Embedded Linux kernel usage

Michael Opdenacker
Thomas Petazzoni
Free Electrons

Latest update: Feb 21, 2011,
Document sources, updates and translations:
http://free-electrons.com/docs/kernel-usage
Corrections, suggestions, contributions and translations are welcome!

Contents

Compiling and booting

Compiling and booting Linux

Linux kernel sources

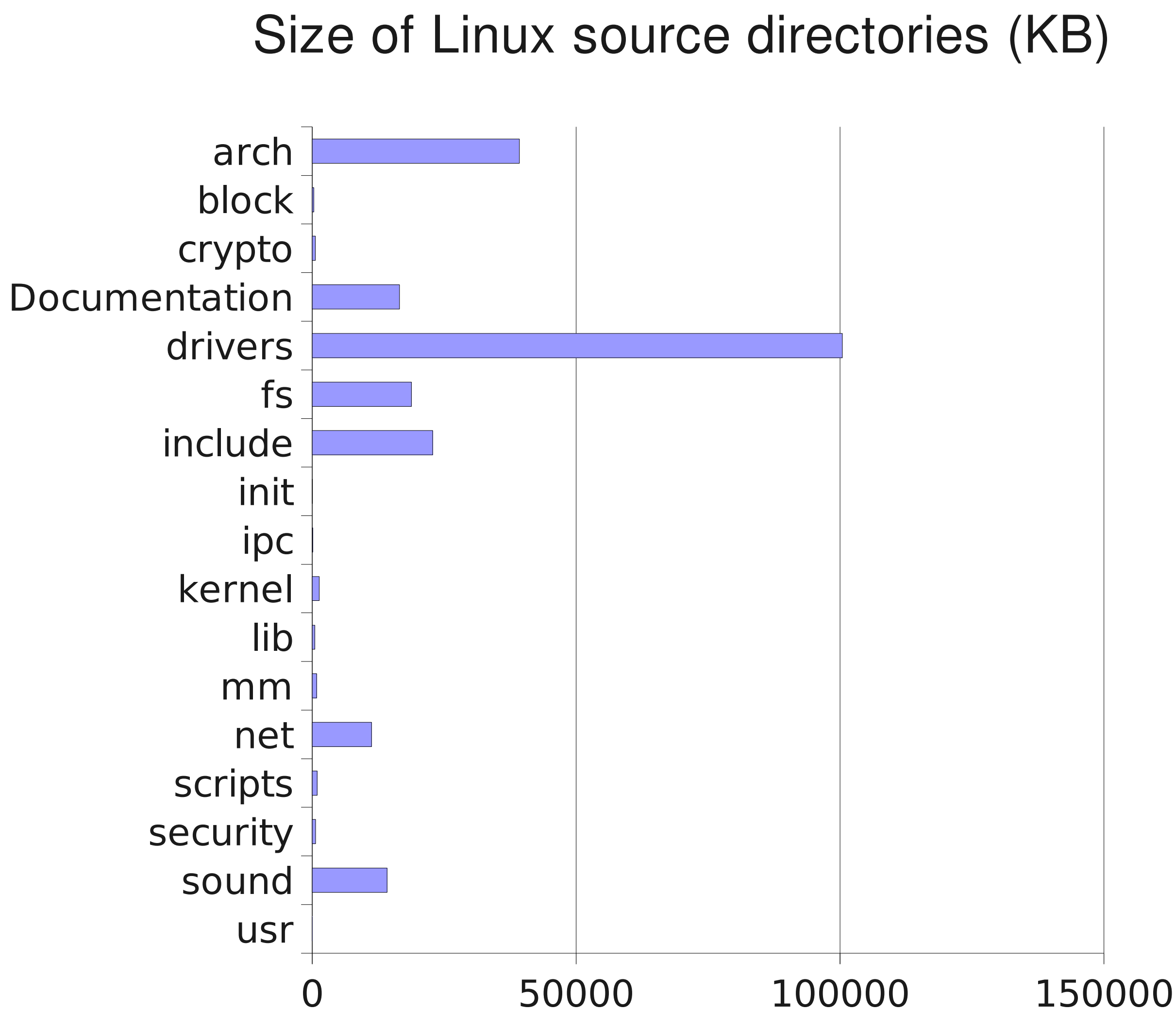
Location of kernel sources

- The official version of the Linux kernel, as released by Linus Torvalds is available at http://www.kernel.org
  - This version follows the well-defined development model of the kernel
  - However, it may not contain the latest development from a specific area, due to the organization of the development model and because features in development might not be ready for mainline inclusion
- Many kernel sub-communities maintain their own kernel, with usually newer but less stable features
  - Architecture communities (ARM, MIPS, PowerPC, etc.), device drivers communities (I2C, SPI, USB, PCI, network, etc.), other communities (real-time, etc.)
  - They generally don't release official versions, only development trees are available

Linux kernel size (1)

- Linux 2.6.37 sources:
  Raw size: 412 MB (37,300 files, approx 14,000,000 lines)
  `gzip` compressed tar archive: 89 MB
  `bzip2` compressed tar archive: 71 MB (better)
  `lzma` compressed tar archive: 61 MB (best)
- Minimum Linux 2.6.29 compiled kernel size with `CONFIG_EMBEDDED`, for a kernel that boots a QEMU PC (IDE hard drive, ext2 filesystem, ELF executable support):
  532 KB (compressed), 1325 KB (raw)
- Why are these sources so big?
  Because they include thousands of device drivers, many network protocols, support many architectures and filesystems...
- The Linux core (scheduler, memory management...) is pretty small!

Linux kernel size (2)



Linux 2.6.17

Measured with:

`du -s --apparent-size`

Getting Linux sources

- Full tarballs
  - Contain the complete kernel sources
  - Long to download and uncompress, but must be done at least once
  - Example:
    http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.14.7.tar.bz2
- Incremental patches between versions
  - It assumes you already have a base version and you apply the correct patches in the right order
  - Quick to download and apply
  - Examples
    http://kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.bz2 (2.6.13 to 2.6.14)
    http://kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.7.bz2 (2.6.14 to 2.6.14.7)
- All previous kernel versions are available in
  http://kernel.org/pub/linux/kernel/

Using the patch command

The patch command applies changes to files in the current directory:

- Making changes to existing files
- Creating or deleting files and directories

patch usage examples:

- patch -p<n> < diff_file
- cat diff_file | patch -p<n>
- bzcat diff_file.bz2 | patch -p<n>
- zcat diff_file.gz | patch -p<n>

n: number of directory levels to skip in the file paths

You can reverse a patch with the -R option

You can test a patch with the --dry-run option

Anatomy of a patch file

A patch file is the output of the `diff` command

```
diff -Nru a/Makefile b/Makefile
--- a/Makefile  2005-03-04 09:27:15 -08:00
+++ b/Makefile  2005-03-04 09:27:15 -08:00
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
 SUBLEVEL = 11
-EXTRAVERSION =
+EXTRAVERSION = .1
 NAME=Woozy Numbat

 # *DOCUMENTATION*
```

- `diff` command line
- File date info
- Line numbers in files
- Context info: 3 lines before the change
  Useful to apply a patch when line numbers changed
- Removed line(s) if any
- Added line(s) if any
- Context info: 3 lines after the change

Applying a Linux patch

Linux patches...

