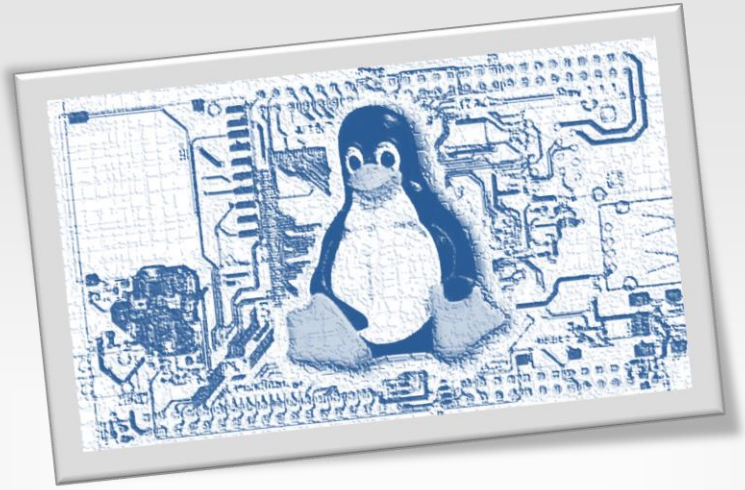


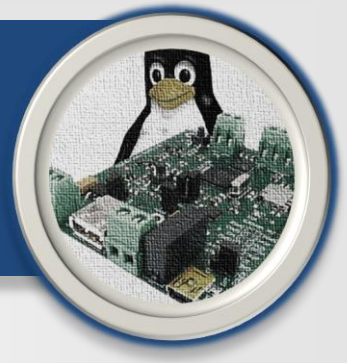
Building Embedded Linux
(Full Tutorial for ARM)

Information Technology Institute (ITI)

Sherif Mousa

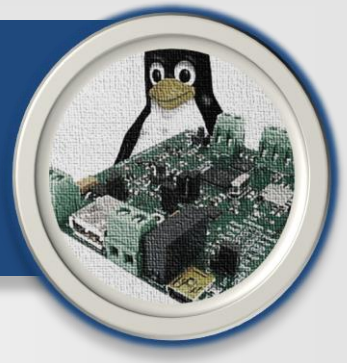
Using Linux in an embedded project is easier, and more fun, than ever ☺

Tips for Linux

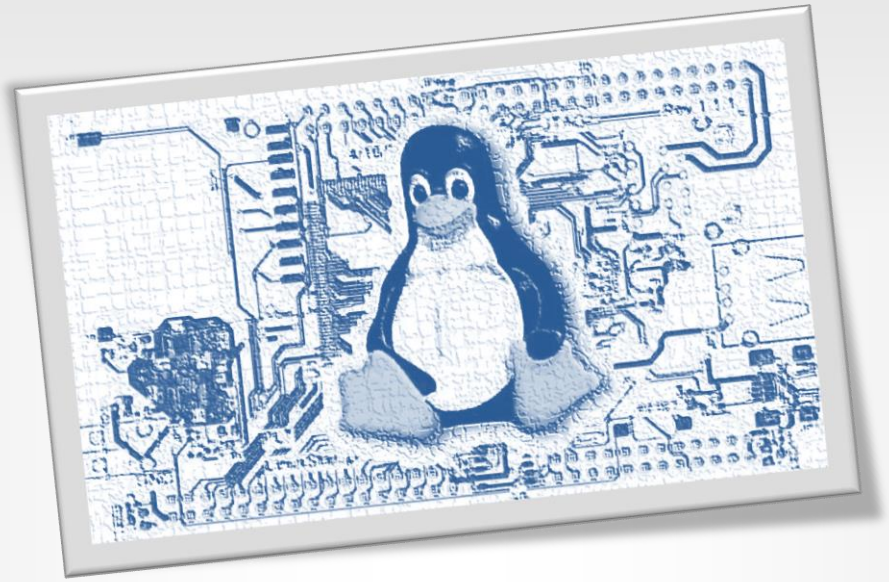


- Read instructions and tips carefully.
- Always read error messages carefully, in particular the first one which is issued.
- Never stay stuck with a strange problem more than 5 minutes.
- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the chown –R command to give the new files back to your regular user.

Tips for Linux

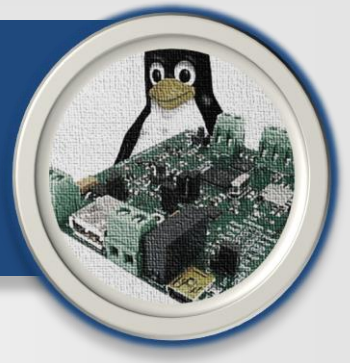


- You should only use the root user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.



Hardware for Linux systems

Processor and Arch.

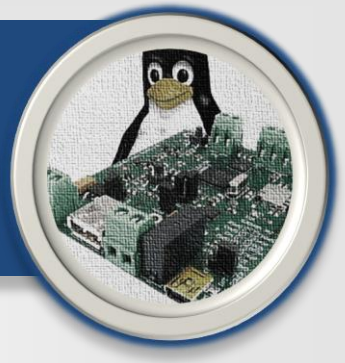


- The Linux kernel and most other architecture-dependent component support a wide range of 32 and 64 bits architectures
  - x86 and x86-64, as found on PC platforms, but also embedded systems (multimedia, industrial)
  - ARM, with hundreds of different SoC (multimedia, industrial)
  - PowerPC (mainly real-time, industrial applications)

RAM & Storage

- RAM: a very basic Linux system can work within 8 MB of RAM, but a more realistic system will usually require at least 32 MB of RAM. Depends on the type and size of applications.
- Storage: a very basic Linux system can work within 4 MB of storage, but usually more is needed.
  - Flash storage is supported, both NAND and NOR flash, with specific filesystems.
  - Block storage including SD/MMC cards and eMMC.

