

Linux Kernel Training: Lecture 1

Building the Software for BEAGLEBONE BLACK

Sam Protsenko

March 23, 2020

GlobalLogic

Agenda

1. Hardware Overview
2. Software Overview
3. Perspective on Building

Organization

Linux Kernel ProCamp Details

- Mon/Fri, 3pm - 5pm
- Schedule: [link](#)
- Target: BEAGLEBONE BLACK and QEMU
- Host: Personal laptop (Ubuntu 18.04) or Training Centre PC
- Training Centre PC:
 - Press F9 on boot (show boot menu)
 - Select second drive (TS64GSSD370S, 64 GB)
 - Login: Lin-Ker
 - Password: 123

- Oleksandr Redchuk <oleksandr.redchuk@globallogic.com>
- Sam Protsenko <semen.protsenko@globallogic.com>
- Andrii Lukin <andrii.lukin@globallogic.com>
- Vitaliy Vasylyskyy <vitaliy.vasylyskyy@globallogic.com>
- Illia Smyrnov <illia.smyrnov@globallogic.com>
- Ruslan Bilovol <ruslan.bilovol@globallogic.com>

Hardware Overview

Embedded Systems

Embedded Systems:

- Mobile
- Automotive
- Networking
- Smart TVs, game consoles, set-top boxes
- IoT
- Medical
- Aerospace
- Industry



Embedded Programming

Differences from regular system:

- Cross-compiling
- Flashing
- Serial console
- Testing concerns
- Working with hardware
- Non discoverable buses on board (device tree, platform drivers)



