Embedded Systems Programming
OS Linux - Toolchain

Iwona Kochańska

Gdansk University of Technology
Elements of embedded Linux

- **Toolchain**: consists of the compiler and other tools needed to create code for target device.
- **Bootloader**: necessary to initialize the board and to load and boot the Linux kernel.
- **Kernel**: heart of the system, managing system resources and interfacing with hardware.
- **Root filesystem**: contains the libraries and programs that are run once the kernel has completed its initialization.
# Starting up embedded Linux

<table>
<thead>
<tr>
<th>BIOS</th>
<th>Hardware init (RAM, PCI bus, USB, video, keyboard, disks, etc.)</th>
<th>Enumerate disks</th>
<th>Find and load boot loader from disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot loader</td>
<td>Enumerate bootable OS's</td>
<td>User interaction (optional)</td>
<td>Load and run OS (Linux: kernel+initrd)</td>
</tr>
<tr>
<td>Kernel</td>
<td>Hardware init (Remaining HW)</td>
<td>Kernel init</td>
<td>Decompress initrd and run init</td>
</tr>
<tr>
<td>Init</td>
<td>Mount root and other filesystems</td>
<td>Start system and network services</td>
<td>Start getty &amp; display manager</td>
</tr>
<tr>
<td>Login Prompt</td>
<td>Authorize user</td>
<td>Setup session</td>
<td></td>
</tr>
</tbody>
</table>

After the kernel has booted

- The kernel seeks to find a root filesystem:
  - initramfs
  - a filesystem specified by root= on the kernel command line
- Executes a program which, by default, is:
  - /init for initramfs,
  - /sbin/init for a regular filesystem.
- The init program has root privilege and since it is the first process to run, it has a process ID (PID) of 1.
- If, for some reason, init cannot be started, the kernel will panic.
After the kernel has booted

- The init program is the ancestor of all other processes
- The job of the init program is to take control of the system and set it running.
- It may be as simple as a shell command running a shell script
- In the majority of cases, it is a dedicated init daemon.
The tasks of init daemon

Init manages the lifecycle of the system, from boot up to shutdown:

- At boot:
  - starts daemon programs,
  - configures system parameters
  - configures other things needed to get the system into a working state.

- Optionally, it launches daemons, such as `getty` on terminals which allow a login shell.

- It adopts processes that become orphaned as a result of their immediate parent terminating and there being no other processes in the thread group.

- It responds to any of init’s immediate children terminating by catching the signal SIGCHLD and collecting the return value to prevent them becoming zombie processes.

- Optionally, it restarts those daemons that have terminated.

- It handles system shutdown.
The kernel will get a root filesystem as:

- a ramdisk, passed as a pointer from the bootloader,
- by mounting the block device given on the kernel command line by the root= parameter.

Once it has a root filesystem, the kernel will execute the first program, by default named `init`,

The `init` program begins processing scripts, start other programs, and so on, by calling system functions in the C library, which translate into kernel system calls.
Root filesystem - minimum components

- **init**: The program that starts everything off, usually by running a series of scripts.
- **shell**: Needed to give you a command prompt but, more importantly, to run the shell scripts called by init and other programs.
- **daemons**: Various server programs, started by init.
- **libraries**: Usually, the programs mentioned so far are linked with shared libraries which must be present in the root filesystem.
- **Configuration files**: The configuration for init and other daemons is stored in a series of ASCII text files, usually in the /etc directory.
Root filesystem - minimum components

- **Device nodes**: The special files that give access to various device drivers.

- **/proc and /sys**: Two pseudo filesystems that represent kernel data structures as a hierarchy of directories and files. Many programs and library functions read these files.

- **kernel modules**: If you have configured some parts of your kernel to be modules, they will be here, usually in /lib/modules/[kernel version].

In addition, there are the system application or applications that make the device do the job it is intended for, and the runtime end user data that they collect.
Linux does not care about the layout of files and directories beyond the existence of the program named by `init=` or `rdinit=`.

However, many programs expect certain files to be in certain places.

