PiLFS: Beyond Linux From Scratch

Table of Contents

Cet article présente les recettes en complément de Beyond Linux From Scratch permettent d'étendre le système.

Les scripts de démarrage PiLFS

L'archive tar PiLFS Bootscripts est une petite collection de scripts et de correctifs provenant de diverses sources spécifiques au Raspberry Pi. Les scripts suivants peuvent être installés :

- **networkfix**: Ce petit correctif indique au noyau de garder 8 Mo de RAM libres à tout moment pour le trafic réseau entrant. Sans ce correctif, il est facile de suspendre le Pi lorsqu'on travaille beaucoup sur SSH. Tout ce qu'il fait est d'ajouter `vm.min_free_kbytes=8192` à `/etc/sysctl.conf`
- **swapfix**: Le script d'initialisation d'échange LFS d'origine essaiera d'activer l'espace d'échange avant que le système de fichiers racine ne soit correctement monté. Cela ne fonctionnera pas si vous utilisez un fichier d'échange au lieu d'une partition d'échange dédiée. Ce correctif ajuste simplement le timing de ce script d'initialisation.
- **fake-hwclock**: Le Pi n'a pas de moyen de garder le temps entre les redémarrages. Ce script

ajoute une fonction de sauvegarde/restauration pour définir une date approximative jusqu'à ce que la date exacte puisse être obtenue via NTP.
- **rngd**: Script de démarrage/arrêt du démon générateur de nombres aléatoires à utiliser avec le périphérique RNG matériel présent sur le Pi.
- **switch-cpu-governor**: Fait passer le gouverneur **cpufreq** de “économie d'énergie” par défaut à “à la demande”, ce qui permet d'overclocker le Pi.
- **sshd-keygen**: Modification simple du script d'initialisation BLFS sshd qui générera de nouvelles clés si elles sont manquantes.
- **fanshim**: Fournit un contrôle automatique du Fan SHIM sur Raspberry Pi 4.
- **rc.local**: Un script d'initialisation vide dans lequel on peut ajouter ses propres applications à démarrer automatiquement.

Ajout d'un fichier d'échange

Pour créer un fichier d'échange où count est le nombre de mégaoctets que vous souhaitez :

```
dd if=/dev/zero of=/swapfile bs=1M count=512
mkswap /swapfile
swapon -v /swapfile
```

Pour activer le fichier d'échange au démarrage, ajoutez une ligne à /etc/fstab

```
/swapfile swap swap pri=1 0 0
```

Utiliser sur le générateur de nombres aléatoires du matériel Pi

La puce BCM2835 sur le Pi contient une source d'entropie matérielle qui peut être exploitée par des applications crypto-lourdes pour améliorer la qualité du caractère aléatoire et pourrait même aider à accélérer certaines opérations.

Il faut construire **rngd** qui à son tour alimentera les applications avec une qualité aléatoire à partir du RNG matériel :

```
./autogen.sh
./configure --prefix=/usr
make
make install
```

`make install-rngd` à partir des scripts d'amorçage PiLFS s'occupe de démarrer le démon.

Overclocker le Pi

Le 19 septembre 2012, la Fondation Raspberry Pi a introduit un “mode turbo” Pi, qui active dynamiquement l'overclock et l'overvolt sous le contrôle d'un pilote cpufreq, sans affecter votre garantie.

La première étape consiste à ajouter un ensemble de valeurs d'overclocking à votre /boot/config.txt. Voici les valeurs prédéfinies offertes par la configuration raspi de Raspbian :

Variable	Pas de OC	Modeste	Moyen	Élevé	Turbo
arm_freq	700	800	900	950	1000
core_freq	250	250	250	250	500
sdram__freq	400	400	450	450	600
sur_tension	0	0	2	6	6

La dernière étape consiste à faire passer le gouverneur cpufreq de la valeur par défaut « powersave » à « ondemand » avec cette commande :

```
echo "ondemand" > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

`make install-switch-cpu-governor` à partir des scripts de démarrage PiLFS s'occupe de cette étape au démarrage.

On peut garder un œil sur la fréquence et la température actuelles avec ces commandes :On peut

