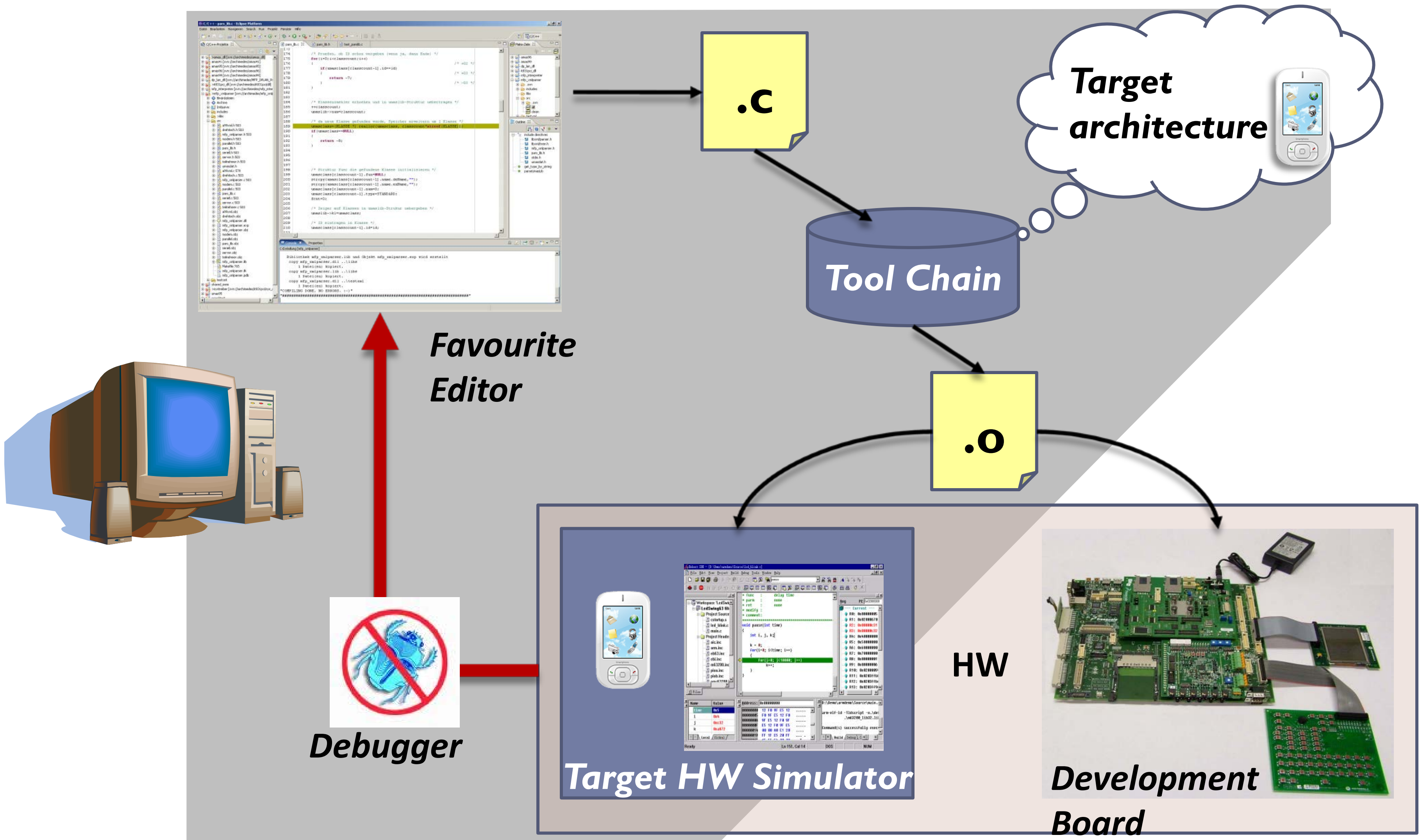
# Application Development for Embedded Systems

Andrea Bartolini
a.bartolini@unibo.it

# Application Cross-Development

- Cross development is the separation of the build environment from the target environment.
- Embedded computers where a device has extremely limited resources, are typically not powerful enough to run a compiler, a file system, or a development environment.
- Since debugging and testing may also require more resources than are available on an embedded system, cross-compilation can be less involved and less prone to errors than native compilation.

