Embedded Systems Programming
OS Linux - Toolchain

Iwona Kochańska

Gdansk University of Technology
Embedded software

- **Toolchain**: compiler and tools for hardware-dependent software development
- **Bootloader**: initializes hardware and loads OS kernel
- **Kernel**: a heart of a system, manages resources and communicates with hardware
- **Root filesystem**: libraries and applications
What is toolchain?

**Toolchain** - set of tools that compiles source code into executables that can run on the target device, and includes:

- a compiler,
- a linker,
- run-time libraries

Toolchain is needed to build the other three elements of an embedded Linux system:

- the bootloader,
- the kernel,
- the root filesystem.
What is Toolchain?

- Toolchain has to be able to compile code written in assembly, C, and C++ since these are the languages used in the base open source packages.
- Usually, toolchains for Linux are based on components from the GNU project (http://www.gnu.org).
Toolchain components

Standard GNU toolchain consists of three main components:

- **Binutils**: binary utilities including assembler, linker, ld. It is available at http://www.gnu.org/software/binutils/.

- **GNU Compiler Collection (GCC)**: the compilers for C and other languages, include C++, Objective-C, Objective-C++, Java, Fortran, Ada, and Go. It is available at http://gcc.gnu.org/.

- **C library**: a standardized API based on the POSIX specification which is the principle interface to the operating system kernel from applications.

... and a copy of the Linux kernel headers.
Cross-compilation

Basic definitions:

- **build**: Also called the local platform, the platform that you physically perform the compiling on (for example PC with Debian distribution)
- **target**: The destination platform (for example Raspberry Pi)
- **host**: OS providing virtual environment
- **guest**: OS in virtual environment
There are two main problems in cross-compiling:

- All C/C++ include files and libraries for the target must be available on the build platform.
- The cross-compiler and related tools must generate code suitable for the target platform.
Types of toolchains

There are several types of toolchains, based on the architecture of build, host and target machines:

▶ Native (common in desktop computers)

<table>
<thead>
<tr>
<th>build machine</th>
<th>host machine</th>
<th>target machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch. A</td>
<td>arch. A</td>
<td>arch. A</td>
</tr>
</tbody>
</table>

▶ Cross-compilation

<table>
<thead>
<tr>
<th>build machine</th>
<th>host machine</th>
<th>target machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch. A</td>
<td>arch. A</td>
<td>arch. B</td>
</tr>
</tbody>
</table>
Types of toolchains

- **Cross-native**
  - **build** machine: arch. A
  - **host** machine: arch. B
  - **target** machine: arch. B

- **Canadian**
  - **build** machine: arch. A
  - **host** machine: arch. B
  - **target** machine: arch. C
The toolchain has to be built according to the capabilities of the target CPU, which includes:

- **CPU architecture**: arm, mips, x86_64, and so on
- **Big- or little-endian operation**: Some CPUs can operate in both modes, but the machine code is different for each
- **Floating point support**: Not all versions of embedded processors implement a hardware floating point unit, in which case, the toolchain can be configured to call a software floating point library instead
- **Application Binary Interface (ABI)**: The calling convention used for passing parameters between function calls
ARM ABI architectures

- **OABI** - Old Application Binary Interface (OABI) - discontinued 2011
- **EABI** - Extended Application Binary Interface (EABI) - general purpose (integer) registers
- **EABIHF** - Hard Float Extended Application Binary Interface - uses floating point registers.
GNU prefix

GNU uses a prefix consisting of a tuple of three or four components separated by dashes:

- **CPU**: arm, mips, x86_64, el = little-endian, eb = big-endian;
  e.g. armeb = big-endian ARM
- **Vendor**, e.g. poky
- **Kernel**, e.g. linux
- **Operating system**: a name for the user space component, which might be gnu or uclibcgnu. The ABI may be appended (gnueabi, gnueabihf, uclibcgnueabi, or uclibcgnueabihf).

$$\text{gcc } \text{--dumpmachine} \text{ x86_64--linux--gnu}$$
C library

▶ The programming interface to the Unix operating system is defined in the C language (POSIX standards).

▶ The C library is the implementation of that interface (gateway to the kernel for Linux programs).