```
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
/opt/vc/bin/vcgencmd measure_temp
```

Redimensionner la partition ext4

Lorsqu'on écrit une image sur la carte SD, on souhaite généralement redimensionner la partition de données pour l'adapter à l'espace libre disponible sur la carte.

- Démarrer fdisk `fdisk /dev/mmcblk0` et appuyer sur **p** pour afficher la table de partition
- Noter le secteur de démarrage (540672 sur l'image de base PiLFS) de la partition Linux.
- Appuyer maintenant sur **d** et **2** pour le supprimer.
- Appuyer ensuite sur **n**, **p**, **2**, le secteur Start, **enter** et **w** (Cela recrée la partition ext4 avec la taille maximale disponible).
- Enfin, redémarrer Pi, puis effectuer le redimensionnement réel : `resize2fs /dev/mmcblk0p2`

Remplacement Memcpy/Memset optimisé

Ces versions accélérées par ARM des fonctions sélectionnées de **string.h** remplacent les routines par défaut de la bibliothèque glibc. Pour les utiliser, récupérer une copie de ce référentiel github :

```
wget https://github.com/bavison/arm-mem/archive/master.tar.gz -O arm-mem.tar.gz
```

Ensuite, les compiler dans une bibliothèque partagée, la mettre en place et veiller à ce qu'elle soit utilisée en remplacement des fonctions de la glibc.

Pour Raspberry Pi 1 & 2:

```
make
cp -v libarmmem-v6l.so /usr/lib
echo "/usr/lib/libarmmem-v6l.so" >> /etc/ld.so.preload
```

Pour Raspberry 3 & 4:

```
make
cp -v libarmmem-v7l.so /usr/lib
echo "/usr/lib/libarmmem-v7l.so" >> /etc/ld.so.preload
```

Compilation du noyau

Commencer par récupérer les dernières sources du noyau:

```
wget https://github.com/raspberrypi/linux/archive/rpi-5.10.y.tar.gz
```

Après le déballage, vérifier que les sources sont en parfait état :

```
make mrproper
```

Exécuter la commande sed suivante pour que le noyau apparaisse exactement comme le noyau officiel (avec une chaîne de version 5.10.X+) :

```
sed -i 's/EXTRAVERSION =.*/EXTRAVERSION = +/'Makefile
```

Créer un fichier .config de noyau par défaut pour votre modèle Raspberry Pi:

- Pour Raspberry Pi 1: `make bcmrpi_defconfig`
- Pour Raspberry Pi 2 & 3: `make bcm2709_defconfig`
- Pour Raspberry Pi 4 : `make bcm2711_defconfig`

On peut également personnaliser la configuration du noyau :

`make menuconfig`

On peut maintenant construire le noyau, puis installer les modules du noyau :

```
make
make zImage dtbs modules_install
cp arch/arm/boot/dts/*.dtb /boot/
cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
```

Enfin, copier le noyau compressé du modèle Pi sur la partition de démarrage:

- Pour Raspberry Pi 1 : `cp -v arch/arm/boot/zImage /boot/kernel.img`
- Pour Raspberry Pi 2 & 3: `cp -v arch/arm/boot/zImage /boot/kernel7.img`
- Pour Raspberry Pi 4 : `cp -v arch/arm/boot/zImage /boot/kernel7l.img`

Compilation de modules de noyau externes

On peut simplement compiler un module de noyau tiers compatible avec le dernier noyau officiel de Raspberry Pi. Par exemple pour compiler le module DirectFB Fusion sans avoir à construire un noyau complet.

Il faut récupérer les sources complètes du noyau:

```
wget https://github.com/raspberrypi/linux/archive/rpi-5.10.y.tar.gz
```

Après le déballage, s'assurer que les sources sont en parfait état :

```
make mrproper
```

Exécuter la commande sed suivante pour que le module se retrouve au bon endroit (/lib/modules/5.10.X+) :

