LFS, a tool to build Linux From Scratch based images

fabien.lementec@gmail.com

Contents

1 Overview **3**

2 Internals **4**

3 Software installation process **7**

1 Overview

LFS is customisable tool automating the creation of *LINUX from scratch* based disk images. It is currently implemented as a monolithic but configurable BASH script and runs on a LINUX host.

LFS has been designed and implemented with the following considerations in mind:

- the core engine source code must be easy to understand, so that the code is part of the documentation. The number of source files is limited, keeping LFS monolithic yet fully tunable,
- tunning must not require LFS source modification. Tuning is done by passing variables to and from scripts sourced or executed by the core. The exported and required variables must be well known, and all start with the *LFS_* prefix,
- common operations, such as adding a new software, must be simple and must require a minimal amount of effort,
- specific operations must be possible, for instance, installing a software in a special location. It requires the execution of a user written script. In this case, all the LFS context is exported to the script. *LFS_* variables and internal routines can be used, limiting the code redundancy.

2 Internals

This section describes how LFS sequences the target system installation procedure.

The target system is described by 3 components:

- a *board* describes the target platform, the software version and their corresponding configuration file for this platform,
- a set of *software* and packages, as well as the method to retrieve, build and install them if required by the board configuration,
- an *environment* that is used to setup the target system according to its operating context.

The main LFS script is invoked using:

```
LFS_THIS_BOARD_NAME={comex,quadmo,qseven,rpib,bbb} \
LFS_THIS_ENV_NAME={esrf,home,stick} \
$LFS_TOP_DIR/sh/do_lfs.sh
```

LFS_THIS_ENV_NAME is used to choose the operating environment, especially regarding the network configuration:

- *esrf*: ESRF typical environment,
- *home*: user specific environment,
- *stick*: USB stick, for 2 stages install.

LFS_THIS_BOARD_NAME is used to choose the target platform:

- *comex*
  - x86 kontron type10 com express modules
  - http://www.kontron.com/products/computeronmodules/com-express/
- *quadmo*
  - x86 qseven modules
  - http://www.seco.com/prods/boards/qseven-boards/quadmo747-e6xx.html
- *qseven* (should be called *qmx6*)
  - arm qseven modules
  - http://www.congatec.com/en/products/qseven/conga-qmx6.html
- *rpib*
  - raspberry pi modules
  - http://www.raspberrypi.org/
- *bbb*
  - beagle bone black modules
  - http://beagleboard.org/Products/BeagleBone+Black

LFS first checks for a list of required software. An error occurs if one of them is not found.

LFS_TOP_DIR is set to the LFS topmost directory. First, LFS reads the default global configuration in:

```
$LFS_TOP_DIR/sh/do_default_globals.sh
```

The following variables are set to default values:

```
LFS_WORK_DIR: the working directory path
LFS_TAR_DIR: the directory to retrieve tarballs
LFS_SRC_DIR: the directory to extract tarballs
LFS_TARGET_INSTALL_DIR: the target system rootfs mountpoint
LFS_HOST_INSTALL_DIR: the directory used to install host softwares
LFS_BUILD_DIR: the directory used to build softwares
LFS_CROSS_COMPILE: the toolchain compilation prefix
LFS_HOST_ARCH: the host architecture
LFS_DISK_DEV: the disk image device path
LFS_DISK_IMAGE: the disk image file path
LFS_DISK_EMPTY_SIZE: the disk empty partition size, in MB
LFS_DISK_BOOT_SIZE: the dist boot partition size, in MB
LFS_DISK_ROOT_SIZE: the disk root partition size, in MB
```

Then, LFS reads user specific global configuration in:

```
$LFS_TOP_DIR/sh/do_user_globals.sh
```

This file may override one of the previous variable, and set user specific variables such as network proxy settings ...

Then, the board configuration is sourced in:

```
$LFS_TOP_DIR/board/$LFS_THIS_BOARD_NAME/do_conf.sh
```

It is used to set partition sizes and types:

```
LFS_DISK_EMPTY_SIZE: the size of the empty partition, in MB
LFS_DISK_BOOT_SIZE: the boot partition size, in MB
LFS_DISK_BOOT_FS_TYPE: the boot filesystem type, (vfat or ext2)
LFS_DISK_ROOT_SIZE: the root partition size, in MB
LFS_DISK_ROOT_FS_TYPE= the root filesystem type, (vfat or ext2)
```

A missing or zero sized partition will not be present in the image. Note that the partitions are created in this order: empty, boot, root.

The board configuration script should also set a correct toolchain and architecture for the target platform:

```
LFS_TARGET_ARCH: the architecture type
LFS_CROSS_CROMPILE: the toolchain prefix
```

Finally, it is used to select the software versions. For instance:

```
export LFS_LINUX_VERS=3.6.11
export LFS_PCIUTILS_VERS=3.1.10
...
```

Then, the root filesystem is installed. Enabled softwares are retrieved, configured, built and installed. This process is detailed in the section *Software installation process*.

Then, the environment configuration is applied. To do so, the script:

```
$LFS_TOP_DIR/env/$LFS_THIS_ENV_NAME/do_post_rootfs.sh
```

is executed. This script handles tasks such as user creation, network setup ...