# Application Cross-Development



# Toolchain

- The first and most essential product to develop applications on any embedded device is a cross-toolchain

- A set of tools running on a host machine, used to
  1. Compile high-level source code into target object code
  2. Link pre-existing collections of object files (libraries)
  3. Assemble the whole thing into an executable object by the target machine

- Let us take a closer look..

# GNU Toolchain

- A collection of programming tools produced by the GNU Project. These tools form a toolchain (suite of tools used in a serial manner) used for developing applications and operating systems.

- It plays a vital role in development of Linux kernel, and software for embedded systems.

- Projects included in the GNU toolchain are:
  - GNU Compiler Collection (GCC): Suite of compilers for several programming languages;
  - GNU Binutils: Suite of tools including linker, assembler and other tools;
  - GNU Debugger (GDB): Code debugging tool;
  - GNU make: Automation tool for compilation and build;
  - GNU build system (autotools):

# Compiling code

source code

pre-processor

header files

compiler

assembler .s

assembler

object code

linker

executable file

**compiler**

**binutils**

**library**

# Compiling code

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0

main()
{
    int i;
    i = 5 * 2;
    printf("5 times 2 is %d.\n", i);
    printf("TRUE is %d.\n", TRUE);
    printf("FALSE is %d.\n", FALSE);
}
```

Handled by the pre-processor

Handled by the compiler

Implemented in C library

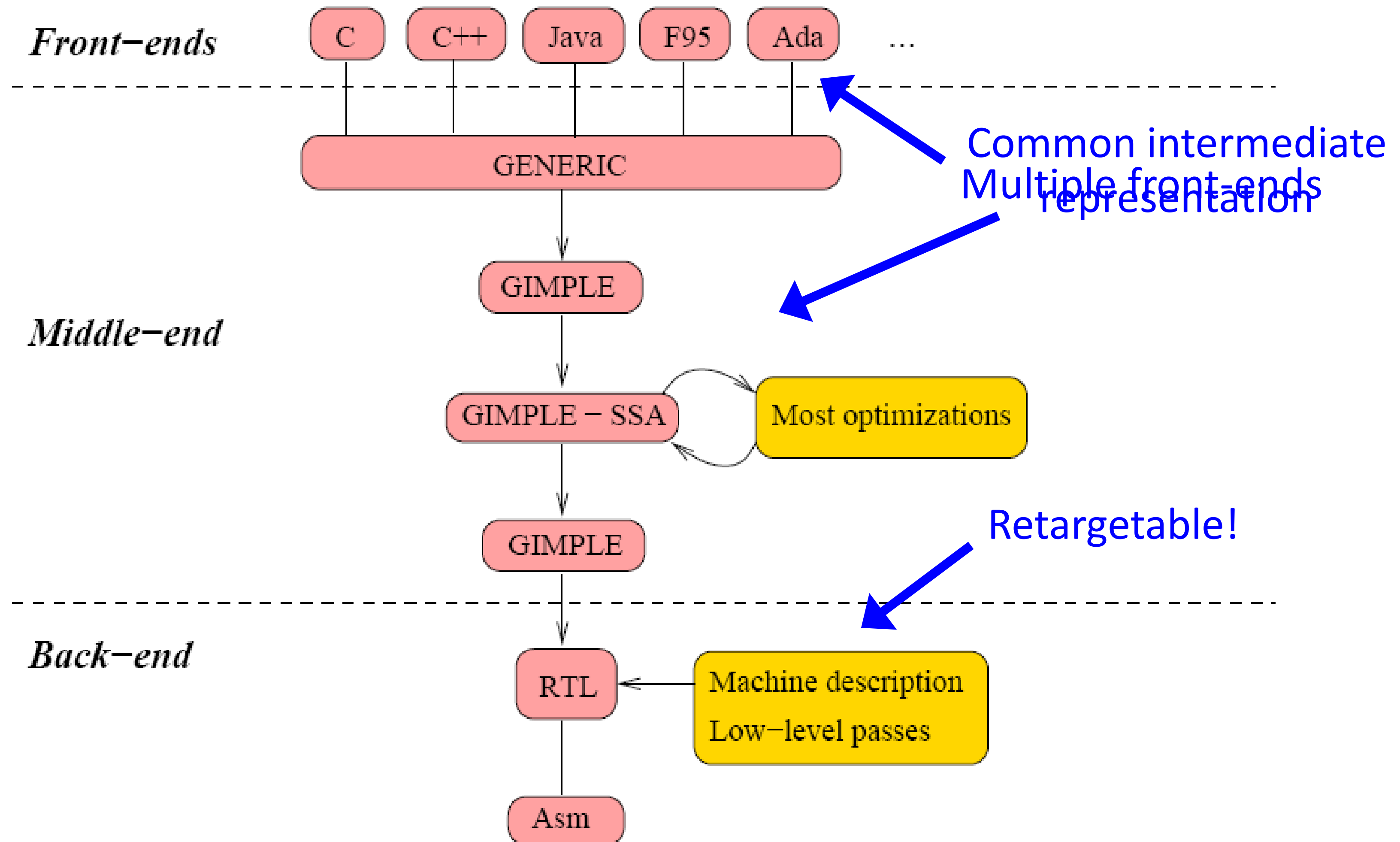
# Compiling code

- The pre-processor handles
  - Macros (#define)
  - Inclusions (#include)
  - Conditional code inclusion (#ifdef, #if)
  - Language extensions (#pragma).
- The compiler processes source code and turns it into assembler modules.
- The assembler converts them to  target binary code.
- The linker takes the object files and searches library files to find the routines it calls. It calculates the address references and incorporates any symbolic information to create an executable file format.