```
sed -i 's/EXTRAVERSION =.*/EXTRAVERSION = +/' Makefile
```

Créer un fichier `.config` de noyau par défaut pour le modèle Raspberry Pi:

- Pour Raspberry Pi 1: `make bcmrpi_defconfig`
- Pour Raspberry Pi 2 & 3: `make bcm2709_defconfig`
- Pour Raspberry Pi 4: `make bcm2711_defconfig`

Il faut maintenant récupérer un fichier important pour le modèle Pi à partir du dépôt du micrologiciel et le placer à la racine de l'arborescence source:

- Pour Raspberry Pi 1: `wget https://github.com/raspberrypi/firmware/raw/master/extra/Module.symvers`
- Pour Raspberry Pi 2 & 3: `wget https://github.com/raspberrypi/firmware/raw/master/extra/Module7.symvers`
- Pour Raspberry Pi 4: `wget`

```
https://github.com/raspberrypi/firmware/raw/master/extra/Module7l.symvers
```

Ensuite, utiliser la commande qui compilera tous les petits morceaux de noyau supplémentaires nécessaires à la construction de modules :

```
make modules_prepare
```

Et la dernière étape nécessaire est un lien symbolique de construction pointant vers les sources du noyau :

```
ln -sv $PWD /lib/modules/5.10.X+/build
```

On doit maintenant pouvoir compiler le module de noyau tiers. Il faut actualiser lee fichier `modules.dep` avant de modifier le nouveau module, comme ceci :

```
depmod
```

Construction des modules de l'espace utilisateur

Les bibliothèques côté ARM utilisées sur Raspberry Pi sont généralement installés dans `/opt/vc/lib` et incluent la source du code côté ARM à interfacer avec : **EGL**, **mmal**, **GLESv2**, **vcos**, **openmaxil**, **vchiq_arm**, **bcm_host**, **WFC**, **OpenVG**.

Broadcom a ouvert les bibliothèques de l'espace utilisateur ARM, on peut remplacer le contenu de `/opt/vc` du référentiel de firmware par une autre version. Donc bien que l'espace utilisateur 64 bits ne soit pas officiellement pris en charge, certaines bibliothèques peuvent être compilées avec cmake pour effectuer la construction.

Récpérer les sources de l'espace utilisateur :

```
wget https://github.com/raspberrypi/userland/archive/master.tar.gz -O
userland.tar.gz
```

Et maintenant, construire et installer (renommer/sauvegarder /opt/vc actuel avant l'installation) :

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
make install
```

Créer les exemples `hello_pi` inclus pour vérifier que les nouvelles bibliothèques fonctionnent correctement :

```
cd /opt/vc/src/hello_pi
chmod +x rebuild.sh
./rebuild.sh
```

hello_font nécessite FreeType pour être construit.

Construction de OMXPlayer

Le lecteur vidéo **OMXplayer** permet d'exploiter les performance vidéo du Pi et peut être utilisé depuis la console.

Installer d'abord les prérequis suivants (dans l'ordre) :

- ALSA-lib
- ALSA-utils
- FreeType
- FFmpeg (ajouter –disable-muxer=hls pour contourner un bogue lors de la compilation sur RPi 3)
- ALSA-plugins
- Boost
- PCRE
- D-Bus

Télécharger la dernière source **OMXPlayer** à partir du référentiel github :

```
wget https://github.com/popcornmix/omxplayer/archive/master.tar.gz -O
omxplayer.tar.gz
```

Il faut adapter Makefile:

```
sed -i '/include Makefile.include/d' Makefile
sed -i 's:INCLUDES+=*:&-I/opt/vc/include -
I/opt/vc/include/interface/vcos/pthreads -
I/opt/vc/include/interface/vmcs_host/linux -I/usr/include/freetype2 -
I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include :' Makefile
sed -i 's:LDFLAGS+=*:&-L/opt/vc/lib :' Makefile
sed -i 's/$(STRIP)/strip/' Makefile
sed -i '/cp -a ffmpeg_compiled/d' Makefile
```

Maintenant, on peut construire et copier le binaire:

```
make
cp -v omxplayer.bin /usr/bin/omxplayer
cp -v omxplayer.1 /usr/share/man/man1
```