Kernel: Big Picture

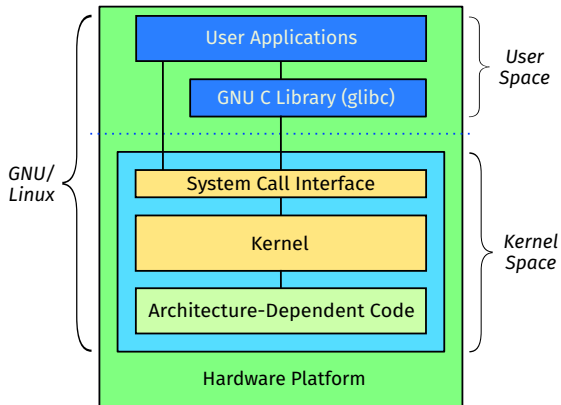


Figure 1: The fundamental architecture of the GNU/Linux operating system

ARM Cortex A/R/M Families

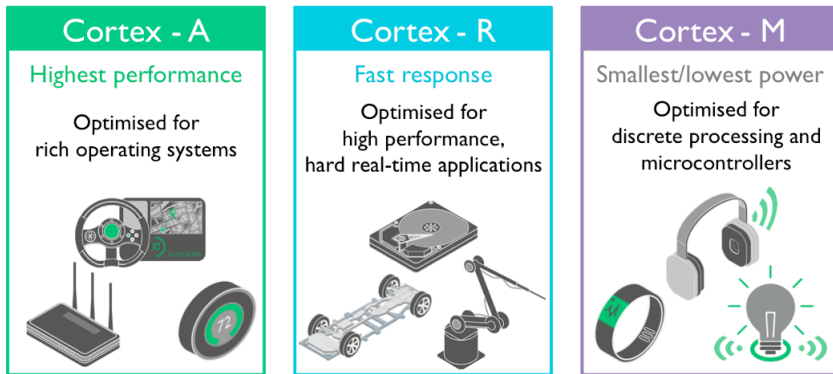


Figure 2: ARM Cortex Processor Families

AM335x SoC

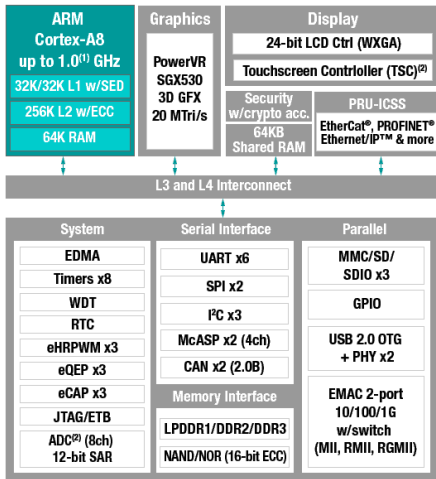


Figure 3: AM335x Functional Diagram

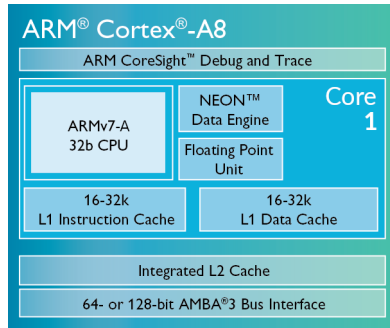
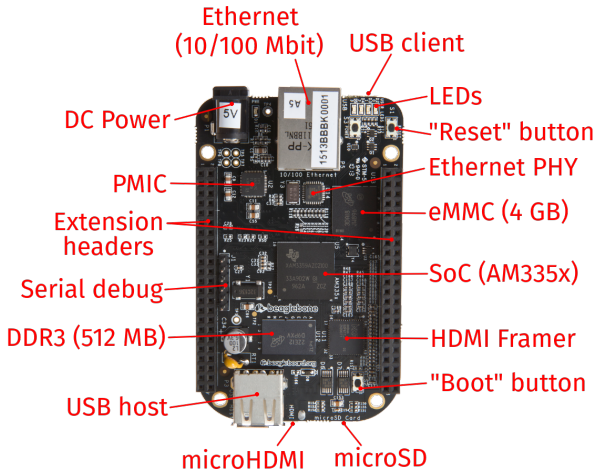


Figure 4: Cortex-A8 Processor Architecture

BeagleBone Black



BeagleBone Black: Pros and Cons

Pros:

- Open Hardware
 - Public TRM
 - Schematic
 - PCB files
- Supported in upstream
 - Kernel
 - U-Boot
- Conventional ARM architecture
- Very popular
- Low cost (\$55)

Cons:

- Old 32-bit architecture
- Single core processor
- Android is not supported officially
- No WiFi