Communication

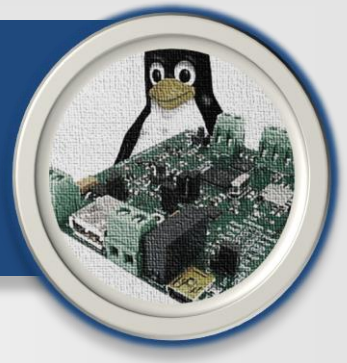


- The Linux kernel has support for many common communication busses
  - USB
  - SPI (Serial Peripheral Interface)
  - CAN (Controller Area Network)
  - 1-wire
- And also extensive networking support
  - Ethernet, Wi, Bluetooth, CAN, etc.
  - IPv4, IPv6, TCP, UDP, etc.
  - Firewalling, advanced routing.

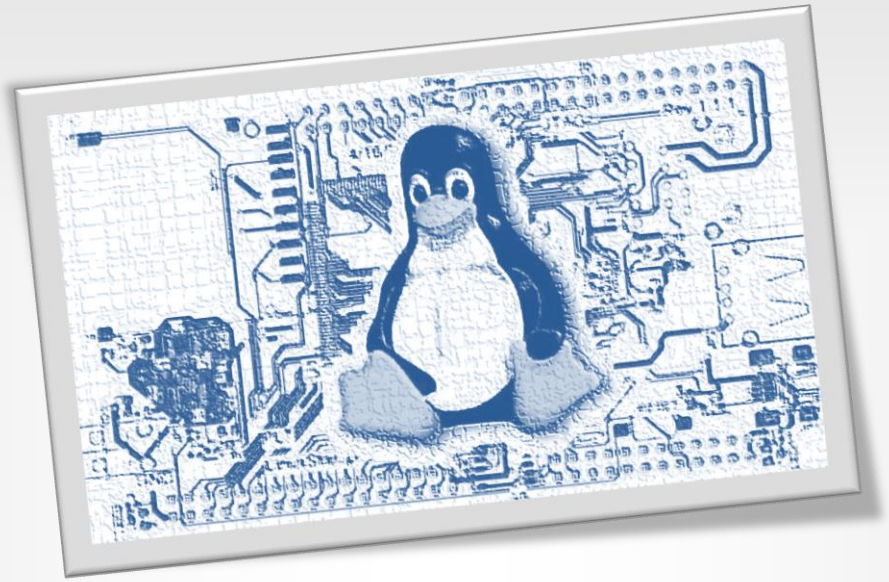
Types of HW platforms

- **Evaluation platforms** from the SoC vendor. Usually expensive, but many peripherals are built-in. Generally unsuitable for real products.
- **Component on Module**, a small board with only CPU/RAM/ash and a few other core components, with connectors to access all other peripherals. Can be used to build end products for small to medium quantities.
- **Custom platform**. Schematics for evaluation boards or development platforms are more and more commonly freely available, making it easier to develop custom platforms. (Open Hardware)

Tips for HW

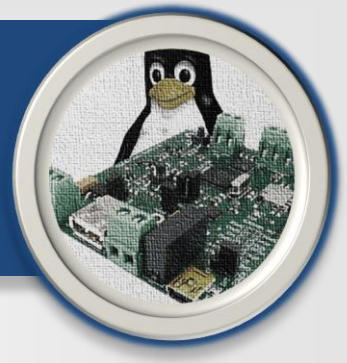


- Make sure the hardware you plan to use is already supported by the Linux kernel, and has an open-source bootloader.
- Having support in the official versions of the projects (kernel, bootloader) is a lot better: quality is better, and new versions are available.
- Some SoC vendors and/or board vendors do not contribute their changes back to the mainline Linux kernel.



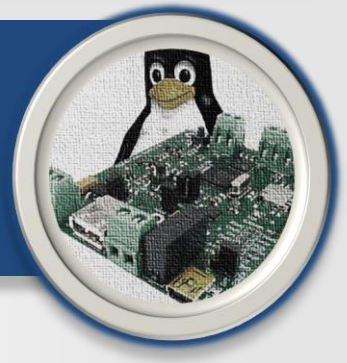
Software Components

Software Components



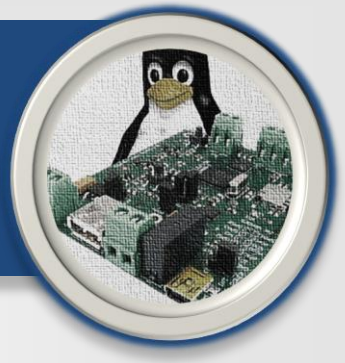
- Cross-compilation ToolChain
- Bootloader
- Linux Kernel
- System libraries (C library)
- Filesystem with other libraries & apps.