# GCC (GNU Compiler Collection)

- The **GNU Compiler Collection** (usually shortened to **GCC**) is a set of compilers produced for various programming languages by the GNU Project.
- It is a tool used by nearly every embedded engineer, even those who don't target Linux
- When starting an embedded project, the first tool needed is a cross-compiler, a compiler that generates code intended to work on a machine different from the one on which the code generation occurred.
- When used in an embedded project, GCC capably does cross-compilation, without complaint
- GCC is available for most embedded platforms, for example Symbian, Freescale Power Architecture-based chips, Playstation and Sega Dreamcast

# GCC Architecture



# GNU Binutils

- The **GNU Binary Utilities**, or **binutils**, is a collection of programming tools for the manipulation of object code in various object file formats.
- The most important tools are

  - **as** – ***Assembler***: Transforms the assembly code produced by the compiler into a binary format executable by the target machine
  - **ld** – ***Linker***: Takes as input several object files and generates a single executable, resolving interactions between symbols
  - **ar** – ***Archiver***: Combines together (possibly) multiple object files into a library. This could be later linked to other applications either statically or dynamically
  - **objdump** – ***Dumper***: It allows recreating an assembler file from an object file. It can also dump additional information about different segments of the output file format

# Object Archives (Libraries)

- In computer science, a library is a **collection of subroutines** or classes used to develop software
- Libraries **contain code and data that provide services** to independent programs. This allows the sharing and changing of code and data in a modular fashion
- Executables and libraries make references known as links to each other through the process known as **linking**, which is typically done by a linker
- Two types of libraries:
  1. **Static** libraries
  2. **Dynamic** / shared libraries

# Object Archives (Libraries)

- Here is an example program which makes a call to the external function sqrt in the math library 'libm.a'