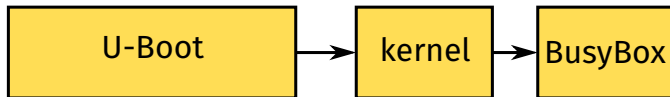
Software Overview

Software Components

- U-Boot
- Linux kernel
- BusyBox

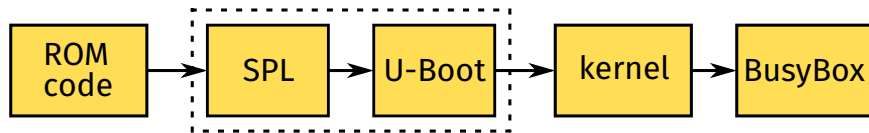
Software Components

- U-Boot
- Linux kernel
- BusyBox



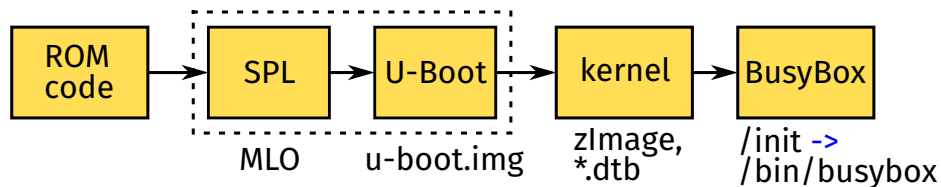
Software Components

- U-Boot
- Linux kernel
- BusyBox



Software Components

- U-Boot
- Linux kernel
- BusyBox

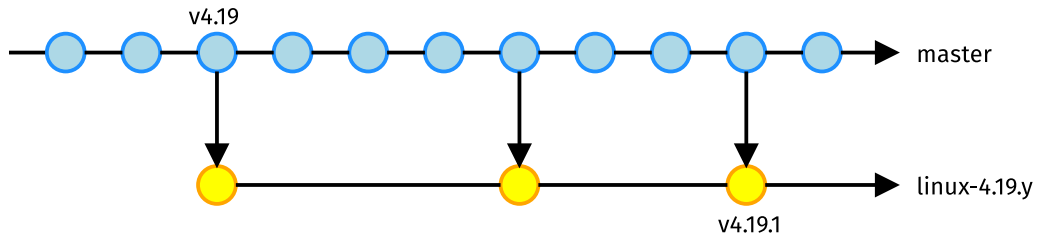


Building Steps

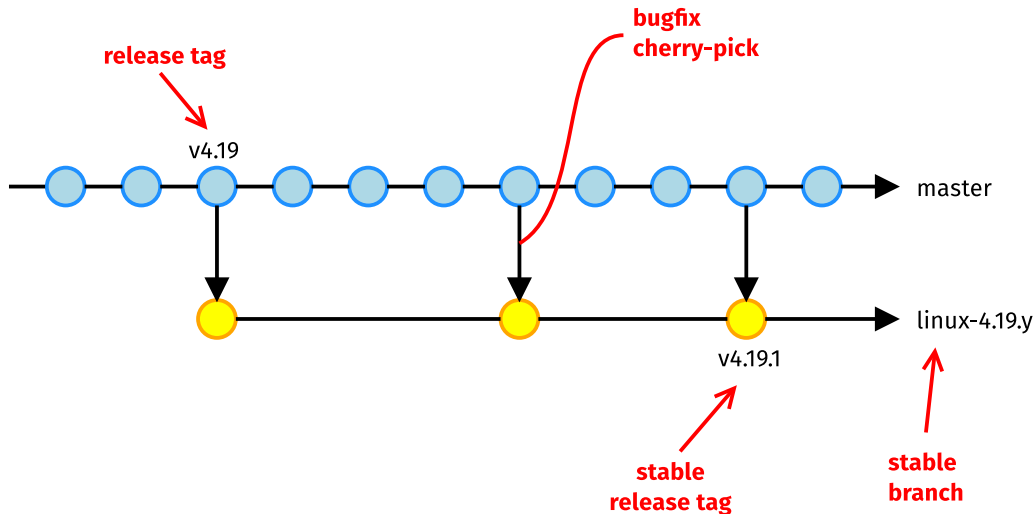
1. Obtain the software
2. Checkout to desired branch or tag
3. Consult with **README** and **INSTALL**
4. Install all build dependencies
5. Configure shell environment for cross-compiling
6. **Configure the software for build with options you desire**
7. **Build the software**
8. **Install/flash the built software**

