler pour configurer le package.

En plus de ces lignes minimalement requises, plusieurs variables facultatives peuvent être définies pour répondre aux besoins du package considéré :

FOO_KCONFIG_EDITORS	une liste séparée par des espaces des éditeurs kconfig à prendre en charge, par exemple menuconfig **xconfig**. Par défaut, menuconfig.
FOO_KCONFIG_FRAGMENT_FILES	une liste séparée par des espaces de fichiers de fragments de configuration qui sont fusionnés avec le fichier de configuration principal. Les fichiers fragmentés sont généralement utilisés lorsque l'on souhaite rester synchronisé avec un fichier de configuration (def) en amont, avec quelques modifications mineures.
FOO_KCONFIG_OPTS	options supplémentaires à transmettre lors de l'appel des éditeurs **kconfig**. Cela peut nécessiter d'inclure **$(FOO_MAKE_OPTS**), par exemple. Par défaut, vide.
FOO_KCONFIG_FIXUP_CMDS	une liste de commandes shell nécessaires pour corriger le fichier de configuration après l'avoir copié ou exécuté un éditeur **kconfig**. De telles commandes peuvent être nécessaires pour assurer une configuration cohérente avec une autre configuration de Buildroot, par exemple. Par défaut, vide.
FOO_KCONFIG_DOTCONFIG	chemin (avec nom de fichier) du fichier `.config`, relatif à l'arborescence source du package. La valeur par défaut, .config, devrait être bien adaptée à tous les packages qui utilisent l'infrastructure **kconfig** standard héritée du noyau Linux ; certains packages utilisent un dérivé de kconfig qui utilise un emplacement différent.
FOO_KCONFIG_DEPENDENCIES	la liste des packages (probablement des packages hôtes) qui doivent être construits avant que le kconfig de ce package ne soit interprété. Rarement utilisé. Par défaut, vide.
FOO_KCONFIG_SUPPORTS_DEFCONFIG	si le système **kconfig** du paquet prend en charge l'utilisation des fichiers defconfig ; peu de paquets ne le font pas. Par défaut, **YES**.

Infrastructure pour les packages basés sur erlang

. didacticiel sur le package erlang

Voyons d'abord comment écrire un fichier .mk pour un package basé sur les erlang, avec un exemple :

```
01:
################################################################################
####
02: #
03: # erlang-foobar
04: #
05:
################################################################################
####
06:
07: ERLANG_FOOBAR_VERSION = 1.0
08: ERLANG_FOOBAR_SOURCE = erlang-foobar-$(ERLANG_FOOBAR_VERSION).tar.xz
09: ERLANG_FOOBAR_SITE = http://www.foosoftware.org/download
10: ERLANG_FOOBAR_DEPENDENCIES = host-libaaa libbbb
11:
12: $(eval $(rebar-package))
```

- A la ligne 7, on déclare la version du package.
- Aux lignes 8 et 9, on déclare le nom du tarball (xz-ed tarball recommandé) et l'emplacement du tarball sur le Web. Buildroot téléchargera automatiquement l'archive tar depuis cet emplacement.
- À la ligne 10, on déclare les dépendances, afin qu'elles soient construites avant le démarrage du processus de construction de notre paquet.
- A la ligne 12, on invoque la macro rebar-package qui génère toutes les règles Makefile qui permettent réellement de construire le package.

référence du package de erlang

La macro principale de l'infrastructure du package rebar est rebar-package. Elle est similaire à la macro de package générique. La possibilité d'avoir des packages hôtes est également disponible, avec la macro host-rebar-package.

Tout comme l'infrastructure générique, l'infrastructure rebar fonctionne en définissant un certain nombre de variables avant d'appeler la macro rebar-package.

Tout d'abord, toutes les variables d'informations sur les métadonnées du package qui existent dans l'infrastructure générique _ ENSEMBLE existent également dans l'infrastructure de rebar :

Quelques variables supplémentaires, spécifiques à l'infrastructure d'armatures, peuvent également être définies. Beaucoup d'entre eux ne sont utiles que dans des cas très spécifiques, les packages typiques n'en utiliseront donc que quelques-uns.

ERLANG_FOOBAR_USE_AUTOCONF	pour spécifier que le package utilise autoconf à l'étape de configuration. Lorsqu'un package définit cette variable sur YES, l'infrastructure des autotools est utilisée. on peut également utiliser certaines des variables de l'infrastructure des autotools : **ERLANG_FOOBAR_CONF_ENV**, **ERLANG_FOOBAR_CONF_OPTS**, **ERLANG_FOOBAR_AUTORECONF**, **ERLANG_FOOBAR_AUTORECONF_ENV** et **ERLANG_FOOBAR_AUTORECONF_OPTS**.
ERLANG_FOOBAR_USE_BUNDLED_REBAR	pour spécifier que le package a une version groupée de rebar et qu'il doit être utilisé. Les valeurs valides sont **YES** ou **NO** (valeur par défaut). Si le package regroupe un utilitaire de barre d'armature, mais peut utiliser celui générique fourni par Buildroot, dites simplement **NO** (c'est-à-dire, ne pas spécifier pas cette variable). Défini uniquement s'il est obligatoire d'utiliser l'utilitaire d'armature fourni dans ce package.
ERLANG_FOOBAR_REBAR_ENV	pour spécifier des variables d'environnement supplémentaires à transmettre à l'utilitaire Rebar.
ERLANG_FOOBAR_KEEP_DEPENDENCIES	pour conserver les dépendances décrites dans le fichier rebar.config. Les valeurs valides sont **YES** ou **NO** (valeur par défaut). À moins que cette variable ne soit définie sur YES, l'infrastructure d'armature supprime ces dépendances dans un hook post-correctif pour s'assurer que l'armature ne les télécharge ni ne les compile.

Avec l'infrastructure de erlang, toutes les étapes requises pour créer et installer les packages sont déjà définies, et elles fonctionnent généralement bien pour la plupart des packages basés sur les erlang. Cependant, si nécessaire, il est toujours possible de personnaliser ce qui est fait à une étape particulière :

- En ajoutant un hook post-opération (après extraction, patch, configuration, compilation ou installation). Voir Section « hooks disponibles dans les différentes étapes de construction » pour plus de détails.
- En remplaçant l'une des étapes. Par exemple, même si l'infrastructure de erlang est utilisée, si le fichier `.mk` du package définit sa propre variable **ERLANG_FOOBAR_BUILD_CMDS**, elle sera utilisée à la place de la barre d'armature par défaut.une. Cependant, l'utilisation de cette méthode doit être limitée à des cas très spécifiques. Ne pas l'utiliser dans le cas général.

Infrastructure pour les packages basés sur Waf

waf-package tutoriel

Voyons d'abord comment écrire un fichier .mk pour un package basé sur Waf, avec un exemple :

```
01:
################################################################################
####
02: #
03: # libfoo
04: #
05:
################################################################################
####
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_CONF_OPTS = --enable-bar --disable-baz
11: LIBFOO_DEPENDENCIES = bar
12:
13: $(eval $(waf-package))
```

- A la ligne 7, on déclare la version du package.
- Aux lignes 8 et 9, on déclare le nom du tarball (xz-ed tarball recommandé) et l'emplacement du tarball sur le Web. Buildroot téléchargera automatiquement l'archive tar depuis cet emplacement.
- À la ligne 10, on indique à Buildroot quelles options activer pour libfoo.
- À la ligne 11, on indique à Buildroot les dépendances de libfoo.
- A la ligne ligne 13, on invoque la macro waf-package qui génère toutes les règles Makefile qui permettent réellement de construire le package.

référence du paquet waf

La macro principale de l'infrastructure de paquets Waf est waf-package. Elle est similaire à la macro de package générique.

Tout comme l'infrastructure générique, l'infrastructure Waf fonctionne en définissant un certain nombre de variables avant d'appeler la macro waf-package.