- Always to apply to the `x.y.<z-1>` version
  Downloadable in `gzip`
  and `bzip2` (much smaller) compressed files.
- Always produced for `n=1`
  (that's what everybody does... do it too!)
- Linux patch command line example:

```
cd linux-2.6.13
bzcat ../patch-2.6.14.bz2 | patch -p1
bzcat ../patch-2.6.14.7.bz2 | patch -p1
cd ..; mv linux-2.6.13 linux-2.6.14.7
```

- Keep patch files compressed: useful to check their signature later.
  You can still view (or even edit) the uncompressed data with `vim`:
  `vim patch-2.6.14.bz2` (on the fly (un)compression)

> You can make `patch` 30% faster by using `-sp1` instead of `-p1` (**s**ilent)
>
> Tested on patch-2.6.23.bz2

Ketchup - Easy access to kernel sources

http://www.selenic.com/ketchup/

- Makes it easy to download a specific version.
  Takes care of downloading and applying patches
- Example: downloading the latest kernel version

```
> mkdir linux-2.6.31
> cd linux-2.6.31
> ketchup -G 2.6.31.6
None -> 2.6.31.6
Downloading linux-2.6.31.6.tar.bz2
Unpacking linux-2.6.31.6.tar.bz2
```

- Now getting back to an older version (from the same directory)

```
> ketchup -G 2.6.29
2.6.31.6 -> 2.6.29
Applying patch-2.6.31.6.bz2 -R
Applying patch-2.6.31.bz2 -R
Downloading patch-2.6.30.bz2
Applying patch-2.6.30.bz2 -R
```

The `-G` option of ketchup disables source signature checking.

See http://kernel.org/signature.html for details about enabling kernel source integrity checking.

Practical lab – Kernel sources

- Get the sources
- Apply patches

Compiling and booting Linux

Kernel configuration

Kernel configuration (1)

- The kernel contains thousands of device drivers, filesystem drivers, network protocols and other configurable items
- Thousands of options are available, that are used to selectively compile parts of the kernel source code
- The kernel configuration is the process of defining the set of options with which you want your kernel to be compiled
- The set of options depends
  - On your hardware
  - On the capabilities you would like to give to your kernel

Kernel configuration (2)

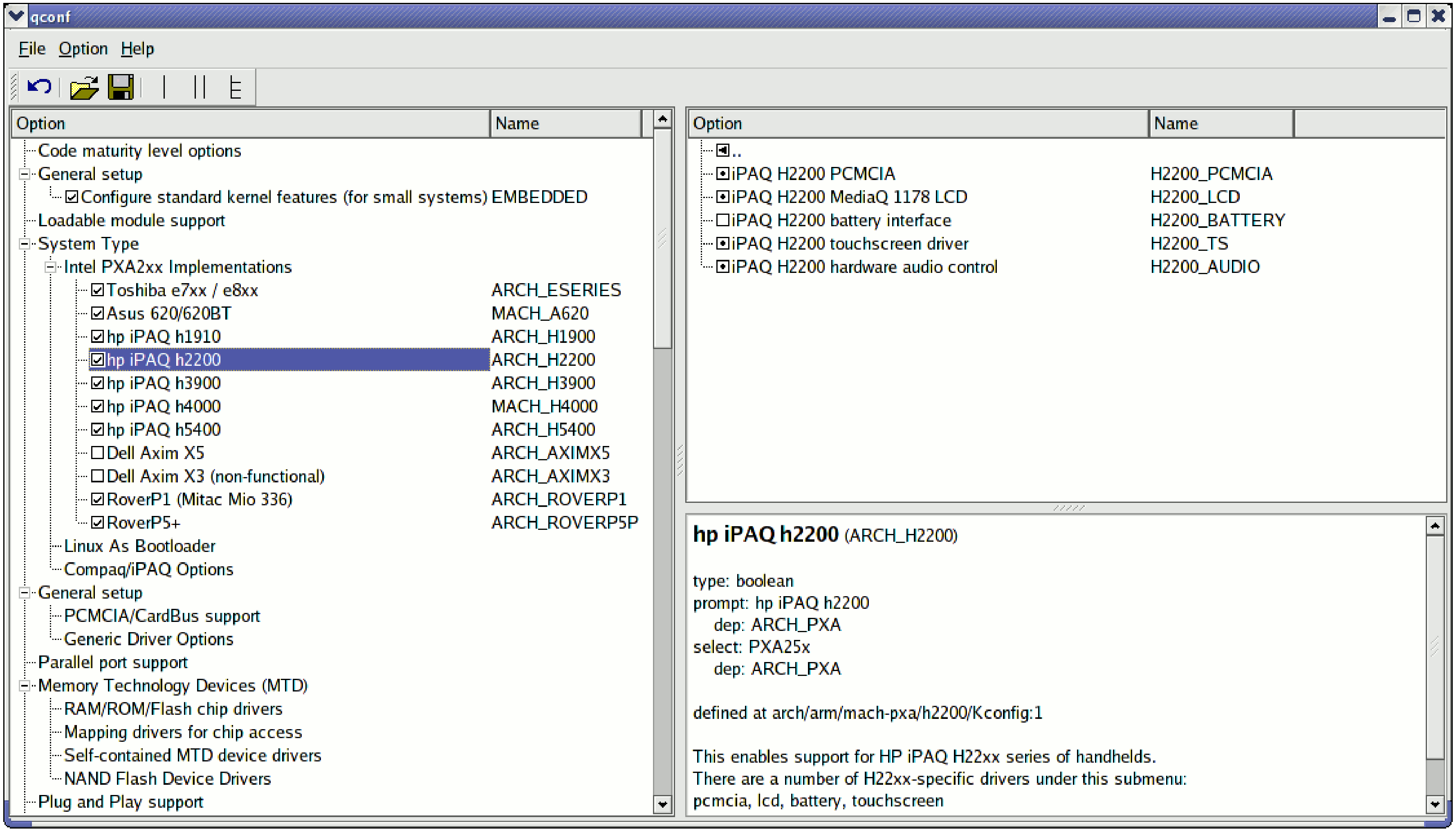
- The configuration is stored in the `.config` file at the root of kernel sources
  - Simple text file, `key=value` style
- As options have dependencies, typically never edited by hand, but through graphical interfaces :
  - `make [xconfig|gconfig|menuconfig|oldconfig]`
  - These are targets from the main kernel Makefile. Run `make help` to get a list of all available targets.
- To modify a kernel in a GNU/Linux distribution:
  the configuration files are usually released in `/boot/`, together with kernel images: `/boot/config-2.6.17-11-generic`