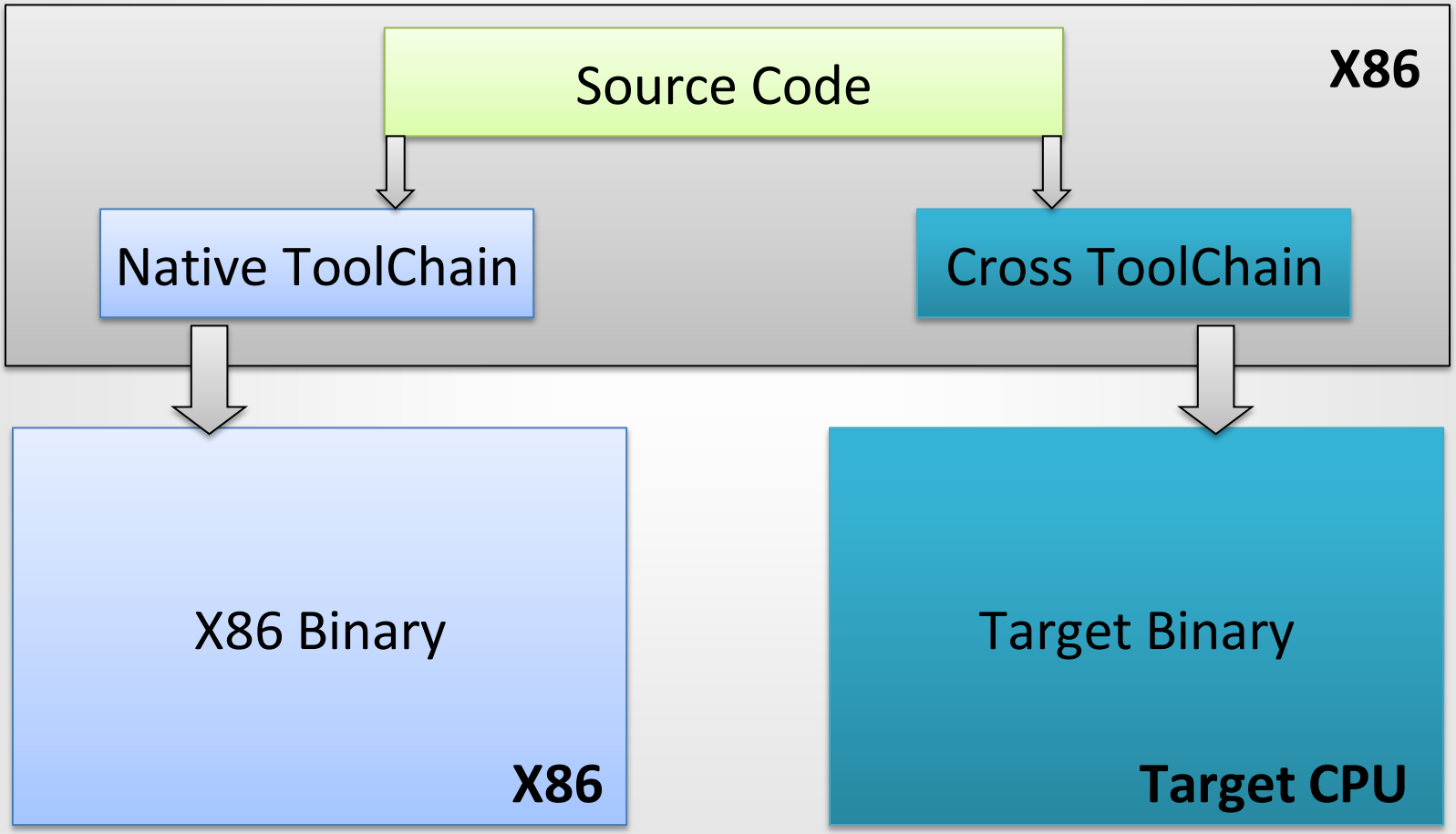
Embedded Linux Output

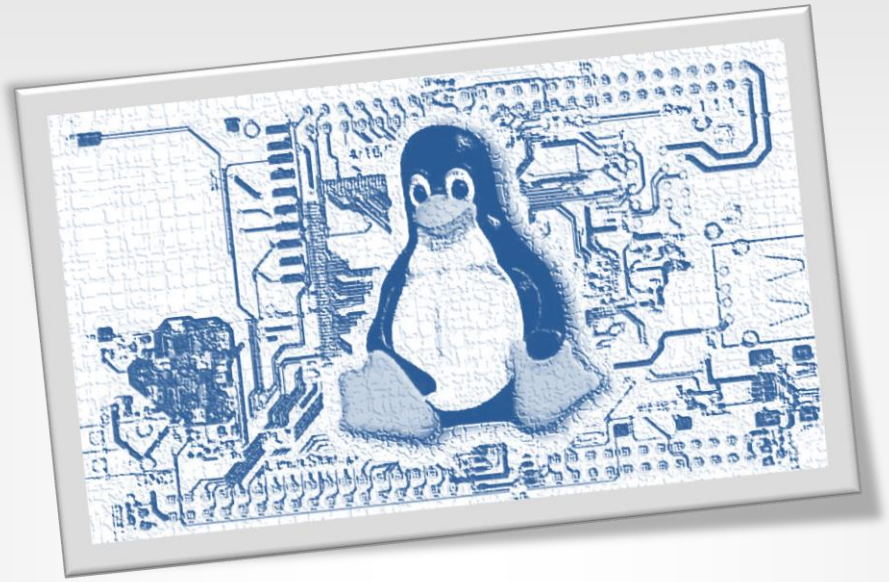


- Board Support Package development
  - A BSP contains a bootloader and kernel with the suitable device drivers for the targeted hardware
- System integration
  - Integrate all the components, bootloader, kernel, third-party libraries and applications and in-house applications into a working system
- Development of applications
  - Normal Linux applications, but using specially chosen libraries