Basic layout of a Linux system is defined in the **Filesystem Hierarchy Standard (FHS)**.
Directory layout

- Embedded devices have a sub-set based on need but it usually includes the following:
  - `/bin`: programs essential for all users
  - `/dev`: device nodes and other special files
  - `/etc`: system configuration
  - `/lib`: essential shared libraries, for example, those that make up the C library
  - `/proc`: the proc filesystem
  - `/sbin`: programs essential to the system administrator
  - `/sys`: the sysfs filesystem
  - `/tmp`: a place to put temporary or volatile files
  - `/usr`: as a minimum, this should contain the directories `/usr/bin`, `/usr/lib` and `/usr/sbin`, which contain additional programs, libraries, and system administrator utilities
  - `/var`: a hierarchy of files and directories that may be modified at runtime, for example, log messages, some of which must be retained after boot
The difference between `/bin` and `/sbin`: `/sbin` need not be included in the search path for non-root users.

`/usr` is that it may be in a separate partition from the root filesystem so it cannot contain anything that is needed to boot the system up.
Creating a staging directory on the host computer

```bash
$ mkdir ~/rootfs
$ cd ~/rootfs
$ mkdir bin dev etc home lib proc sbin sys
$ mkdir usr/bin usr/lib usr/sbin
$ mkdir var/log
```

```bash
$ tree -d
  ├── bin
  │    └── log
  └── var
      ├── bin
      │    └── log
      └── lib
```

```bash
$ tree -d
  ├── bin
  │    └── log
  └── var
      ├── bin
      │    └── log
      └── lib
```

$ tree -d
  ├── bin
  │    └── log
  └── var
      ├── bin
      │    └── log
      └── lib
Every process (every running program) belongs to a user and one or more groups.

The **user** is represented by a 32-bit **user ID** or **UID**.
- Information about users is kept in `/etc/passwd`.

Groups are represented by a **group ID** or **GID**
- Information kept in `/etc/group`.

A root user: **UID = 0**, root group: **GID = 0**.

The root user
- Also called the **super-user**
- Bypasses most permission checks
- Can access all the resources in the system.

Security in Linux-based systems is mainly about restricting access to the root account.
POSIX files access permissions

- Each file and directory has an owner and belongs to exactly one group.
- The level of access a process has to a file or directory is controlled by a set of access permission flags (mode of the file).
  - three collections of three bits:
    - first collection - owner of the file,
    - second collection - members of the same group as the file
    - third collection - rest of the world.
  - The bits are for read (r), write (w), and execute (x) permissions on the file.
  - Since three bits fit into an octal digit
POSIX files access permissions

<table>
<thead>
<tr>
<th></th>
<th>Owner permissions</th>
<th>Group permissions</th>
<th>World permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>r - - - - - -</td>
<td>- - - - - - - -</td>
<td>- - - - - - - -</td>
</tr>
<tr>
<td>200</td>
<td>- - - - - - w</td>
<td>- - - - - - - -</td>
<td>- - - - - - - -</td>
</tr>
<tr>
<td>100</td>
<td>- - - - - - x</td>
<td>- - - - - - - -</td>
<td>- - - - - - - -</td>
</tr>
<tr>
<td>040</td>
<td>- - - - - - r</td>
<td>- - - - - - - -</td>
<td>- - - - - - - -</td>
</tr>
<tr>
<td>020</td>
<td>- - - - - - w</td>
<td>- - - - - - - -</td>
<td>- - - - - - - -</td>
</tr>
<tr>
<td>010</td>
<td>- - - - - - x</td>
<td>- - - - - - - -</td>
<td>- - - - - - - -</td>
</tr>
<tr>
<td>004</td>
<td>- - - - - - r</td>
<td>- - - - - - - -</td>
<td>- - - - - - - -</td>
</tr>
<tr>
<td>002</td>
<td>- - - - - - w</td>
<td>- - - - - - - -</td>
<td>- - - - - - - -</td>
</tr>
<tr>
<td>001</td>
<td>- - - - - - x</td>
<td>- - - - - - - -</td>
<td>- - - - - - - -</td>
</tr>
</tbody>
</table>
POSIX files access permissions

There is a further group of three bits that have special meanings:

- **SUID (4):** If the file is an executable, change the effective UID of the process to that of the owner of the file.
  - probably the most often used.
  - gives non-root users a temporary privilege escalation to super-user to perform a task.
  - example - ping program is normally owned by the root and has the SUID bit set so that, when running the ping, it executes with UID 0 regardless of your UID.