make xconfig

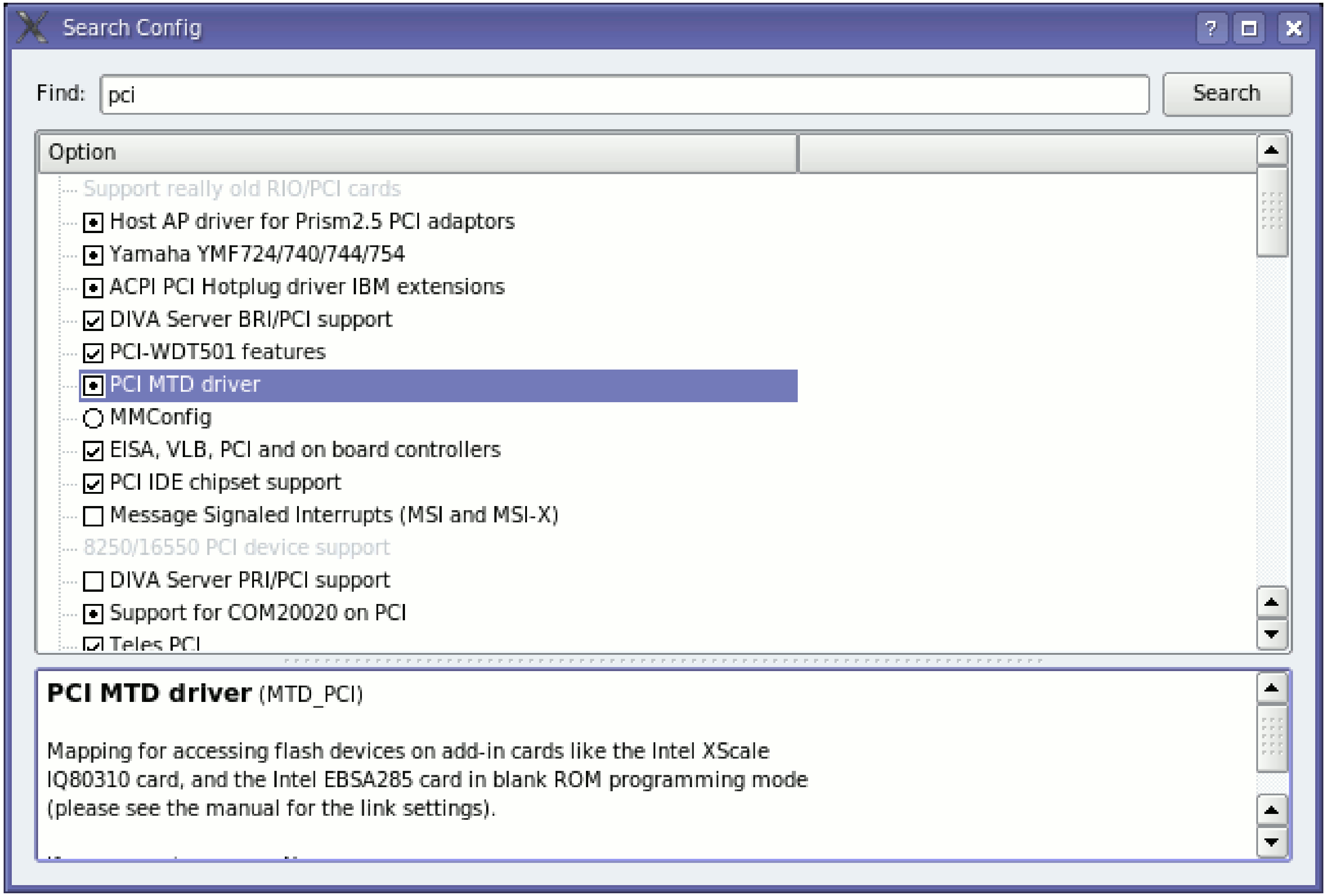
`make xconfig`

- The most common graphical interface to configure the kernel.
- Make sure you read `help -> introduction: useful options!`
- File browser: easier to load configuration files
- New search interface to look for parameters
- Required Debian / Ubuntu packages: `libqt3-mt-dev`, `g++`

make xconfig screenshot



make xconfig search interface

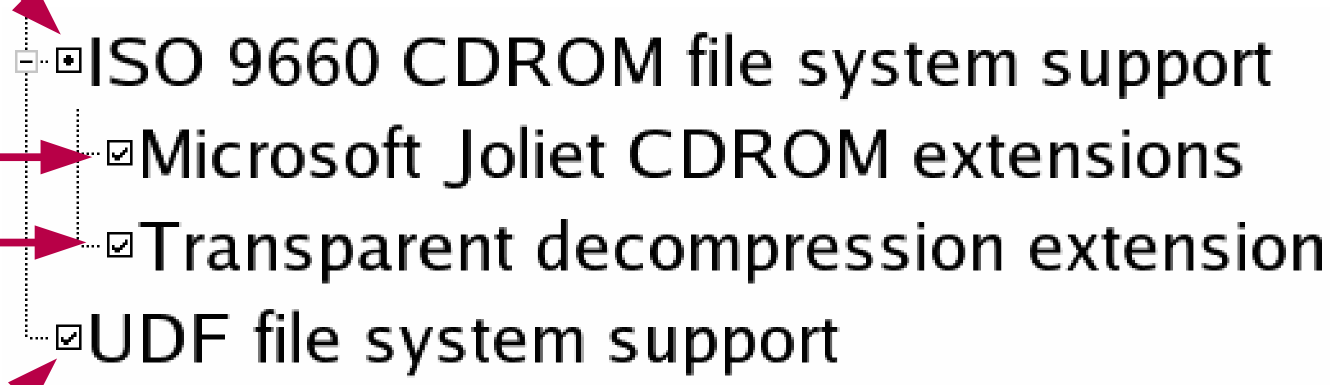


Looks for a keyword in the description string

Allows to select or unselect found parameters.

Compiled as a module (separate file)
`CONFIG_ISO9660_FS=m`

Driver options
`CONFIG_JOLIET=y`
`CONFIG_ZISOFS=y`



Compiled statically into the kernel
`CONFIG_UDF_FS=y`

Corresponding .config file excerpt

```
#
# CD-ROM/DVD Filesystems
#
CONFIG_ISO9660_FS=m
CONFIG_JOLIET=y
CONFIG_ZISOFS=y
CONFIG_UDF_FS=y
CONFIG_UDF_NLS=y

#
# DOS/FAT/NT Filesystems
#
# CONFIG_MSDOS_FS is not set
# CONFIG_VFAT_FS is not set
CONFIG_NTFS_FS=m
# CONFIG_NTFS_DEBUG is not set
CONFIG_NTFS_RW=y
```

Section name
(helps to locate settings in the interface)

All parameters are prefixed
with `CONFIG_`

Kernel option dependencies

- There are dependencies between kernel options
- For example, enabling a network driver requires the network stack to be enabled
- Two types of dependencies
  - *depends on* dependencies. In this case, option A that depends on option B is not visible until option B is enabled
  - *select* dependencies. In this case, with option A depending on option B, when option A is enabled, option B is automatically enabled
  - *make xconfig* allows to see all options, even those that cannot be selected because of missing dependencies. In this case, they are displayed in gray