Cross-ToolChain

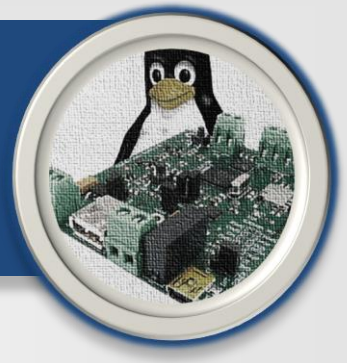




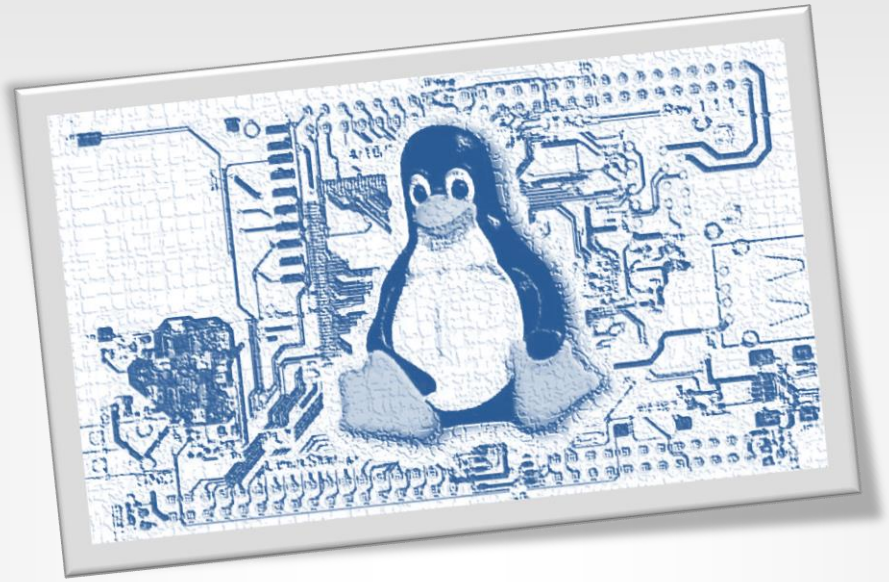


Where to get Cross-ToolChain

Where to get Cross-ToolChain

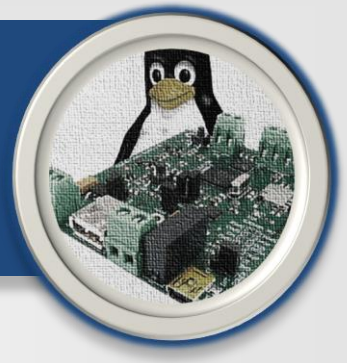


- http://elinux.org/Toolchains
- Codesourcery, ready-made
  - http://www.mentor.com/embedded-software/codesourcery
- Linaro, ready made
  - http://www.linaro.org/
- Buildroot, tool to build
  - http://www.buildroot.net/
- Crosstool-NG, tool to build
  - http://crosstool-ng.org/



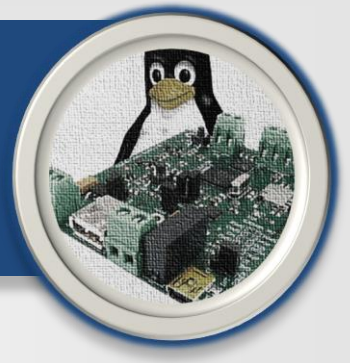
Building Cross-ToolChain

Cross-ToolChain



- Three machines must be distinguished when discussing toolchain creation
  - The **build** machine, where the toolchain is built.
  - The **host** machine, where the toolchain will be executed.
  - The **target** machine, where the binaries created by the toolchain are executed.

Cross-ToolChain components



binutils

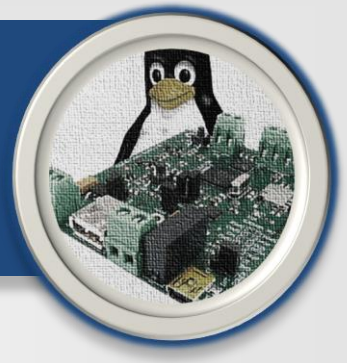
Kernel Headers

C/C++ libs

GCC Compiler

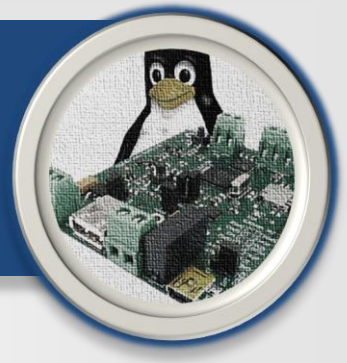
GDB (optional)

Binutils



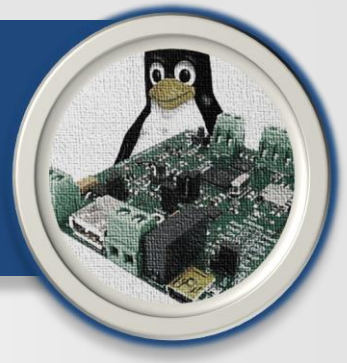
- Binutils is a set of tools to generate and manipulate binaries for a given CPU architecture
  - as, the assembler, that generates binary code from assembler source code
  - ld, the linker
  - ar, ranlib, to generate .a archives, used for libraries
  - objdump, readelf, size, nm, strings, to inspect binaries. Very useful analysis tools!
  - strip, to strip useless parts of binaries in order to reduce their size
- http://www.gnu.org/software/binutils/

Kernel Headers



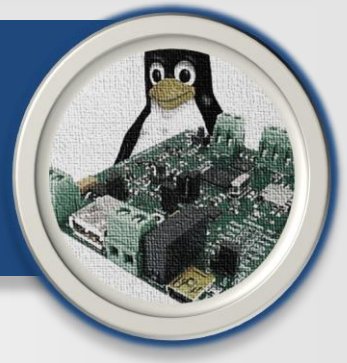
- The C library and compiled programs needs to interact with the kernel
  - Available system calls and their numbers
  - Constant definitions
  - Data structures, etc.
- Therefore, compiling the C library requires kernel headers, and many applications also require them.

GCC

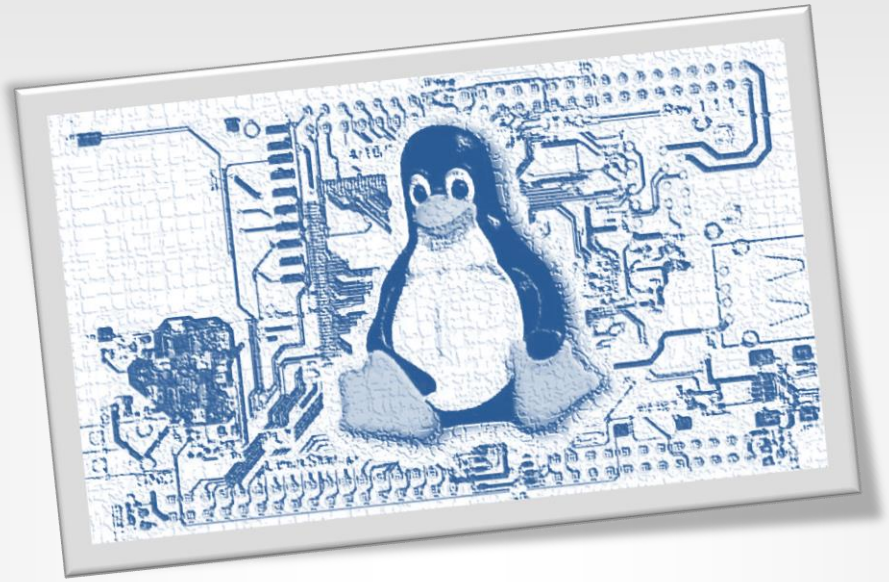


- GNU Compiler Collection, the famous free software compiler
- Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, and generate code for a large number of CPU architectures, including ARM, AVR, Blackfin, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, i386, x86 64, IA64, Xtensa, etc.
- http://gcc.gnu.org/