- **SGID (2):** If the file is an executable, change the effective GID of the process to that of the group of the file.

- **Sticky (1):** In a directory, restrict deletion so that one user cannot delete files that are owned by another user.

- This is usually set on /tmp and /var/tmp.
POSIX files access permissions

**chmod** command - set permissions

Example: set SUID on /bin/ping in staging root directory

```
$ cd ~/rootfs
$ ls -l bin/ping
-rwxr-xr-x 1 root root 35712 Feb  6 09:15 bin/ping
$ sudo chmod 4755 bin/ping
$ ls -l bin/ping
-rwsr-xr-x 1 root root 35712 Feb  6 09:15 bin/ping
```

s - last file listing
Ownership and permissions of the files that will be placed on the target device:

- restrict sensitive resources to be accessible only by the root
- run as many of the programs using non-root users so that, if they are compromised by an outside attack, they offer as few system resources to the attacker as possible.
- Example: the device node /dev/mem gives access to system memory. Should be owned by root, belong to the root group and have a mode of 600, which denies read and write access to all but the owner.
The **init** program

- the first program to be run (PID 1).
- runs as the root user and so has maximum access to system resources.
- runs shell scripts which start daemons (programs that run in the background with no connection to a terminal)
The **shell** programs:

- to run scripts and to give a command-line prompt
- lists programs to run,
- passes information between programs.
- probably not necessary in a production device
- useful for development, debugging, and maintenance.
The shell programs:

- **bash**: a superset of the Unix Bourne shell, with many extensions or bashisms.
- **ash**: based on the Bourne shell, and has a long history with the BSD variants of Unix.
  - Busybox has a version of ash which has been extended to make it more compatible with bash.
  - much smaller than bash and hence is a very popular choice for embedded systems.
- **hush**: a very small shell (used in bootloaders).
  - useful on devices with very little memory.
  - there is a version in BusyBox.

Always test the shell scripts on the target!
Utilities programs:

- make a shell useful
- even for a basic root filesystem, there are approximately 50 utilities
  - problem of tracking down the source code for each and cross compiling it
  - problem of size of executable files
BusyBox

BusyBox - project instigated in 1996 by Bruce Perens for the Debian installer so that he could boot Linux from a 1.44 MB floppy disk.

- written from scratch to perform the essential functions of those essential Linux utilities.
- 80:20 rule: the most useful 80% of a program is implemented in 20% of the code.
- BusyBox tools implement a subset of the functions of the desktop equivalents, but they do enough to be useful in the majority of cases.
- combines all the tools together into a single binary, making it easy to share code between them.
BusyBox

- Collection of applets, each of which exports its main function in the form `[applet]_main`.
- Example: the `cat` command is implemented in `coreutils/cat.c` and exports `cat_main`.
- The main function of BusyBox itself dispatches the call to the correct applet based on the command-line arguments.

Syntax:

```
$ busybox cat my_file.txt
```

- Run busybox with no arguments to get a list of all the applets that have been compiled.
BusyBox

- Create a symbolic link from `/bin/cat` to `/bin/busybox`:

  ```
  $ ls -l bin/cat bin/busybox
  -rwxr-xr-x 1 chris chris 892868 Feb 2 11:01 bin/busybox
  lrwxrwxrwx 1 chris chris 7 Feb 2 11:01 bin/cat -> busybox
  ```

- When you type `cat` at the command line, `busybox` is the program that actually runs.