```
#include <math.h>
#include <stdio.h>
int main (void) {
 double x = sqrt (2.0);
 printf ("The square root of 2.0 is %f\n", x);
 return 0;}
```

- To enable the compiler to link the sqrt function to the main program 'calc.c' we need to supply the library 'libm.a'

- gcc -L/usr/lib -lm -o calc calc.c

# Runtime Libraries

- Compilers only generate a small subset of high-level languages facilities and commands from built-in routines.

- It relies on libraries to provide the full range of functions that the language offers:
  - Processor dependent: mathematical functions, string manipulation and similar features that use the processor and don't need to communicate with peripherals;

  - I/O dependent: defines the hardware that the software need to access. The library routine either drives directly the hardware or calls the operating system to perform its task;

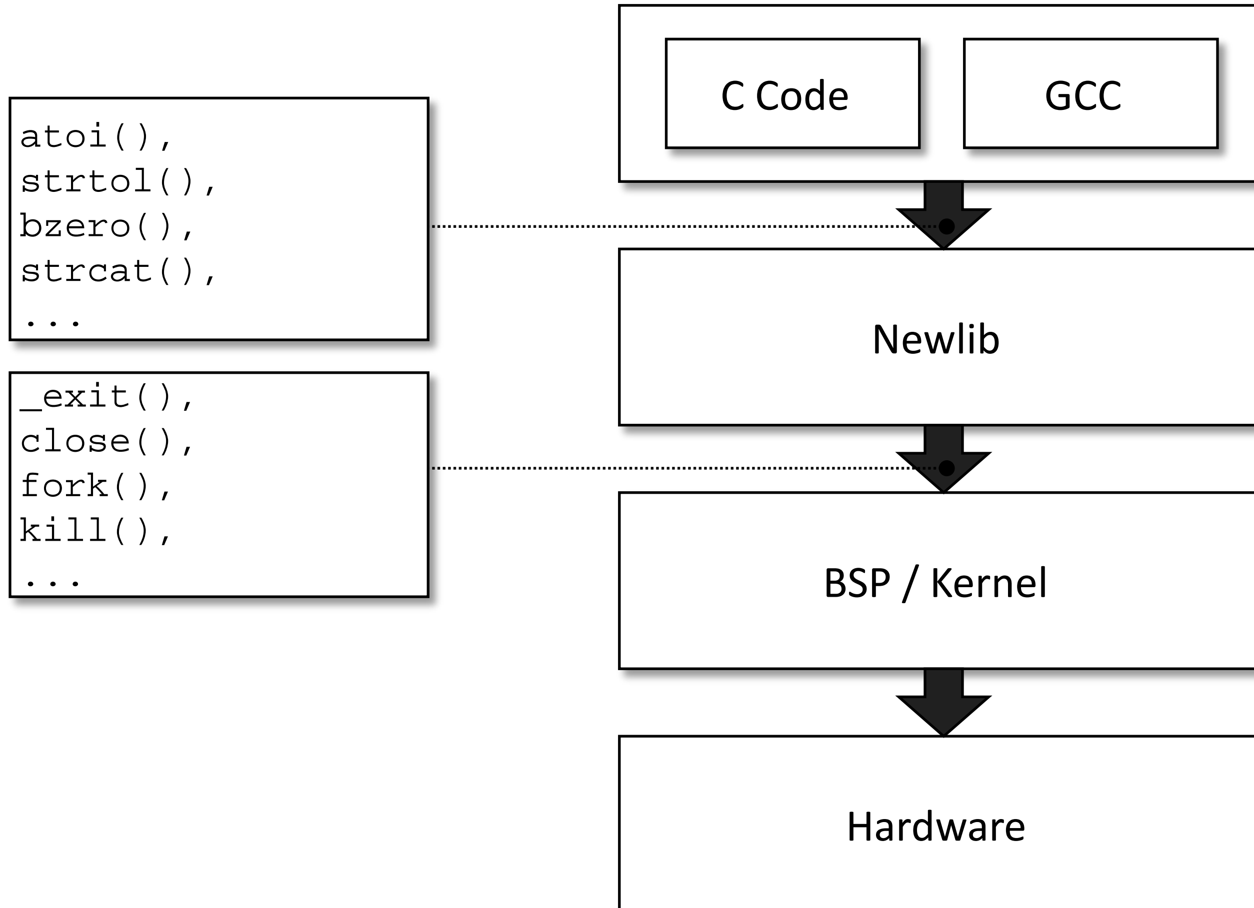
  - System calls: typical routines are those which dinamically allocate memory, task control commands, use semaphores, etc;

  - Exit routines: used to terminate programs free up the memory used by the application.

# Newlib

- **Newlib** is a C standard library implementation intended for use on embedded systems.
- It is a conglomeration of several library parts, all under free software licenses that make them easily usable on embedded products.
- The section System Calls of the newlib documentation describes how it can be used with many operating systems.