Perspective on Building

Kernel Branching Strategy

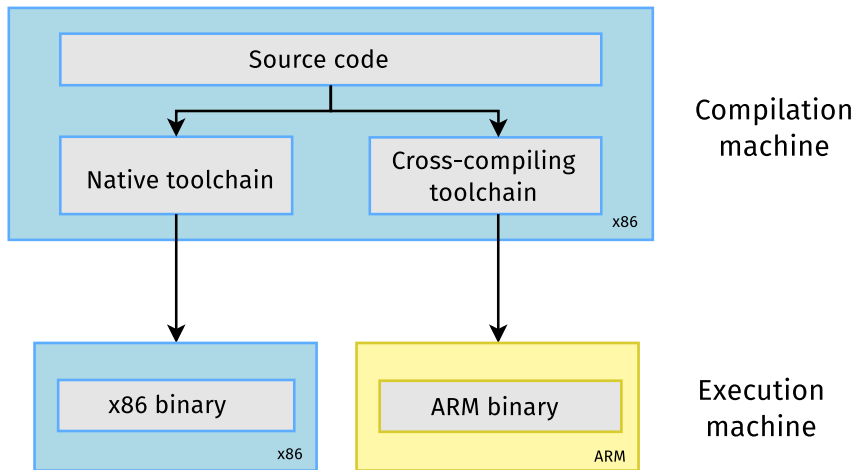


Kernel Branching Strategy



- Git tags: for releases (e.g. `v4.19`)
- Git branches: for stable releases (e.g. `linux-4.19.y`)
- Some stable branches are LTS (Long Term Support)
- When possible, let's use stable branches (for reliability)
- When stable branches are not available, let's use release tags

Toolchain (page 1)



- Set of tools for cross-compiling:
 1. `gcc`
 2. binutils: `ld`, `as`, `objdump`, `objcopy`, `readelf`, etc.
 3. `glibc` and other system libraries (optional)
 4. Linux kernel headers (optional)
 5. `gdb` (optional)

- Set of tools for cross-compiling:
 1. `gcc`
 2. binutils: `ld`, `as`, `objdump`, `objcopy`, `readelf`, etc.
 3. `glibc` and other system libraries (optional)
 4. Linux kernel headers (optional)
 5. `gdb` (optional)
- Toolchain types:
 - Bare-metal targeted (`arm-eabi`): for U-Boot and kernel
 - Linux targeted (`arm-linux-gnueabihf`): for BusyBox
- In our case: host = x86_64, target = ARMv7-A
- We use only GNU (GCC-based) toolchains

Toolchain tuple examples:

- **arm-foo-none-eabi**, bare-metal toolchain targeting the ARM architecture, from vendor *foo*
- **arm-unknown-linux-gnueabi**, Linux toolchain targeting the ARM architecture, using the EABI ABI and the glibc C library, from an *unknown* vendor
- **armeb-linux-uclibcgnueabi**, Linux toolchain targeting the ARM big-endian architecture, using the EABI ABI and the uClibc C library
- **mips-img-linux-gnu**, Linux toolchain targeting the MIPS architecture, using the glibc C library, provided by Imagination Technologies

- Regular compilation on host system (using *native toolchain*):

```
$ gcc main.c
```

- Regular compilation on host system (using *native toolchain*):

```
$ gcc main.c
```

- Cross-compilation for ARM target (part before `gcc` is called *prefix*):

```
$ /toolchain/path/bin/arm-eabi-gcc main.c
```

- Regular compilation on host system (using *native toolchain*):

```
$ gcc main.c
```

- Cross-compilation for ARM target (part before `gcc` is called *prefix*):

```
$ /toolchain/path/bin/arm-eabi-gcc main.c
```

- More universal way:

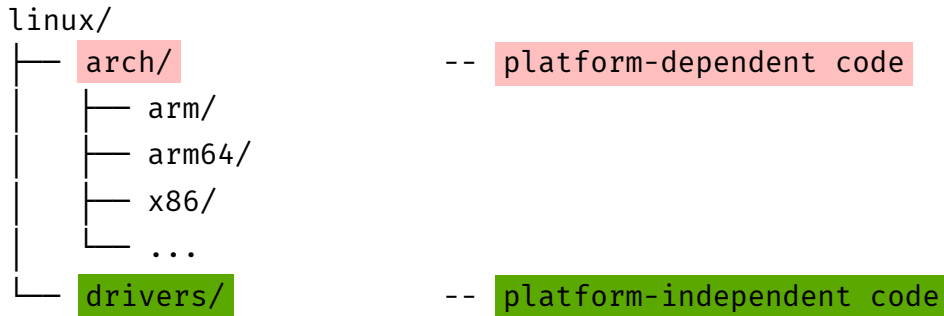
```
$ PATH=/toolchain/path/bin:$PATH
```

```
$ CROSS_COMPILE=arm-eabi-
```

```
$ ${CROSS_COMPILE}gcc main.c
```