Tout d'abord, toutes les variables d'informations de métadonnées de package qui existent dans l'infrastructure générique existent également dans l'infrastructure Waf : **LIBFOO_VERSION**, **LIBFOO_SOURCE**, **LIBFOO_PATCH**, **LIBFOO_SITE**, **LIBFOO_SUBDIR**, **LIBFOO_DEPENDENCIES**, **LIBFOO_INSTALL_STAGING**, LIBFOO_INSTALL_TARGET.

Une variable supplémentaire, spécifique à l'infrastructure Waf, peut également être définie.

LIBFOO_SUBDIR	peut contenir le nom d'un sous-répertoire à l'intérieur du package qui contient le fichier wscript principal. Ceci est utile, si par exemple, le fichier `wscript` principal n'est pas à la racine de l'arbre extrait par l'archive. Si HOST_**LIBFOO_SUBDIR** n'est pas spécifié, la valeur par défaut est **LIBFOO_SUBDIR**.

LIBFOO_NEEDS_EXTERNAL_WAF	peut être défini sur **YES** ou **NO** pour indiquer à Buildroot d'utiliser l'exécutable waf fourni. S'il est défini sur **NO**, la valeur par défaut, Buildroot utilisera l'exécutable waf fourni dans l'arborescence des sources du package ; s'il est défini sur **YES**, Buildroot se téléchargera, installera waf en tant qu'outil hôte et l'utilisera pour créer le package.
LIBFOO_WAF_OPTS	pour spécifier des options supplémentaires à passer au script waf à chaque étape du processus de construction du paquet : configuration, construction et installation. Par défaut, vide.
LIBFOO_CONF_OPTS	pour spécifier des options supplémentaires à passer au script waf pour l'étape de configuration. Par défaut, vide.
LIBFOO_BUILD_OPTS	pour spécifier des options supplémentaires à passer au script waf lors de l'étape de construction. Par défaut, vide.
LIBFOO_INSTALL_STAGING_OPTS	pour spécifier des options supplémentaires à transmettre au script waf lors de l'étape d'installation intermédiaire. Par défaut, vide.
LIBFOO_INSTALL_TARGET_OPTS	pour spécifier des options supplémentaires à passer au script waf lors de l'étape d'installation de la cible. Par défaut, vide.

Infrastructure pour les packages basés sur Meson

didacticiel meson-package

Meson est un système de construction open source censé être à la fois extrêmement rapide et, plus important encore, aussi convivial que possible. Il utilise Ninja comme outil compagnon pour effectuer les opérations de construction réelles.

Voyons comment écrire un fichier .mk pour un package basé sur Meson, avec un exemple :

```
01:
################################################################################
####
02: #
03: # foo
04: #
05:
################################################################################
####
06:
07: FOO_VERSION = 1.0
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.gz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-3.0+
11: FOO_LICENSE_FILES = COPYING
12: FOO_INSTALL_STAGING = YES
13:
14: FOO_DEPENDENCIES = host-pkgconf bar
15:
16: ifeq ($(BR2_PACKAGE_BAZ),y)
```

```
17: FOO_CONF_OPTS += -Dbaz=true
18: FOO_DEPENDENCIES += baz
19: else
20: FOO_CONF_OPTS += -Dbaz=false
21: endif
22:
23: $(eval $(meson-package))
```

Le Makefile commence par la définition des variables standard pour la déclaration des packages (lignes 7 à 11).

- À la ligne ligne 23, on invoque la macro meson-package qui génère toutes les règles Makefile qui permettent réellement de construire le paquet.

Dans l'exemple, **host-pkgconf** et bar sont déclarés comme dépendances dans **FOO_DEPENDENCIES** * à la ligne 14 car le fichier de construction Meson de foo utilise **pkg-config** pour déterminer les drapeaux de compilation et les bibliothèques du paquet bar.

il n'est pas nécessaire d'ajouter host-meson dans la variable **FOO_DEPENDENCIES** d'un package, puisque cette dépendance de base est automatiquement ajoutée selon les besoins par l'infrastructure du package Meson.

Si le package “baz” est sélectionné, la prise en charge de la fonctionnalité “baz” dans “foo” est activée en ajoutant -Dbaz=true à FOO_CONF_OPTS * à la ligne 17, comme spécifié dans le mefichier son_options.txt dans l'arborescence des sources “foo”. Le package “baz” est également ajouté à **FOO_DEPENDENCIES**. La prise en charge de base est explicitement désactivée * à la ligne 20, si le package n'est pas sélectionné.

Pour résumer, pour ajouter un nouveau package basé sur meson, l'exemple Makefile peut être copié textuellement puis édité pour remplacer toutes les occurrences de **FOO** par le nom en majuscule du nouveau package et mettre à jour les valeurs des variables standard.

référence du paquet méson

La macro principale de l'infrastructure de paquets Meson est meson-package. Elle est similaire à la macro de package générique. La possibilité d'avoir des packages cible et hôte est également disponible, avec la macro **host-meson-package**.

Tout comme l'infrastructure générique, l'infrastructure Meson fonctionne en définissant un certain nombre de variables avant d'appeler la macro **meson-package**.

Tout d'abord, toutes les variables d'informations de métadonnées de package qui existent dans l'infrastructure générique existent également dans l'infrastructure Meson : **FOO_VERSION**, **FOO_SOURCE**, **FOO_PATCH**, **FOO_SITE**, **FOO_SUBDIR**, **FOO_DEPENDENCIES**, **FOO_INSTALL_STAGING**, **FOO_INSTALL_TARGET**.

Quelques variables supplémentaires, spécifiques à l'infrastructure Meson, peuvent également être définies. Beaucoup d'entre eux ne sont utiles que dans des cas très spécifiques, les packages

typiques n'en utiliseront donc que quelques-uns.

FOO_SUBDIR	peut contenir le nom d'un sous-répertoire à l'intérieur du package qui contient le fichier principal meson.build. Ceci est utile si, par exemple, le fichier principal meson.build n'est pas à la racine de l'arborescence extraite par l'archive. Si **HOST_FOO_SUBDIR** n'est pas spécifié, la valeur par défaut est **FOO_SUBDIR**.
FOO_CONF_ENV	pour spécifier des variables d'environnement supplémentaires à passer à meson pour l'étape de configuration. Par défaut, vide.
FOO_CONF_OPTS	pour spécifier des options supplémentaires à passer à meson pour l'étape de configuration. Par défaut, vide.
FOO_CFLAGS	pour spécifier les arguments du compilateur ajoutés à la propriété **c_args** du fichier cross-compile.conf spécifique au package. Par défaut, la valeur de TARGET_CFLAGS.
FOO_CXXFLAGS	pour spécifier les arguments du compilateur ajoutés à la propriété cpp_args du fichier cross-compile.conf spécifique au package. Par défaut, la valeur de **TARGET_CXXFLAGS**.
FOO_LDFLAGS	pour spécifier les arguments du compilateur ajoutés aux propriétés **c_link_args** et **cpp_link_args** du fichier cross-compile.conf spécifique au package. Par défaut, la valeur de **TARGET_LDFLAGS**.
FOO_MESON_EXTRA_BINARIES	pour spécifier une liste de programmes séparés par des espaces à ajouter à la section [binaries] du fichier de configuration meson cross-compilation.conf. Le format est nom-programme='/chemin/vers/programme', sans espace autour du signe =, et avec le chemin du programme entre guillemets simples. Par défaut, vide. Buildroot définit déjà les valeurs correctes pour c, cpp, ar, strip et pkgconfig.
FOO_MESON_EXTRA_PROPERTIES	pour spécifier une liste de propriétés séparées par des espaces à ajouter à la section [properties] du fichier de configuration meson cross-compilation.conf. Le format est nom-propriété=<valeur> sans espace autour du signe = et avec des guillemets simples autour des valeurs de chaîne. Par défaut, vide. Buildroot définit déjà des valeurs pour **needs_exe_wrapper**, **c_args**, **c_link_args**, **cpp_args**, **cpp_link_args**, **sys_root** et **pkg_config_libdir**.
FOO_NINJA_ENV	pour spécifier des variables d'environnement supplémentaires à transmettre à ninja, l'outil compagnon de meson en charge des opérations de construction. Par défaut, vide.
FOO_NINJA_OPTS	pour spécifier une liste de cibles séparées par des espaces à construire. Par défaut, vide, pour construire la ou les cibles par défaut.