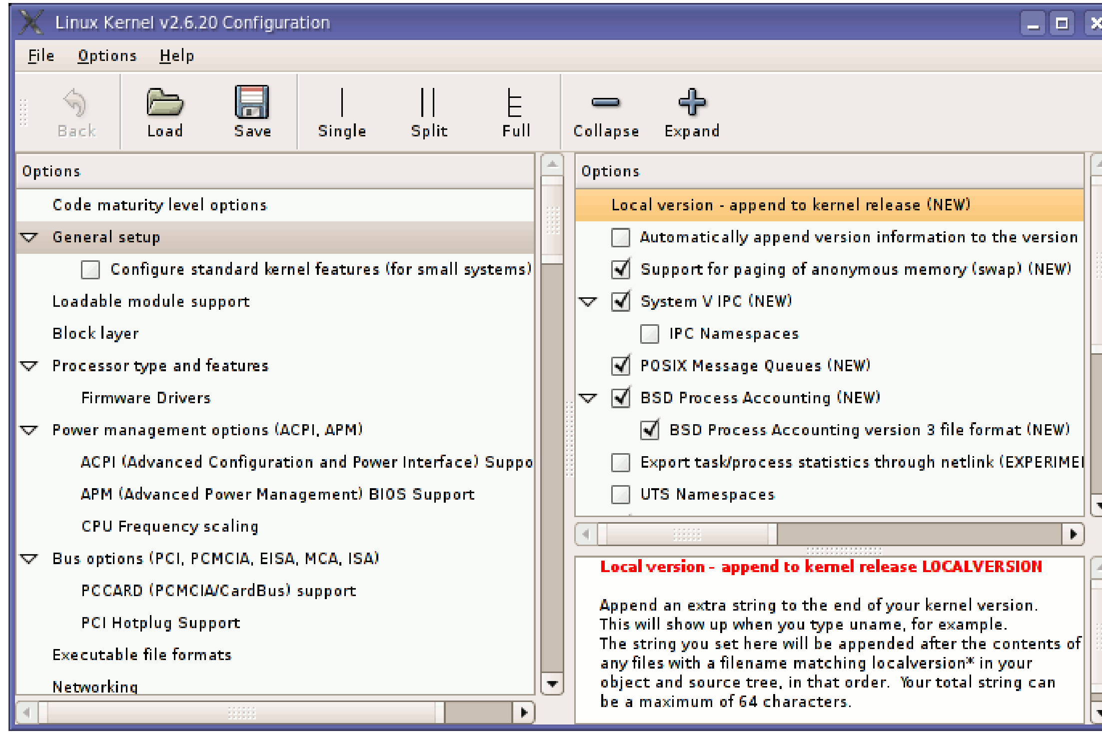
make gconfig

`make gconfig`

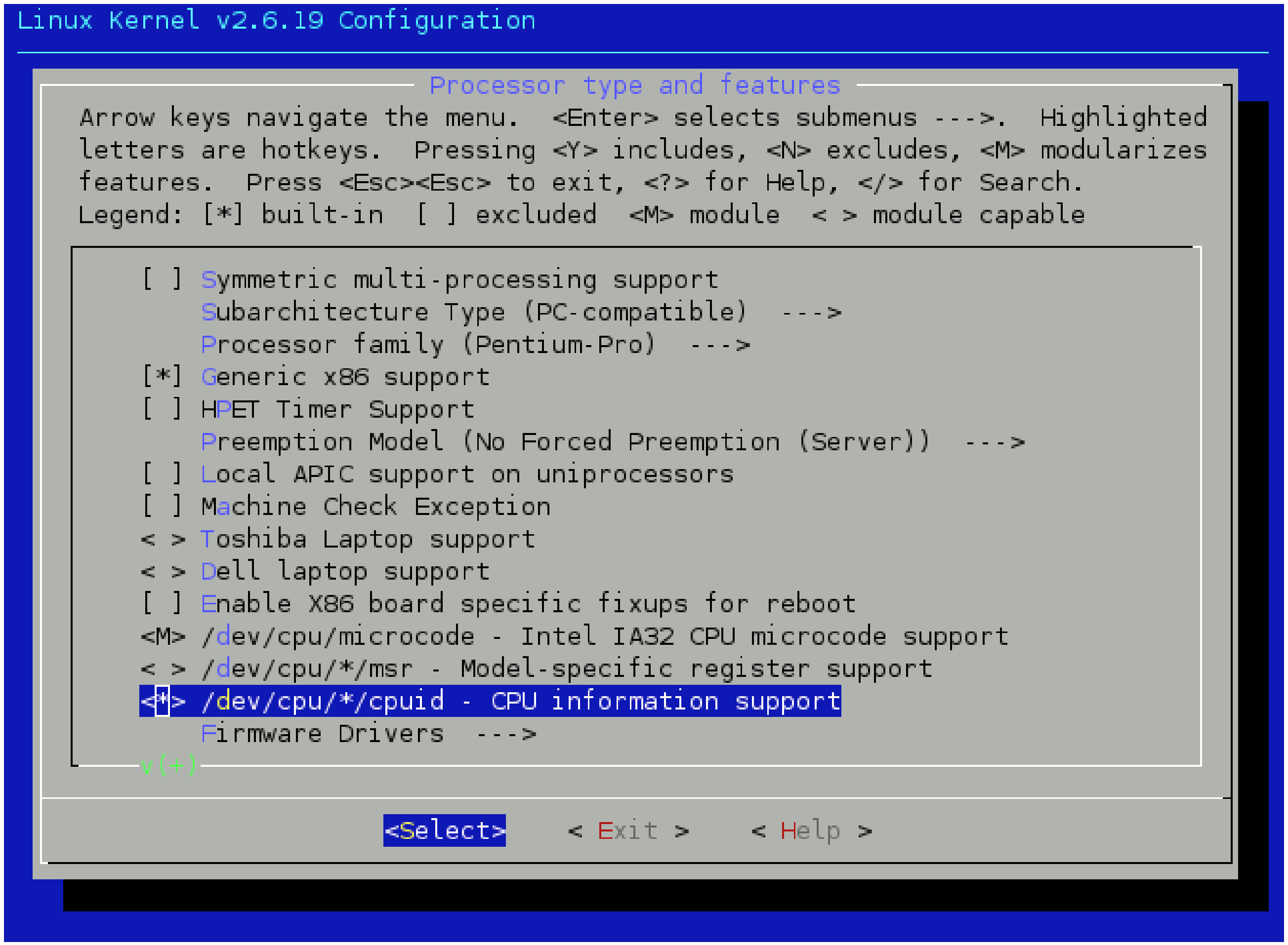
New GTK based graphical configuration interface. Functionality similar to that of `make xconfig`.

Just lacking a search functionality.

Required Debian packages: `libglade2-dev`



make menuconfig



`make menuconfig`

Useful when no graphics are available. Pretty convenient too!

Same interface found in other tools: BusyBox, buildroot...

Required Debian packages: `libncurses-dev`

make oldconfig

`make oldconfig`

- Needed very often!
- Useful to upgrade a `.config` file from an earlier kernel release
- Issues warnings for configuration parameters that no longer exist in the new kernel.
- Asks for values for new parameters

If you edit a `.config` file by hand, it's strongly recommended to run `make oldconfig` afterwards!

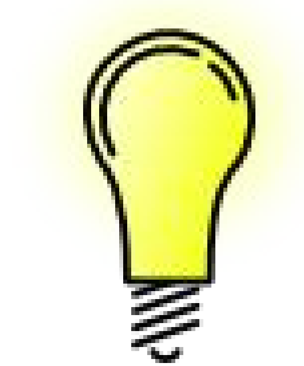
`make allnoconfig`

- Only sets strongly recommended settings to `y`.
- Sets all other settings to `n`.
- Very useful in embedded systems to select only the minimum required set of features and drivers.
- Much more convenient than unselecting hundreds of features one by one!

