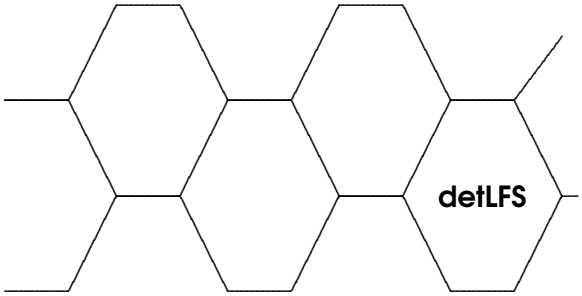

detLFS - Getting started

Thomas Dettbarn dettus@dettus.net

August 6, 2016

Contents

Chapter 1

Introduction

Hello. Welcome. Thank you for your interest in the system builder `detLFS`. I started this project, because I had a Raspberry Pi lying around, and I had to do something very specific to it at work. The distribution images offered by the raspberrypi.org homepage where good, but not what I was looking for. Other system builders like crosstool-ng, buildroot or Yocto seemed popular, but quite exhaustive. I was looking for something smaller. Something everybody could understand and modify.
So, out of my stubborness, `detLFS` was born. Its goal is to provide the advanced Raspberry Pi user with a minimalistic set of scripts to build their very own Linux system. Including a Kernel, busybox as a shell replacement and a working GCC, which is always a tedious process.

At its core is a selection of shell scripts, which are less than 150 lines long, and pretty straight forward. There are no conditions in them, no loops, just pure and utter commands. One after the other.
Understanding what the scripts do is thus not only easy, but also imperative. If you have not done so, just look at them. They are going to be explained in greater detail in chapter 3.1.

1.1 Nomenclature

Host system The scripts will require to be run on a Desktop PC. This is the host. So far, only Ubuntu 14.04 and 16.04 have been used for this, building everything under OpenBSD failed.

Target system This will be the Raspberry Pi, albeit only because of the Kernel which is downloaded via git and the bootloader. In theory, the scripts can be expanded to include other eval boards.

Chapter 2

Building a system

Building a system is as easy as 1, 2, 3a, 3b. It can be downloaded and unpacked on the commandline:

```
% wget http://www.dettus.net/detLFS/detLFS_0.04.gar.gz
% tar xvfz detLFS_0.04.gar.gz
% cd detLFS_0.04
% ls
0_getit.sh          4_mksdcard.sh      logo/
1_buildtools.sh     bsd_twoclause.txt  readme.txt
2_basesystem.sh     config_busybox     runall.sh
3a_comppackages.sh  config_kernel      skeldir/
3b_ownpackages.sh   helloworld.c
```

Note that all the files in here start with either a number or a lower case letter. This is because generated files and directories will start with upper case letters. After running the scripts, the directory looks like this:

```
% sh 0_getit.sh          #download the packages
% sh 1_buildtools.sh     #build the cross compiler
% sh 2_basesystem.sh     #for a minimalistic system
% sh 3a_comppackages.sh  #build the compilers
% sh 3b_ownpackages.sh   #build your own packages
% sudo sh 4_mksdcard.sh  #HAZARDOUS
% ls
0_getit.sh          Build/          Helloworld_shared.app
1_buildtools.sh     config_busybox  Helloworld_static.app
2_basesystem.sh     config_kernel   readme.txt
3a_comppackages.sh  Destination/    runall.sh
3b_ownpackages.sh   Downloads/      skeldir/
4_mksdcard.sh       helloworld.c    Sources/
bsd_twoclause.txt   logo/           Tools/
Mnt/
```

Which will take approximately three hours. `4_mksdcard.sh` will not run out of the box, it needs a handful of changes. It is also the one that needs root priviledges and is therefore DANGEROUS.

Once the system has been build and copied onto an SD card, you can use it. See chapter 4 for that.

2.1 Prerequisites

The build system was tested successfully on Ubuntu 14.04, as well as Ubuntu 16.04. The requirements are quite moderate: gcc-4.8.4 was installed, gawk, git, as well as Imagemagick and netpbm. Among the usual suspects were make, tar, gzip, bzip2 and xz. Not even a working cross compiler is needed, the scripts can build everything they need. Even though they are small, running them results in at least 11 Gigabytes of downloaded sources and binaries. The final system is either 64 Mbyte or 700 MBytes large. Depending on how many scripts were running. You will need to have an SD card that size. As well as a Raspberry Pi to run everything on.

