Khóa học : "Basic Embedded Linux"

- **Email** : training.laptrinhnhung@gmail.com
- **Website** : http://hethongnhung.com – http://laptrinhnhung.com

Cross Compiler ToolChain

Hello World

1. Write hello.c:

```
udooer@udooneo:~$ cat hello.c
#include <stdio.h>

void main() {
        printf("Hello World!\n");
}
```

2. Compile hello.c -> hello_local

```
udooer@udooneo:~$ gcc -o hello_local hello.c
```

Hello World

3. Check hello_local

```
udooer@udooneo:~$ file hello_local
hello_local: ELF 32-bit LSB  executable, ARM,
```

4. Run ./hello_local

```
udooer@udooneo:~$ ./hello_local
Hello World!
```

Hello World

1. Write hello.c:

```
udooer@udooneo:~$ cat hello.c
#include <stdio.h>

void main() {
        printf("Hello World!\n");
}
```

2. Compile hello.c -> hello_x86

```
lec@lec-vm:~$ gcc hello.c -o hello_x86
```

Development Host - Wrong way

Hello World

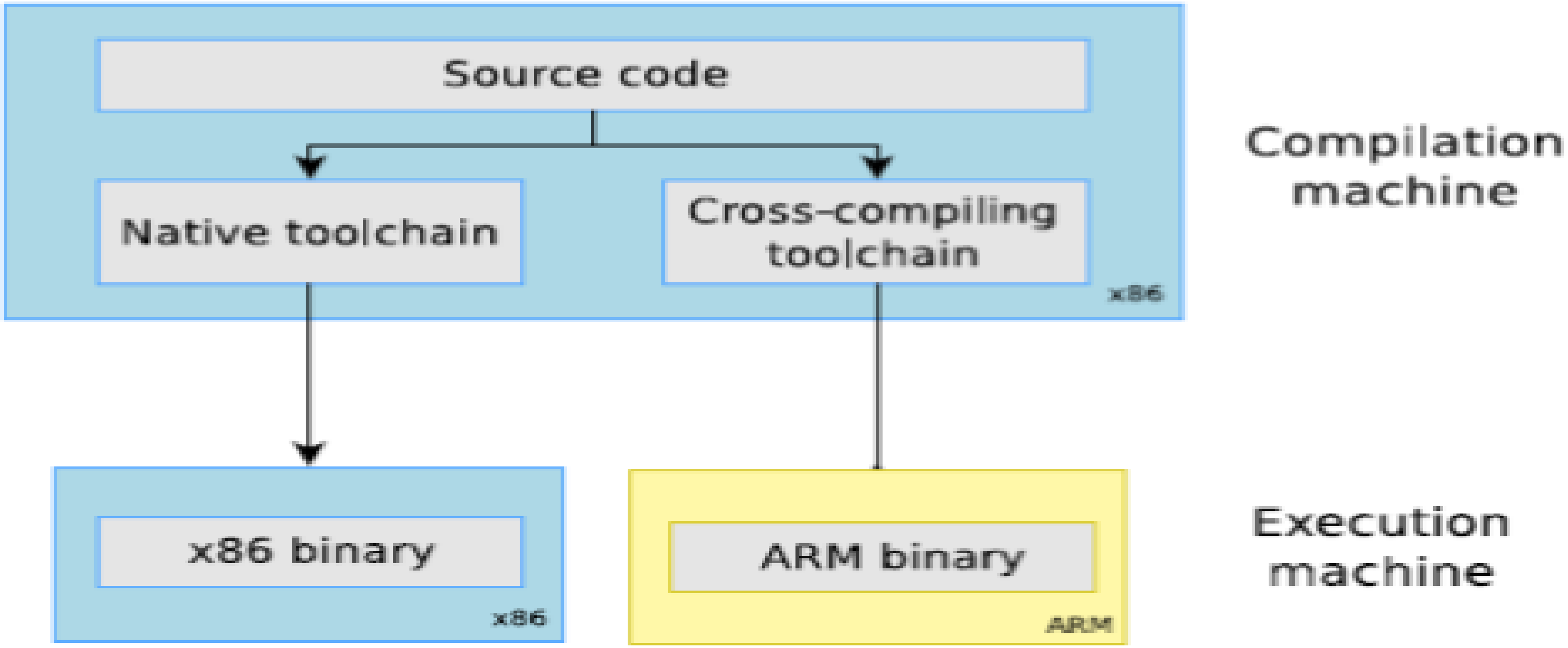
3. Check hello_x86

```
hello_x86: ELF 32-bit LSB  executable, Intel 80386,
```