Intégration de packages basés sur Cargo

Cargo est le gestionnaire de paquets pour le langage de programmation Rust. Il permet à l'utilisateur de construire des programmes ou des bibliothèques écrits en Rust, mais il télécharge et gère également leurs dépendances, pour assurer des constructions reproductibles. Les colis de fret sont appelés “caisses”.

Fichier Config.in du package basé sur Cargo

Le fichier Config.in du package foo basé sur Cargo doit contenir :

```
01: config BR2_PACKAGE_FOO
02:     bool "foo"
03:     depends on BR2_PACKAGE_HOST_RUSTC_TARGET_ARCH_SUPPORTS
04:     select BR2_PACKAGE_HOST_RUSTC
05:     help
06:       This is a comment that explains what foo is.
07:
08:       http://foosoftware.org/foo/
```

Fichier .mk du package basé sur Cargo

Buildroot ne fournit pas (encore) d'infrastructure de packages dédiée aux packages basés sur Cargo. On va donc expliquer comment écrire un fichier .mk pour un tel package. Commençons par un exemple :

```
01:
################################################################################
####
02: #
03: # foo
04: #
05:
################################################################################
####
06:
07: FOO_VERSION = 1.0
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.gz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-3.0+
11: FOO_LICENSE_FILES = COPYING
12:
13: FOO_DEPENDENCIES = host-rustc
14:
15: FOO_CARGO_ENV = CARGO_HOME=$(HOST_DIR)/share/cargo
16:
17: FOO_BIN_DIR = target/$(RUSTC_TARGET_NAME)/$(FOO_CARGO_MODE)
18:
19: FOO_CARGO_OPTS = \
20:     $(if $(BR2_ENABLE_DEBUG),,--release) \
21:     --target=$(RUSTC_TARGET_NAME) \
22:     --manifest-path=$(@D)/Cargo.toml
23:
24: define FOO_BUILD_CMDS
```

```
25:     $(TARGET_MAKE_ENV) $(FOO_CARGO_ENV) \
26:             cargo build $(FOO_CARGO_OPTS)
27: endef
28:
29: define FOO_INSTALL_TARGET_CMDS
30:     $(INSTALL) -D -m 0755 $(@D)/$(FOO_BIN_DIR)/foo \
31:             $(TARGET_DIR)/usr/bin/foo
32: endef
33:
34: $(eval $(generic-package))
```

Le Makefile commence par la définition des variables standard pour la déclaration des packages (lignes 7 à 11).

Comme on le voit * à la ligne 34, il est basé sur l'infrastructure des packages génériques. Ainsi, il définit les variables requises par cette infrastructure particulière, où Cargo est invoqué :

- **FOO_BUILD_CMDS** : Cargo est appelé pour effectuer la construction. Les options nécessaires pour configurer la compilation croisée du package sont passées via **FOO_CONF_OPTS**.
- **FOO_INSTALL_TARGET_CMDS** : L'exécutable binaire généré est installé sur la cible.

Afin que Cargo soit disponible pour la construction, **FOO_DEPENDENCIES** doit contenir host-cargo.

Pour résumer, pour ajouter un nouveau package basé sur Cargo, l'exemple Makefile peut être copié textuellement puis édité pour remplacer toutes les occurrences de FOO par le nom en majuscule du nouveau package et mettre à jour les valeurs des variables standard.

À propos de la gestion des dépendances

Une caisse peut dépendre d'autres bibliothèques des dépôts crates.io ou git, listées dans son fichier Cargo.toml. Avant de commencer une construction, Cargo les télécharge généralement automatiquement. Cette étape peut également être effectuée indépendamment, via la commande cargo fetch.

Cargo maintient un cache local de l'index de registre et des git checkouts des caisses, dont l'emplacement est donné par **$CARGO_HOME**. Comme indiqué dans l'exemple Makefile du package * à la ligne 15, cette variable d'environnement est définie sur **$(HOST_DIR)/share/cargo**.

Ce mécanisme de téléchargement de dépendances n'est pas pratique lors de l'exécution d'une construction hors ligne, car Cargo ne parviendra pas à récupérer les dépendances. Dans ce cas, il est conseillé de générer une archive tar des dépendances à l'aide du fournisseur de fret et de l'ajouter à **FOO_EXTRA_DOWNLOADS**.

Infrastructure pour Go

Cette infrastructure s'applique aux packages Go qui utilisent le système de construction standard et utilisent des dépendances groupées.

tutoriel golang-package

Voyons d'abord comment écrire un fichier .mk pour un package go, avec un exemple :

```
01:
################################################################################
####
02: #
03: # foo
04: #
05:
################################################################################
####
06:
07: FOO_VERSION = 1.0
08: FOO_SITE = $(call github,bar,foo,$(FOO_VERSION))
09: FOO_LICENSE = BSD-3-Clause
10: FOO_LICENSE_FILES = LICENSE
11:
12: $(eval $(golang-package))
```

- A la ligne 7, on déclare la version du package.
- À la ligne 8, on déclare l'emplacement en amont du paquet, ici récupéré sur Github, car un grand nombre de paquets Go sont hébergés sur Github.
- Aux lignes 9 et 10, on donne des détails sur la licence du package.
- A la ligne 12, on invoque la macro golang-package qui génère toutes les règles Makefile qui permettent réellement de construire le package.

référence du package golang

Dans leur fichier Config.in, les packages utilisant l'infrastructure golang-package doivent dépendre de **BR2_PACKAGE_HOST_GO_TARGET_ARCH_SUPPORTS** car Buildroot ajoutera automatiquement une dépendance sur host-go à ces packages. Si on a besoin de la prise en charge de CGO dans le package, ajouter une dépendance sur **BR2_PACKAGE_HOST_GO_TARGET_CGO_LINKING_SUPPORTS**.

La macro principale de l'infrastructure du package Go est golang-package. Elle est similaire à la macro de package générique. La possibilité de créer des packages hôtes est également disponible, avec la macro host-golang-package. Les packages hôtes créés par la macro host-golang-package doivent dépendre de **BR2_PACKAGE_HOST_GO_HOST_ARCH_SUPPORTS**.

Tout comme l'infrastructure générique, l'infrastructure Go fonctionne en définissant un certain nombre de variables avant d'appeler le package golang.

Toutes les variables d'informations de métadonnées de package qui existent dans l'infrastructure de package générique existent également dans l'infrastructure Go : **FOO_VERSION**, **FOO_SOURCE**, **FOO_PATCH**, **FOO_SITE**, **FOO_SUBDIR**, **FOO_DEPENDENCIES**, **FOO_LICENSE**, **FOO_LICENSE_FILES**, **FOO_INSTALL_STAGING**, etc.

Iil n'est pas nécessaire d'ajouter host-go dans la variable FOO_DEPENDENCIES d'un package, puisque cette dépendance de base est automatiquement ajoutée selon les besoins par l'infrastructure du package Go.

Quelques variables supplémentaires, spécifiques à l'infrastructure Go, peuvent éventuellement être définies, en fonction des besoins du package. Beaucoup d'entre eux ne sont utiles que dans des cas très spécifiques, les packages typiques n'en utiliseront donc que quelques-uns, voire aucun.