- Its primary use is on embedded systems that lack any kind of operating system; in that case it calls a "board support package" that can do things like write a byte of output on a serial port, or read a sector from a disk or other memory device.
- As of 2007, devkitARM, a popular toolchain for programming homebrew software for Nintendo DS and Game Boy Advance systems, includes Newlib as its C library

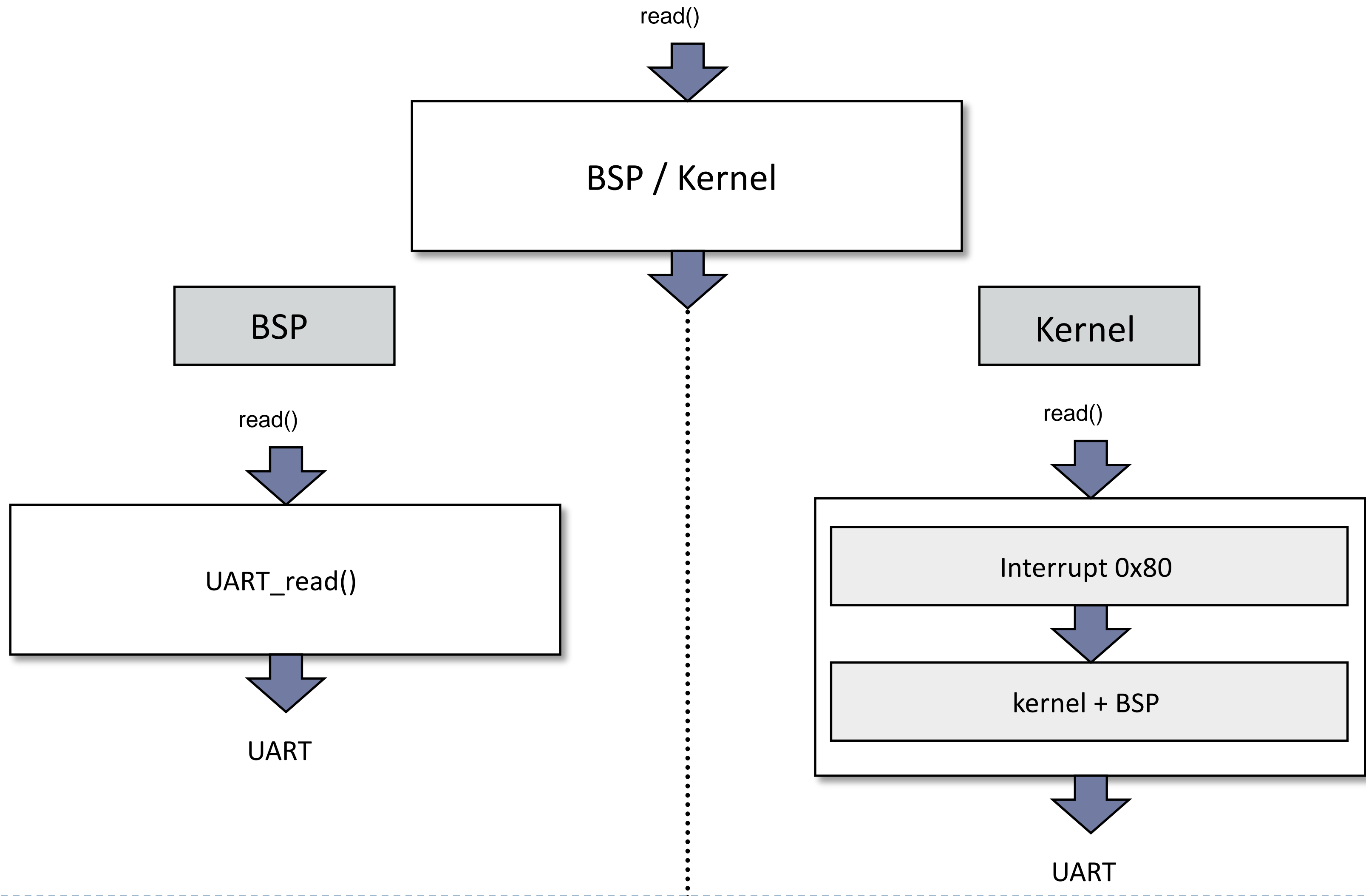
# Newlib





Carlo Caione <carlo.caione@unibo.it>

# Newlib





Carlo Caione <carlo.caione@unibo.it>

# Building a Toolchain
# An example: arm-elf-gcc

- We've introduced the three main software blocks composing a toolchain
  - **Binutils** – Target specific tools
  - **Gcc** – Compiler for target machine
  - **C library** – Target specific C library

- We now describe the entire process of building a complete toolchain for a ARM7 processor

# Building the toolchain for ARM

- The source code for binutils, gcc and newlib can be downloaded at
  - http://www.gnu.org/software/binutils/
  - http://gcc.gnu.org/mirrors.html
  - ftp://sources.redhat.com/pub/newlib/index.html

- The standard procedure to build and install each of the products consists of three steps
  - *Configuration*
  - *Build*
  - *Install*

# Building the toolchain for ARM

- For each of the product, create a separate source and build directory