Specifying Architecture

- Kernel supports many CPU architectures:



- We need to choose which architecture to build for:

```
$ export ARCH=arm
```

- Shell environment configuration for building U-Boot/kernel/BusyBox:

```
$ export ARCH=arm
```

```
$ export PATH=/toolchain/path/bin:$PATH
```

```
$ export CROSS_COMPILE=arm-eabi-
```

- Makefile utilizes those env vars

Take Five

Kbuild: User's Perspective

Building: General Steps

- All projects (U-Boot, Linux kernel and BusyBox) use Kbuild
- Build steps: configuration, build, installation
- **Configuration** (generate `.config` file):

```
$ make defconfig
```

- **Build:**

```
$ make
```

- **Installation:**

```
$ make install
```

Building: Custom Configuration

- Sometimes existing `defconfig` is not enough
- How can we customize our configuration?
 - Using `make menuconfig`
 - Using `merge_config.sh` script
 - Using old `.config` file

Building: Custom Configuration

- Sometimes existing `defconfig` is not enough
- How can we customize our configuration?
 - Using `make menuconfig`
 - Using `merge_config.sh` script
 - Using old `.config` file
- Example: kernel configuration using `merge_config.sh`:

```
$ ./scripts/kconfig/merge_config.sh \
  arch/arm/configs/multi_v7_defconfig \
  fragments/bbb.cfg
```

Building: .config Example

Excerpt from `.config` file:

```
CONFIG_USE_OF=y
CONFIG_DEFAULT_HOSTNAME="(none)"
CONFIG_CMDLINE=""
# CONFIG_PREEMPT is not set
CONFIG_I2C_GPIO=m
CONFIG_LOG_BUF_SHIFT=17
# CONFIG_SLAB is not set
CONFIG_USB=y
CONFIG_SND_USB_AUDIO=m
```

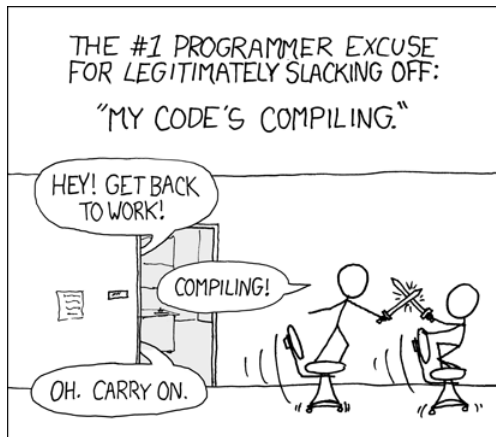
Kernel Modules

- Every driver is a module
- Kernel modules can be:
 - Loadable: “=m”
 - Built-in: “=y”
- Kernel loadable module (**.ko** file) is some code that can be loaded into kernel space (i.e. added to running kernel as a plugin)

Kernel Modules

- Every driver is a module
- Kernel modules can be:
 - Loadable: “=m”
 - Built-in: “=y”
- Kernel loadable module (**.ko** file) is some code that can be loaded into kernel space (i.e. added to running kernel as a plugin)
- How it works:
 - **multi_v7_defconfig** is common for all ARMv7 systems (so the single **zImage** can be used)
 - Device Tree file covers SoC and board differences
 - Needed modules (for particular board) can be loaded in run-time
- It's not always convenient to load a lot of modules

How to Speed-Up the Build? (page 1)



How to Speed-Up the Build? (page 2)

- **Kbuild tracks all dependencies very well!**

- *Clean build* (try to avoid it):

```
$ make  
// Do some changes to source code  
$ make distclean  
$ make
```

- Use *incremental build* instead:

```
$ make  
// Do some changes to source code  
$ make
```

How to Speed-Up the Build? (page 3)