4. Run ./hello_x86

```
udooer@udooneo:~$ ./hello_x86
-bash: ./hello_x86: cannot execute binary file: Exec format error
```

Cross-Compiling



Set up environment

1. Install Cross Compiler

```
lec@lec-vm:~$ sudo apt-get install gcc-4.7-arm-linux-gnueabihf
Reading package lists... Done
Building dependency tree
```

2. A new GCC

```
lec@lec-vm:~$ ll /usr/bin/arm-linux-gnueabihf-gcc-4.7
-rwxr-xr-x 1 root root 511148 feb 26  2014 /usr/bin/arm-linux-gnueabihf-gcc-4.7*
```

Hello World

1. Write hello.c:

```
udooer@udooneo:~$ cat hello.c
#include <stdio.h>

void main() {
        printf("Hello World!\n");
}
```

2. Compile hello.c -> hello_cross_compile

```
lec@lec-vm:~$ /usr/bin/arm-linux-gnueabi-gcc -o hello_cross_compile hello.c
```

Development Host

3. Check hello_cross_compile

```
lec@lec-vm:~$ file hello_cross_compile
hello_cross_compile: ELF 32-bit LSB  executable, ARM,
```

4. Copy to UDOO

```
lec@lec-vm:~$ scp hello_cross_compile udooer@192.168.7.2:~/
udooer@192.168.7.2's password:
hello_cross_compile
```

5. Run ./hello_cross_compile

```
udooer@udooneo:~$ ./hello_local
Hello World!
```

Kernel configuration

Kernel configuration and build system

- The kernel configuration and build system is based on multiple Makefiles
- One only interacts with the main *Makefile*, present at the **top directory** of the kernel source tree
- Interaction takes place
  - using the *make* tool, which parses the *Makefile*
  - through various **targets**, defining which action should be done (configuration, compilation, installation, etc.). Run *make help* to see all available targets.
- Example
  - *cd linux-3.6.x/*
  - *make <target>*

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 (for device drivers, etc.)
  - On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.)

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 or text interfaces:
  - *make xconfig, make gconfig* (graphical)
  - *make menuconfig, make nconfig* (text)
  - You can switch from one to another, they all load/save the same *.config* file, and show the same set of options
- To modify a kernel in a GNU/Linux distribution: the configuration files are usually released in */boot/*, together with kernel images: */boot/config-3.2.0-31-generic*

Kernel or module?

- The **kernel image** is a **single file**, resulting from the linking of all object files that correspond to features enabled in the configuration
  - This is the file that gets loaded in memory by the bootloader
  - All included features are therefore available as soon as the kernel starts, at a time where no filesystem exists
- Some features (device drivers, filesystems, etc.) can however be compiled as **modules**
  - These are plugins that can be loaded/unloaded dynamically to add/remove features to the kernel
  - Each **module is stored as a separate file in the filesystem**, and therefore access to a filesystem is mandatory to use modules
  - This is not possible in the early boot procedure of the kernel, because no filesystem is available

Kernel option types

There are different types of options

- *bool* options, they are either
  - *true* (to include the feature in the kernel) or
  - *false* (to exclude the feature from the kernel)
- *tristate* options, they are either
  - *true* (to include the feature in the kernel image) or
  - *module* (to include the feature as a kernel module) or
  - *false* (to exclude the feature)
- *int* options, to specify integer values
- *hex* options, to specify hexadecimal values
- *string* options, to specify string values

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 the ones that cannot be selected because of missing dependencies. In this
case, they are displayed in gray.