Then, the disk image is finalized and available at:

```
$LFS_DISK_IMAGE
```

It can be put into a physical disk, for instance using the LINUX *dd* command:

```
$> dd if=$LFS_DISK_IMAGE of=/dev/sdX
```

NOTE
After this command, the SDCARD must be removed and reinsert for the kernel to see the new partition layout. otherwise, trying to mount one of the /dev/sdX partition will probably fail.

3 Software installation process

The software installation process iterates over subdirectories in:

```
$LFS_TOP_DIR/soft
```

For each directory, LFS reads the file:

```
do_conf.sh
```

This file sets the following variables:

```
LFS_THIS_SOFT_IS_ENABLED: set to 1 if the software is to be installed
LFS_THIS_SOFT_DEPS: the space separated names of softwares that must
be installed before the software can be installed itself
LFS_THIS_SOFT_PATCHES: the space separated names of patches to be
applied before the sources are compiled
LFS_THIS_SOFT_URL: the url the software tarball can be retrieved at
LFS_IS_CROSS_COMPILED: set to 0 if the software is compiled for the
host system
```

NOTE
LFS_THIS_SOFT_DEPS will be replaced by *LFS_THIS_SOFT_DEP[]*, an array containing containing a list of dependencies.
LFS_THIS_SOFT_PATCHES will be replaced by *LFS_THIS_SOFT_PATCH[]*, an array containing containing a list of patches.
NOTE
LFS_THIS_SOFT_URL will be replaced by *LFS_THIS_SOFT_URL[]*, an array containing a list of possible urls.

If a given software is enabled, an url is given and the tarball does not already exist, the source tarball is retrieved from:

```
$LFS_THIS_SOFT_URL
```

The retrieved tarball is put in:

```
$LFS_TAR_DIR
```

The following schemes are supported:

- *file://*,
- *http://*,
- *https://*,
- *ftp://*,

- *git://* (not implemented),
- *svn://* (not implemented)

The following extensions are supported:

- *.tar*,
- *.tar.gz*,
- *.tgz*,
- *.tar.bz2*,
- *.xz* (not implemented)

Then, if the source does not already exist, the tarball is extracted in the directory:

```
$LFS_SRC_DIR/$LFS_THIS_SOFT_NAME
```

A build directory is created:

```
$LFS_BUILD_DIR/$LFS_THIS_SOFT_NAME
```

If the build directory already exist, the software installation process stops. Otherwise, the installation process is then divided into 3 stages:

- *pre_build*, the preparing stage,
- *build*, the build stage,
- *post_build*, the finalistion stage.

These stages can be configured by variables specified in the configuration file, or fully driven by dedicated scripts, as explained in the following sections. The reason to split the installation is to minimize the overall efforts required if a software specific operations must be performed at a particular stage. For instance, finalizing *dropbear* installation requires creating symbolic links manually, but preparing and building are automatically handled.

3.1 The pre_build stage

The *pre_build* stage does whatever is needed to prepare the install process. The variable:

```
LFS_THIS_SOFT_PRE_BUILD_METHOD={autotools,kbuild}
```

may be set to indicate a default method to use.

If *autoconf* is used, the usual sequence:

```
./bootstrap (optional)
./configure
```

is used to prepare the build process.

If *kbuild* is used, the file:

```
$LFS_TOP_DIR/board/$LFS_BOARD_NAME/$LFS_THIS_SOFT_NAME-$LFS_THIS_SOFT_VERS.config
```

is copied (linux kernel, busybox, crosstool-ng ...) .

In both cases, the arrays:

```
LFS_THIS_SOFT_PRE_BUILD_ARG[]
LFS_THIS_SOFT_PRE_BUILD_ENV[]
```

can be used to pass values to the preparing stage.

If the variable:

```
LFS_THIS_SOFT_PRE_BUILD_METHOD
```

is left empty, the script:

```
do_pre_build.sh
```

is executed if existing. Otherwise, nothing is done.

3.2 The build stage

The *build* stage mainly consists of compiling the sources. The variable:

```
LFS_THIS_SOFT_BUILD_METHOD={make}
```

may be set to indicate the compilation method. In this case, make is run and the arrays:

```
LFS_THIS_SOFT_BUILD_ARG[]
LFS_THIS_SOFT_BUILD_ENV[]
```

can be used to pass values to the process.

If the variable:

```
LFS_THIS_SOFT_BUILD_METHOD
```

is empty, the script:

```
do_build.sh
```

is executed if existing. Otherwise, nothing is done.

3.3 The post_build stage

The *post_build* stage finalizes the installation procedure. It mostly consists of installing the binaries and configuration files. The variable:

```
LFS_THIS_SOFT_POST_BUILD_METHOD={make_install}
```

may be set to indicate the installation method, in which case make install is run. In this case, the arrays:

```
LFS_THIS_SOFT_POST_BUILD_ARG[]
LFS_THIS_SOFT_POST_BUILD_ENV[]
```

can be used to pass values to the process.

If the variable:

```
LFS_THIS_SOFT_POST_BUILD_METHOD
```

is empty, the script:

```
do_post_build.sh
```

is executed if existing. Otherwise, nothing is done.