```
$ROOT> mkdir binutils-src
$ROOT> mkdir gcc-src
$ROOT> mkdir newlib-src
$ROOT> mkdir binutils-build
$ROOT> mkdir gcc-build
$ROOT> mkdir newlib-build
```

- Then create an INSTALL folder for the entire toolchain

```
$ROOT> mkdir INSTALL
```

# Building the toolchain for ARM

- Build and install the binutils specifying the arm-elf target

```
$ROOT> cd binutils-build
$ROOT> ../binutils-src/configure --target=arm-elf          \
                                 --prefix=../INSTALL       \
                                 --program-prefix=arm-elf-
$ROOT> make all install
```

- At the end of the process, in the folder $ROOT/INSTALL/bin we'll find the cross-tools

```
$ROOT> ll INSTALL/bin
```

# Building the toolchain for ARM

- Add to your $PATH environment variable the path to the binutils just installed, so that the compiler can use them to build the cross-compiler

```
$ROOT> export PATH=$ROOT/INSTALL/bin:$PATH
```

- Build and install the cross-compiler specifying the arm-elf target and the location of the C library headers

```
$ROOT> cd gcc-build
$ROOT> ../gcc-src/configure --target=arm-elf              \
                              --prefix=../INSTALL         \
                              --program-prefix=arm-elf-   \
                              --enable-languages=c,c++    \
                              --with-newlib               \
                --with-headers=../newlib-src/newlib/libc/include
$ROOT> make all-gcc install-gcc
```

# Building the toolchain for ARM

- Build and install the C library

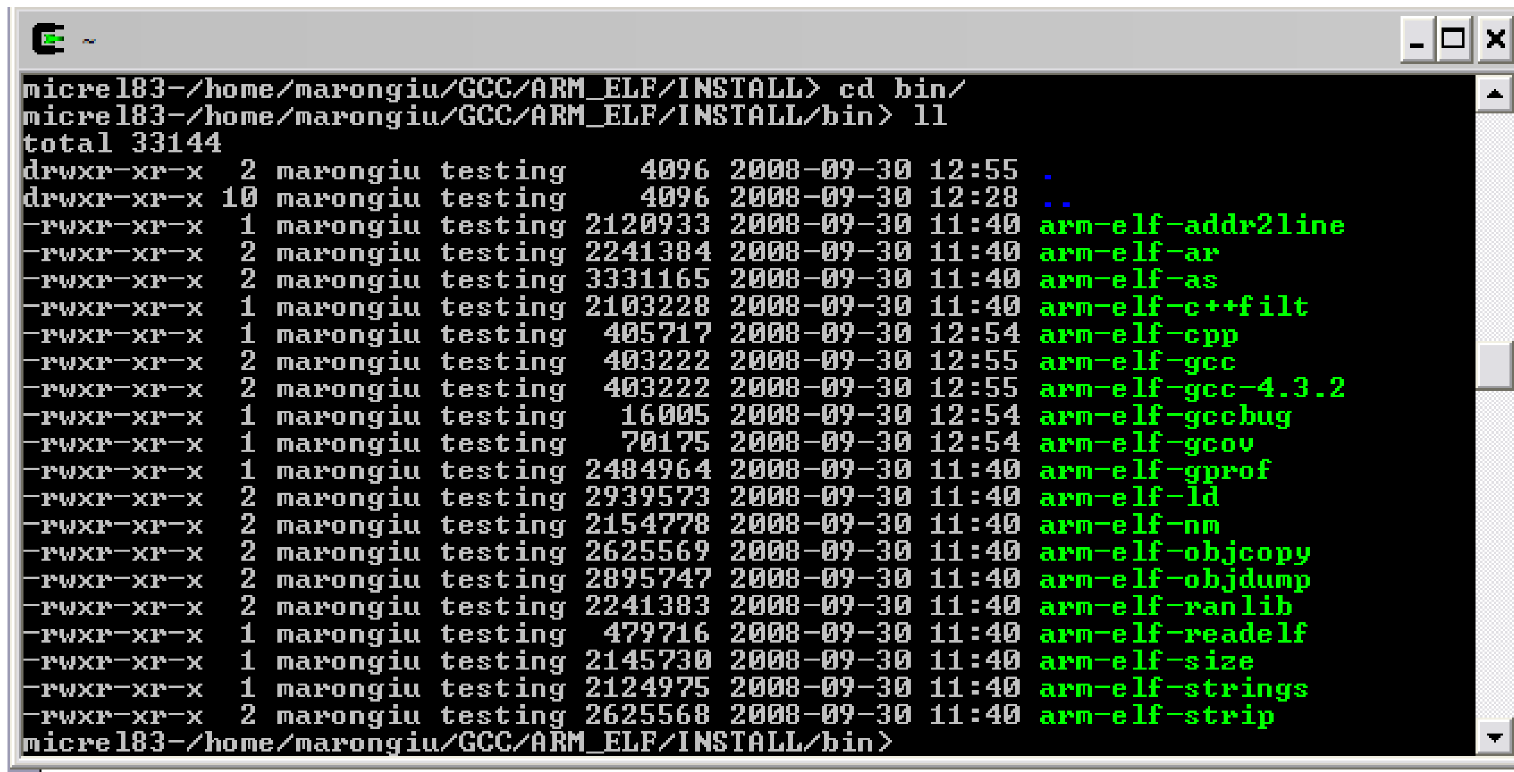
```
$ROOT> cd newlib-build
$ROOT> ../newlib-src/configure --target=arm-elf                    \
                              --prefix=../INSTALL
$ROOT> make all install
```

- Finally, install the entire compiler

```
$ROOT> cd gcc-build
$ROOT> make all install
```

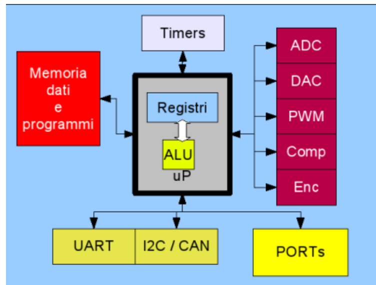
# Building the toolchain for ARM

- Take a look at the $ROOT/INSTALL/bin folder