▶ Whenever the C library needs the services of the kernel it will use the kernel **system call** interface to transition between **user space** and **kernel space**.
C libraries

- **glibc** - standard GNU C library, most complete implementation of the POSIX API

- **eglibc** - embedded GLIBC; a series of patches to glibc which added configuration options and support for architectures not covered by glibc; merged back into glibc as of version 2.20

- **uClibc** - micro controller C library; first developed to work with uClinux (Linux for CPUs without memory management units), but has since been adapted to be used with full Linux

- **musl libc** - new C library designed for embedded systems.
Building a toolchain using crosstool-NG

Crosstool-NG (2007) - a way to create a stand-alone cross toolchain from source SoC or board vendor

Install:

$ sudo apt-get install automake bison chrpath flex g++ gperf gawk libexpat1-dev libncurses5-dev libstdc++1-dev libtool python2.7-dev texinfo

Download sources from http://crosstool-ng.org/download/crosstool-ng and:

$ tar xf crosstool-ng-1.9.3.tar.bz2
$ cd crosstool-ng-1.9.3
$ ./configure
$ make
$ make install

–enable-local option - program will be installed into the current directory, instead of /usr/local/bin.
Building a toolchain using crosstool-NG

Launch the crosstool menu:

$ ./ct-ng

List of sample configurations

$ ./ct-ng list-samples

Select target configuration:

$ ./ct-ng arm-cortex_a8-linux-gnueabi

Make changes using the configuration menu:

$ ./ct-ng menuconfig

The configuration data is saved into a file named .config.

Build the toolchain:

$ ./ct-ng build
Building a toolchain using crosstool-NG

Toolchain directory:
arm−cortex_a8−linux−gnueabi / bin /
Add directory to the cross compiler to PATH:
$ PATH = / ... / arm−cortex_a8−linux−gnueabihf / bin : $PATH
Compile helloworld.c:
$ arm−cortex_a8−linux−gnueabihf−gcc helloworld.c −o helloworld
Check the type of the executable:
$ file helloworld
helloworld: ELF 32−bit LSB executable, ARM, version 1 (SYSV)
Building a toolchain using crosstool-NG

Check the cross-compiler version:

$ arm–cortex_a8–linux–gnuabi–gcc --version

Check the cross-compiler configuration:

$ arm–cortex_a8–linux–gnuabi–gcc –v

Print out the the range of architecture-specific options available:

$ arm–cortex_a8–linux–gnuabihf–gcc --target–help
Building a toolchain using crosstool-NG

The **toolchain sysroot** - a directory which contains subdirectories for libraries, header files, and other configuration files.

- It can be set when the toolchain is configured through
  
  ```
  –with-sysroot= 
  ```
  
  or it can be set on the command line, using –sysroot=.

- Check the location of the default sysroot:

  ```
  $ arm–cortex_a8–linux–gnueabi–gcc –print–sysroot 
  ```
Building a toolchain using crosstool-NG

Sysroot contains:

- **lib**: shared objects for the C library and the dynamic linker/loader,
- **ld-linux usr/lib**: the static library archives for the C library and any other libraries that may be installed subsequently
- **usr/include**: headers for all the libraries
- **usr/bin**: utility programs that run on the target, such as the ldd command
- **usr/share**: used for localization and internationalization
- **sbin**: Provides the ldconfig utility, used to optimize library loading paths

Some of these are needed on the development host to compile programs, others are needed on the target at runtime.
# Tools of a toolchain

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr2line</td>
<td>Converts program addresses into filenames and numbers</td>
</tr>
<tr>
<td>ar</td>
<td>Archive utility used to create static libraries.</td>
</tr>
<tr>
<td>as</td>
<td>GNU assembler</td>
</tr>
<tr>
<td>c++filt</td>
<td>demangle C++ and Java symbols</td>
</tr>
<tr>
<td>cpp</td>
<td>C preprocessor</td>
</tr>
<tr>
<td>elfedit</td>
<td>update the ELF header of ELF files</td>
</tr>
<tr>
<td>g++</td>
<td>GNU C++ front-end</td>
</tr>
<tr>
<td>gcc</td>
<td>GNU C front-end</td>
</tr>
<tr>
<td>gcov</td>
<td>code coverage tool</td>
</tr>
<tr>
<td>gdb</td>
<td>GNU debugger</td>
</tr>
<tr>
<td>gprof</td>
<td>program profiling tool</td>
</tr>
</tbody>
</table>
# Tools of a toolchain

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ld</code></td>
<td>GNU linker</td>
</tr>
<tr>
<td><code>nm</code></td>
<td>list of symbols from object files</td>
</tr>
<tr>
<td><code>objcopy</code></td>
<td>used to copy and translate object files</td>
</tr>
<tr>
<td><code>objfump</code></td>
<td>display information from object files</td>
</tr>
<tr>
<td><code>ranlib</code></td>
<td>creates or modifies an index in a static library, making the linking stage faster</td>
</tr>
<tr>
<td><code>readelf</code></td>
<td>displays information about files in ELF object format</td>
</tr>
<tr>
<td><code>size</code></td>
<td>lists section sizes and the total size</td>
</tr>
<tr>
<td><code>strings</code></td>
<td>display strings of printable characters in files</td>
</tr>
<tr>
<td><code>strip</code></td>
<td>strip an object file of debug symbol tables</td>
</tr>
</tbody>
</table>
C library