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
- Search interface to look for parameters
- Required Debian / Ubuntu packages: *libqt4-dev g++*

make gconfig

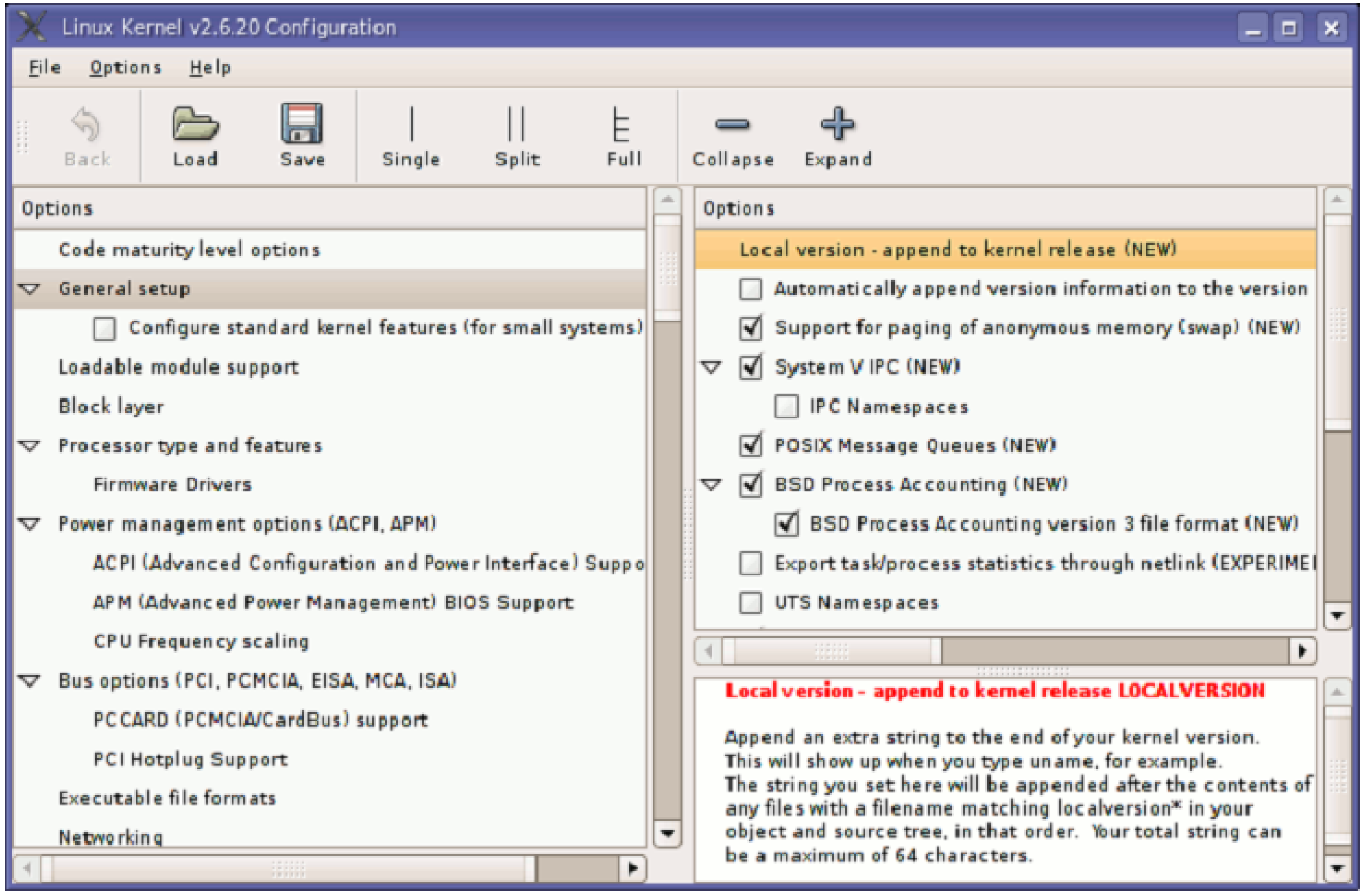
make gconfig

- GTK based graphical configuration interface.

Functionality similar to that of make *xconfig*.

