

Porting Ubuntu to DFC

Park Ju Hyung 20170224



Inha University

Table of Contents

1. Introduction	2
2. Preparing the microSD	3
3. Preparing the kernel	3
3.1. Obtaining the kernel source	3
3.2. Setting up the toolchain	3
3.3. Building the provided Ubuntu kernel	4
3.4. Setting up the kernel configuration	4
3.4.1. Setting up the kernel configuration - for Ubuntu	5
3.5. Kernel modules	6
3.6. Building the kernel	6
3.7. Booting the kernel	7
4. Preparing Ubuntu	8
5. TODOs	9

1. Introduction

Linux distribution(a.k.a. distro) such as Ubuntu is consisted of two software categories. One being the Linux kernel and the other being softwares running on top of the kernel, also known as the userspace softwares.

The Linux kernel is a monolithic kernel, which means most of the essential drivers required by the operating system are integrated to the kernel. Thanks to this monolithic design, the userspace softwares mostly don't require platform-specific tweaks as the Linux kernel already took care of those.

So it's critical to get the Linux kernel running without issues, and compatible with the distro you're trying to install.

We are going to talk about how to prepare the Linux kernel and setup the root filesystem for Ubuntu.

2. Preparing the microSD

Due to the insufficient space on the integrated MTD storage, we are going to use microSD to boot Ubuntu instead.

Use gparted to create 2 partitions on the microSD. MSDOS(MBR) partition table is recommended.

Create the first partition(which will be /dev/mmcblk0p1) as FAT32 and give it about 256~512MB. This partition will be used to store the kernel and device tree blob.

Create the second partition(which will be /dev/mmcblk0p2) as EXT4. This partition will be used as the Ubuntu's root file-system. Use "Label File System" to give this partition a name. This name will be picked up during the boot process.

3. Preparing the kernel

The first step for developing a kernel, is to build and run **without any changes**(from the vendor) before making any changes, to make sure the development environment(such as toolchain¹) is properly setup.

All you need to build a kernel, is a proper kernel source, kernel configuration and toolchain settings.

¹ Collection of softwares required to build the Linux kernel, including GCC and binutils.

3.1. Obtaining the kernel source

The Linux kernel is licensed under GNU General Public License(GPL) which makes the developers obliged to release every kernel's sources ever shipped to a product. The Linux kernel is mostly distributed under a tarball(.tar) archive or in a Git repository format. If the former method is used, you can use `tar -xf` to extract the tarball. If the latter method is used, use `git clone` to access the kernel sources. If you cannot receive the Linux kernel sources, contact the manufacturer or ask support from gpl-violations.org.

3.2. Setting up the toolchain

The Linux kernel is built using gcc² and the gcc used for the build needs to support the targeted architecture. We are targeting arm64(aarch64), so you need to download the aarch64 toolchain.

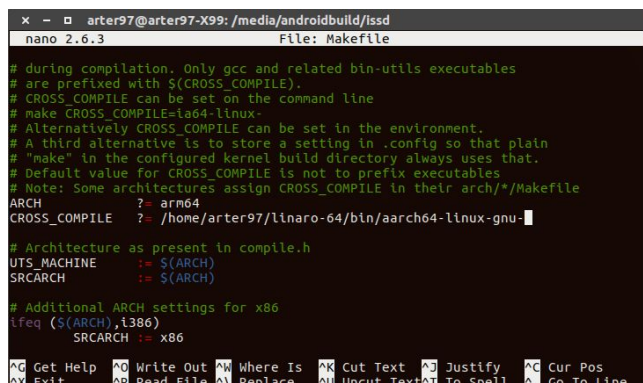
“aarch64-linux-gnu” toolchain from Linaro is recommended :

<https://releases.linaro.org/components/toolchain/binaries>

Please note that using the latest GCC version may result in build errors as warning/error detection is being improved on every GCC updates.

After extracting the toolchain, review that the extracted directory includes “bin/aarch64-linux-gnu-gcc” and “bin/aarch64-linux-gnu-ld”. Open Makefile located at the root of the Linux kernel source and find the declaration point of the “CROSS_COMPILE” variable. You must add the full, absolute path to the extracted toolchain directory and add “bin/aarch64-linux-gnu-” at the end. The individual executables such as gcc, ld and strip will be used by having the variable CROSS_COMPILE as a prefix.

The Linux kernel defaults to the host architecture to be used for building target. Since we are cross-compiling³ to arm64, we should also define the variable ARCH as arm64 as well.