C Library

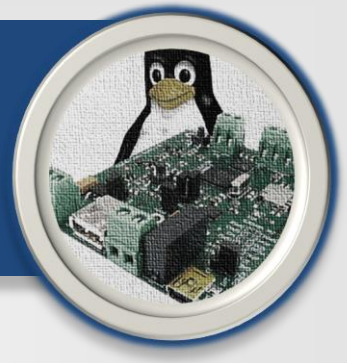


- The C library is
  - Interface between the applications and the kernel
  - Provides the well-known standard C API to ease application development
- Several C libraries are available:
  - glibc, uClibc, eglibc, dietlibc, newlib, etc.
- The choice of the C library must be made at the time of the cross-compiling toolchain generation, as the GCC compiler is compiled against a specific c C library.
- We will use uClibc: http://www.uclibc.org/



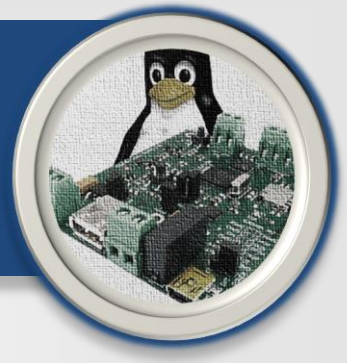
crosstool-ng

crosstool-ng



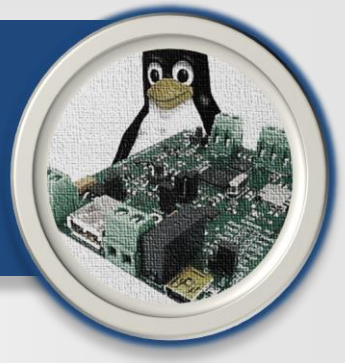
- Don’t forget the required pkgs
  - **`sudo apt-get install ncurses-dev bison texinfo flex autoconf automake libtool libexpat1-dev libncurses5-dev patch curl cvs build-essential subversion gawk python-dev gperf`**
- http://crosstool-ng.org
- Untar the source code, then build it
  - **`./configure --enable-local`**
  - **`make`**

crosstool-ng



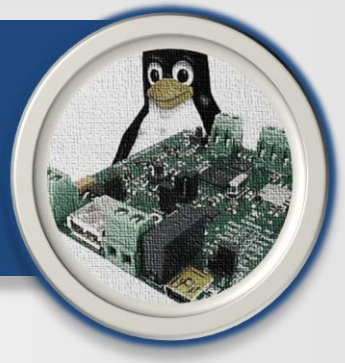
- You can use help
  - `./ct-ng help`
  - `./ct-ng list-samples`
- We will use
  - `./ct-ng arm-unknown-linux-uclibcgnueabi`

crosstool-ng



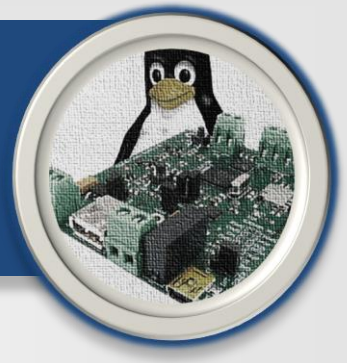
- You can configure it again
  - `./ct-ng menuconfig`
- We will use
  - `./ct-ng arm-unknown-linux-uclibcgnueabi`
- In Path and misc options:
  - Prefix directory
  - Change **Maximum log level** to see to **DEBUG**
- In Toolchain options:
  - Set **Tuple's alias** to **arm-linux**

crosstool-ng



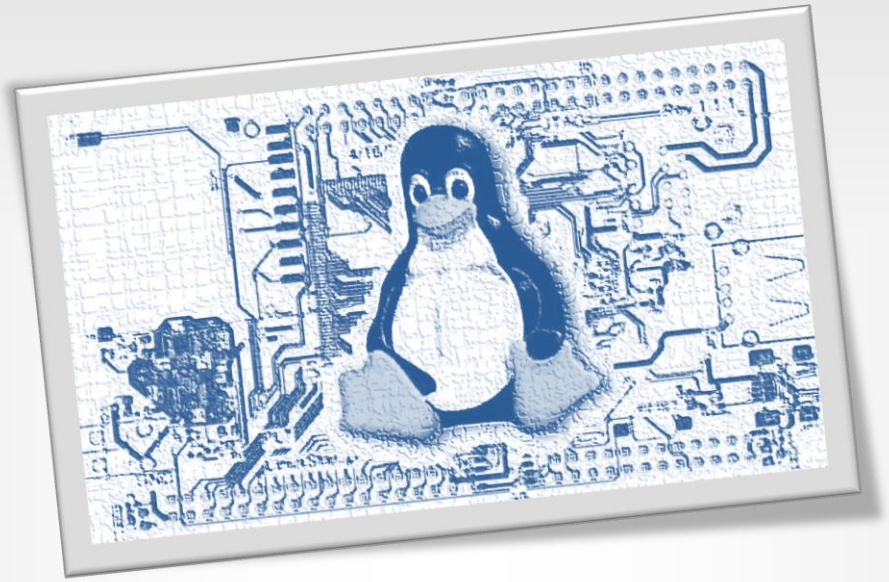
- In Debug facilities:
- Enable **gdb**, **strace** and **ltrace**.
- Remove the other options (**dmalloc** and **duma**).
- In gdb options:
  - Make sure that the **Cross-gdb** and **Build a static gdbserver** options are enabled; the other options are not needed.
  - Set **gdb version** to **7.4.1**

crosstool-ng



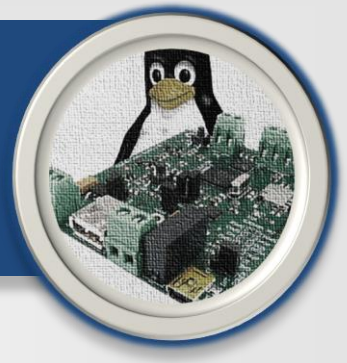
- Build
  - Create in you home directory
    - **src** >> save the tarballs it will download
    - **x-tools** >> the outbut
  - **`./ct-ng build`**