OMXPlayer est livré avec une police gratuite à utiliser pour les sous-titres, il faut la placer où il s'attend à ce qu'elle soit :

```
mkdir -p /usr/share/fonts/truetype/freefont
cp -v fonts/FreeSans* /usr/share/fonts/truetype/freefont
```

Parfois, la console d'arrière-plan brille pendant la lecture d'une vidéo, ou tout aussi ennuyeuse, reste noire après la lecture d'une vidéo. Cela peut être corrigé avec l'utilitaire **fbset** et une simple fonction wrapper dans le fichier `~/.profile`:

```
function omxplay() { omxplayer -r -t1 □□"$@" ; fbset -depth 8 && fbset -
depth 16 ;}
```

Construction de SDL

SDL (**S**imple **D**irectMedia **L**ayer) est une Bibliothèque logicielle permettant de développer des programmes gérant le son, la vidéo, le clavier, la souris et le lecteur CD.

Pour SDL, les seuls prérequis sont ALSA-lib et ALSA-utils (pour pouvoir régler le volume avec alsamixer). Construisons!

```
wget https://github.com/vanfanel/SDL12-kms-dispmanx/archive/master.tar.gz -O
SDL12-dispmanx.tar.gz
sed -i 's:DISPMANX_INCLUDES="*:&-I/opt/vc/include/interface/vmcs_host/linux
:' configure
./configure --prefix=/usr --disable-static --enable-video-dispmanx --
disable-video-kms --disable-video-fbcon --disable-video-x11 --disable-video-
directfb
make
make install
```

C'est tout ce qu'il y a vraiment à faire, mais il faut sassurer que l'application SDL génère une petite résolution (320×240) sans effets de mise à l'échelle fantaisistes, etc., de sorte que lorsqu'on démarre l'application, on doit voir quelque chose comme ceci :

```
dispmanx: Opening display[0]...
Using internal program mode: width=320 height=240
Using physical mode: 1920 x 1080 16 bpp
```

Construction de Kodi (XBMC)

Commençons par les dépendances nécessaires à la construction de Kodi.

Tout d'abord, on a besoin d'un runtime Java pour créer les fichiers d'interface. Java n'est pas requis

pour exécuter Kodi, il n'est utilisé que pendant la construction. Sr la page Java SE Embedded Downloads récupérer ejdk-8u65-linux-arm-sflt.gz.

Décompresser et placer le chemin de l'exécutable Java dans le PATH :

```
cp -rv ejdk1.8.0_65/linux_arm_sflt/jre /opt/java
export PATH="$PATH:/opt/java/bin"
```

Il faut installer les dépendances suivantes :

- Boost
- SWIG
- ALSA-lib
- ALSA-utils
- CURL
- GnuTLS
- libXML
- libXSLT
- FreeType
- Fontconfig
- FriBidi
- libMPEG2
- libMAD
- libJPEG
- libSamplerate
- libOGG
- libVorbis
- FLAC
- libTIFF
- LZO
- CMake
- Which
- Zip
- UnZip
- SQLite
- libPNG
- PCRE
- Jaspe
- Git
- SDL (uniquement nécessaire pour le packer de texture. Ne pas –disable-sdl-dlopen)
- SDL-Image
- libASS
- libcdio
- Taglib
- libModPlug

On a également besoin de **yajl**, qui compile comme ceci :

```
mkdir build
cd build
```

```
cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_BUILD_TYPE=Release ..
make
make install
```

Et enfin TinyXML qui a besoin d'un patch pour produire une librairie partagée :

```
patch -Np1 -i ../tinyxml_2_6_2-shared_lib.patch
make
make install
```

Ce sont les dépendances minimales pour la construction de **Kodi**. Il faudra peut-être ajouter des logiciels supplémentaires que **Kodi** peut utilisé, comme **Samba** et **Avahi**, etc. En particulier, installer également **libgpg-error**, **libgcrypt** et **libmicrohttpd** et supprimer `--disable-webserver` de la ligne de configuration ci-dessous, afin de pouvoir contrôler à distance **Kodi** depuis un navigateur ou un téléphone, ce qui est une fonctionnalité très pratique à avoir sur le Pi.