FOO_GOMOD	Le package doit spécifier son nom de module Go dans la variable **FOO_GOMOD**. S'il n'est pas spécifié, la valeur par défaut est URL-domain/1st-part-of-URL/2nd-part-of-URL, par exemple **FOO_GOMOD** prendra la valeur github.com/bar/foo pour un package qui spécifie **FOO_SITE** = $(call github, barre, truc, **$(FOO_VERSION)**). L'infrastructure du package Go générera automatiquement un fichier go.mod minimal dans l'arborescence source du package s'il n'existe pas.
FOO_LDFLAGS **FOO_TAGS**	peuvent être utilisés pour passer respectivemente **LDFLAGS** ou les **TAGS** à la commande go build.
FOO_BUILD_TARGETS	peut être utilisé pour passer la liste des cibles qui doivent être construites. Si **FOO_BUILD_TARGETS** n'est pas spécifié, sa valeur par défaut est .. buildroot a alors deux cas : * **FOO_BUILD_TARGETS** Dans ce cas, on suppose qu'un seul binaire sera produit, et que par défaut on le nomme d'après le nom du paquet. Si cela n'est pas approprié, le nom du binaire produit peut être remplacé à l'aide de **FOO_BIN_NAME**. * **FOO_BUILD_TARGETS** n'est pas. Dans ce cas, on parcoure les valeurs pour construire chaque cible, et pour chacune produit un binaire qui est le composant non-répertoire de la cible. Par exemple si `FOO\_BUILD\_TARGETS = cmd/docker cmd/dockerd` les binaires produites sont **docker** et **dockerd**.
FOO_INSTALL_BINS	peut être utilisé pour transmettre la liste des binaires qui doivent être installés dans /usr/bin sur la cible. Si **FOO_INSTALL_BINS** n'est pas spécifié, il utilise par défaut le nom du package en minuscules.

Avec l'infrastructure Go, toutes les étapes nécessaires à la création et à l'installation des packages sont déjà définies et fonctionnent généralement bien pour la plupart des packages basés sur Go. Cependant, si nécessaire, il est toujours possible de personnaliser ce qui est fait à une étape particulière :

- En ajoutant un hook post-opération (après extraction, patch, configuration, compilation ou installation). Voir Section « hooks disponibles dans les différentes étapes de construction » pour plus de détails.
- En remplaçant l'une des étapes. Par exemple, même si l'infrastructure Go est utilisée, si le fichier .mk du package définit sa propre variable FOO_BUILD_CMDS, elle sera utilisée à la place de celle par défaut de Go. Cependant, l'utilisation de cette méthode doit être limitée à des cas très spécifiques. Ne pas l'utiliser dans le cas général.

Infrastructure pour les packages basés sur QMake

tutoriel qmake-package

Voyons d'abord comment écrire un fichier .mk pour un package basé sur QMake, avec un exemple :

```
01:
################################################################################
####
02: #
03: # libfoo
04: #
05:
################################################################################
####
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_CONF_OPTS = QT_CONFIG+=bar QT_CONFIG-=baz
11: LIBFOO_DEPENDENCIES = bar
12:
13: $(eval $(qmake-package))
```

- A la ligne 7, on déclare la version du package.
- Aux lignes 8 et 9, on déclare le nom du tarball (xz-ed tarball recommandé) et l'emplacement du tarball sur le Web. Buildroot téléchargera automatiquement l'archive tar depuis cet emplacement.
- À la ligne 10, on indique à Buildroot quelles options activer pour libfoo.
- À la ligne 11, on indique à Buildroot les dépendances de libfoo.
- A la ligne ligne 13, on invoque la macro qmake-package qui génère toutes les règles Makefile qui permettent réellement de construire le paquet.

qmake-package référence

La macro principale de l'infrastructure de packages QMake est qmake-package. Elle est similaire à la macro de package générique.

Tout comme l'infrastructure générique, l'infrastructure QMake fonctionne en définissant un certain nombre de variables avant d'appeler la macro qmake-package.

Tout d'abord, toutes les variables d'informations de métadonnées de package qui existent dans l'infrastructure générique existent également dans l'infrastructure QMake : **LIBFOO_VERSION**, **LIBFOO_SOURCE**, **LIBFOO_PATCH**, **LIBFOO_SITE**, **LIBFOO_SUBDIR**, **LIBFOO_DEPENDENCIES**, **LIBFOO_INSTALL_STAGING**, LIBFOO_INSTALL_TARGET.

Une variable supplémentaire, spécifique à l'infrastructure QMake, peut également être définie.

LIBFOO_CONF_ENV	pour spécifier des variables d'environnement supplémentaires à transmettre au script qmake pour l'étape de configuration. Par défaut, vide.

LIBFOO_CONF_OPTS	pour spécifier des options supplémentaires à passer au script qmake pour l'étape de configuration. Par défaut, vide.
LIBFOO_MAKE_ENV	pour spécifier des variables d'environnement supplémentaires à la commande make pendant les étapes de construction et d'installation. Par défaut, vide.
LIBFOO_MAKE_OPTS	pour spécifier des cibles supplémentaires à transmettre à la commande make lors de l'étape de construction. Par défaut, vide.
LIBFOO_INSTALL_STAGING_OPTS	pour spécifier des cibles supplémentaires à transmettre à la commande make lors de l'étape d'installation intermédiaire. Par défaut, install.
LIBFOO_INSTALL_TARGET_OPTS	pour spécifier des cibles supplémentaires à transmettre à la commande make lors de l'étape d'installation de la cible. Par défaut, install.
LIBFOO_SYNC_QT_HEADERS	pour exécuter syncqt.pl avant qmake. Certains packages en ont besoin pour avoir un répertoire d'inclusion correctement rempli avant d'exécuter la construction.

Infrastructure pour les packages créant des modules de noyau

Buildroot offre une infrastructure d'assistance pour faciliter l'écriture de packages qui construisent et installent des modules du noyau Linux. Certains packages ne contiennent qu'un module noyau, d'autres packages contiennent des programmes et des bibliothèques en plus des modules noyau. L'infrastructure d'assistance de Buildroot prend en charge les deux cas.

tutoriel du module noyau

Commençons par un exemple sur la façon de préparer un package simple qui ne construit qu'un module de noyau, et aucun autre composant :

```
01:
################################################################################
####
02: #
03: # foo
04: #
05:
################################################################################
####
06:
07: FOO_VERSION = 1.2.3
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.xz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-2.0
11: FOO_LICENSE_FILES = COPYING
12:
13: $(eval $(kernel-module))
```

```
14: $(eval $(generic-package))
```

- Les lignes 7 à 11 définissent les métadonnées habituelles pour spécifier la version, le nom de l'archive, l'URI distant où trouver la source du paquet, les informations de licence.
- À la ligne 13, on invoque l'infrastructure d'assistance du module du noyau, qui génère toutes les règles et variables Makefile appropriées pour construire ce module du noyau.
- A la ligne 14, on invoque l'infrastructure de paquets génériques.

La dépendance à linux est automatiquement ajoutée, il n'est donc pas nécessaire de la spécifier dans **FOO_DEPENDENCIES**.

Contrairement à d'autres infrastructures de packages, on invoque explicitement une seconde infrastructure. Cela permet à un paquet de construire un module noyau, mais aussi, si nécessaire, d'utiliser n'importe laquelle des autres infrastructures de paquets pour construire des composants utilisateur normaux (bibliothèques, exécutables...). L'utilisation de l'infrastructure noyau-module seule n'est pas suffisante ; une autre infrastructure de package doit être utilisée.