And wait



Using Cross-ToolChain

Using Cross-ToolChain



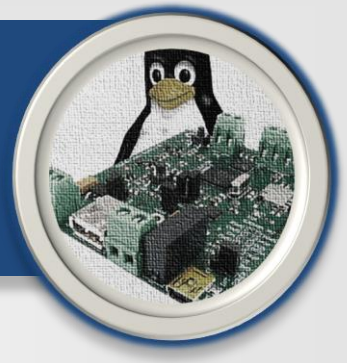
- Now you have your own built toolchain from ARM arch. Inside x-tools directory.
- Add the bin directory to your PATH environment variable to be able to use it from anywhere

```
export PATH=$PATH:PATH_TO_BIN_INSIDE_TOOLCHAIN
```

- You can add this line to bash startup script to be global every time you log in.

```
sudo gedit /etc/bash.bashrc
```

Compile a simple program



- Compile any C program to check the toolchain

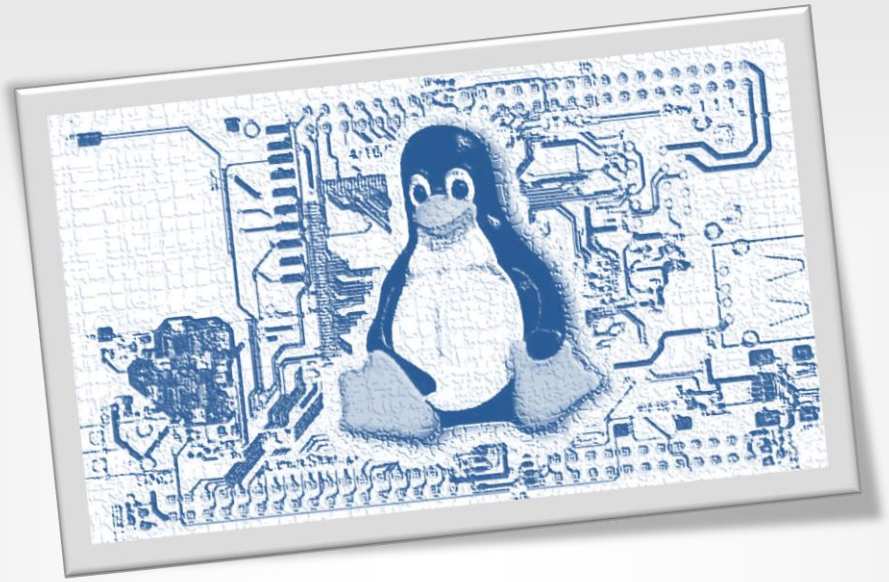
```
arm-linux-gcc hello.c -o hello
```

- Or by using static compiling

```
arm-linux-gcc -static hello.c -o hellostatic
```

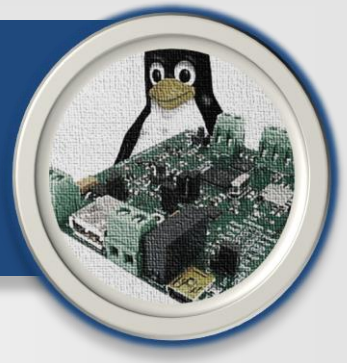
- You can view the file type

```
file hello
```



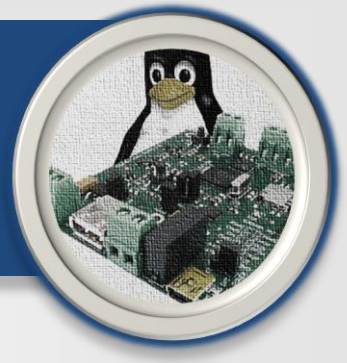
Kernel Compilation

Kernel Compilation



- Where to get Linux Kernel source
  - https://www.kernel.org/
- You can download any Linux version according to your needs.

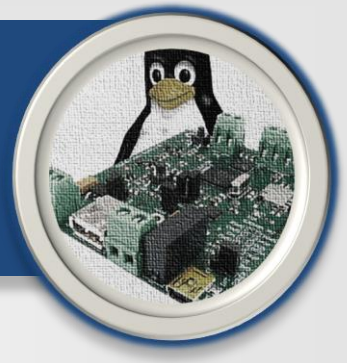
Kernel patching



- Changes to the kernel sources are distributed as patch files. The patch utility is used to apply a series of edits to a set of source files. So, for example, if you have the 2.0.29 kernel source tree and you wanted to move to the 2.0.30 source tree, you would obtain the 2.0.30 patch file and apply the patches (edits) to that source tree:

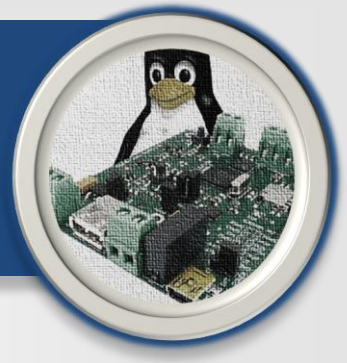
```
patch -p1 < patch-2.0.30
```

Linux source dir



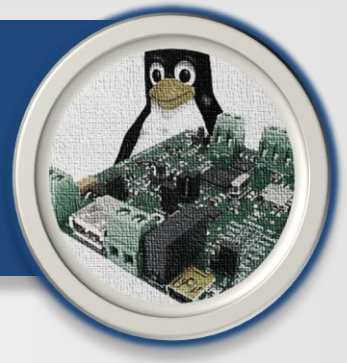
- arch
  - The arch subdirectory contains all of the architecture specific kernel code. It has further subdirectories, one per supported architecture, for example i386 and alpha.
- include
  - The include subdirectory contains most of the include files needed to build the kernel code. It too has further subdirectories including one for every architecture supported. The include/asm subdirectory is a soft link to the real include directory needed for this architecture, for example include/asm-i386. To change architectures you need to edit the kernel makefile and rerun the Linux kernel configuration program.
- init
  - This directory contains the initialization code for the kernel and it is a very good place to start looking at how the kernel works.