2.2 Cleanup after build

After the system has been build, only `Destination/` needs to be saved. The directories `Build/`, `Downloads/`, `Mnt/`, `Sources/` and `Tools/` can go:

```
% rm -rf Build Downloads Mnt Sources Tools
% rm -rf Destination # if you want
```

2.3 Pre-build configuration

Before you want to build your system, you have to make up your mind what you want to have in it. I needed something to run on a Raspberry Pi2 with a Touchscreen attached to it and a Cherry Keryboard. And I wanted to have a fancy and colourful bootlogo, as well as a root user. For this, the directories `skeldir/` and `logo/` are important, as well as the two files `config_busybox` and `config_kernel`.

2.3.1 The hostname

The hostname can be changed simply by typing

```
% cat skeldir/etc/hostname
detlfs
% echo "newhostname" >skeldir/etc/hostname
```

2.3.2 Changing to a UART console

Since my machine was connected directly to a monitor, I did not need the serial port. This is reflected by the `cmdline.txt` in `skeldir/boot/`:

```
% cat skeldir/boot/cmdline.txt
dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2
rootfstype=ext4 elevator=deadline fsck.repair=yes
init=/sbin/init rootwait
```

Note that this is a single line. Change this to

```
% cat skeldir/boot/cmdline.txt
dwc_otg.lpm_enable=0 console=ttyAMA0,115200
root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline
fsck.repair=yes init=/sbin/init rootwait
```

That SHOULD DO THE TRICK. I HAVE NOT TRIED IT OUT YET! (sorry)

2.3.3 The boot logo

When the system is booting, it is displaying a nice little logo. For this, basically any picture with 80x80 pixels and no more than 224 colours can be used. Just overwrite the `mylogo.xpm` in `logo/`:

```
% convert WHATEVER.png -scale \!80x80 logo/mylogo.xpm.
```

It will be converted into its final format during the run of `1_buildtools.sh`.

2.3.4 Kernel and busybox configuration

Configuration of the kernel and for busybox can be performed by editing the `config_kernel` and `config_busybox`. Those files have been configured to work with my Raspberry, and the `2_basesystem.sh` will use them during its run. If you prefer to have a menu driven interface, and are not bothered by the sudden user interaction, please edit it. The following lines

```
make ARCH=arm CROSS_COMPILE=$TOOLSDIR/bin/arm-linux-gnuea
## configuration of the kernel can be done by choosing on
cat $DETLFSROOT/config_kernel | sed -e 's?CONFIG_CROSS_CO
#vimdiff .config $DETLFSROOT/config_kernel
#make ARCH=arm menuconfig
### pick one!
make ARCH=arm CROSS_COMPILE=$TOOLSDIR/bin/arm-linux-gnuea
...
make ARCH=arm CROSS_COMPILE=$TOOLSDIR/bin/arm-linux-gnuea
## configuration of busybox can be done by choosing one o
cat $DETLFSROOT/config_busybox | sed -e 's?CONFIG_CROSS_C
#vimdiff .config ../../../config_busybox
#make ARCH=arm menuconfig
### pick one!
make ARCH=arm CROSS_COMPILE=$TOOLSDIR/bin/arm-linux-gnuea
```

should become

```
make ARCH=arm CROSS_COMPILE=$TOOLSDIR/bin/arm-linux-gnuea
## configuration of the kernel can be done by choosing on
#cat $DETLFSROOT/config_kernel | sed -e 's?CONFIG_CROSS_C
#vimdiff .config $DETLFSROOT/config_kernel
make ARCH=arm menuconfig
### pick one!
make ARCH=arm CROSS_COMPILE=$TOOLSDIR/bin/arm-linux-gnuea
...
make ARCH=arm CROSS_COMPILE=$TOOLSDIR/bin/arm-linux-gnuea
## configuration of busybox can be done by choosing one o
#cat $DETLFSROOT/config_busybox | sed -e 's?CONFIG_CROSS_
#vimdiff .config ../../../config_busybox
make ARCH=arm menuconfig
### pick one!
make ARCH=arm CROSS_COMPILE=$TOOLSDIR/bin/arm-linux-gnuea
```

2.3.5 Network, System time etc.

I don't know.