Prenons un exemple plus complexe :

```
01:
################################################################################
####
02: #
03: # foo
04: #
05:
################################################################################
####
06:
07: FOO_VERSION = 1.2.3
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.xz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-2.0
11: FOO_LICENSE_FILES = COPYING
12:
13: FOO_MODULE_SUBDIRS = driver/base
14: FOO_MODULE_MAKE_OPTS = KVERSION=$(LINUX_VERSION_PROBED)
15:
16: ifeq ($(BR2_PACKAGE_LIBBAR),y)
17: FOO_DEPENDENCIES = libbar
18: FOO_CONF_OPTS = --enable-bar
19: FOO_MODULE_SUBDIRS += driver/bar
20: else
21: FOO_CONF_OPTS = --disable-bar
22: endif
23:
24: $(eval $(kernel-module))
26: $(eval $(autotools-package))
```

Ici, on voit que buildroot a un paquet basé sur autotools, qui construit également le module noyau situé dans le sous-répertoire driver/base et, si libbar est activé, le module noyau situé dans le sous-

répertoire driver/bar, et définit la variable **KVERSION** à transmettre au système de construction Linux lors de la construction du ou des modules.

référence du module du noyau

La macro principale pour l'infrastructure du module du noyau est kernel-module. Contrairement à d'autres infrastructures de packages, il n'est pas autonome et nécessite que l'une des autres macros *-package soit appelée après lui.

La macro kernel-module définit les hooks **post-build** et **post-target-install** pour construire les modules du noyau. Si le `.mk` du paquet a besoin d'accéder aux modules du noyau construits, il doit le faire dans un hook **post-build**, enregistré après l'appel à kernel-module. De même, si le `.mk` du paquet a besoin d'accéder au module du noyau après son installation, il doit le faire dans un hook **post-install**, enregistré après l'appel à kernel-module. Voici un exemple :

```
$(eval $(kernel-module))

define FOO_DO_STUFF_WITH_KERNEL_MODULE
    # Do something with it...
endef
FOO_POST_BUILD_HOOKS += FOO_DO_STUFF_WITH_KERNEL_MODULE

$(eval $(generic-package))
```

Enfin, contrairement aux autres infrastructures de paquets, il n'y a pas de variante host-kernel-module pour construire un module noyau hôte.

Les variables supplémentaires suivantes peuvent éventuellement être définies pour configurer davantage la construction du module du noyau :

FOO_MODULE_SUBDIRS	peut être défini sur un ou plusieurs sous-répertoires (par rapport au répertoire supérieur des sources du paquet) où se trouvent les sources du module du noyau. S'il est vide ou non défini, les sources du ou des modules du noyau sont considérées comme étant situées en haut de l'arborescence des sources du paquet.
FOO_MODULE_MAKE_OPTS	peut être défini pour contenir des définitions de variables supplémentaires à transmettre au système de construction Linux.

on peut également référencer (mais on ne peut pas définir !) ces variables :

- **LINUX_DIR** contient le chemin d'accès à l'endroit où le noyau Linux a été extrait et construit.
- **LINUX_VERSION** contient la chaîne de version telle que configurée par l'utilisateur.
- **LINUX_VERSION_PROBED** contient la chaîne de version réelle du noyau, récupérée avec l'exécution de `make -C $(LINUX\_DIR) kernelrelease`
- **KERNEL_ARCH** contient le nom de l'architecture actuelle, comme arm, mips…

Infrastructure pour les documents asciidoc

Bien que Buildroot ne contienne qu'un seul document écrit en AsciiDoc, il existe, comme pour les packages, une infrastructure de rendu des documents utilisant la syntaxe AsciiDoc.

Toujours comme pour les packages, l'infrastructure AsciiDoc est disponible depuis une arborescence externe br2. Cela permet à la documentation d'une arborescence externe br2 de correspondre à la documentation Buildroot, car elle sera rendue dans les mêmes formats et utilisera la même mise en page et le même thème.

tutoriel asciidoc-document

Tandis queles infrastructures de package sont suffixées par -package, les infrastructures de document sont suffixées par -document. Ainsi, l'infrastructure AsciiDoc est nommée asciidoc-document.

Voici un exemple pour rendre un document AsciiDoc simple.

```
01:
################################################################################
02: #
03: # foo-document
04: #
05:
################################################################################
06:
07: FOO_SOURCES = $(sort $(wildcard $(pkgdir)/*))
08: $(eval $(call asciidoc-document))
```

- A la ligne 7, le Makefile déclare quelles sont les sources du document. Actuellement, on s'attend à ce que les sources du document soient uniquement locales ; Buildroot n'essaiera pas de télécharger quoi que ce soit pour rendre un document. Ainsi, indiquer où se trouvent les sources. Habituellement, la chaîne ci-dessus est suffisante pour un document sans structure de sous-répertoires.
- A la ligne 8, on appelle la fonction asciidoc-document, qui génère tout le code Makefile nécessaire pour rendre le document.

référence du document asciidoc

La liste des variables qui peuvent être définies dans un fichier .mk pour donner des informations de métadonnées est (en supposant que le nom du document est foo) :

FOO_SOURCES	obligatoire, définit les fichiers source du document.
FOO_RESOURCES	facultatif, peut contenir une liste de chemins séparés par des espaces vers un ou plusieurs répertoires contenant des ressources dites (comme CSS ou des images). Par défaut, vide.

FOO_DEPENDENCIES	facultatif, la liste des packages (probablement des packages hôtes) qui doivent être construits avant de construire ce document.

Il existe également des hooks supplémentaires (voir Section « hooks disponibles dans les différentes étapes de construction » pour des informations générales sur les hooks), qu'un document peut définir pour définir des actions supplémentaires à effectuer à différentes étapes :

FOO_POST_RSYNC_HOOKS	pour exécuter des commandes supplémentaires après que les sources ont été copiées par Buildroot. Cela peut par exemple être utilisé pour générer une partie du manuel avec des informations extraites de l'arborescence. Par exemple, Buildroot utilise ce hook pour générer les tableaux dans les annexes.
FOO_CHECK_DEPENDENCIES_HOOKS	pour exécuter des tests supplémentaires sur les composants requis pour générer le document. Dans AsciiDoc, il est possible d'appeler des filtres, c'est-à-dire des programmes qui analyseront un bloc AsciiDoc et le restitueront de manière appropriée (par exemple ditaa ou aafigure).
FOO_CHECK_DEPENDENCIES_<FMT>_HOOKS	pour exécuter des tests supplémentaires pour le format spécifié <FMT> (voir la liste des formats rendus, ci-dessus).

Voici un exemple complet qui utilise toutes les variables et tous les hooks :

```
01: ################################################################################
02: #
03: # foo-document
04: #
05: ################################################################################
06:
07: FOO_SOURCES = $(sort $(wildcard $(pkgdir)/*))
08: FOO_RESOURCES = $(sort $(wildcard $(pkgdir)/ressources))
09:
10: define FOO_GEN_EXTRA_DOC
11:     /path/to/generate-script --outdir=$(@D)
12: endef
13: FOO_POST_RSYNC_HOOKS += FOO_GEN_EXTRA_DOC
14:
15: define FOO_CHECK_MY_PROG
16:     if ! which my-prog >/dev/null 2>&1; then \
17:         echo "You need my-prog to generate the foo document"; \
18:         exit 1; \
19:     fi
20: endef
21: FOO_CHECK_DEPENDENCIES_HOOKS += FOO_CHECK_MY_PROG
```

```
22:
23: define FOO_CHECK_MY_OTHER_PROG
24:     if ! which my-other-prog >/dev/null 2>&1; then \
25:         echo "You need my-other-prog to generate the foo document as
PDF"; \
26:         exit 1; \
27:     fi
28: endef
29: FOO_CHECK_DEPENDENCIES_PDF_HOOKS += FOO_CHECK_MY_OTHER_PROG
30:
31: $(eval $(call asciidoc-document))
```

Infrastructure spécifique au package du noyau Linux

Le package du noyau Linux peut utiliser certaines infrastructures spécifiques basées sur des hooks de package pour créer des outils du noyau Linux ou/et créer des extensions du noyau Linux.

outils-du-noyau-linux

Buildroot offre une infrastructure d'assistance pour créer des outils d'espace utilisateur pour la cible disponible dans les sources du noyau Linux. Comme leur code source fait partie du code source du noyau, un package spécial, linux-tools, existe et réutilise les sources du noyau Linux qui s'exécute sur la cible.

