Linux cheat sheet

This cheat sheet is designed to show the essential technical details of configuring and building Linux with an emphasis on embedded systems to allow programmers and engineers to get up the inital learning curve as speedily as possible.

- **Bootloaders**
  - **U-Boot**
- **Kernel startup**
  - **The initial RAM filesystem, initramfs**
  - **/linuxrc**
  - **The root file system, rootfs**
  - **The init thread**
- **Building the kernel**
  - **Kernel version information**
  - **Kernel configuration**
  - **Kernel configuration editors**
  - **Makefile targets**
- **Useful commands**
  - **Viewing the kernel version**
  - **Viewing running processes**
  - **Viewing library dependencies**
  - **Disassembling object code**
  - **Managing kernel modules**
  - **Finding files**
- **Filesystems and devices**
  - **Flash memory**
  - **Memory Technology Device (MTD)**
  - **Unsorted Block Images (UBI)**
  - **Device trees**
  - **Displays**
  - **Real Time Clock (RTC)**

Bootloaders

The bootloader initializes the hardware and then loads and runs the Linux kernel. Normally the Linux kernel is stored in in a compressed form in flash and is decompressed into RAM before execution.

U-Boot

U-Boot is a popular open source bootloader. It supports booting the kernel from memory, network or serial port using various protocols and has built-in **device tree** support.

U-Boot can update the in-memory **device tree** with platform-specific information on boot. These are called fixups in the U-Boot source. Fixups are contained in the *boards* birectory of the U-Boot source code, are board specific and not normally documented in any place other than the U-Boot code. The function where this occurs within U-Boot is the function *ft_board_setup()*.

Kernel startup

The kernel performs some architecture-specific configuration and enters function start_kernel() which can be found in init/main.c in the kernel source tree. The startup of the kernel continues until device drivers are loaded and the network adapter is configured.

The initial RAM filesystem initramfs

After the hardware has been initialized the kernel searches for something to execute by first unpacking the initial RAM filesystem initramfs which is stored as a compressed cpio archive.

/linuxrc

If the file /linuxrc exists it is run and once it's execution finishes kernel startup continues.

The root file system, rootfs

The kernel then mounts the root file system using the root and rootfstype kernel parameters. The following command loads a JFFS2 root file system stored at /dev/mtd1.

```
root=/dev/mtd1 rootfstype=jffs2
```

The init thread

The first thread that is started eventually becomes the kernel thread init() with a process id (PID) of 1 and this thread is the parent of all Linux processes in user space. The last step of init() is to use a function

called run_init_process() to attempt to load and execute one of a set of specified files as the init process. This function does not return on success but if the init process cannot be started the code falls through to a call to kernel panic. See the code fragment below from main.c.

```
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");

panic(...);
```

A custom initial process can be specified on the kernel command line by setting the init= command line parameter to specify a file that runs after the kernel boot has finished.

Building the kernel

Kernel version information

The top level makefile in the kernel source tree contains version information and as an example the information for kernel 2.6.14 is shown below:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 14
EXTRAVERSION =
NAME = Affluent Albatross
```

If it's installed the kernel source code is normally be found in /usr/src/linux-v.p.s where v = VERSION, p = PATCHLEVEL and s = SUBLEVEL. The running kernel version information can be viewed by **this command**. It is strongly recommended that you specify a custom kernel build by setting EXTRAVERSION. Setting EXTRAVERSION to -red will result in the kernel version information displaying 'Linux version 2.6.14-red...'.

Building the kernel creates System.map containing the kernel symbols for debugging and an ELF format executable file named vmlinux. This is not however normally the file executed to run the kernel, see **Makefile targets** for more information.

Kernel configuration

The .config file in the root of the kernel source tree contains configuration information used to build a kernel and because this file is prefixed with a dot it is a hidden file in Linux.

The command 'make mrproper' restores the .config file to its default state, overwriting any changes that the you may have made.

A section of a .config file is shown below.

```
# USB support
#
CONFIG_USB=m
```

The =m declares that the USB subsystem is included as a dynamically loaded module loaded after the kernel has booted. To statically link in the USB subsystem this is set to =y.

During the kernel build the .config file is transformed into a C header file that can be found in include/linux/autoconf.h.

Kernel configuration editors

To ease the setting up of the kernel configuration stored in the .config file various configuration editors can be built with the kernel build makefile using the commands shown below. The first invocation will build the configuration editor and subsequent invocations will run the editior immediately.