- Just lacking a search functionality.
- Required Debian packages:

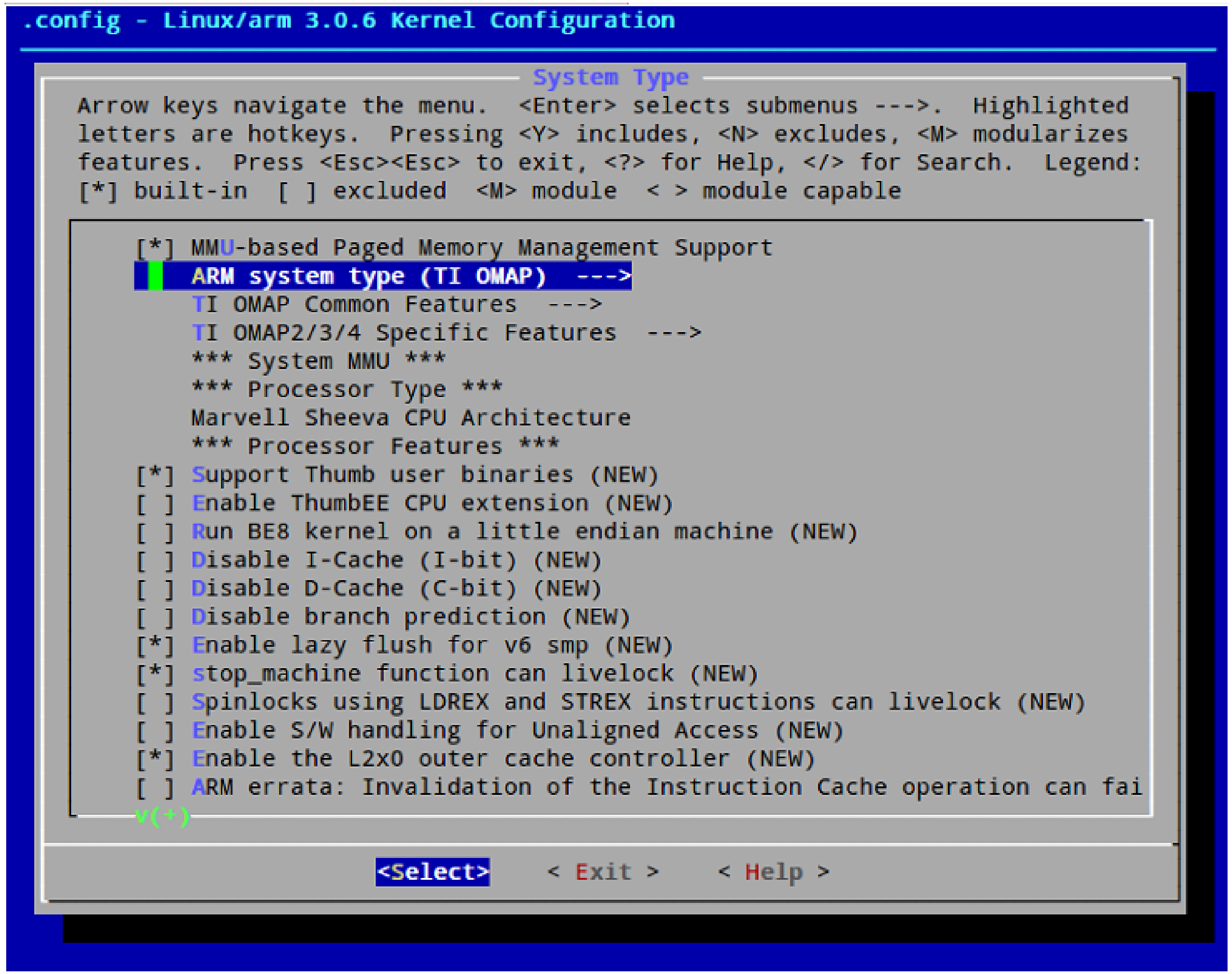
libglade2-dev



make menuconfig

make menuconfig

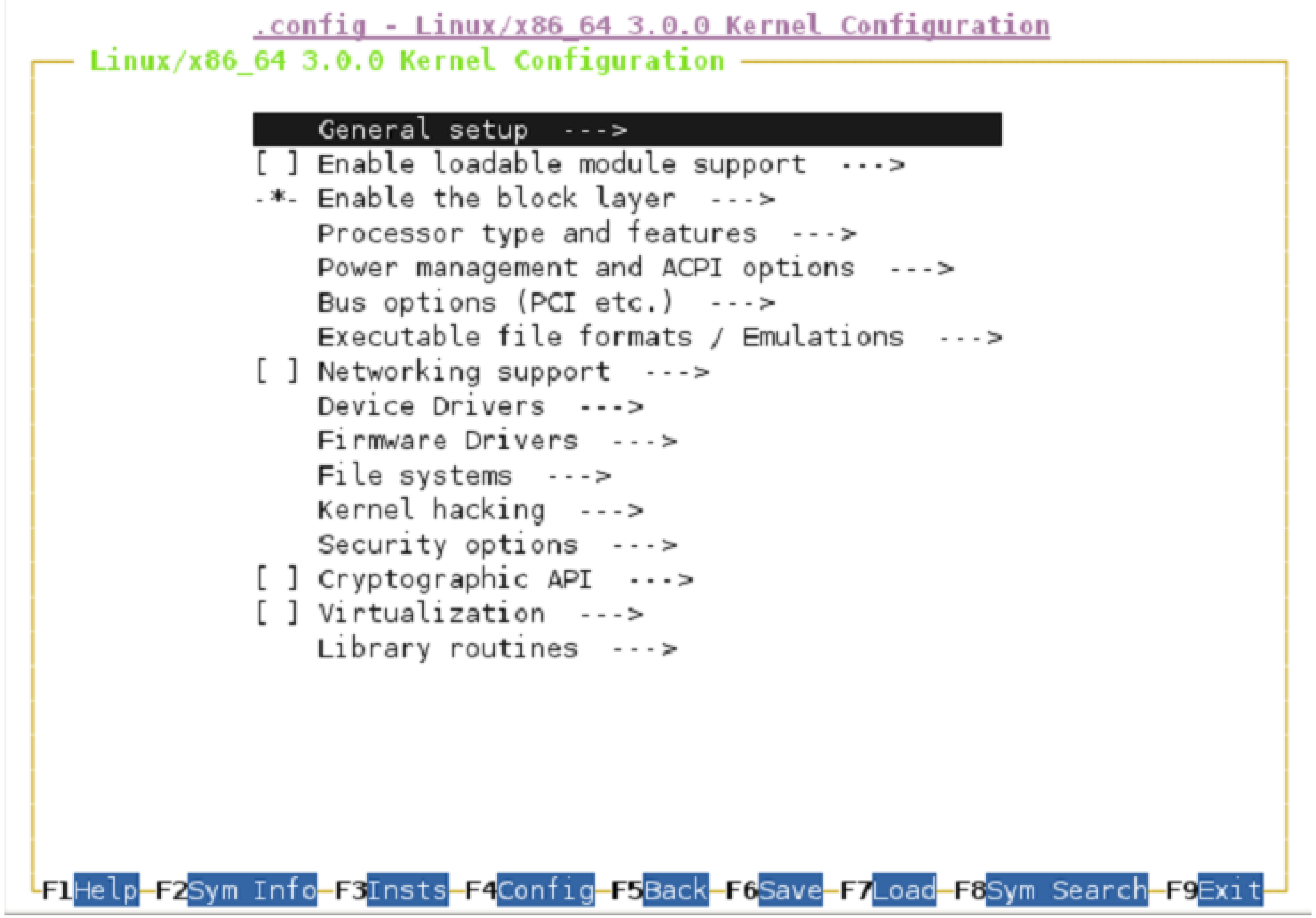
- Useful when no graphics are available. Pretty convenient too!
- Same interface found in other tools: BusyBox, Buildroot...
- Required Debian packages: *libncurses-dev*



make nconfig

make nconfig

- A newer, similar text interface
- More user friendly (for example, easier to access help information).
- 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 (while *xconfig* and *menuconfig* silently set default values for new parameters).

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

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*, *oldconfig*...) 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 architectures.
- 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 a higher level
- We will see later how to override this behaviour, to allow the configuration of kernels for a different architecture

Compiling and installing the kernel for the host system

Kernel compilation

- *make*
  - in the main kernel source directory
  - Remember to run multiple jobs in parallel if you have multiple CPU cores. Example: *make -j 4*
  - No need to run as root!
- Generates
  - *vmlinux*, the raw uncompressed kernel image, in 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.
  - *arch/<arch>/boot/dts/*.dtb*, compiled Device Tree files (on some architectures)
  - 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. Generally not used when compiling for an embedded system, as it installs files on the development workstation.
- 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 the new kernel into account.