Regardons un exemple d'outil Linux. Pour un nouvel outil Linux nommé foo, créer une nouvelle entrée de menu dans le package/linux-tools/Config.in existant. Ce fichier contiendra les descriptions d'options liées à chaque outil du noyau qui sera utilisé et affiché dans l'outil de configuration. Cela ressemblerait essentiellement à:

```
01: config BR2_PACKAGE_LINUX_TOOLS_FOO
02:     bool "foo"
03:     select BR2_PACKAGE_LINUX_TOOLS
04:     help
05:       This is a comment that explains what foo kernel tool is.
06:
07:       http://foosoftware.org/foo/
```

Le nom de l'option commence par le préfixe **BR2_PACKAGE_LINUX_TOOLS_**, suivi du nom de l'outil en majuscule (comme c'est le cas pour les packages).

Contrairement aux autres packages, les options du package linux-tools apparaissent dans le menu du noyau Linux, sous le sous-menu Outils du noyau Linux, et non dans le menu principal des packages cibles.

Ensuite, pour chaque outil Linux, ajouter un nouveau fichier .mk.in nommé package/linux-tools/linux-tool-foo.mk.in. Cela ressemblerait essentiellement à:

```
01: ################################################################################
02: #
03: # foo
04: #
05: ################################################################################
06:
07: LINUX_TOOLS += foo
08:
09: FOO_DEPENDENCIES = libbbb
10:
11: define FOO_BUILD_CMDS
12:     $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools foo
13: endef
14:
15: define FOO_INSTALL_STAGING_CMDS
16:     $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools \
17:             DESTDIR=$(STAGING_DIR) \
18:             foo_install
19: endef
20:
21: define FOO_INSTALL_TARGET_CMDS
22:     $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools \
23:             DESTDIR=$(TARGET_DIR) \
24:             foo_install
25: endef
```

- À la ligne 7, on enregistre l'outil Linux foo dans la liste des outils Linux disponibles.
- À la ligne 9, on spécifie la liste des dépendances sur lesquelles repose cet outil. Ces dépendances sont ajoutées à la liste des dépendances du package Linux uniquement lorsque l'outil foo est sélectionné.
- Le reste du Makefile, lignes 11-25 définit ce qui doit être fait aux différentes étapes du processus de construction de l'outil Linux comme pour un paquet générique. Ils ne seront en fait utilisés que lorsque l'outil foo est sélectionné. Les seules commandes prises en charge sont **_BUILD_CMDS**, **_INSTALL_STAGING_CMDS** et **_INSTALL_TARGET_CMDS**.

Il ne faut pas appeler $(eval $(generic-package)) ou toute autre infrastructure de paquets ! Les outils Linux ne sont pas des packages en eux-mêmes, ils font partie du package linux-tools.

extensions du noyau linux

Certains packages fournissent de nouvelles fonctionnalités qui nécessitent la modification de l'arborescence du noyau Linux. Cela peut être sous la forme de correctifs à appliquer sur l'arborescence du noyau, ou sous la forme de nouveaux fichiers à ajouter à l'arborescence. L'infrastructure d'extensions du noyau Linux de Buildroot fournit une solution simple pour le faire automatiquement, juste après l'extraction des sources du noyau et avant l'application des correctifs du noyau. Des exemples d'extensions empaquetées à l'aide de ce mécanisme sont les extensions temps réel Xenomai et RTAI, ainsi que l'ensemble de pilotes d'écrans LCD hors arbre fbtft.

Regardons un exemple sur la façon d'ajouter une nouvelle extension Linux foo.

Tout d'abord, créer le package foo qui fournit l'extension : ce package est un package standard ; voir les chapitres précédents sur la façon de créer un tel package. Ce paquet est en charge du téléchargement de l'archive des sources, de la vérification du hash, de la définition des informations de licence et de la construction des outils de l'espace utilisateur s'il y en a.

Créer ensuite l'extension Linux proprement dite : créer une nouvelle entrée de menu dans le fichier linux/Config.ext.in existant. Ce fichier contient les descriptions d'options liées à chaque extension de noyau qui sera utilisée et affichée dans l'outil de configuration. Cela ressemblerait essentiellement à:

```
01: config BR2_LINUX_KERNEL_EXT_FOO
02:     bool "foo"
03:     help
04:       This is a comment that explains what foo kernel extension is.
05:
06:       http://foosoftware.org/foo/
```

Ensuite, pour chaque extension Linux, ajouter un nouveau fichier .mk nommé linux/linux-ext-foo.mk. Il doit essentiellement contenir :

```
01:
################################################################################
####
02: #
03: # foo
04: #
05:
################################################################################
####
06:
07: LINUX_EXTENSIONS += foo
08:
09: define FOO_PREPARE_KERNEL
10:     $(FOO_DIR)/prepare-kernel-tree.sh --linux-dir=$(@D)
11: endef
```

- À la ligne 7, on ajoute l'extension Linux foo à la liste des extensions Linux disponibles.
- Aux lignes 9-11, on définit ce que doit faire l'extension pour modifier l'arborescence du noyau

Linux ; celle-ci est spécifique à l'extension linux et peut utiliser les variables définies par le package foo, comme : **$(FOO_DIR)** ou **$(FOO_VERSION)**... ainsi que toutes les variables Linux, comme : **$(LINUX_VERSION)** ou **$(LINUX_VERSION_PROBED)**, **$(KERNEL_ARCH)**... Voir la définition de ces variables du noyau.

hooks disponibles dans les différentes étapes de construction

L'infrastructure générique (et par conséquent également les autotools dérivés et les infrastructures cmake) permet aux packages de spécifier des hooks. Ceux-ci définissent d'autres actions à effectuer après les étapes existantes. La plupart des hooks ne sont pas vraiment utiles pour les packages génériques, car le fichier .mk a déjà un contrôle total sur les actions effectuées à chaque étape de la construction du package.

Les points d'accroche suivants sont disponibles :

- **LIBFOO_PRE_DOWNLOAD_HOOKS**
- **LIBFOO_POST_DOWNLOAD_HOOKS**
- **LIBFOO_PRE_EXTRACT_HOOKS**
- **LIBFOO_POST_EXTRACT_HOOKS**
- **LIBFOO_PRE_RSYNC_HOOKS**
- **LIBFOO_POST_RSYNC_HOOKS**
- **LIBFOO_PRE_PATCH_HOOKS**
- **LIBFOO_POST_PATCH_HOOKS**
- **LIBFOO_PRE_CONFIGURE_HOOKS**
- **LIBFOO_POST_CONFIGURE_HOOKS**
- **LIBFOO_PRE_BUILD_HOOKS**
- **LIBFOO_POST_BUILD_HOOKS**
- **LIBFOO_PRE_INSTALL_HOOKS** (pour les packages hôtes uniquement)
- **LIBFOO_POST_INSTALL_HOOKS** (pour les packages hôtes uniquement)
- **LIBFOO_PRE_INSTALL_STAGING_HOOKS** (pour les packages cibles uniquement)
- **LIBFOO_POST_INSTALL_STAGING_HOOKS** (pour les packages cibles uniquement)
- **LIBFOO_PRE_INSTALL_TARGET_HOOKS** (pour les packages cibles uniquement)
- **LIBFOO_POST_INSTALL_TARGET_HOOKS** (pour les packages cibles uniquement)
- **LIBFOO_PRE_INSTALL_IMAGES_HOOKS**
- **LIBFOO_POST_INSTALL_IMAGES_HOOKS**
- **LIBFOO_PRE_LEGAL_INFO_HOOKS**
- **LIBFOO_POST_LEGAL_INFO_HOOKS**

Ces variables sont des listes de noms de variables contenant des actions à effectuer à ce point de hook. Cela permet d'enregistrer plusieurs hameçons à un point d'hameçon donné. Voici un exemple:

```
define LIBFOO_POST_PATCH_FIXUP
        action1
        action2
endef

LIBFOO_POST_PATCH_HOOKS += LIBFOO_POST_PATCH_FIXUP
```

Utilisation du hook POST_RSYNC

Le hook **POST_RSYNC** est exécuté uniquement pour les packages qui utilisent une source locale, via la méthode du site local ou le mécanisme **OVERRIDE_SRCDIR**. Dans ce cas, les sources de package sont copiées à l'aide de rsync depuis l'emplacement local dans le répertoire de construction buildroot. La commande rsync ne copie cependant pas tous les fichiers du répertoire source. Les fichiers appartenant à un système de contrôle de version, comme les répertoires .git, .hg, etc. ne sont pas copiés. Pour la plupart des packages, cela suffit, mais un package donné peut effectuer des actions supplémentaires à l'aide du hook **POST_RSYNC**.