```
$make gconfig
$make menuconfig
$make xconfig
```

gconfig is a GTK based editor, *menuconfig* is a text based editor using ncurses and *xconfig* is a X11 based editor using QT.

Makefile targets

Typing make help will display a list of targets that can be generated from the source tree. Specifying make with no target generates the kernel for the default architecture.

Many architectures require binary targets specific to the architecture and bootloader in use. One common target is zImage and in many architectures this is the default target image. The bzImage target is specific to x86-based PC compatible architectures.

Useful commands

Viewing the kernel version

You can check the version of a running kernel from the command line as follows:

```
$cat /proc/version
Linux version 2.6.14...
```

Additional text after the version number has been removed for clarity but it normally contains details of the compiler used to build the kernel.

Viewing running processes

To display all running processes in the system use the following command.

```
$ps -aux | more
```

- a - all processes
- u - user format
- x - including processes without a controlling terminal

Processes with names ending in d and a TTY field of ? are daemons e.g. syslogd, klogd, crond and lpd.

Viewing library dependencies

To display all of the shared libraries that an application uses (the library dependencies) use the following command.

```
$ldd filename
```

Where filename is the name of the file you wish to view dependencies for. The edited example below shows the dependencies for ls.

```
$ldd /bin/ls
        linux-gate.so.1 =>  (0x0050c000)
        libselinux.so.1 => /lib/i386-linux-gnu/libselinux.so.1 (0x0069b000)
        ...
        libattr.so.1 => /lib/i386-linux-gnu/libattr.so.1 (0x00f15000)
```

Alternatively use

```
$readelf -d filename
```

This command has the advantage that files for any processor architecure may be examined.

To list the symbols contained in a library file use:

```
$nm libname
```

Disassembling object code

The objdump utility can be used to disassemble files containing object code. The following example will disassemble the object code contained in the .text section of *programname*.

```
$objdump -S -marm -j .text programname
```

- S - show source with disassembly
- m - processor architecture
- j - show only the following section (.text in the example)

Managing kernel modules

Kernel modules may be compiled into the kernel so that they are always available or they may be loaded when required.

The names of all modules compiled into the kernel (whether being used or not) can be listed using:

```
$cat /lib/modules/$(uname -r)/modules.builtin
```

The names of all the kernel modules dynamically loaded into the kernel (not compiled in) can be listed using the following command:

```
$cat /proc/modules
```

On embedded systems all required kernel modules may be compiled into the kernel in which case /proc/modules will be empty.

Modules may have information embedded into them describing things such as the parameters it takes, the author, a description and the licence. To view this information about a specific module type:

```
$modinfo modulename
```

Where *modulename* is the name of the kernel module who's information you wish to view.

There are two ways to insert modules into or remove modules from a running kernel. The *modprobe* command is the safe way as it

automatically inserts or removes modules that the module depends upon. The commands *insmod* and *rmmod* ignore module dependencies and incorrect usage may crash the system.

To install a kernel module and all modules that it depends upon use the following command.

```
$modprobe modulename
```

To remove a kernel module and all modules that it depends upon use the following command.

```
$modprobe -r modulename
```

These commands use a file named *modprobe.conf* to determine the module dependencies. An error will be displayed if the module being added or removed has no definition in *modprobe.conf*.

To install a kernel module ignoring any module dependencies use the following command.

```
$insmod pathname
```

Where *pathname* is the absolute pathname of the kernel module file you wish to install. Modules that this module depends on will not be automatically loaded. To remove a kernel module use the following command.

```
$rmmod modulename
```

Where *modulename* is the name of the kernel module you wish to remove. Removing a module that is in use will result in an error message being displayed unless -f (force removal) or -w (wait until free, blocking new use of the module) options are specified on the command line. Forcing the removal of a module that is in use may crash the system. Removing a module that another module depends on may also result in a system crash, see below to remove a module and its dependent modules.

Finding files

The find command can be used to find files in a specified path.

```
find /bin -name myfile
```

Finds all files named myfile in the /bin directory and all of it's subdirectories.

Filesystems and devices

Flash memory

Flash memory is normally used for storage in embedded Linux systems. The Linux kernel needs to know the flash memory layout and this can be hardcoded in the **MTD** driver or stored in the device tree.

MTD

The Memory Technology Device (MTD) subsystem provides a device independent interface for accessing flash memory. MTD devices such as /dev/mtd0 basically represent raw flash and are used by **UBI** and flash file systems such as **UBIFS**, **JFFS2** or **YAFFS**. To view the available MTD partitions type:

```
$ cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00100000 00020000 "u-boot"
mtd1: 00060000 00020000 "env"
mtd2: 00600000 00020000 "linux"
mtd3: 02000000 00020000 "rootfs"
mtd4: 06700000 00020000 "userfs"
```

Information about an MTD partition can be displayed by using the following command. The example displays information for mtd3.