Si le téléviseur prend en charge CEC, créer également **libCEC** afin de pouvoir utiliser la télécommande du téléviseur pour contrôler **Kodi**:

```
wget https://github.com/Pulse-Eight/platform/archive/1.0.10.tar.gz
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_BUILD_TYPE=Release ..
make
make install
```

```
wget https://github.com/Pulse-Eight/libcec/archive/libcec-3.0.1.tar.gz
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_BUILD_TYPE=Release -
DRPI_INCLUDE_DIR=/opt/vc/include -DRPI_LIB_DIR=/opt/vc/lib ..
sed -i 's/#define LIB_INFO.*/#define LIB_INFO ("3.0.1")/'
../src/libcec/env.h
sed -i '/\\n, compiled on/d' ../src/libcec/env.h
make
make install
```

On va certainement avoir besoin d'un fichier d'échange pour créer **Kodi**, alors il faut le configurer d'abord.

Maintenant, récupérer la dernière version stable de **Kodi**:

```
wget https://github.com/xbmc/xbmc/archive/15.2-Isengard.tar.gz
```

Si on construit pour Raspberry Pi 2, appliquer ce correctif pour remplacer les spécifications cibles de

la plate-forme raspberry-pi :

```
patch -Np1 -i ../xbmc-15.2-Isengard-rpi2-target.patch
```

Mettre les sources en forme avec un bootstrap :

```
./bootstrap
```

Maintenant, on peut exécuter configure :

```
CFLAGS="-I/opt/vc/include -I/opt/vc/include/interface/vcos/pthreads -
I/opt/vc/include/interface/vmcs_host/linux" CXXFLAGS="-I/opt/vc/include -
I/opt/vc/include/interface/vcos/pthreads -
I/opt/vc/include/interface/vmcs_host/linux" LDFLAGS="-L/opt/vc/lib"
./configure --prefix=/usr --disable-debug --disable-gl --enable-gles --
disable-joystick --disable-x11 --disable-ssh --disable-samba --disable-
dvdcss --disable-avahi --disable-mysql --disable-webserver --disable-
optical-drive --with-platform=raspberry-pi --enable-player=omxplayer
```

Le script de configuration s'arrêtera à mi-chemin et créera la propre version de **Kodi** de **FFMPEG** car il ne sera pas actuellement compilé par rapport à la version officielle.

Puis fabriquer

```
make
make install
```

Ajout de la prise en charge Bluetooth intégrée

Les Raspberry Pi 3 et 4 sont équipés d'un contrôleur Bluetooth intégré.

Pour l'utiliser, il faut construire **BlueZ** - la pile de protocoles Bluetooth.

Commencer par installer ces dépendances (dans l'ordre) :

- Glib
- D-Bus
- CMake
- libical

Avant de créer **BlueZ**, il faut appliquer ce correctif qui contient divers correctifs spécifiques à Pi :

```
patch -Np1 -i ../bluez-5.40-rpi-fixes.patch
```

Il faut également ajouter `--enable-deprecated` à la ligne de configuration afin de créer quelques

binaires encore nécessaires.

Une fois **BlueZ** installé, une dernière étape est nécessaire pour faire fonctionner Bluetooth.

Pour Raspberry Pi 3 :

```
cat > /etc/bluetooth/uart.conf << "EOF"
# Start uart.conf
# Attach serial devices via UART HCI to BlueZ stack
# Use one line per device
# See the hciattach man page for options

ttyAMA0 bcm43xx 921600 noflow

# End of uart.conf
EOF
```

Pour Raspberry Pi 4 :

```
cat > /etc/bluetooth/uart.conf << "EOF"
# Start uart.conf
# Attach serial devices via UART HCI to BlueZ stack
# Use one line per device
# See the hciattach man page for options

ttyAMA0 bcm43xx 3000000 flow

# End of uart.conf
EOF
```

Après un redémarrage, le contrôleur Bluetooth devrait être opérationnel et prêt à l'action. Essayer l'appairage avec un appareil BT comme ceci :

```
bluetoothctl
power on
agent on
scan on
pair XX:XX:XX:XX:XX:XX
```

Construction de Wayland / Weston

Le bureau accéléré par GPU sur le Pi qu'on attend tous se rapproche de plus en plus chaque jour.