- `BusyBox` only has to check the command tail passed in `argv[0]`, which will be `/bin/cat`, extract the application name, `cat`, and do a table look-up to match `cat` with `cat_main` (`libbb/appletlib.c`):

  ```
  applet_name = argv[0];
  applet_name = bb_basename(applet_name);
  run_applet_and_exit(applet_name, argv);
  ```
BusyBox

- BusyBox has over 300 applets including:
  - an `init` program
  - several shells of varying levels of complexity
  - utilities for most admin tasks
  - simple version of the vi editor

- A typical installation of BusyBox consists of a single program with a symbolic link for each applet, but which behaves exactly as if it were a collection of individual applications.
Building BusyBox

- BusyBox uses the same Kconfig and Kbuild system of the kernel
- getting the source:

  ```
  $ git clone git://busybox.net/busybox.git
  $ cd busybox
  $ git checkout 1_24_1
  ```

- Default configuration:

  ```
  $ make distclean
  $ make defconfig
  ```

- Run make menuconfig to fine tune the configuration - set the install path in Busybox Settings | Installation Options (CONFIG_PREFIX) to point to the staging directory.
Building BusyBox

- Cross compile in the usual way:

  ```
  $ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
  ```

  - The result is the executable, `busybox`.
  - For a defconfig build for ARM v7a, it comes out at about 900 KiB. If that is too big, it can be slimed down by configuring out the utilities.

- Install BusyBox:

  ```
  $ make install
  ```

  This will copy the binary to the directory configured in CONFIG_PREFIX and create all the symbolic links to it.
Libraries

- Programs are linked with libraries - need for copy shared libraries from the toolchain to the staging directory. Which libraries?
- One option is to copy all of them since they must be of some use
  - full glibc is large (thousands of MB); may use uClibc or Musel libc libraries instead.
- pick only required libraries

```
$ cd ~/rootfs
$ arm-cortex_a8-linux-gnueabihf-readelf -a bin/busybox | grep "program interpreter"
  [Requesting program interpreter: /lib/ld-linux-armhf.so.3]
$ arm-cortex_a8-linux-gnueabihf-readelf -a bin/busybox | grep "Shared library"
0x00000001 (NEEDED)  Shared library: [libm.so.6]
0x00000001 (NEEDED)  Shared library: [libc.so.6]
```
Libraries

- Find sysroot:
  
  ```bash
  $ arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot
  ```

- remember as environmental variable:
  
  ```bash
  $ export SYSRoot=`arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot`
  ```

- check `/lib/ld-linux-armhf.so.3`:
  
  ```bash
  $ ls -l $SYSRoot/lib/ld-linux-armhf.so.3
  [...]/sysroot/lib/ld-linux-armhf.so.3 -> ld-2.19.so
  ```

Repeat procedure. Copy all symbolic links and a file to a target.
Libraries - reducing size

- Libraries and programs are often compiled with a symbol table information built in (with debug switch, -g).
- A quick and easy way to save space is to strip them:

```bash
$ file rootfs/lib/libc-2.19.so
rootfs/lib/libc-2.19.so: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked
$ ls -og rootfs/lib/libc-2.19.so
-rwxrwxr-x 1 1547371 Feb 5 10:18 rootfs/lib/libc-2.19.so
$ arm-cortex_a8-linux-gnueabi-strip rootfs/lib/libc-2.19.so
$ file rootfs/lib/libc-2.19.so
rootfs/lib/libc-2.19.so: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked
$ ls -l rootfs/lib/libc-2.19.so
-rwxrwxr-x 1 chris chris 1226024 Feb 5 10:19 rootfs/lib/libc-2.19.so
$ ls -og rootfs/lib/libc-2.19.so
-rwxrwxr-x 1 1226024 Feb 5 10:19 rootfs/lib/libc-2.19.so
```

- When stripping kernel modules, use the following command (otherwise the symbols needed to relocate the module code will be stripped out and the module will fail to load):

```
strip --strip-unneeded <module name>
```
Device nodes

- In Unix everything is a file - except network interfaces, which are sockets
- Most devices in Linux are represented by device nodes
- A **device node** may refer to:
  - **block device** - mass storage device such as SD cards or hard drive
  - **character device** - anything else, once again with the exception of network interfaces.
- The conventional location for device nodes is the directory `/dev`
  (for example, a serial port may be represented by the device node `/dev/ttyS0`).
Device nodes

- Device nodes are created using the program `mknod` (short for make node):

  ```
  mknod <name> <type> <major> <minor>
  ```

  - **name** - name of the device node
  - **type** is either, c for character devices, and b for block.
  - major number and a minor number are used by the kernel to route file requests to the appropriate device driver code.
  - list of standard major and minor numbers in the kernel source in Documentation/devices.txt.