² Using an alternative C compiler such as clang is considered experimental at this point.

³ Building for architecture that differs from the host architecture.
(e.g. Building for arm64 on a x86 machine is “cross-compiling”)

3.3. Building the provided Ubuntu kernel

The kernel source used in development for porting Ubuntu is available at :

<https://github.com/DFC-OpenSource/issd-kernel>

You can use this kernel source and skip *steps* 3.4 to 3.5. Proceed to *step* 3.6 after typing ``make ls2085a_issd_ubuntu_defconfig``. Please refer to *step* 3.4 for more details on this command.

3.4. Setting up the kernel configuration

The Linux kernel is the most successful collaborative project in history of humanity. To accommodate various people's needs, the Linux kernel has thousands of kernel options to customize. These kernel options includes architecture-specific options, device drivers options, userspace options, network management options, memory management options, debugging options and many more.

The kernel configuration is located in "arch/<architecture name>/configs". In our case, the kernel configuration is located at "arch/arm64/configs/ls2085a_issdconfig".

The exported kernel config, also known as "defconfig", requires to have a "_defconfig" postfix in order to be properly recognized by Makefile. So you need to rename the kernel config. The exported kernel config is a shortened version of a kernel config that contains the settings that differs from the default values. The advantage of using this method, is that distributing the kernel config is easier since the maintainer doesn't have to keep track all thousands of settings. Also, it's more portable since the default values on the Linux kernel sources are untouched. (e.g. Generating full Linux 4.9 kernel config from Linux 4.1's defconfig is easy and straightforward.)

After locating the defconfig, we need to use it to generate a full kernel config. Type ``make <defconfig name>`` to generate it. The generated kernel config will be saved to ".config".

In our case, you can type ``make ls2085a_issd_defconfig``(assuming that you've changed the defconfig filename to ls2085a_issd_defconfig).

You can also customize most of the values in the kernel config. Type ``make menuconfig`` to open up the kernel config customizer. You can get a full description for each configs(if written) and disable/enable manually, or set it to build as a module(if the module option is given).

3.4.1. Setting up the kernel configuration - for Ubuntu

(As mentioned earlier, customizing the kernel config is recommended to do *after* confirming that the kernel boots with the stock kernel config.)

Different Linux distros need different kernel configuration. For example, systemd (which is an init system managing all userspace processes) on Ubuntu requires `CONFIG_FHANDLE` to be enabled for properly setting up mountspaces, while Android requires `CONFIG_SECURITY_SELINUX` to be enabled for managing security. To ensure the best compatibility, matching the kernel config for the distro is essential.

Ubuntu kernel's configuration is obtainable by merging Ubuntu's kernel patches. Those patches are available at <http://kernel.ubuntu.com/~kernel-ppa/mainline>. (You can get the build log file too, which is useful to see which exact commands did Canonical use for building a kernel.)

After merging the kernel patches by

```
`git am /path/to/patch/file/name.patch` or  
`patch -p1 < /path/to/patch/file/name.patch`,  
you can find Ubuntu kernel configs at debian.master/config.
```

The following commands can be used in our case :

```
# This command will generate default Ubuntu arm64 kernel config  
cat \  
    debian.master/config/config.common.ubuntu \  
    debian.master/config/arm64/config.common.arm64 \  
    debian.master/config/arm64/config.flavour.generic > .config  
# This command will append our platform-specific kernel config,  
# overriding the defaults  
cat arch/arm64/configs/ls2085a_issd_defconfig >> .config
```

3.5. Kernel modules

Here is an example of a kernel config:

```
# CONFIG_SATA_AHCI is not set  
CONFIG_SATA_AHCI=y  
CONFIG_SATA_AHCI=m
```

The first one indicates that the driver will not be a part of the build.

The second one indicates that the driver will be integrated to the kernel image. Integrating the driver to the kernel image itself will make it impossible to swap it without

rebuilding the kernel. If you're in the middle of debugging a driver, building it as a module is encouraged.

The third one indicates that the driver will be built as a module, not integrated to the kernel image.

It's generally a good idea to integrate drivers which are most likely to be loaded and used anyways. By doing this, we can avoid using initramfs entirely as we don't need to ship kernel modules in the initramfs and depend on it to mount the root volume.

3.6. Building the kernel

After setting the toolchain, and generating the full kernel config to ".config", you're ready to initiate the build. Type ``make -j<CPU number>`` (e.g. ``make -j12`` on a 12-core CPU machine). Alternatively, we can also use ``make deb-pkg -j<CPU number>`` to generate debian packages for the built binaries. (We can make use of this to easily ship updated kernels to the users.)