Ces instructions permettront d'obtenir un environnement **Wayland**/**Weston** “pur” sans dépendances X11, ce qui signifie évidemment que la couche de compatibilité **XWayland** ne sera pas construite. Si on a besoin de pouvoir exécuter des applications X dans Weston, il faut suivre le guide Xorg BLFS

pour que le serveur Xorg soit opérationnel en premier.

La première chose à faire est d'activer le pilote VC4 d'Eric Anholt dans /boot/config.txt en ajoutant ou en décommentant cette ligne :

```
dtoverlay=vc4-kms-v3d
```

Le pilote VC4 d'Eric Anholt ne fonctionne pas sur Raspberry Pi 1 ou Zero en raison de contraintes de mémoire.

L'ancien pilote **vc4-fkms-v3d** (driver Broadcom VideoCore 4 présent dans Raspberry Pi) ne sera plus pris en charge à l'avenir. Il s'appuie fortement sur des éléments spécifiques aux ordinateurs Pi et non directement pris en charge par Linux. Le nouveau pilote **vc4-kms-v3d** introduit avec Bullseye est a privilégié, mais comme il est nouveau, il y aura des ajustements nécessaires (en particulier pour les personnes qui dépendaient des anciens pilotes et logiciels fkms).

Lorsque le nouveau pilote fonctionne mal, il faut utiliser: `dtoverlay=vc4-fkms-v3d`

Une fois redémarré, il est nécessaire d'installer cette liste de dépendances (dans l'ordre) :

- libPNG
- libJPEG
- Pixman
- Glib
- libXML
- Freetype
- Fontconfig
- libdrm
- Wayland
- Wayland-Protocols

Compiler Mesa pour le pilote VC4/V3D :

```
meson --prefix=/usr --sysconfdir=/etc -Dbuildtype=release -
Dplatforms="wayland" -Dvulkan-drivers="broadcom" -Ddri-drivers="" -Dgallium-
drivers="vc4,v3d,kmsro" -Dglx=disabled build
ninja -C build && ninja -C build install
```

Et quelques autres dépendances :

- Cairo (ajouter `--enable-glesv2 --enable-egl` dans la ligne de config)
- MTdev
- XKeyboardConfig (`./configure --prefix=/usr --disable-runtime-deps`)

- libxkbcommon (`meson --prefix=/usr -Denable-docs=false -Denable-x11=false ..`)
- libevdev

Ensuite, libinput qui utilise meson et se construit comme ceci :

```
meson --prefix=/usr -Dudev-dir=/lib/udev -Ddocumentation=false -
Dlibwacom=false -Ddebug-gui=false -Dtests=false build
ninja -C build && ninja -C build install
udevadm hwdb --update
```

Construire Weston:

```
wget https://wayland.freedesktop.org/releases/weston-9.0.0.tar.xz
meson --prefix=/usr -Dbuildtype=release -Dxwayland=false -Dbackend-x11=false
-Dweston-launch=false -Dimage-webp=false -Dlauncher-logind=false -Dbackend-
drm-screencast-vaapi=false -Dbackend-rdp=false -Dcolor-management-
colord=false -Dcolor-management-lcms=false -Dsystemd=false -Dremoting=false
-Ddemo-clients=false -Dpipewire=false build
ninja -C build && ninja -C build install
```

Créer un répertoire pour les fichiers de socket et de verrouillage de **Weston**, comme ceci :

```
echo "/run/shm/wayland dir 1700 root root" >> /etc/sysconfig/createfiles
```

Et définir la variable d'environnement **XDG_RUNTIME_DIR** pour qu'elle pointe là, comme ceci :

```
echo "export XDG_RUNTIME_DIR=/run/shm/wayland" >> ~/.profile
```

Une dernière variable d'environnement pour le pilote **Vulkan**, comme ceci :

```
echo "export
VK_ICD_FILENAMES=/usr/share/vulkan/icd.d/broadcom_icd.armv7l.json" >>
~/.profile
```

Redémarrer le Pi et exécuter **weston** pour donner un bureau minimal accéléré et un terminal.