- Device nodes can be created:
  - manually by using the mknod command
  - automatically (at runtime) using one of the device managers to create them automatically
Device nodes

➢ To boot with BusyBox just two nodes are needed:
  ➢ **console** - accessible to root, the owner of the device node, so the access permissions are 600.
  ➢ **null** device - should be readable and writable by everyone (mode 666).
  ➢ setting the mode when creating the node: `-m` option to `mknod`

```
$ cd ~/rootfs
$ sudo mknod -m 666 dev/null c 1 3
$ sudo mknod -m 600 dev/console c 5 1
$ ls -l dev
total 0
crw------- 1 root root 5, 1 Oct 28 11:37 console
crw-rw-rw- 1 root root 1, 3 Oct 28 11:37 null
```

➢ to delete device nodes - by using the standard `rm` command
The proc and sysfs filesystems

- **proc** and **sysfs** are two pseudo filesystems that give a window onto the inner workings of the kernel.
- Both represent **kernel data as files** in a hierarchy of directories:
  - When you read one of the files, the contents you see do not come from disk storage, it has been formatted on-the-fly by a function in the kernel.
- Some files are also writable, meaning that a kernel function is called with the new data.
- Proc and sysfs should be mounted on the directories /proc and /sys:
  ```bash
  mount -t proc proc /proc
  mount -t sysfs sysfs /sys
  ```
The proc filesystem

**proc** - part of Linux since the early days

- Its original purpose was to expose information about processes to user space
- There is a directory for each process named `/proc/<PID>` which contains information about its state.
- **ps** - process list command, reads the `/proc/<PID>` files to generate its output.
- `/proc/cpuinfo` - information about the CPU,
- `/proc/interrupts` - information about interrupts
- `/proc/sys` - files that display and control the state and behavior of kernel sub-systems, especially scheduling, memory management, and networking.
- **man page proc** - reference for the `/proc/` files
The sysf filesystem

**sysfs** - since Linux 2.6

- introduced to export a subset of the data in an ordered way
- sysfs exports a very ordered hierarchy of files relating to devices and the way they are connected to each other
Mounting filesystem

- **mount** command - allows to attach one filesystem to a directory within another, forming a hierarchy of filesystems.
  
  ```
  mount [-t vfstype] [-o options] device directory
  ```

- **example:**
  
  ```
  mount -t ext4 /dev/mmcblk0p1 /mnt
  ```

- **root filesystem** - filesystem at the top, which was mounted by the kernel when it booted

- **mounting proc filesystem:**
  - there is no device node, /dev/proc, since it is a pseudo filesystem, not a real one
  - the mount command requires a device as a parameter.
  - have to give a string where the device should go, but it does not matter much what that string is. These two commands achieve exactly the same result:
    ```
    mount -t proc proc /proc
    mount -t proc nodevice /proc
    ```
Kernel modules

- Kernel modules need to be installed into the root filesystem, using the kernel `make modules_install` target.
- This will copy them into the directory `/lib/modules/<kernel version>` together with the configuration files needed by the `modprobe` command.
- This way a dependency between the kernel and the root filesystem creates.
Transfering the root filesystem to the target

Transfering the root filesystem from the staging directory to the target:

- **ramdisk**: a filesystem image that is loaded into RAM by the bootloader.
  - easy to create and have no dependencies on mass storage drivers.
  - can be used in fall-back maintenance mode when the main root filesystem needs updating.
  - early user space in mainstream Linux distributions
  - can even be used as the main root filesystem in small embedded devices
  - a compressed ramdisk uses the minimum amount of storage but still consumes RAM.
  - the contents are volatile so another storage type to store permanent data (such as configuration parameters) is needed.
Transfering the root filesystem to the target

Transfering the root filesystem from the staging directory to the target:

- **disk image**: a copy of the root filesystem formatted and ready to be loaded onto a mass storage device on the target.
  - example: an image in ext4 format ready to be copied onto an SD card, jffs2 format ready to be loaded into flash memory via the bootloader.
  - the most common option.
Creating a boot ramdisk