Linux source dir



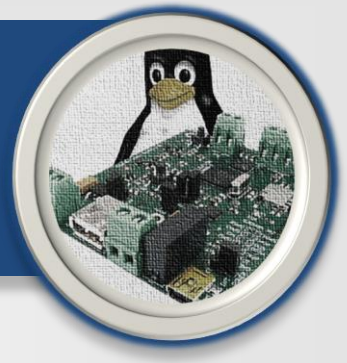
- mm
  - This directory contains all of the memory management code. The architecture specific memory management code lives down in arch/*/mm/, for example arch/i386/mm/fault.c.
- drivers
  - All of the system's device drivers live in this directory. They are further sub-divided into classes of device driver, for example block.
- ipc
  - This directory contains the kernels inter-process communications code.
- modules
  - This is simply a directory used to hold built modules.
- fs
  - All of the file system code. This is further sub-divided into directories, one per supported file system, for example vfat and ext2.

Linux source dir



- kernel
  - The main kernel code. Again, the architecture specific kernel code is in arch/*/kernel.
- net
  - The kernel's networking code.
- lib
  - This directory contains the kernel's library code. The architecture specific library code can be found in arch/*/lib/.
- scripts
  - This directory contains the scripts (for example awk and tk scripts) that are used when the kernel is configured.

Kernel Compilation



- Configure the Linux kernel

```
make vexpress_defconfig ARCH=arm CROSS_COMPILE=arm-linux-
```

- Edit the default configuration

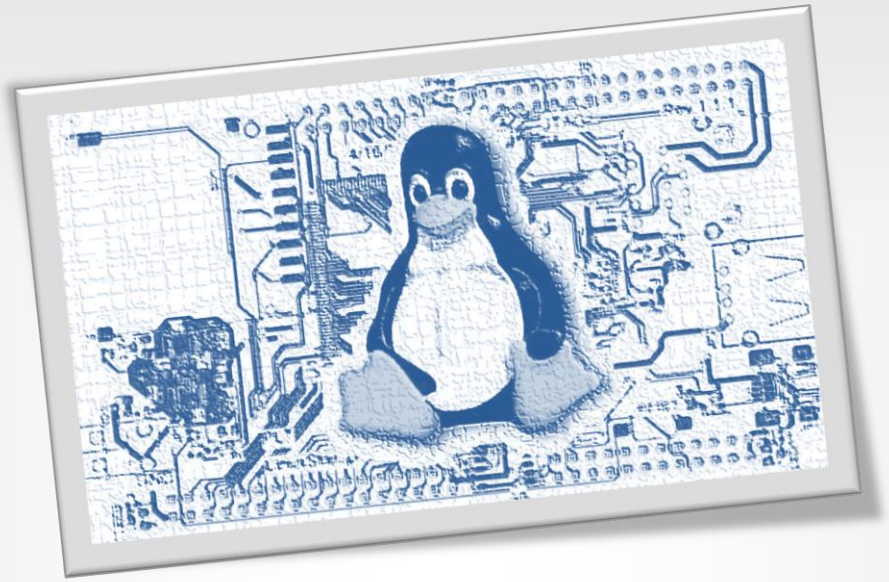
```
make menuconfig
```

- Build the kernel image

```
make all ARCH=arm CROSS_COMPILE=arm-linux-
```

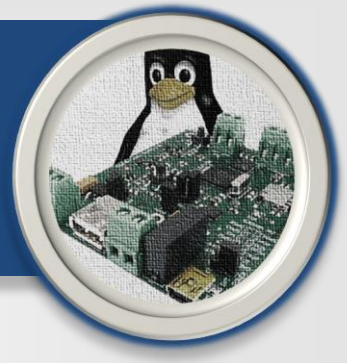
- You will find the kernel image in

```
arch/arm/boot/zImage
```



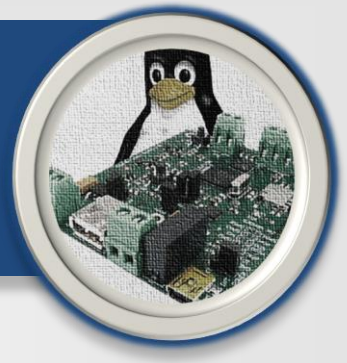
HW Emulators

HW Emulators



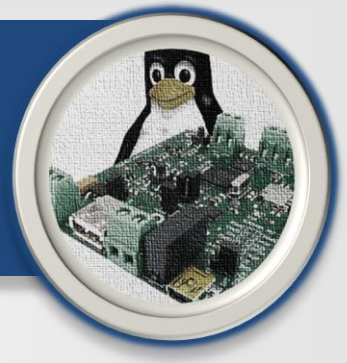
- Hardware emulation is the process of imitating the behavior of one or more pieces of hardware (typically a system under design) with another piece of hardware, typically a special purpose emulation system.

Qemu



- QEMU is a generic and open source machine emulator and virtualizer.
- QEMU lets you run another operating system on top of your existing OS.
- When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.

Qemu



- Install Qemu on Ubuntu

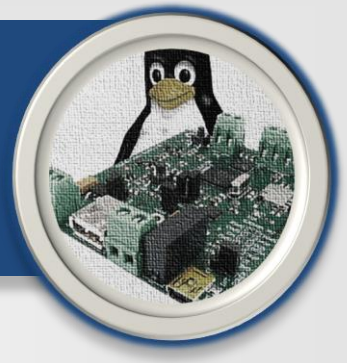
```
sudo apt-get install qemu-system
```

- Test Qemu running

```
qemu-system-i386 -m 256M -kernel       \
/boot/vmlinuz-3.2.0-57-generic-pae     \
-initrd /boot/initrd.img-3.2.0-57-generic-pae
```