Construction de QT / Falkon

QT est une boîte à outils GUI complète, facile à construire et qui fonctionne bien dans **Wayland** et directement depuis la console.

C'est également le moyen le plus rapide d'obtenir **Falkon**, un navigateur basé sur Chromium, fonctionnant sur le Pi.

Le dernier QT est la v5.13.0, mais il ne semble pas bien fonctionner avec Wayland, il faut donc s'en tenir à la v5.12.4 pour le moment.

QT est hautement modulaire - récupèrer le grand tarball à source unique contenant tous les modules afin de construire tout de dont on a besoin.

```
wget
https://download.qt.io/archive/qt/5.12/5.12.4/single/qt-everywhere-src-5.12.4.tar.xz
```

qtbase

Tout d'abord, le module de base QT :

```
cd qtbase
./configure -prefix /opt/qt5 -opensource -confirm-license -opengl es2 -no-pch -nomake examples -no-use-gold-linker -DMESA_EGL_NO_X11_HEADERS
make
make install
```

QT sera installé dans `/opt/qt5`, il faut s'assurer que les bibliothèques et les binaires peuvent être trouvés par d'autres packages :

```
cat > /etc/ld.so.conf.d/qt5.conf << "EOF"
/opt/qt5/lib
EOF

ldconfig
```

```
export PATH="$PATH:/opt/qt5/bin"
export PKG_CONFIG_PATH="$PKG_CONFIG_PATH:/opt/qt5/lib/pkgconfig"
```

Avec les bibliothèques de base en place, on peut passer à la construction de tous les modules QT nécessaires pour faire fonctionner **Falkon** sous **Wayland**:

```
cd qtdeclarative && qmake && make && make install && cd ..
```

```
cd qtwebchannel && qmake && make && make install && cd ..
```

```
cd qtwayland && qmake && make && make install && cd ..
```

```
cd qttools/src/designer && qmake && make && make install && cd ..
```

```
cd qttools/src/linguist && qmake && make && make install && cd ..
```

qtwebengine

Le module du moteur Web QT a quelques dépendances :

- Python2
- D-Bus
- NSS
- Opus
- libwebp
- FFmpeg

La construction de **qtwebengine** est une affaire particulièrement gourmande en mémoire, un gros fichier d'échange est recommandé.

On peut également restreindre le nombre de tâches de compilation parallèles avec export `NINJAJOBS=1` avant la compilation.

```
cd qtwebengine && qmake -- -system-ffmpeg && make && make install
```

Quelles que soient les applications QT que l'on construira à partir de maintenant, peuvent être démarrées en mode **wayland** ou **eglfs** (console directe).

On peut spécifier le mode explicitement avec un argument à l'application - par exemple `-platform wayland` ou on peut le définir dans une variable d'environnement comme ceci :

```
echo "export QT_QPA_PLATFORM=wayland" >> ~/.profile
```

Falkon

Suivre simplement la recette **BLFS Falkon** pour construire, à une exception près pour abandonner la dépendance X :

```
cmake -DCMAKE_INSTALL_PREFIX=/usr \
      -DCMAKE_BUILD_TYPE=Release  \
      -DNO_X11=1 \
      ..
```

Pour que le navigateur utilise une police décente, ajouter la famille de polices **DejaVu**:

```
wget
```

```
http://sourceforge.net/projects/dejavu/files/dejavu/2.37/dejavu-fonts-ttf-2.37.tar.bz2
tar xvf dejavu-fonts-ttf-2.37.tar.bz2 && cd dejavu-fonts-ttf-2.37
install -v -d -m755 /usr/share/fonts/dejavu
install -v -m644 ttf/*.ttf /usr/share/fonts/dejavu
fc-cache -v /usr/share/fonts/dejavu
```