- A Linux boot ramdisk (initial RAM filesystem or initramfs) - compressed cpio archive.
- cpio - old Unix archive format, similar to TAR and ZIP but easier to decode (and so requiring less code in the kernel).
- Configure kernel with CONFIG_BLK_DEV_INITRD to support initramfs.
- Different ways to create a boot ramdisk:
  - as a standalone cpio archive - the most flexibility, but not all bootloaders have the facility to load a separate ramdisk,
  - as a cpio archive embedded in the kernel image,
  - as a device table which the kernel build system processes as part of the build.
The sequence of instructions creates the archive, compresses it and adds a U-Boot header ready for loading onto the target:

```
$ cd ~/rootfs
$ find . | cpio -H newc -ov --owner root:root > ../initramfs.cpio
$ cd ..
$ gzip initramfs.cpio
$ mkimage -A arm -O linux -T ramdisk -d initramfs.cpio.gz uRamdisk
```

**final size (example):**

- uRamdisk file - 2.9 MiB, with no kernel modules.
- zImage file - 4.4 MiB
- U-Boot - 440 KiB

= 7.7 MiB storage needed to boot the board.
Standalone ramdisk

- If size is a real problem:
  - Make the kernel smaller by leaving out drivers and functions you don’t need
  - Make BusyBox smaller by leaving out utilities you don’t need
  - Use uClibc or musl libc in place of glibc
  - Compile BusyBox statically
Booting the ramdisk

- Run a shell on the console to interact with the device. Add
  \[ \text{rdinit}=\text{/bin/sh} \]
  to the kernel command line and boot the device.

- Example:
  ```
  fatload mmc 0:1 0x80200000 zImage
  fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
  fatload mmc 0:1 0x81000000 uRamdisk
  setenv bootargs console=tty00,115200 rdinit=/bin/sh
  bootz 0x80200000 0x81000000 0x80f00000
  ```

- Mounting proc - write a shell script that contains things that need to be done at boot-up and give that as the parameter to rdinit=
  ```
  #!/bin/sh
  /bin/mount -t proc proc /proc
  /bin/sh
  ```
Booting the ramdisk with QEMU

- QEMU has the option -initrd to load initramfs into memory:

```bash
$ cd ~/rootfs $ QEMU_AUDIO_DRV=none qemu-system-arm -m 256M -nographic
-M vexpress-a9 -kernel zImage
-append "console=ttyAMA0 rdinit=/bin/sh"
-dtb vexpress-v2p-ca9.dtb -initrd initramfs.cpio.gz
```
Building a ramdisk cpio into the kernel image

- In some cases, it is preferable to build the ramdisk into the kernel image, for example, if the bootloader cannot handle a ramdisk file.
- Change the kernel configuration and set CONFIG_INITRAMFS_SOURCE to the full path of the cpio archive.
- Menuconfig - General setup | Initramfs source file(s).
- It has to be the uncompressed cpio file ending in .cpio; not the gzipped version.
- Build the kernel (it is larger than before).
Building a ramdisk cpio into the kernel image

» Booting with QEMU:

```bash
$ cd ~/rootfs $ QEMU_AUDIO_DRV=none qemu-system-arm -m 256M -nographic -M vexpress-a9 -kernel zImage -append "console=ttyAMA0 rdiinit=/bin/sh" -dtb vexpress-v2p-ca9.dtb
```

» Booting on the target:

```bash
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
setenv bootargs console=tty00,115200 rdiinit=/bin/sh bootz 0x80200000 - 0x80f00000
```

» Rebuild the kernel each time you change the contents of the ramdisk and regenerate the .cpio file!
Program Init

- Over the years, there have been many init programs, one of them is init from BusyBox.