```
$mtd_debug info /dev/mtd3

mtd.type = MTD_NANDFLASH
mtd.flags = MTD_CAP_NANDFLASH
mtd.size = 33554432 (32M)
mtd.erasesize = 131072 (128K)
mtd.writesize = 2048 (2K)
mtd.oobsize = 64
regions = 0
```

UBI

Unsorted Block Images (UBI) is a volume management system for raw flash devices which uses **MTD** to manage multiple logical volumes on a single physical flash device and perform wear levelling. UBI is used by the **UBIFS** file system.

If UBI is compiled as a kernel module specify the MTD name or number in the module arguments to attach an MTD device.

```
$modprobe ubi mtd=rootfs
$modprobe ubi mtd=3 mtd=4
```

The commands above attach 'rootfs', mtd3 and mtd4. The preferable way to attach MTD devices is by name because device numbers may change if the flash layout changes.

If UBI is compiled in to the kernel kernel module specify the MTD name or number in a ubi.mtd= kernel boot parameter.

```
ubi.mtd=rootfs ubi.mtd=4
```

Lastly ubiattach and ubidetach can be used to add or remove UBI volumes with kernel versions 2.6.25 onwards. These commands take the MTD or UBI device number as a parameter and do not support MTD names.

```
ubiattach -m 4 /dev/ubi_ctrl
ubidetach -d 4 /dev/ubi_ctrl
```

Device trees

Modern embedded Linux systems use device trees to contain a description of the hardware. These can be statically linked into the kernel or passed to the kernel at compile time, allowing hardware devices to be added or removed from a Linux system by recompiling the device tree. Once a Linux system has booted the device tree can be read at /proc/device-tree.

The device tree is an architecture-independent data structure and any device tree compiler can compile a device tree suitable for an embedded device. Device tree source code is contained in files with a .dts suffix and is compiled to a 'blob' with a .dtb suffix for use.

To install the device tree compiler.

```
$sudo apt-get install device-tree-compiler
```

To compile example.dts to example.dtb use the following command.

```
$dtc -I dts -O dtb example.dts > example.dtb
```

To get the device tree as readable text in the file dtb.txt from a compiled tree.

```
$dtc -I dtb -O dts example.dtb > dtb.txt
```

Displays

All kernels from kernels version 2.1.109 onwards implement the framebuffer device which allows programs to manipulate the graphics through a well defined interface without knowing anything about the hardware. Framebuffer devices are named /dev/fbn where n is a digit denoting the device.

The fbset command can be used to inspect and change display settings.

```
$fbset /dev/fb0
mode "800x480-0"
        geometry 800 480 800 480 16
        timings 0 0 0 0 0 0 0
        accel true
        rgba 5/11,6/5,5/0,0/0
endmode
```

The /dev/fb0 display geometry is an 800 x 480 display with 16-bit colour depth. The second 800 480 pair define a virtual display size that can be larger than the physical display.

The timings set the length of one pixel in picoseconds followed by the left, right, top and bottom margins respectively. The last two numbers are the horizontal sync length in pixels and vertical sync length in pixel lines. The timings in this example are all zero as the display characteristics are set in the device tree.

The accel true line means that hardware acceleration is enabled.

The rgba line defines the bits used for the Red, Green, Blue and Alpha colours. The format is number of bits/start bit. In the example 5/11 means 5 bits for Red starting at bit 11 i.e. 11 to 15, 6/5 is 6 bits for Green from bit 5 i.e 5 to 10 and 5/0 is 5 bits for Blue from bit 0 i.e. 0 to 4.

Real Time Clock (RTC)

The Real Time Clocks are accessed at /dev/rtc*n*. Multiple clocks are supported by the kernel and /dev/rtc can be used to access the default clock.

The utility *hwclock* reads and writes to the Real Time Clock. The most commonly used command line options to hwclock are listed below.

-f specify the RTC to use.
-r read the time from the RTC.
-s write the RTC time to the system time.
-w write the system time to the RTC.

The following command writes the time held by /dev/rtc1 to the system time.

```
hwclock -f /dev/rtc1 -s
```