This can take a few minutes to a few hours.

When the build is complete, you'll have the kernel image and the kernel modules(with .ko extension) in each driver's directory.

The kernel image is located at `arch/*/boot/(z)Image(.gz)`, in our case, `arch/arm64/boot/Image.gz`.

This is the final kernel image. In the desktop version of Ubuntu, this is also the one saved to `/boot/vmlinuz`. The 'z' character in "zImage" and the ".gz" extension in "Image.gz" means that the kernel image is compressed. Compressing the kernel image reduces the final binary size.

The Linux kernel loads the modules from `"/lib/modules/$(uname -r)"` by default. To install the kernel modules, you can either use the packaging method mentioned earlier or install it manually by copying `/tmp/kernel/lib/modules` to the target device's `/lib/modules` after running ``make modules_install INSTALL_MOD_PATH=/tmp/kernel``.

3.7. Booting the kernel

For our development board, the process for booting the kernel can be simplified in following steps:

1. Convert kernel image to uboot kernel image
2. Boot to uboot console (using UART and minicom)
3. Load the kernel image from the microSD card to the RAM
4. Load the device tree blob from the microSD card to the RAM
5. Execute the kernel from the RAM

You need 2 files to be present in the microSD card :
The kernel and the device tree blob⁴.

`mkimage` command is used to convert vmlinux kernel format to uboot kernel format.
In our case, the correct command is:

```
`mkimage -A arm64 -O linux -T kernel -C none -a 0x80080000 -e  
0x80080000 -n "ISSD" -d arch/arm64/boot/Image uimage.bin`
```

The output binary uimage.bin is the one used by uboot. Please note that we are using "Image" as the input file, not "Image.gz" because uboot bootloader used in our development board doesn't support gzip compressed kernel image.

You now need to use UART serial console in order to access uboot console. After connecting the serial USB to the host, use `sudo minicom -D /dev/ttyUSB0` to open UART serial console protocol. You need to reboot first to intercept uboot's booting process. uboot will count down approximately 3 seconds and ask you if you want uboot console. After entering uboot console, use `fatload mmc 0 0xa0000000 uimage.bin` to load the kernel to the memory. Following the same syntax, use `fatload mmc 0 0x80000000 issd.dtb` to load the device tree blob.

You should also set Linux command line(cmdline) at this point. cmdline tells Linux how to boot/initialize. For example, we can set which device Linux should use as a root device.

```
Type `setenv bootargs "console=ttyS0,115200 root=/dev/mmcblk0p2  
rootwait earlycon=uart8250,mmio,0x21c0500,115200" `.
```

The console, earlycon parameters tells the kernel that it should print logs to UART serial port.

The root parameter tells the kernel that it should use /dev/mmcblk0p2 as the root device. (If we want to boot from SSD, we can also do "root=/dev/sda1" as well.)

You are now ready to boot the kernel(of course, you need to install Ubuntu at /dev/mmcblk0p2 beforehand).

Type `bootm 0xa0000000 - 0x80000000` to boot the kernel.

The first parameter means to load the kernel image from 0xa0000000.

The second parameter is `-`, which means that initramfs is unused.

The third parameter means to load the device tree blob from 0x80000000.

If everything is done properly, you should see Linux booting up.

⁴ http://elinux.org/Device_Tree_What_It_Is, http://elinux.org/index.php?title=Device_Tree_Reference

4. Preparing Ubuntu

While we prepared the kernel, the rest of the userspace still needs to be configured. We can use the `debootstrap` command to prepare the root filesystem.