En principe, le hook peut contenir n'importe quelle commande qu'on veut. Un cas d'utilisation spécifique, cependant, est la copie intentionnelle du répertoire de contrôle de version à l'aide de rsync. La commande rsync qu'onutilise dans le hook peut, entre autres, utiliser les variables suivantes :

- **$(SRCDIR)**: le chemin vers le répertoire source remplacé
- **$(@D)**: le chemin d'accès au répertoire de construction

Hook target-finalize

Les packages peuvent également enregistrer des hooks dans **LIBFOO_TARGET_FINALIZE_HOOKS**. Ces hooks sont exécutés après la construction de tous les packages, mais avant la génération des images du système de fichiers. Ils sont rarement utilisés et le paquet n'en a probablement pas besoin.

Intégration de Gettext et interaction avec les packages

De nombreux packages prenant en charge l'internationalisation utilisent la bibliothèque **gettext**. Les dépendances de cette bibliothèque sont assez compliquées et méritent donc quelques explications.

La bibliothèque **glibc C** intègre une implémentation complète de **gettext**, prenant en charge la traduction. La prise en charge des langues natives est donc intégrée à la glibc.

D'autre part, les bibliothèques **uClibc** et **musl C** ne fournissent qu'une implémentation stub de la fonctionnalité **gettext**, qui permet de compiler des bibliothèques et des programmes à l'aide de fonctions **gettext**, mais sans fournir les capacités de traduction d'une implémentation complète de **gettext**. Avec de telles bibliothèques C, si un véritable support de langage natif est nécessaire, il peut être fourni par la bibliothèque libintl du paquet **gettext**.

Pour cette raison, et afin de s'assurer que la prise en charge de la langue native est correctement gérée, les packages dans Buildroot qui peuvent utiliser la prise en charge NLS doivent :

- S'assurer que la prise en charge NLS est activée lorsque **BR2_SYSTEM_ENABLE_NLS=y**. Cette opération est effectuée automatiquement pour les packages autotools et ne doit donc être effectuée que pour les packages utilisant d'autres infrastructures de packages.
- Ajouter **$(TARGET_NLS_DEPENDENCIES)** à la variable **<pkg>_DEPENDENCIES** du package. Cet ajout doit être fait de manière inconditionnelle : la valeur de cette variable est automatiquement ajustée par l'infrastructure centrale pour contenir la liste des packages concernés. Si la prise en charge NLS est désactivée, cette variable est vide. Si la prise en

charge NLS est activée, cette variable contient host-gettext afin que les outils nécessaires à la compilation des fichiers de traduction soient disponibles sur l'hôte. De plus, si uClibc ou musl sont utilisés, cette variable contient également gettext afin d'obtenir l'implémentation complète de gettext.
- Si nécessaire, ajouter **$(TARGET_NLS_LIBS)** aux drapeaux de l'éditeur de liens, afin que le package soit lié à libintl. Ce n'est généralement pas nécessaire avec les packages autotools car ils détectent généralement automatiquement qu'ils doivent être liés à libintl. Cependant, les packages utilisant d'autres systèmes de construction ou des packages problématiques basés sur des outils automatiques peuvent en avoir besoin. **$(TARGET_NLS_LIBS)** doit être ajouté sans condition aux drapeaux de l'éditeur de liens, car le noyau le rend automatiquement vide ou défini sur **-lintl** en fonction de la configuration.

Aucune modification ne doit être apportée au fichier Config.in pour prendre en charge NLS.

Enfin, certains packages nécessitent certains utilitaires gettext sur la cible, comme le programme gettext lui-même, qui permet de récupérer les chaînes traduites, à partir de la ligne de commande. Dans un tel cas, le colis doit :

- utiliser select **BR2_PACKAGE_GETTEXT** dans leur fichier Config.in, indiquant dans un commentaire ci-dessus qu'il s'agit uniquement d'une dépendance d'exécution.
- ne pas ajouter de dépendance gettext dans la variable **DEPENDENCIES** de leur fichier .mk.

Trucs et astuces

Nom du package, nom de l'entrée de configuration et relation de la variable makefile

Dans Buildroot, il existe une relation entre :

- le nom du package, qui est le nom du répertoire du package (et le nom du fichier *.mk) ;
- le nom de l'entrée de configuration déclarée dans le fichier Config.in ;
- le préfixe de la variable makefile.

Il est obligatoire de maintenir une cohérence entre ces éléments, en utilisant les règles suivantes :

- le répertoire du package et le nom *.mk sont le nom du package lui-même (par exemple : **package/foo-bar_boo/foo-bar_boo.mk**) ;
- le nom de la cible de création est le nom du paquet lui-même (par exemple : **foo-bar_boo**) ;
- l'entrée de configuration est le nom du package en majuscules avec . et -caractères remplacés par _, préfixés par **BR2_PACKAGE_** (ex. : **BR2_PACKAGE_FOO_BAR_BOO**) ;
- le préfixe de la variable de fichier *.mk est le nom du package en majuscules avec . et -caractères remplacés par _ (ex : **FOO_BAR_BOO_VERSION**).

Comment vérifier le style de codage

Buildroot fournit un script dans utils/check-package qui vérifie les fichiers nouveaux ou modifiés pour le style de codage. Ce n'est pas un validateur de langage complet, mais il détecte de nombreuses erreurs courantes. Il est destiné à s'exécuter dans les fichiers réels qu'on crée ou modifie, avant de

créer le correctif à soumettre.

Ce script peut être utilisé pour les packages, les makefiles du système de fichiers, les fichiers Config.in, etc. Il ne vérifie pas les fichiers définissant les infrastructures des packages et certains autres fichiers contenant du code commun similaire.

Pour l'utiliser, exécuter le script check-package, en indiquant quels fichiers sont créés ou modifiés :

```
$ ./utils/check-package package/new-package/*
```

Si on a le répertoire utils dans le chemin, on peut également exécuter :

```
$ cd package/new-package/
$ check-package *
```

L'outil peut également être utilisé pour les packages dans un br2-external :

```
$ check-package -b /path/to/br2-ext-tree/package/my-package/*
```

Comment tester un package

Une fois qu'on a ajouté un nouveau package, il est important de le tester dans différentes conditions : est-ce qu'il est compatible avec toutes les architectures ? Est-ce qu'il se construit avec les différentes bibliothèques C ? A-t-il besoin de fils, NPTL ? Etc...

Buildroot exécute des constructeurs automatiques qui testent en permanence des configurations aléatoires. Cependant, ceux-ci ne construisent que la branche principale de l'arborescence git, et le nouveau package sophistiqué n'est pas encore là.

Buildroot fournit un script dans utils/test-pkg qui utilise les mêmes configurations de base que celles utilisées par les autobuilders afin qu'on puisse tester le package dans les mêmes conditions.