Undoing configuration changes

A frequent problem:

- After changing several kernel configuration settings, your kernel no longer works.
- If you don't remember all the changes you made, you can get back to your previous configuration:
  `> cp .config.old .config`
- All the configuration interfaces of the kernel (`xconfig`, `menuconfig`, `allnoconfig`...) keep this `.config.old` backup copy.



Configuration per architecture

- The set of configuration options is architecture dependent
  - Some configuration options are very architecture-specific
  - Most of the configuration options (global kernel options, network subsystem, filesystems, most of the device drivers) are visible in all-architecture
- By default, the kernel build system assumes that the kernel is being built for the host architecture, i.e native compilation
- The architecture is not defined inside the configuration, but at an higher level
- We will see later how to override this behaviour, to allow the configuration of kernels for a different architecture

Overview of kernel options (1)

- General setup
  - *Prompt for development/incomplete code* allows to be able to enable drivers or features that are not considered as completely stable yet
  - *Local version - append to kernel release* allows to concatenate an arbitrary string to the kernel version that an user can get using `uname -r`. Very useful for support!
  - *Support for swap*, can usually be disabled on most embedded devices
  - *Configure standard kernel features (for small systems)* allows to remove features from the kernel to reduce its size. Powerful, use with care!

Overview of kernel options (2)

- Loadable module support
  - Allows to enable or completely disable module support. If your system doesn't need kernel modules, best to disable since it saves a significant amount of space and memory
- Enable the block layer
  - If CONFIG_EMBEDDED is enabled, the block layer can be completely removed. Embedded systems using only Flash storage can safely disable the block layer
- Processor type and features (x86) or System type (ARM) or CPU selection (MIPS)
  - Allows to select the CPU or machine for which the kernel must be compiled
  - On x86, only optimization-related, on other architectures very important since there's no compatibility

Overview of kernel options (3)

- Kernel features
  - Tickless system, which allows to disable the regular timer tick and use on-demand ticks instead. Improves power savings
  - High resolution timer support. By default, the resolution of timer is the tick resolution. With high resolution timers, the resolution is as precise as the hardware can give
  - Preemptible kernel enables the preemption inside the kernel code (the userspace code is always preemptible). See our real-time presentation for details
- Power management
  - Global power management option needed for all power management related features
  - Suspend to RAM, CPU frequency scaling, CPU idle control, suspend to disk

Overview of kernel options (4)

- Networking support
  - The network stack
  - Networking options
    - Unix sockets, needed for a form of inter-process communication
    - TCP/IP protocol with options for multicast, routing, tunneling, Ipsec, Ipv6, congestion algorithms, etc.
    - Other protocols such as DCCP, SCTP, TIPC, ATM
    - Ethernet bridging, QoS, etc.
  - Support for other types of network
    - CAN bus, Infrared, Bluetooth, Wireless stack, WiMax stack, etc.

Overview of kernel options (5)

- Device drivers
  - MTD is the subsystem for Flash (NOR, NAND, OneNand, battery-backed memory, etc.)
  - Parallel port support
  - Block devices, a few misc block drivers such as loopback, NBD, etc.
  - ATA/ATAPI, support for IDE disk, CD-ROM and tapes. A new stack exists
  - SCSI
    - The SCSI core, needed not only for SCSI devices but also for USB mass storage devices, SATA and PATA hard drives, etc.
    - SCSI controller drivers

Overview of kernel options (6)

- Device drivers (cont)
  - SATA and PATA, the new stack for hard disks, relies on SCSI
  - RAID and LVM, to aggregate hard drivers and do replication
  - Network device support, with the network controller drivers. Ethernet, Wireless but also PPP
  - Input device support, for all types of input devices: keyboards, mices, joysticks, touchscreens, tablets, etc.
  - Character devices, contains various device drivers, amongst them
    - serial port controller drivers
    - PTY driver, needed for things like SSH or telnet
  - I2C, SPI, 1-wire, support for the popular embedded buses
  - Hardware monitoring support, infrastructure and drivers for thermal sensors

Overview of kernel options (7)

- Device drivers (cont)
  - Watchdog support
  - Multifunction drivers are drivers that do not fit in any other category because the device offers multiple functionality at the same time
  - Multimedia support, contains the V4L and DVB subsystems, for video capture, webcams, AM/FM cards, DVB adapters
  - Graphics support, infrastructure and drivers for framebuffers
  - Sound card support, the OSS and ALSA sound infrastructures and the corresponding drivers
  - HID devices, support for the devices that conform to the HID specification (Human Input Devices)

Overview of kernel options (8)

- Device drivers (cont)
  - USB support
    - Infrastructure
    - Host controller drivers
    - Device drivers, for devices connected to the embedded system
    - Gadget controller drivers
    - Gadget drivers, to let the embedded system act as a mass-storage device, a serial port or an Ethernet adapter
  - MMC/SD/SDIO support
  - LED support
  - Real Time Clock drivers
  - Voltage and current regulators
  - Staging drivers, crappy drivers being cleaned up

Overview of kernel options (9)

- For some categories of devices the driver is not implemented inside the kernel
  - Printers
  - Scanners
  - Graphics drivers used by X.org
  - Some USB devices
- For these devices, the kernel only provides a mechanism to access the hardware, the driver is implemented in userspace

Overview of kernel options (10)

- File systems
  - The common Linux filesystems for block devices: ext2, ext3, ext4
  - Less common filesystems: XFS, JFS, ReiserFS, GFS2, OCFS2, Btrfs
  - CD-ROM filesystems: ISO9660, UDF
  - DOS/Windows filesystems: FAT and NTFS
  - Pseudo filesystems: proc and sysfs
  - Miscellanous filesystems, with amongst other Flash filesystems such as JFFS2, UBIFS, SquashFS, cramfs
  - Network filesystems, with mainly NFS and SMB/CIFS
- Kernel hacking
  - Debugging features useful for kernel developers

Compiling and installing the kernel for the host system

Kernel compilation

- `make`
  - in the main kernel source directory
  - Remember to run `make -j 4` if you have multiple CPU cores to speed up the compilation process
  - No need to run as root !
- Generates
  - `vmlinux`, the raw uncompressed kernel image, at the ELF format, useful for debugging purposes, but cannot be booted
  - `arch/<arch>/boot/*Image`, the final, usually compressed, kernel image that can be booted
    - `bzImage` for x86, `zImage` for ARM, `vmImage.gz` for Blackfin, etc.
  - All kernel modules, spread over the kernel source tree, as `.ko` files.

Kernel installation

- `make install`
  - Does the installation for the host system by default, so needs to be run as root
- Installs
  - `/boot/vmlinuz-<version>`
    Compressed kernel image. Same as the one in `arch/<arch>/boot`
  - `/boot/System.map-<version>`
    Stores kernel symbol addresses
  - `/boot/config-<version>`
    Kernel configuration for this version
- Typically re-runs the bootloader configuration utility to take into account the new kernel.

Module installation

- `make modules_install`
  - Does the installation for the host system by default, so needs to be run as root
- Installs all modules in /lib/modules/<version>/
  - `kernel/`
    Module `.ko` (Kernel Object) files, in the same directory structure as in the sources.
  - `modules.alias`
    Module aliases for module loading utilities. Example line:
    `alias sound-service-?-0 snd_mixer_oss`
  - `modules.dep`
    Module dependencies
  - `modules.symbols`
    Tells which module a given symbol belongs to.