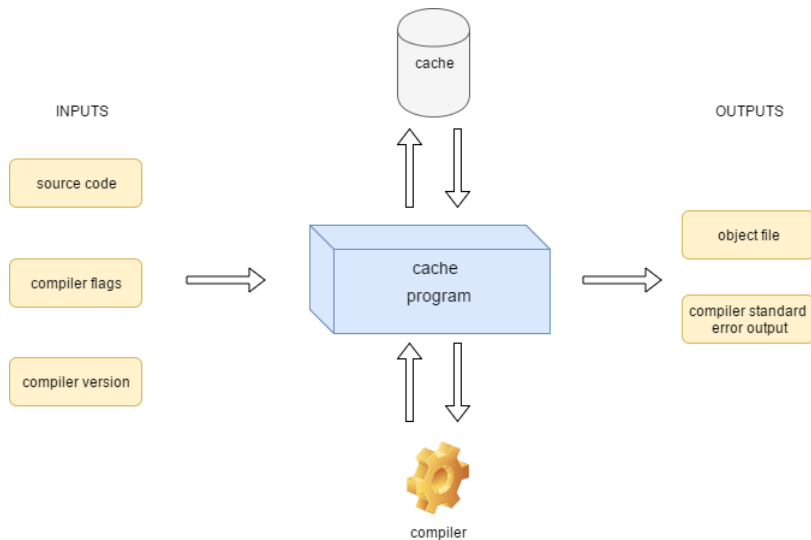
- Distribute the compilation between CPU cores using multi-threading build:

```
$ make -j4
```

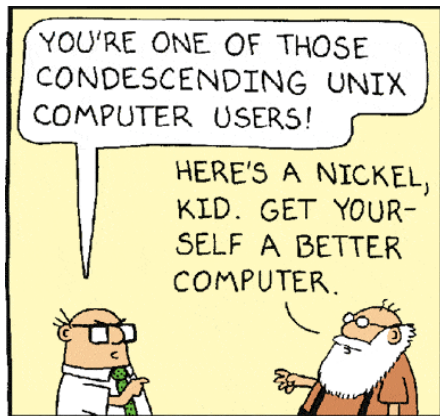
- Use **ccache** tool:
 - Caches **.o** files on the first build
 - On next build, if some **.o** files are unchanged, cached versions will be used
 - **ccache** creates a hash by **.c** file content and by build command
 - ...So if you change the toolchain, cache won't be used
 - Speed up for clean build is usually 5 times
 - Can be used as a wrapper:

```
$ ccache gcc main.c
```

How to Speed-Up the Build? (page 4)



How to Speed-Up the Build? (page 5)



RootFS

RootFS

What is RootFS?

- Filesystem that is needed to make userspace work
- Mounted to “/”
- Crucial component is `init` tool
- Besides of that: libc, kernel modules, tools, config files...

RootFS

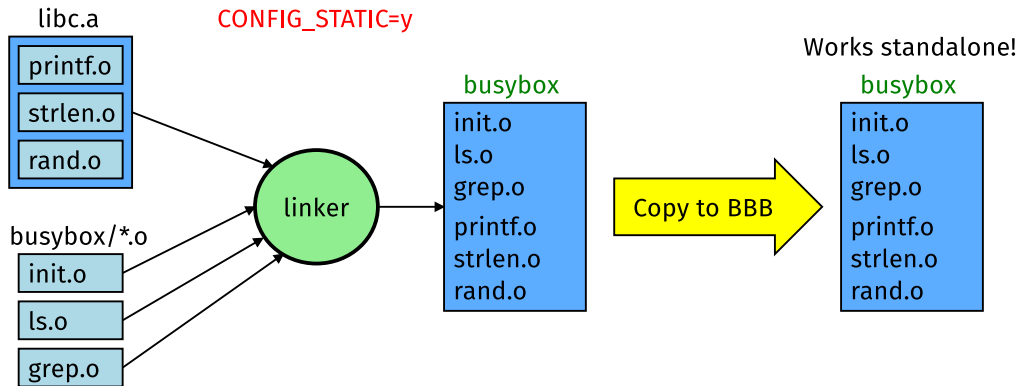
What is RootFS?