- **init** begins by reading the configuration file, `/etc/inittab`

  ::sysinit:/etc/init.d/rcS
  ::askfirst:--/bin/ash

- The first line runs a shell script, rcS, when init is started.
- The second line prints the message Please press Enter to activate this console to the console, and starts a shell when you press Enter.
- The leading - before /bin/ash means that it will be a login shell, which sources /etc/profile and $HOME/.profile before giving the shell prompt.
Program Init

▶ BusyBox init provides a default inittab if none is present in the root filesystem.
▶ The script /etc/init.d/rcS is the place to put initialization commands that need to be performed at boot, for example, mounting the proc and sysfs filesystems:

```bash
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
```

rcS should be executable!
▶ Try it out on QEMU by changing -append parameter:

```
-append "console=ttyAMA0 rdinit=/sbin/init"
```
▶ Try it out on target:

```
setenv bootargs console=tty00,115200 rdinit=/sbin/init
```
Configuring user accounts

- It is not good practice to run all programs as root
- It is preferable to create unprivileged user accounts and use them where full root is not necessary.
- User names are configured in `/etc/passwd`
- There is one line per user, with seven fields of information separated by colons:
  - The login name
  - A hash code used to verify the password, or more usually an x to indicate that the password is stored in `/etc/shadow`
  - UID
  - GID
  - A comment field, often left blank
  - The user’s home directory
Configuring user accounts

- Example - create root (UID 0) and daemon (UID 1):

```
root:x:0:0:root:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/false
```

Setting the shell for user daemon to /bin/false ensures that any attempt to log on with that name will fail.

- Various programs have to read /etc/passwd so as to be able to look up UIDs and names - security problem!

- To reduce the exposure of this sensitive information, the passwords are stored in /etc/shadow (accessible only as root) and an x is placed in the password field.
Configuring user groups

- Group names are stored in `/etc/group`. The format is as follows:
  - The name of the group
  - The group password, usually an `x` character, indicating that there is no group password
  - The GID
  - An optional list of users who belong to this group, separated by commas.

- Example:
  ```
  root:x:0:
daemon:x:1:
  ```
Adding user accounts to the root filesystem

- Add to the staging directory etc/passwd, etc/shadow (permission 600), and etc/group
- **getty** - program for starting the login procedure (part of BusyBox)
- Starting **init** program reads file `/etc/inittab`
- **inittab** restarts **getty** when a login shell is terminated, so inittab should read like this:

```
::sysinit:/etc/init.d/rc5
::respawn:/sbin/getty 115200 console
```
Starting a daemon process

- Example of daemon program: **syslogd** (accumulate log messages from other programs, mostly other daemons)
- Starting the daemon - adding a line to **etc/inittab**:
  ```
  ::respawn:syslogd -n
  ```
  
  **respawn** - if the program terminates, it will be automatically restarted;
  
  **-n** - run as a foreground process.
  
  The log is written to /var/log/messages.
Configuring the network

- Basic network configuration - Ethernet interface (eth0) and a simple IP v4 configuration as a minimum
- BusyBox ifup/ifdown - for simple use cases
- The main network configuration is stored in /etc/network/interfaces.
- Directories in the stage directory:
  
  ```
  etc/network
  etc/network/if-pre-up.d
  etc/network/if-up.d
  var/run
  ```
Configuring the network

- **/etc/network/interfaces** for **static IP** address:

```plaintext
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet static
    address 10.0.0.42
    netmask 255.255.255.0
    network 10.0.0.0
```

- **/etc/network/interfaces** for **dynamic IP** address allocated using DHCP:

```plaintext
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet dhcp
```

- **udchpdc** - BusyBox DHCP client (needs `/usr/share/udhcpc/default.script`). There is a suitable default in the BusyBox source code in the directory examples//udhcpc/simple.script.
Network components for glibc

- **glibc** uses the **name service switch (NSS)** to control the way that names are resolved to numbers for networking and users; i.e.:
  - User names resolved to UIDs via the file `/etc/passwd`;
  - network services such as HTTP resolved to the service port number via `/etc/services`
- Configured by `/etc/nsswitch.conf`
- Everything is resolved by the correspondingly named file in `/etc`, except for the host names, which may additionally be resolved by a DNS lookup.
- `/etc/hosts` should, at least contain, the loopback address:
  
  ```
  127.0.0.1 localhost
  ```
Network components for glibc

- Libraries that perform the name resolution:
  - plugins loaded as needed based on the contents of nsswitch.conf,
  - do not show up as dependencies if you use readelf or similar.