Tout d'abord, créer un extrait de configuration contenant toutes les options nécessaires pour activer le package, mais sans aucune option d'architecture ou de chaîne d'outils. Par exemple, créons un extrait de configuration qui active simplement libcurl, sans aucun backend TLS :

```
$ cat libcurl.config
BR2_PACKAGE_LIBCURL=y
```

Si le package a besoin de plus d'options de configuration, on peut les ajouter à l'extrait de configuration. Par exemple, voici comment tester libcurl avec openssl en tant que backend TLS et le programme curl :

```
$ cat libcurl.config
BR2_PACKAGE_LIBCURL=y
BR2_PACKAGE_LIBCURL_CURL=y
```

```
BR2_PACKAGE_OPENSSL=y
```

Exécuter ensuite le script test-pkg, en lui indiquant quel extrait de configuration utiliser et quel package tester :

```
$ ./utils/test-pkg -c libcurl.config -p libcurl
```

Par défaut, test-pkg construira le package par rapport à un sous-ensemble des chaînes d'outils utilisées par les autobuilders, qui a été sélectionné par les développeurs Buildroot comme étant le sous-ensemble le plus utile et le plus représentatif. Si on veut tester toutes les chaînes d'outils, passer l'option -a (dans tous les cas, les chaînes d'outils internes sont exclues car elles prennent trop de temps à construire).

La sortie répertorie toutes les chaînes d'outils testées et le résultat correspondant (extrait, les résultats sont faux) :

```
$ ./utils/test-pkg -c libcurl.config -p libcurl
                armv5-ctng-linux-gnueabi [ 1/11]: OK
              armv7-ctng-linux-gnueabihf [ 2/11]: OK
                        br-aarch64-glibc [ 3/11]: SKIPPED
                           br-arcle-hs38 [ 4/11]: SKIPPED
                            br-arm-basic [ 5/11]: FAILED
                    br-arm-cortex-a9-glibc [ 6/11]: OK
                     br-arm-cortex-a9-musl [ 7/11]: FAILED
                     br-arm-cortex-m4-full [ 8/11]: OK
                               br-arm-full [ 9/11]: OK
                      br-arm-full-nothread [10/11]: FAILED
                        br-arm-full-static [11/11]: OK
11 builds, 2 skipped, 2 build failed, 1 legal-info failed
```

Les résultats signifient :

- **OK** : la compilation a réussi.
- **SKIPPED** : une ou plusieurs options de configuration répertoriées dans l'extrait de configuration n'étaient pas présentes dans la configuration finale. Cela est dû à des options ayant des dépendances non satisfaites par la chaîne d'outils, comme par exemple un package qui dépend de BR2_USE_MMU avec une chaîne d'outils noMMU. Les options manquantes sont signalées dans missing.config dans le répertoire de construction de sortie (~/br-test-pkg/TOOLCHAIN_NAME/ par défaut).
- **FAILED**: la construction a échoué. Inspecter le fichier journal dans le répertoire de génération de sortie pour voir ce qui n'a pas fonctionné :
  - la construction réelle a échoué,
  - les informations légales ont échoué,
  - l'une des étapes préliminaires (téléchargement du fichier de configuration, application de la configuration, exécution de dirclean pour le package) a échoué.

En cas d'échec, on peut simplement réexécuter le script avec les mêmes options (après avoir corrigé le package) ; le script tentera de reconstruire le package spécifié avec -p pour toutes les chaînes d'outils, sans avoir besoin de reconstruire toutes les dépendances de ce package.

Le script test-pkg accepte quelques options, pour lesquelles on peut obtenir de l'aide en exécutant :

```
$ ./utils/test-pkg -h
```

Comment ajouter un package depuis GitHub

Les packages sur GitHub n'ont souvent pas de zone de téléchargement avec des archives de publication. Cependant, il est possible de télécharger des tarballs directement depuis le dépôt sur GitHub. Comme GitHub est connu pour avoir modifié les mécanismes de téléchargement dans le passé, la fonction d'assistance github doit être utilisée comme indiqué ci-dessous.

```
# Utilise une balise ou un ID de validation complet
FOO_VERSION = 1.0
FOO_SITE = $(appel github,<utilisateur>,<paquet>,v$(FOO_VERSION))
```

* Le **FOO_VERSION** peut être une balise ou un ID de validation.
* Le nom de l'archive généré par github correspond à celui par défaut de Buildroot (par exemple : foo-f6fb6654af62045239caed5950bc6c7971965e60.tar.gz), il n'est donc pas nécessaire de le spécifier dans le fichier .mk.
* Lorsqu'on utilise un ID de validation comme version, il faut utiliser les 40 caractères hexadécimaux complets.
* Lorsque la balise contient un préfixe tel que v dans la version 1.0, la variable VERSION doit contenir uniquement 1,0 et le v doit être ajouté directement dans la variable SITE, comme illustré ci-dessus. Cela garantit que la valeur de la variable VERSION peut être utilisée pour correspondre aux résultats de release-monitoring.org.

Si le paquet qu'on souhaite ajouter a une section de publication sur GitHub, le responsable peut avoir téléchargé une archive tar de version, ou la version peut simplement pointer vers l'archive tar générée automatiquement à partir de la balise git. S'il y a une archive de version téléchargée par le responsable, buildroot préférons l'utiliser car elle peut être légèrement différente (par exemple, elle contient un script de configuration, buildroot n'a donc pas besoin de faire AUTORECONF).

on peut voir sur la page de publication s'il s'agit d'une archive tar téléchargée ou d'une balise git :

- elle a été téléchargée par le responsable et on doit utiliser ce lien (dans cet exemple : mongrel2-v1.9.2.tar.bz2) pour spécifier **FOO_SITE**, et ne pas utiliser l'assistant github.
- D'un autre côté, s'il n'y a que le lien “Code source”, il s'agit d'une archive tar générée automatiquement et on doit utiliser la fonction d'assistance github.

Comment ajouter un package depuis Gitlab

De la même manière que la macro github décrite dans la Section « Comment ajouter un paquet depuis GitHub », Buildroot fournit également la macro gitlab à télécharger depuis les référentiels Gitlab. Il peut être utilisé pour télécharger des archives tar générées automatiquement et produites

par Gitlab, que ce soit pour des balises ou des commits spécifiques :

```
# Utilise une balise ou un ID de validation complet
FOO_VERSION = 1.0
FOO_SITE = $(call gitlab,<user>,<package>,v$(FOO_VERSION))
```

Par défaut, il utilisera une archive .tar.gz, mais Gitlab fournit également des archives .tar.bz2, donc en ajoutant une variable <pkg>_SOURCE, cette archive .tar.bz2 peut être utilisée :

```
# Utilise une balise ou un ID de validation complet
FOO_VERSION = 1.0
FOO_SITE = $(call gitlab,<user>,<package>,v$(FOO_VERSION))
FOO_SOURCE = foo-$(FOO_VERSION).tar.bz2
```

S'il existe une archive tar spécifique téléchargée par les développeurs en amont dans https://gitlab.com/<project>/releases/, ne pas utiliser pas cette macro, mais utiliser plutôt directement le lien vers l'archive tar.

Conclusion

Comme on peut le constater, l'ajout d'un package logiciel à Buildroot consiste simplement à écrire un Makefile à l'aide d'un exemple existant et à le modifier en fonction du processus de compilation requis par le package.

Lorsqu'on empaquete un logiciel qui pourrait être utile à d'autres personnes, ne pas oublier pas d'envoyer un correctif à la liste de diffusion Buildroot (voir Section 22.5, « Soumettre des correctifs ») !

1)
utilisés pour les opérations atomiques, ils sont disponibles en variantes fonctionnant sur 1 octet, 2 octets, 4 octets et 8 octets. Étant donné que différentes architectures prennent en charge les opérations atomiques sur différentes tailles, un symbole de dépendance est disponible pour chaque taille

2) , 3) , 4)
remplacer X_Y par la version appropriée