The following command is recommended to use (from x86 host):

```
debootstrap \
    --arch=arm64 \
    --include=aptitude,asciidoc,autogen,automake,axel,baobab,bash-completion,bc,bison,b
uild-essential,busybox,checkbox-gui,command-not-found,desktop-base,diffutils,dmz-cursor-the
me,dnsutils,doc-base,dos2unix,dosfstools,doxygen,ed,exfat-fuse,exfat-utils,ffmpeg,file,find
utils,firefox,flex,fonts-dejavu-extra,fonts-guru,fonts-guru-extra,fonts-kacst,fonts-kacst-o
ne,fonts-khmeros-core,fonts-lao,fonts-liberation,fonts-lklug-sinhala,fonts-lohit-guru,fonts
-nanum,fonts-noto-cjk,fonts-opensymbol,fonts-sil-abyssinica,fonts-sil-padauk,fonts-stix,fon
ts-symbola,fonts-takao-pgothic,fonts-thai-tlwg,fonts-tibetan-machine,fonts-tlwg-garuda,font
s-tlwg-garuda-ttf,fonts-tlwg-kinnari,fonts-tlwg-kinnari-ttf,fonts-tlwg-laksaman,fonts-tlwg-
laksaman-ttf,fonts-tlwg-loma,fonts-tlwg-loma-ttf,fonts-tlwg-mono,fonts-tlwg-mono-ttf,fonts-
tlwg-norasi,fonts-tlwg-norasi-ttf,fonts-tlwg-purisa,fonts-tlwg-purisa-ttf,fonts-tlwg-sawasde
ee,fonts-tlwg-sawasdee-ttf,fonts-tlwg-typewriter,fonts-tlwg-typewriter-ttf,fonts-tlwg-typis
t,fonts-tlwg-typist-ttf,fonts-tlwg-typo,fonts-tlwg-typo-ttf,fonts-tlwg-umpush,fonts-tlwg-um
push-ttf,fonts-tlwg-waree,fonts-tlwg-waree-ttf,ftp,gdb,gedit,gettext,grep,gvfs-bin,gzip,hdp
arm,hostname,image-magick,info,init,inputattach,iotop,iproute,iputils-arping,iw,kernel-commo
n,libcurl4-gnutls-dev,libexpat1-dev,liblz4-tool,libnotify-bin,libpcre3-dev,libxt-dev,login,
logrotate,lsof,ltrace,lxd,lzop,mediainfo,meld,nano,nautilus,ncurses-base,ncurses-bin,ncurse
s-term,nethogs,ntfs-3g,ntp,openjdk-8-jdk,optipng,p7zip-full,parallel,patchutils,pv,realpath
,rftkill,ssh,strace,subversion,tcl,texinfo,time,u-boot-tools,ubuntu-artwork,ubuntu-minimal,u
buntu-server,ubuntu-session,ubuntu-settings,ubuntu-sounds,ubuntu-system-service,unity-asset
-pool,unity-greeter,unity-scope-home,unrar-free,unzip,uuid-dev,wireless-tools,xcursor-theme
s,xdg-user-dirs-gtk,xdiagnose,xmllto,xorg,zip,zlib1g-dev \
    --components=universe,main \
    --variant=buildd \
    rootfs
```

The `arch` parameter tells the command to prepare the root filesystem for the specified architecture.

The `include` parameter tells the command to include the specified packages. The ones with “ubuntu-” prefix is essential to have a functional Ubuntu system. The rest of the packages are mostly from `ubuntu-desktop` package. The actual `ubuntu-desktop` package was excluded since we do not need `compiz` GUI support.

The `components` parameter tells the command to lookup the specified Ubuntu repositories when downloading packages.

The “`buildd`” option in the `variant` parameter gives the system basic development functions.

The final parameter is used as the output directory.

After the `debootstrap` command has finished, you can copy the entire contents in `rootfs/` to desired storage partition. Using `rsync -a` is recommended as it ensures the source and the destination directory to have the same file contents with the same metadata. (e.g. `rsync -a rootfs/ /media/arter97/microsd_root/``)

After the copying is done, you must update `/etc/fstab` to ensure partitions will be properly mounted during boot. You can use the “LABEL=” parameter with the name we set during the microSD setup⁵.

Now you should boot to newly debootstrapped Ubuntu and finish the setup. Follow the same steps in the “Booting the kernel” section, but use the following command for setting cmdline:

```
setenv bootargs "console=ttyS0,115200 root=/dev/mmcblk0p2 rootwait  
earlycon=uart8250,mmio,0x21c0500,115200 init=/bin/sh rw"
```

This will execute `/bin/sh` immediately after kernel finishes initializing, which means it'll immediately drop you to a shell.

Type ``/debootstrap/debootstrap --second-stage`` to finish bootstrapping and type ``passwd`` to set root account's password.

You can now reboot the board and start Ubuntu using the appropriate uboot commands mentioned earlier.

5. TODOs

- a) Modify uboot to make it automatically boot to microSD's Ubuntu.
- b) Test and validate Infiniband functionality.
- c) Test and validate `issd_nvme` userspace process functionality.

⁵ See <https://wiki.archlinux.org/index.php/Fstab> for more information.