2.4 Preparing the SD card

If you have tried running `4_mksdcard.sh` earlier, you might have noticed that it refused to run at all. This is because it contains a line

```
echo "aborting now." ;  exit  ## COMMENT THIS ONE OUT ONC
```

As the line says, it can be commented out once the script has been understood. Just plug in an SD card into your computer, and use dmesg to figure out which device it is.

```
% dmesg
[950012.353484] sd 11:0:0:2: [sdh] 31116288 512-byte logi
[950012.354814] sd 11:0:0:2: [sdh] No Caching mode page f
[950012.354819] sd 11:0:0:2: [sdh] Assuming drive cache:
[950012.356714] sd 11:0:0:2: [sdh] No Caching mode page f
[950012.356715] sd 11:0:0:2: [sdh] Assuming drive cache:
[950012.361979]  sdh: sdh1 sdh2
```

On my computer, it was /dev/sdh. So edit `4_mksdcard.sh`, ESPECIALLY the line where `MMCCARD` is being set:

```
export MMCCARD="/dev/sdf"
```

into /dev/sdh. Or /dev/mmcblk0 or something. Not /dev/mmcblk0p1. Not /dev/sdh2. Once you have done that AND YOU ARE SURE, comment out the exit:

```
# echo "aborting now." ;  exit  ## COMMENT THIS ONE OUT ON
```

Then you can run the script, and partition the SD card.

```
% sudo sh 4_mksdcard.sh
Command (m for help): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-31116287, default 2048): 2048
Using default value 2048
Last sector, +sectors or +size{K,M,G}: +16M
Command (m for help): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p): p
Partition number (1-4, default 2): 2
First sector (34816-31116287, default 34816): 34816
Using default value 34816
Last sector, +sectors or +size{K,M,G}: +1024M
Command (m for help): t
Partition number (1-4): 1
Hex code (type L to list codes): c
Changed system type of partition 1 to c (W95 FAT32 (LBA))
Command (m for help): p
   Device Boot  Start     End  Blocks Id  System
/dev/sdh1        2048   34815   16384  c  W95 FAT32 (LBA)
/dev/sdh2       34816 2131967 1048576 83  Linux
Command (m for help): w
```

The type for partition one is important. If it is set to anything other than Id=c, your Raspberry will not boot.

After this brief user interaction, the SD card should be finished and bootable. Try it out now!

2.5 Current packages

At the moment of writing this document, the latest version of the packages were

- binutils-2.27
- busybox-1.25.0
- gcc-6.1.0
- glibc-2.24
- gmp-6.1.1
- linux 4.4.15, raspberry pi extensions 4eda74f2dfcc8875482575c79471bde6766de3ad
- make-4.2.1
- mpc-1.0.3
- mpfr-3.1.4

The latest version can always be downloaded by editing `0_getit.sh`.

Chapter 3

The build process explained

Most of the packages are using the autoconf system to build. This offers a `configure` script, which is generating a `Makefile`. The build can be performed in a directory other than the sources, which is why the `Build/` is being created.

3.1 The scripts

When opening the scripts in a text editor, it becomes obvious that instructions for a specfic packet are being grouped within brackets.

3.1.1 `0_getit.sh`

The purpose of this script is to download the packages needed for the target linux, as well as the sources for the cross compiler. Its mode of operation is that it is using wget to download tarballs from their respective servers. Those are placed in the `Downloads/` directory, and are later extracted into the `Sources/` directory.
To ease the design of later scripts, the version number is then removed by renaming the extracted directory. The Linux Kernel is downloaded via git. This is because it is expected that the Raspberry Pi's kernel fork includes some specific patches, hence using their repository. To make sure that the config provided by `detLFS` matches the kernel, a specific revision is chosen.
After the script's completion, no more internet connection is needed.