Kernel cleanup targets

- Clean-up generated files
  (to force re-compiling drivers):
  `make clean`
- Remove all generated files. Needed when switching from one architecture to another
  Caution: also removes your `.config` file!
  `make mrproper`
- Also remove editor backup and patch reject files:
  (mainly to generate patches):
  `make distclean`

Compiling and booting Linux

Linux device files

Character device files

- Accessed through a sequential flow of individual characters
- Character devices can be identified by their `c` type (`ls -l`):

```
crw-rw---- 1 root uucp   4,  64 Feb 23 2004 /dev/ttyS0
crw--w---- 1 jdoe tty  136,   1 Feb 23 2004 /dev/pts/1
crw------- 1 root root  13,  32 Feb 23 2004 /dev/input/mouse0
crw-rw-rw- 1 root root   1,   3 Feb 23 2004 /dev/null
```

- Example devices: keyboards, mice, parallel port, IrDA, Bluetooth port, consoles, terminals, sound, video...

Block device files

- Accessed through data blocks of a given size. Blocks can be accessed in any order.
- Block devices can be identified by their `b` type (`ls -l`):

```
brw-rw----   1 root disk     3,   1 Feb 23  2004 hda1
brw-rw----   1 jdoe floppy   2,   0 Feb 23  2004 fd0
brw-rw----   1 root disk     7,   0 Feb 23  2004 loop0
brw-rw----   1 root disk     1,   1 Feb 23  2004 ram1
brw-------   1 root root     8,   1 Feb 23  2004 sda1
```

- Example devices: hard or floppy disks, ram disks, loop devices...

Device major and minor numbers

As you could see in the previous examples, device files have 2 numbers associated to them:

- First number: *major* number
- Second number: *minor* number
- Major and minor numbers are used by the kernel to bind a driver to the device file. Device file names don't matter to the kernel!
- To find out which driver a device file corresponds to, or when the device name is too cryptic, see `Documentation/devices.txt`.

Device file creation

- Device files are not created when a driver is loaded.
- They have to be created in advance:
  `sudo mknod /dev/<device> [c|b] <major> <minor>`
- Examples:

```
sudo mknod /dev/ttyS0 c 4 64
sudo mknod /dev/hda1 b 3 1
```

- Configure your kernel
- Compile it
- Boot it on a virtual PC
- Modify a root filesystem image by adding entries to the `/dev/` directory

Compiling and booting Linux

Overall system startup

Traditional booting sequence

Bootloader
- Executed by the hardware at a fixed location in ROM / Flash
- Initializes support for the device where the kernel image is found (local storage, network, removable media)
- Loads the kernel image in RAM
- Executes the kernel image (with a specified command line)

Kernel
- Uncompresses itself
- Initializes the kernel core and statically compiled drivers (needed to access the root filesystem)
- Mounts the root filesystem (specified by the root kernel parameter)
- Executes the first userspace program (specified by the init kernel parameter)

First userspace program
- Configures userspace and starts up system services

Kernel command line parameters

The Linux kernel can be given parameters at boot time

- Kernel command line arguments are part of the bootloader configuration settings.
- They are copied to RAM by the bootloader, to a location where the kernel expects them.
- Useful to modify the behavior of the kernel at boot time, without having to recompile it.
- Useful to perform advanced kernel and driver initialization, without having to use complex user-space scripts.

Kernel command line example

HP iPAQ h2200 PDA booting example:

`root=/dev/ram0 \`	Root filesystem (first ramdisk)
`rw \`	Root filesystem mounting mode
`init=/linuxrc \`	First userspace program
`console=ttyS0,115200n8 \`	Console (serial)
`console=tty0 \`	Other console (framebuffer)
`ramdisk_size=8192 \`	Misc parameters...
`cachepolicy=writethrough`	

Hundreds of command line parameters described on `Documentation/kernel-parameters.txt`

Drawbacks

- Assumption that all device drivers needed to mount the root filesystem (storage and filesystem drivers) are statically compiled inside the kernel.
- Assumption can be correct for most embedded systems, where the hardware is known and the kernel can be fine-tuned for the system.
- Assumption is mostly wrong for desktop and servers, since a single kernel image should support a wide range of devices and filesystems
  - More flexibility was needed
  - Modules have this flexibility, but they are not available before mounting the root filesystem
  - Need to handle complex setups (RAID, NFS, etc.)

Solution

- A solution is to include a small temporary root filesystem with modules, in the kernel itself. This small filesystem is called the *initramfs.*
- This initramfs is a gzipped cpio archive of this basic root filesystem
  - A gzipped cpio archive is a kind of zip file, with a much simpler format
- The initramfs scripts will detect the hardware, load the corresponding kernel modules, and mount the real root filesystem.
- Finally the initramfs scripts will run the init application in the real root filesystem and the system can boot as usual.
- The initramfs technique completely replaces init ramdisks (initrds). Initrds were used in Linux 2.4, but are no longer needed.

Booting sequence with initramfs

Bootloader
- Executed by the hardware at a fixed location in ROM / Flash
- Initializes support for the device where the images are found (local storage, network, removable media)
- Loads the kernel image in RAM
- Executes the kernel image (with a specified command line)

Kernel
- Uncompresses itself
- Initializes the kernel core and statically compiled drivers
- Uncompresses an initramfs cpio archive (if existing, in the kernel image or copied to memory by the bootloader) and extracts it to the kernel file cache (no mounting, no filesystem).
- If found in the initramfs, executes the first userspace program: `/init`

unchanged

Userspace: `/init` script (what follows is just a typical scenario)
- Runs userspace commands to configure the device (such as network setup, mounting `/proc` and `/sys`...)
- Mounts a new root filesystem. Switch to it (`switch_root`)
- Runs `/sbin/init`

Userspace: `/sbin/init`
- Runs commands to configure the device (if not done yet in the initramfs)
- Starts up system services (daemons, servers) and user programs

Initramfs features and advantages

- Root filesystem directly embedded in the kernel image, or copied to RAM by the bootloader, simple solution.
- Just a plain compressed cpio archive extracted in the file cache. Neither needs a block nor a filesystem driver.
- Simpler to mount complex filesystems from flexible userspace scripts rather than from rigid kernel code. More complexity moved out to user-space!
- Possible to add non GPL files (firmware, proprietary drivers) in the filesystem. This is not linking, just file aggregation (not considered as a derived work by the GPL).

How to populate an initramfs

Using `CONFIG_INITRAMFS_SOURCE`
in kernel configuration (`General Setup` section)

- Either give an existing `cpio` archive (file name ending with `.cpio`)
- Or give a directory to be archived.
- Any other regular file will be taken as a text specification file (see next page).

see `Documentation/filesystems/ramfs-rootfs-initramfs.txt` and `Documentation/early-userspace/README` in kernel sources.

See also http://www.linuxdevices.com/articles/AT4017834659.html for a nice overview of initramfs (by Rob Landley).