Four main parts implement POSIX standard

- **libc**: main C library that contains the well-known POSIX functions (printf, open, close, read, write)
- **libm**: maths functions (cos, exp, log)
- **libpthread**: all the POSIX thread functions with names beginning with pthread_
- **librt**: The real-time extensions to POSIX, including shared memory and asynchronous I/O

**libc** is always linked in but the others have to be explicitly linked with the -l option

```
arm–cortex_a8–linux–g nueabihf–gcc myprog.c –o myprog –L/lib –l
```
C library

Which libraries have been linked?

```
$ arm-cortex_a8-linux-gnueabihf-readelf -a myprog | grep " Shared library: "
0x00000001 (NEEDED) Shared library: [libm.so.6]
0x00000001 (NEEDED) Shared library: [libc.so.6]
```

Check the run-time linker:

```
$ arm-cortex_a8-linux-gnueabihf-readelf -a myprog | grep " Requesting program interpreter: "
[ Requesting program interpreter: /lib/ld-linux-armhf.so.3 ]
```
Static and dynamic linking

The library code can be linked in two different ways:

▶ **statically** - all the library functions the application calls and their dependencies are pulled from the library archive and bound into the executable;

▶ **dynamically** - references to the library files and functions in those files are generated in the code but the actual linking is done dynamically at runtime
Static libraries

Static linking is useful:

- when building a small system which consists of only BusyBox and some script files, it is simpler to link BusyBox statically and avoid having to copy the runtime library files and linker
- when running a program before the filesystem that holds the runtime libraries is available

Tell gcc to link all libraries statically:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -static helloworld.c
```
Creating a static library:

$ arm--cortex_a8--linux--gnueabihf--gcc --c test1.c
$ arm--cortex_a8--linux--gnueabihf--gcc --c test2.c
$ arm--cortex_a8--linux--gnueabihf--ar rc libtest.a test1.o

Linking libtest into helloworld program:

$ arm--cortex_a8--linux--gnueabihf--gcc helloworld.c --ltest
Shared libraries

- Shared objects linked at runtime,
- Efficient use of storage and system memory, since only one copy of the code needs to be loaded.
- Easy to update library files without having to re-link all the programs that use them.
Shared libraries

Creating a shared library:

$ arm–cortex_a8–linux–gnueabihf–gcc –fPIC –c test1.c
$ arm–cortex_a8–linux–gnueabihf–gcc –fPIC –c test2.c
$ arm–cortex_a8–linux–gnueabihf–gcc –shared –o libtest.so
test1.o  test2.o

**PIC** flag tells GCC to produce code that does not contain references
to specific memory locations for functions and variables

Linking libtest into helloworld program:

$ arm–cortex_a8–linux–gnueabihf–gcc helloworld.c –ltest

The linker will look for libtest.so in the default search path: /lib and
/usr/lib.

Other directories with shared libraries - place a colon-separated list of
paths in the shell variable LD_LIBRARY_PATH
Shared libraries names

**ldd** - searches the standard system library paths and shows which library versions would be used by a particular program.

```
ldd helloworld
```

Shared libraries names:

- **libjpeg.a**: library archive used for static linking
- **libjpeg.so -> libjpeg.so.8.0.2**: symbolic link, used for dynamic linking
- **libjpeg.so.8 -> libjpeg.so.8.0.2**: symbolic link used when loading the library at runtime
- **libjpeg.so.8.0.2**: actual shared library, used at both compile time and runtime
Build systems for cross compiling

- Simple makefiles where the toolchain is controlled by the make variable `CROSS_COMPILE`
- The GNU build system known as Autotools
- CMake (https://cmake.org)
Simple makefiles

- For packages simple to cross compile (the Linux kernel, the U-Boot bootloader, Busybox)
- put the toolchain prefix in the make variable CROSS_COMPILE, for example:
  
  $ make CROSS_COMPILE=arm\-cortex\_a8\-linux\-gnueabihf

- Or set it as a shell variable:
  
  $ export CROSS_COMPILE=arm\-cortex\_a8\-linux\-gnueabihf
  $ make
Autotools refers to a group of tools that are used as the build system in many open source projects. The components:

- GNU Autoconf
- GNU Automake
- GNU Libtool
- Gnulib

Packages that use Autotools come with a script named `configure` that checks dependencies and generates `makefiles`. To configure, build, and install a package for the native operating system:

```bash
$ ./configure
$ make
$ sudo make install
```
For Further Reading I