- Filesystem that is needed to make userspace work
- Mounted to “/”
- Crucial component is `init` tool
- Besides of that: libc, kernel modules, tools, config files...

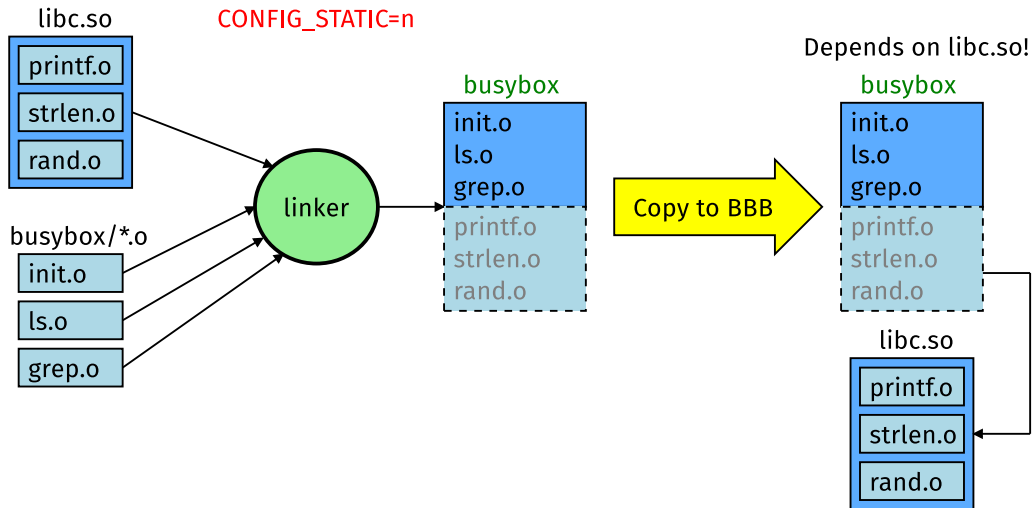
Known rootfs's for BBB:

- Debian
- Yocto/OpenEmbedded
- BuildRoot
- BusyBox

BusyBox Linking: Static



BusyBox Linking: Dynamic



BusyBox Linking: Static vs Dynamic

- **Static linking:** libc (.a) is compiled in your binary
 - Only “**busybox**” binary is needed in rootfs
 - Easier to build and minimal
 - Some networking functions won't work (like **nslookup**, see libnss)
- **Dynamic linking:** libc used as a shared library (.so)
 - Only one copy of libc is used (for all possible apps)
 - Dynamic libraries must be copied in rootfs **/lib** (libc and its dependencies)

BusyBox Applets

- BusyBox is a multi-call binary
- Apps in BusyBox rootfs are just symbolic links:

/bin

```
|— busybox  
|— grep -> busybox  
|— ls    -> busybox
```

- So you can call **ls** tool like this:

```
# busybox ls -l
```

- ...which is identical to this form:

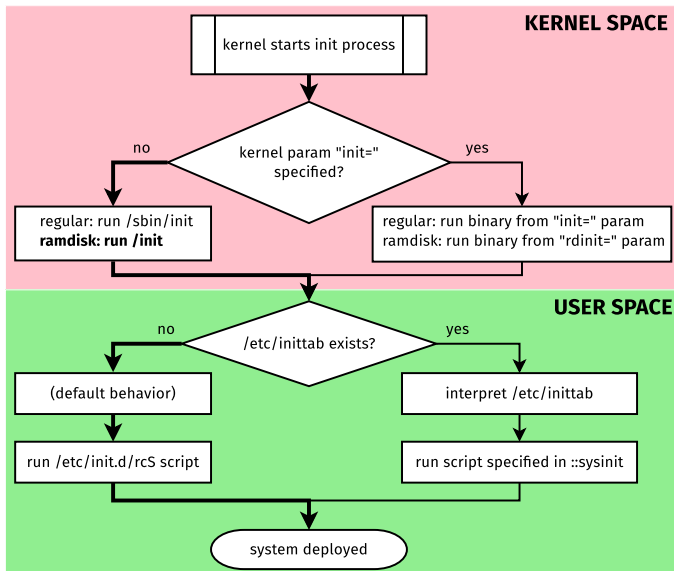
```
# ls -l
```

Init process

- First process started during boot (kernel starts it)
- PID = 1, uid = 0 (root)
- It never exits (daemon)
- Init is a parent for all processes
- Automatically adopts all orphaned processes
- Init is started by the kernel using a hard-coded filename (e.g. `/init`)
- A kernel panic will occur if the kernel is unable to start it
- Most popular init implementations:
 - sysvinit
 - openrc
 - upstart
 - systemd

- **busybox** tool implements init as an applet
- BusyBox's init implementation resembles SysVinit, but more simple
- Doesn't support runlevels (as opposed to SysVinit)
- Starts **/etc/init.d/rcS** script
- (Re)spawns children according to **/etc/inittab** (e.g. **getty**)
- Handles signals (e.g. **reboot** and **poweroff**)

BusyBox init (cont'd)



- No **udev** in BusyBox
- *'mdev is a mini-udev implementation for dynamically creating device nodes in the /dev directory'*
- Requires SysFS support in kernel; it must be mounted to **/sys**
- Can be also used for hot-plugging (e.g. load needed kernel module when some USB device was inserted)
- **mdev -s**: scan **/sys** and populate **/dev**
- **mdev** without params: kernel hotplug helper
- For more details see: **doc/mdev.txt**

BusyBox init: Script Example

- **rc** = “run commands”, **S** = single-user runlevel
 - Example of `/etc/init.d/rcS` file:
-

```
#!/bin/sh
```

```
mount -t sysfs none /sys
```

```
mount -t proc none /proc
```

```
mount -t debugfs none /sys/kernel/debug
```

```
echo /sbin/mdev > /proc/sys/kernel/hotplug
```

```
mdev -s
```

BusyBox rootfs (static, minimal)

```
/
├── bin
│   └── busybox
├── boot
│   ├── am335x-boneblack.dtb
│   └── zImage
├── dev
├── etc
│   └── init.d
│       └── rcS
```

```
├── init -> bin/busybox
├── linuxrc -> bin/busybox
├── proc
├── root
├── sbin
├── sys
├── tmp
└── usr
    ├── bin
    └── sbin
```

Demo: menuconfig

Assignments

Assignment





- Using BBB instructions guide (will be sent out):
 - Go through 1st chapter (“Preparing the Tools”)
 - Go through 2nd chapter (“Obtaining and Building the Software”)
 - Run built software on QEMU, using section 3.1 “QEMU Boot”
- Download TRM and datasheet for AM335x
- Download schematic for BBB
- Proof: send me screenshot of `uname -a` output in your QEMU

Advanced assignment (optional)

- Using TRM, figure out:
 - Which module (TRM section?) is used for setting clocks (gating, DPLL)
 - Which module (TRM section?) is used for pin multiplexing
 - Where GPIO output registers are documented
 - Where UART RX/TX registers are documented
- Using schematic, figure out:
 - Which pins (pads) the user LEDs are connected to
 - How to mux those pins for GPIO (use datasheet and TRM)?
 - Which registers to use for pin muxing and then blinking some LED?

References

Recommended Reading

-  Karim Yaghmour, Jon Masters and others.
Building Embedded Linux Systems.
-  Brian Ward.
How Linux Works, 2nd Edition.
-  Andrew N. Sloss and others.
ARM System Developer's Guide.
-  Robert Love.
Linux Kernel Development, 3rd Edition.

Thank you!