Initramfs specification file example

major minor

```
dir /dev 755 0 0
nod /dev/console 644 0 0 c 5 1
nod /dev/loop0 644 0 0 b 7 0
dir /bin 755 1000 1000
file /bin/busybox /stuff/initramfs/busybox 755 0 0
slink /bin/sh busybox 777 0 0
dir /proc 755 0 0
dir /sys 755 0 0
dir /mnt 755 0 0
file /init /stuff/initramfs/init.sh 755 0 0
```

permissions

user id group id

No need for `root` user access!

Summary

- For embedded systems, two interesting solutions
  - No initramfs: all needed drivers are included inside the kernel, and the final root filesystem is mounted directly
  - Everything inside the initramfs

Compiling and booting Linux

Root filesystem over NFS

Usefulness of rootfs on NFS

Once networking works, your root filesystem could be a directory on your GNU/Linux development host, exported by NFS (Network File System). This is very convenient for system development:

- Makes it very easy to update files (driver modules in particular) on the root filesystem, without rebooting. Much faster than through the serial port.
- Can have a big root filesystem even if you don't have support for internal or external storage yet.
- The root filesystem can be huge. You can even build native compiler tools and build all the tools you need on the target itself (better to cross-compile though).

NFS boot setup (1)

On the host (NFS server)

- Install an NFS server (example: Debian, Ubuntu)
  `sudo apt-get install nfs-kernel-server`
- Add the exported directory to your `/etc/exports` file:
  `/home/rootfs 192.168.1.111(rw,no_root_squash,no_subtree_check)`
  client address NFS server options
- Start or restart your NFS server (example: Debian, Ubuntu)
  `sudo /etc/init.d/nfs-kernel-server restart`

NFS boot setup (2)

On the target (NFS client)

- Compile your kernel with `CONFIG_NFS_FS=y`, `CONFIG_IP_PNP=y` (configure IP at boot time) and `CONFIG_ROOT_NFS=y`
- Boot the kernel with the below command line options:
  `root=/dev/nfs`
  virtual device
  `ip=192.168.1.111`
  local IP address
  `nfsroot=192.168.1.110:/home/nfsroot`
  NFS server IP address Directory on the NFS server

Compiling and booting Linux

Cross-compiling the kernel

Cross-compiling the kernel

When you compile a Linux kernel for another CPU architecture

- Much faster than compiling natively, when the target system is much slower than your GNU/Linux workstation.
- Much easier as development tools for your GNU/Linux workstation are much easier to find.
- To make the difference with a native compiler, cross-compiler executables are prefixed by the name of the target system, architecture and sometimes library. Examples:

```
mips-linux-gcc
m68k-linux-uclibc-gcc
arm-linux-gnueabi-gcc
```

Specifying cross-compilation

The CPU architecture and cross-compiler prefix are defined through the ARCH and CROSS_COMPILE variables in the toplevel Makefile.

- The Makefile defines CC = $(CROSS_COMPILE)gcc
  See comments in Makefile for details
- The easiest solution is to modify the Makefile.
  Example, ARM platform, cross-compiler: arm-linux-gcc

```
ARCH    ?= arm
CROSS_COMPILE  ?= arm-linux-
```

- Other solutions
  - Pass ARCH and CROSS_COMPILE on the make command line
  - Define ARCH and CROSS_COMPILE as environment variables
  - Don't forget to have the values properly set at all steps, otherwise the kernel configuration and build system gets confused

Configuring the kernel

`make xconfig`

- Same as in native compiling
- The set of available options will be different
- Don't forget to set the right board / machine type!

Ready-made config files

```
assabet_defconfig       integrator_defconfig  mainstone_defconfig
badge4_defconfig        iq31244_defconfig     mx1ads_defconfig
bast_defconfig          iq80321_defconfig     neponset_defconfig
cerfcube_defconfig      iq80331_defconfig     netwinder_defconfig
clps7500_defconfig      iq80332_defconfig     omap_h2_1610_defconfig
ebsa110_defconfig       ixdp2400_defconfig    omnimeter_defconfig
edb7211_defconfig       ixdp2401_defconfig    pleb_defconfig
enp2611_defconfig       ixdp2800_defconfig    pxa255-idp_defconfig
ep80219_defconfig       ixdp2801_defconfig    rpc_defconfig
epxa10db_defconfig      ixp4xx_defconfig      s3c2410_defconfig
footbridge_defconfig    jornada720_defconfig  shannon_defconfig
fortunet_defconfig      lart_defconfig        shark_defconfig
h3600_defconfig         lpd7a400_defconfig    simpad_defconfig
h7201_defconfig         lpd7a404_defconfig    smdk2410_defconfig
h7202_defconfig         lubbock_defconfig     versatile_defconfig
hackkit_defconfig       lusl7200_defconfig
```

`arch/arm/configs` example

Using ready-made config files

- Default configuration files available for many boards / machines! Check if one exists in `arch/<arch>/configs/` for your target.
- Example: if you found an `acme_defconfig` file, you can run: `make acme_defconfig`
- Using `arch/<arch>/configs/` is a very good good way of releasing a default configuration file for a group of users or developers.

Like all `make` commands, you must run `make <machine>_defconfig` in the toplevel source directory.

Building the kernel

- Run
  `make`
- Copy
  `arch/<arch>/boot/zImage`
  to the target storage
- You can customize `arch/<arch>/boot/install.sh` so that `make install` does this automatically for you.
- `make INSTALL_MOD_PATH=<dir>/ modules_install`
  and copy `<dir>/lib/modules/` to `/lib/modules/` on the target storage.

Practical lab – Cross-compiling

- Set up a cross-compiling environment
- Configure the kernel `Makefile` accordingly
- Cross-compile the kernel for an `arm` target platform
- On this platform, interact with the bootloader and boot your kernel.

Using kernel modules

Loadable kernel modules

- Modules: add a given functionality to the kernel (drivers, filesystem support, and many others).
- Can be loaded and unloaded at any time, only when their functionality is need.
- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).
- Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later.
- Caution: once loaded, have full access to the whole kernel address space. No particular protection.

Module dependencies

- Some kernel modules can depend on other modules, which need to be loaded first.
- Example: the `usb-storage` module depends on the `scsi_mod`, `libusual` and `usbcore` modules.
- Dependencies are described in `/lib/modules/<kernel-version>/modules.dep`
  This file is generated when you run `make modules_install`.

Kernel log

When a new module is loaded,
related information is available in the kernel log.

- The kernel keeps its messages in a circular buffer
(so that it doesn't consume more memory with many messages)
- Kernel log messages are available through the `dmesg` command.
(“**d**iagnostic **mes**sa**g**e”)
- Kernel log messages are also displayed in the system console
(console messages can be filtered by level using the `loglevel`
kernel, or completely disabled with the `quiet` parameter).
- Note that you can write to the kernel log from userspace too:
`echo “Debug info” > /dev/kmsg`

Module utilities (1)

- `modinfo <module_name>`
  `modinfo <module_path>.ko`
  Gets information about a module: parameters, license, description and dependencies.
  Very useful before deciding to load a module or not.
- `sudo insmod <module_path>.ko`
  Tries to load the given module. The full path to the module object file must be given.