- This command will run your current installed Linux kernel inside Qemu emulator

Qemu ARM



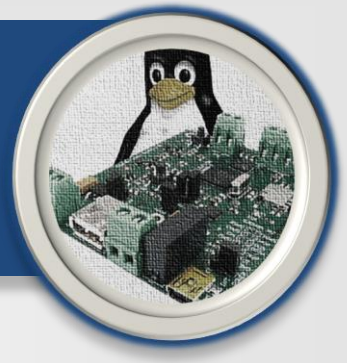
- To run the ARM kernel inside Qemu
- We need the zImage Kernel & initrd image filesystem with init process to run
- Compile a simple C program

```
arm-linux-gcc -static hello.c -o init
```

- Generate the initramfs image

```
echo init | cpio -o --format=newc > initramfs
```

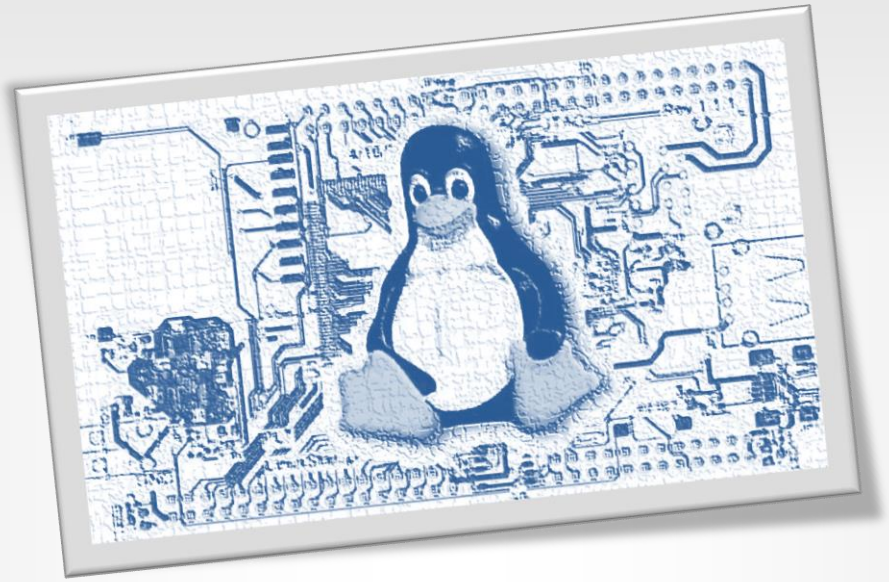
Qemu ARM



- Put the zImage & initramfs together and execute the command

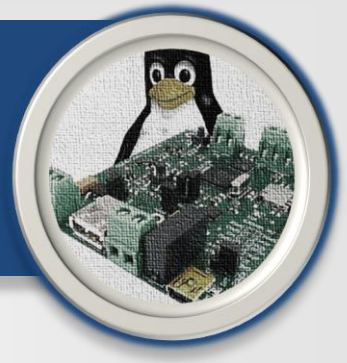
```
qemu-system-arm -M vexpress-a9 \
-kernel zImage -initrd initramfs \
-append "console=tty1“
```

- The Linux system should start and execute the single process init.



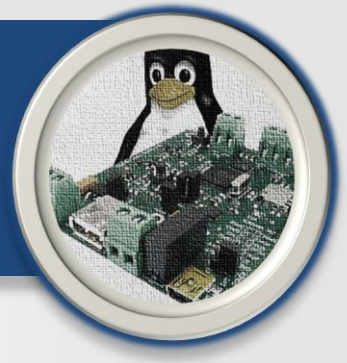
Building full ARM system with Busybox

Linux Kernel



- The same zImage kernel already built.

Busybox



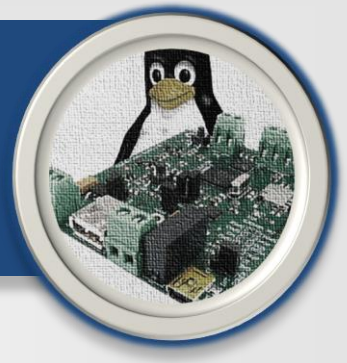
- Inside Busybox source dir, configure & cross-compile for ARM (static)

```
make ARCH=arm CROSS_COMPILE=arm-linux- defconfig
make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
    Busybox Settings -->
        Build Options -->
            Build Busybox as a static binary (CHECK)
```

- Command to start building

```
make CONFIG_PREFIX=/home/user-name/ARM_SYSTEM_DIR
ARCH=arm CROSS_COMPILE=arm-linux- install
```

Complete the filesystem dirs



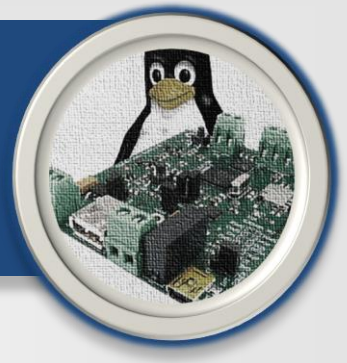
- Inside the ARM_SYSTEM_DIR

```
mkdir proc sys dev etc etc/init.d
gedit etc/init.d/rcS
   #!/bin/sh
   mount -t proc none /proc
   mount -t sysfs none /sys
   /sbin/mdev -s
```

- Change the rcS file permissions

```
chmod +x etc/init.d/rcS
```

Create the root fs image file

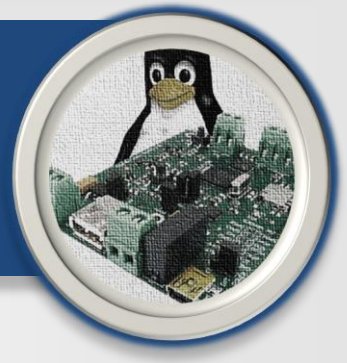


- Inside the ARM_SYSTEM_DIR

```
find . | cpio -o --format=newc > ../rootfs.img
cd ..
gzip -c rootfs.img > rootfs.img.gz
```

- Now you have the kernel image & the root fs image compressed

Run Qemu emulator



```
qemu-system-arm -M vexpress-a9 \
-kernel zImage -initrd rootfs.img.gz \
-append "root=/dev/ram rdinit=/bin/sh“
```

- Now you got a console running, and can try any command or app installed.

eng.sherif.mosa@gmail.com

http://about.me/shatrix