3.1.2 `1_buildtools.sh`

This script is building the cross compiler. Building the cross compiler requires not only the kernel headers, but also the glibc. Which, in turn, requires a working cross compiler. To untie this knot of truely Gordian

proportions, the glibc is in fact being referenced 4 times in this file, the gcc twice. It starts off by performing a build which is ignoring errors, (with make -k). This will do nothing more than to install the headers into the appropriate locations. The first gcc build will create an executable which is installed into the `Tools/` directory, and can be used to properly build and install the glibc. Afterwards the gcc is build again, this time with the glibc.
The cross compiler is tested by compiling the helloworld.c file, once as `Helloworld_shared.app` and `Helloworld_static.app`.

3.1.3 `2_basesystem.sh`

This script is creating the base system, consisting of the Kernel and Busybox. Both builds offer the opportunity to configure them menu-based. Since this requires user interaction, which is breaking the scripted aspect, two predefined configuration files are used. In case user configuration is required, the script can easily be changed by commenting in the make menuconfig line. See chapter 2.3.4 for that.
The script starts by converting the `logo/mylogo.xpm` into the format which is needed so that it can be shown during the boot process. Then, the Kernel and the modules are being build, and copied into the `Destination/` directory. The same goes for busybox. Once the build has been finished, the `skeldir/` is being copied.

This concludes the build of a bootable Linux system for the Raspberry Pi.

3.1.4 `3a_comppackages.sh`

This script is building the compiler packages which can run natively on the Raspberry Pi. It is using the already created cross compiler under `Tools/` for this. Which is why this script is more straight forward than `1_buildtools.sh`. Running this script is optional.

3.1.5 `3b_ownpackages.sh`

This script's purpose is to provide an example to show how to extend the build process. It is setting up the environmental variables and performing the build of the make program again. Adding a new package can be done by using this a template.

3.1.6 `4_mksdcard.sh`

This script will create the final folder `Mnt/`. It is dangerous, since it needs to be run with root priviledge. Please see chapter 2.4 before running it.

3.2 The directorys

3.2.1 `Downloads/`

This directory holds the packages which have been downloaded from the internet.

3.2.2 `Sources/`

This directory contains the extracted sources from the packages. The version numbers have been removed, to make the build scripts easier to understand.

3.2.3 `Build/`

This directoy contains object files and binaries, as well as the temporary files during the build.

3.2.4 `Tools/`

This directoy contains the cross compiler.

3.2.5 `Destination/`

This directory will become the root filesystem on the Raspberry.

3.2.6 `Mnt/`

This is where the SD Card will be mounted.

Chapter 4

Using your system

The login is root, the password is root as well.

4.1 The init scripts

If you look at the `skeldir/etc/` directory, you will notice two files: `inittab` and `rcS`.

```
% cat skeldir/etc/inittab
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mount -o remount,rw /
null::sysinit:/bin/mkdir -p /dev/pts
null::sysinit:/bin/mkdir -p /dev/shm
null::sysinit:/bin/mount -a
null::sysinit:/bin/hostname -F /etc/hostname
::sysinit:/etc/rcS
tty1::respawn:/sbin/getty -L tty1 115200 vt100
tty2::respawn:/sbin/getty -L tty2 115200 vt100
::ctrlaltdel:/sbin/reboot
% cat skeldir/etc/rcS
#!/bin/ash
echo "hello world"
```

Busybox, in its role as init-replacement, is reading from `inittab` , which is creating a new login on console tty1 and tty2. It is also starting the script `/etc/rcS` at system init time, which is nothing more than a "hello world" example at this point.

4.2 Changing the root password

The way I created the password hash, which can be found on the host in `skeldir/etc/shadow`, was by using perl's crypt() function:

```
% perl -e 'printf("%s\n", crypt("root",".l"));'
.lsr3Kl7ExsXU
```

The first parameter is the password, the second one is the salt. Changing the password to "abcde", for example, can be done like this:

```
% perl -e 'printf("%s\n", crypt("abcde","11"));'
11BYYrrX5tm1k
% perl -e 'printf("%s\n", crypt("abcde","12"));'
12yYR42qdsGFc
```

One of those needs to be copied into `skeldir/etc/shadow` instead of `.lsr3Kl7ExsXU`. After building the system, the new password will be "abcde". Obviously, it could be changed on the target as well.