Understanding module loading issues

- When loading a module fails,
  `insmod` often doesn't give you enough details!
- Details are often available in the kernel log.
- Example:

```
> sudo insmod ./intr_monitor.ko
insmod: error inserting './intr_monitor.ko': -1
Device or resource busy
> dmesg
[17549774.552000] Failed to register handler for
irq channel 2
```

Module utilities (2)

- `sudo modprobe <module_name>`
  Most common usage of `modprobe`: tries to load all the modules the given module depends on, and then this module. Lots of other options are available. Modprobe automatically looks in `/lib/modules/<version>/` for the object file corresponding to the given module name.
- `lsmod`
  Displays the list of loaded modules
  Compare its output with the contents of `/proc/modules`!

Module utilities (3)

- `sudo rmmod <module_name>`
  Tries to remove the given module.
  Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)
- `sudo modprobe -r <module_name>`
  Tries to remove the given module and all dependent modules (which are no longer needed after the module removal)

Passing parameters to modules

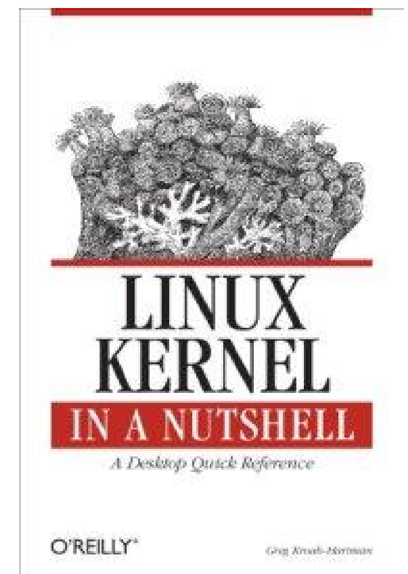
- Find available parameters:
  `modinfo snd-intel8x0m`
- Through `insmod`:
  `sudo insmod ./snd-intel8x0m.ko index=-2`
- Through `modprobe`:
  Set parameters in `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:
  `options snd-intel8x0m index=-2`
- Through the kernel command line,
  when the driver is built statically into the kernel:
  `snd-intel8x0m.index=-2`

driver name
driver parameter name
driver parameter value

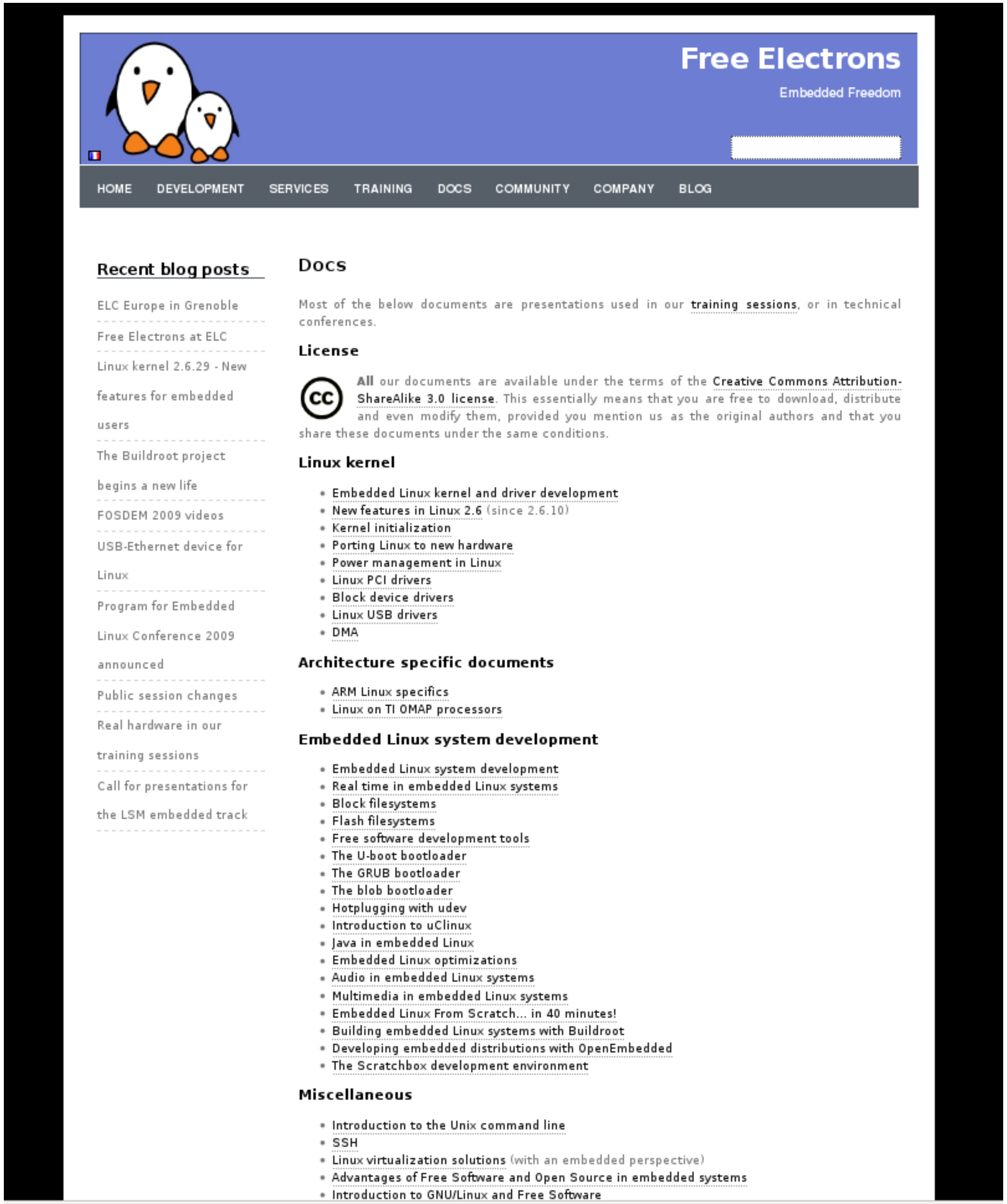
Useful reading

Linux Kernel in a Nutshell, Dec 2006

- By Greg Kroah-Hartman, O'Reilly
  http://www.kroah.com/lkn/
- A good reference book and guide on configuring, compiling and managing the Linux kernel sources.
- Freely available on-line!
  Great companion to the printed book for easy electronic searches!
  Available as single PDF file on
  http://free-electrons.com/community/kernel/lkn/



Related documents



All our technical presentations on http://free-electrons.com/docs

- Linux kernel
- Device drivers
- Architecture specifics
- Embedded Linux system development

How to help

You can help us to improve and maintain this document...

- By sending corrections, suggestions, contributions and translations
- By asking your organization to order development, consulting and training services performed by the authors of these documents (see http://free-electrons.com/).
- By sharing this document with your friends, colleagues and with the local Free Software community.
- By adding links on your website to our on-line materials, to increase their visibility in search engine results.

Free Electrons

Our services

Linux kernel

Linux device drivers
Board support code
Mainstreaming kernel code
Kernel debugging

Embedded Linux Training

All materials released with a free license!

Unix and GNU/Linux basics
Linux kernel and drivers development
Real-time Linux, uClinux
Development and profiling tools
Lightweight tools for embedded systems
Root filesystem creation
Audio and multimedia
System optimization

Custom Development

System integration
Embedded Linux demos and prototypes
System optimization
Application and interface development

Consulting and technical support

Help in decision making
System architecture
System design and performance review
Development tool and application support
Investigating issues and fixing tool bugs