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, modules.dep.bin (binary hashed)
  Module dependencies
- modules.symbols, modules.symbols.bin (binary hashed)
  Tells which module a given symbol belongs to.

Kernel cleanup targets

- Clean-up generated files (to force re-compilation):

make clean

- Remove all generated files. Needed when switching from one architecture to another. Caution: it also removes your .config file!

make mrproper

- Also remove editor backup and patch reject files (mainly to generate patches):

make distclean

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, the prefix is *mips-linux-arm-linux-gnueabi-gcc*, the prefix is *arm-linux-gnueabi-*

Specifying cross-compilation (1)

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

- *ARCH* is the name of the architecture. It is defined by the name of the subdirectory in *arch/* in the kernel sources
- Example: *arm* if you want to compile a kernel for the *arm* architecture.
- *CROSS_COMPILE* is the prefix of the cross compilation tools
- Example: *arm-linux-* if your compiler is *arm-linux-gcc*

Specifying cross-compilation (2)

Two solutions to define *ARCH* and *CROSS_COMPILE*:

- Pass *ARCH* and *CROSS_COMPILE* on the make command line:

make ARCH=arm CROSS_COMPILE=arm-linux- ...

Drawback: it is easy to forget to pass these variables when you run any *make* command, causing your build and configuration to be screwed up.

- Define *ARCH* and *CROSS_COMPILE* as environment variables:

export ARCH=arm

export CROSS_COMPILE=arm-linux-

Drawback: it only works inside the current shell or terminal.

You could put these settings in a file that you source every time you start working on the project. If you only work on a single architecture with always the same toolchain, you could even put these settings in your *~/.bashrc* file to make them permanent and visible from any terminal.

Predefined configuration files

- Default configuration files available, per board or per-CPU family
- They are stored in *arch/<arch>/configs/*, and are just minimal *.config* files
- This is the most common way of configuring a kernel for embedded platforms
- Run *make help* to find if one is available for your platform
- To load a default configuration file, just run

  make acme_defconfig
- This will overwrite your existing *.config* file!
- To create your own default configuration file
- *make savedefconfig*, to create a minimal configuration file
- *mv defconfig arch/<arch>/configs/myown_defconfig*

Configuring the kernel

- After loading a default configuration file, you can adjust the configuration to your needs with the normal *xconfig*, *gconfig* or *menuconfig* interfaces
- As the architecture is different from your host architecture
  - Some options will be different from the native configuration (processor and architecture specific options, specific drivers, etc.)
  - Many options will be identical (filesystems, network protocols, architecture-independent drivers, etc.)

Device Tree

- Many embedded architectures have a lot of non-discoverable hardware.
- Depending on the architecture, such hardware is either described using C code directly within the kernel, or using a special hardware description language in a Device Tree.
- ARM, PowerPC, OpenRISC, ARC, Microblaze are examples of architectures using the Device Tree.
- A Device Tree Source, written by kernel developers, is compiled into a binary Device Tree Blob, passed at boot time to the kernel.
- There is one different Device Tree for each board/platform supported by the kernel, available in *arch/arm/boot/dts/<board>.dtb*.
- The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.

Building and installing the kernel

- Run *make*
- Copy the final kernel image to the target storage
  - can be *zImage, vmlinux, bzImage* in *arch/<arch>/boot*
  - copying the Device Tree Blob might be necessary as well, they are available in *arch/<arch>/boot/dts*
- *make install* is rarely used in embedded development, as the kernel image is a single file, easy to handle
  - It is however possible to customize the make install behavior in *arch/<arch>/boot/install.sh*
- *make modules_install* is used even in embedded development, as it installs many modules and description files
  - *make INSTALL_MOD_PATH=<dir>/ modules_install*
  - The *INSTALL_MOD_PATH* variable is needed to install the modules in the target root filesystem instead of your host root filesystem.

Questions?