- For target device - copy them from the toolchain’s sysroot:

```bash
$ cd ~/rootfs
$ cp -a $TOOLCHAIN_SYSROOT/lib/libnss* lib
$ cp -a $TOOLCHAIN_SYSROOT/lib/libresolv* lib
```
Creating filesystem images with device tables

- **gen_init_cpio** - kernel utility, that creates a cpio file based on format instructions set out in a text file, called a **device table**,
- **device table** - allows a non-root user to create device nodes and to allocate arbitrary UID and GID values to any file or directory.
- Utilities for other filesystem image formats:
  - jffs2: mkfs.jffs2
  - ubifs: mkfs.ubifs
  - ext2: genext2fs
Creating filesystem images with device tables

Syntax:
<name> <type> <mode> <uid> <gid> <major> <minor> <start> <inc> <count>

▶ **type:**
  ▶ f: A regular file
  ▶ d: A directory
  ▶ c: A character special device file
  ▶ b: A block special device file
  ▶ p: A FIFO (named pipe)

▶ **major** and **minor**: the device numbers (device nodes only)

▶ **start**, **inc**, and **count**: (device nodes only) allow to create a
group of device nodes starting from the **minor** number in **start**

▶ filesystems for flash memory. The third, ext2, is a fairly old format
for hard drives.
Creating filesystem images with device tables

Example of device-table.txt:

```
/dev     d  755 0 0 - - - - -
/dev/null c  666 0 0 1 3 0 0 -
/dev/console c  600 0 0 5 1 0 0 -
/dev/tty00 c  600 0 0 252 0 0 0 -
```

Use `genext2fs` to generate a filesystem image of 4 MiB (that is 4,096 blocks of the default size, 1,024 bytes):

```
$ genext2fs -b 4096 -d rootfs -D device-table.txt -U rootfs.ext2
```
Putting the root filesystem onto an SD card

- Example of mounting a filesystem from a normal block device (SD card):
  ```
  $ sudo dd if=rootfs.ext2 of=/dev/mmcblk0p2
  ```
  - first partition of SD card is used for the boot files (MLO and u-boot.img)

- Slot the SD card into the device, and set the kernel command line to `root=/dev/mmcblk0p2`

- Complete boot sequence:
  ```
  fatload mmc 0:1 0x80200000 zImage
  fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
  setenv bootargs console=tty00,115200 root=/dev/mmcblk0p2
  bootz 0x80200000 - 0x80f00000
  ```
Mounting the root filesystem using NFS

- Mount the root filesystem over the network during development
  - gives access to almost unlimited storage
  - updates made to the root filesystem hosted on the development machine are made available on the target immediately.
- Kernel has to be configured with CONFIG_ROOT_NFS.
- Configure Linux to do the mount at boot time by adding the following to the kernel command line:
  
  ```
  root=/dev/nfs
  ```
- Give the details of the NFS export:
  
  ```
  nfsroot=<host−ip>:<root−dir>
  ```
- Configure the network interface that connects to the NFS server so that it is available at boot time, before the init program runs:
  
  ```
  ip=<target−ip>
  ```
Mounting the root filesystem using NFS

- Install and configure an NFS server on host machine:
  
  ```bash
  $ sudo apt-get install nfs-kernel-server
  ``

- The NFS server needs to be told which directories are being exported to the network, which is controlled by `/etc/exports`. Add a line like this one to that file:
  
  ```
  /<path to staging> *(rw,sync,no_subtree_check,no_root_squash)
  ```

- Restart the server to pick up the change:
  
  ```bash
  $ sudo /etc/init.d/nfs-kernel-server restart
  ```

Documentation/filesystems/nfs/nfsroot.txt.
Mounting the root filesystem using NFS

Testing with target machine:

▶ enter commands at the U-Boot prompt:

```plaintext
setenv serverip 192.168.1.1
setenv ipaddr 192.168.1.101
setenv npath [path to staging directory]
setenv bootargs console=ttyO0,115200 root=/dev/nfs rw nfsroot=${serverip}:${npath}

▶ load the kernel and dtb from sdcard:

```plaintext
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
bootz 0x80200000 – 0x80f00000
```