```
micrel83-/home/marongiu/GCC/ARM_ELF/INSTALL> cd bin/
micrel83-/home/marongiu/GCC/ARM_ELF/INSTALL/bin> ll
total 33144
drwxr-xr-x  2 marongiu testing    4096 2008-09-30 12:55 .
drwxr-xr-x 10 marongiu testing    4096 2008-09-30 12:28 ..
-rwxr-xr-x  1 marongiu testing 2120933 2008-09-30 11:40 arm-elf-addr2line
-rwxr-xr-x  2 marongiu testing 2241384 2008-09-30 11:40 arm-elf-ar
-rwxr-xr-x  2 marongiu testing 3331165 2008-09-30 11:40 arm-elf-as
-rwxr-xr-x  1 marongiu testing 2103228 2008-09-30 11:40 arm-elf-c++filt
-rwxr-xr-x  1 marongiu testing  405717 2008-09-30 12:54 arm-elf-cpp
-rwxr-xr-x  2 marongiu testing  403222 2008-09-30 12:55 arm-elf-gcc
-rwxr-xr-x  2 marongiu testing  403222 2008-09-30 12:55 arm-elf-gcc-4.3.2
-rwxr-xr-x  1 marongiu testing   16005 2008-09-30 12:54 arm-elf-gccbug
-rwxr-xr-x  1 marongiu testing   70175 2008-09-30 12:54 arm-elf-gcov
-rwxr-xr-x  1 marongiu testing 2484964 2008-09-30 11:40 arm-elf-gprof
-rwxr-xr-x  2 marongiu testing 2939573 2008-09-30 11:40 arm-elf-ld
-rwxr-xr-x  2 marongiu testing 2154778 2008-09-30 11:40 arm-elf-nm
-rwxr-xr-x  2 marongiu testing 2625569 2008-09-30 11:40 arm-elf-objcopy
-rwxr-xr-x  2 marongiu testing 2895747 2008-09-30 11:40 arm-elf-objdump
-rwxr-xr-x  2 marongiu testing 2241383 2008-09-30 11:40 arm-elf-ranlib
-rwxr-xr-x  1 marongiu testing  479716 2008-09-30 11:40 arm-elf-readelf
-rwxr-xr-x  1 marongiu testing 2145730 2008-09-30 11:40 arm-elf-size
-rwxr-xr-x  1 marongiu testing 2124975 2008-09-30 11:40 arm-elf-strings
-rwxr-xr-x  2 marongiu testing 2625568 2008-09-30 11:40 arm-elf-strip
micrel83-/home/marongiu/GCC/ARM_ELF/INSTALL/bin>
```

# Embedded Application Programming



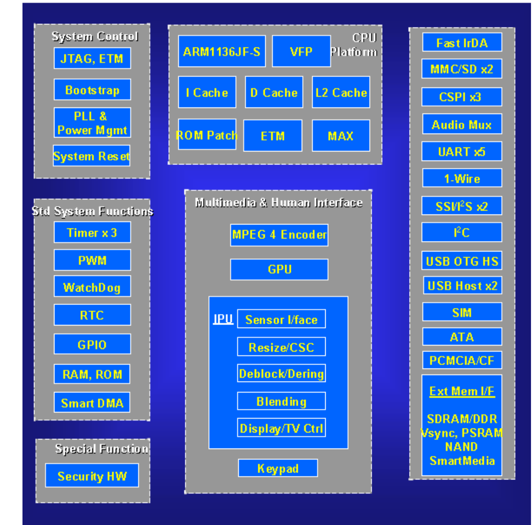

Microcontroller

SW & HW Low Complexity

- The application may use directly the HW
- The programmer know the HW and read and write directly internal register

Custom middleware

- HW programmed by statically linking API
- Standard / Custom API
- Microkernel

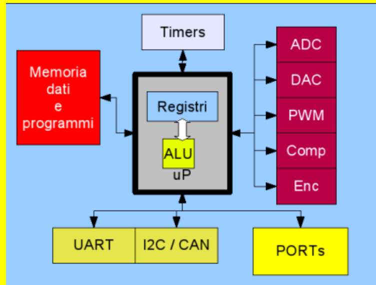



Multimedia SoC

Complex HW

- Programmer needs abstraction layers.
- Embedded O.S.
- Application use HW facilities throught system call & driver

# Embedded Application Programming




Microcontroller

SW & HW Low Complexity

- The application may use directly the HW
- The programmer know the HW and read and write directly internal register

Custom middleware

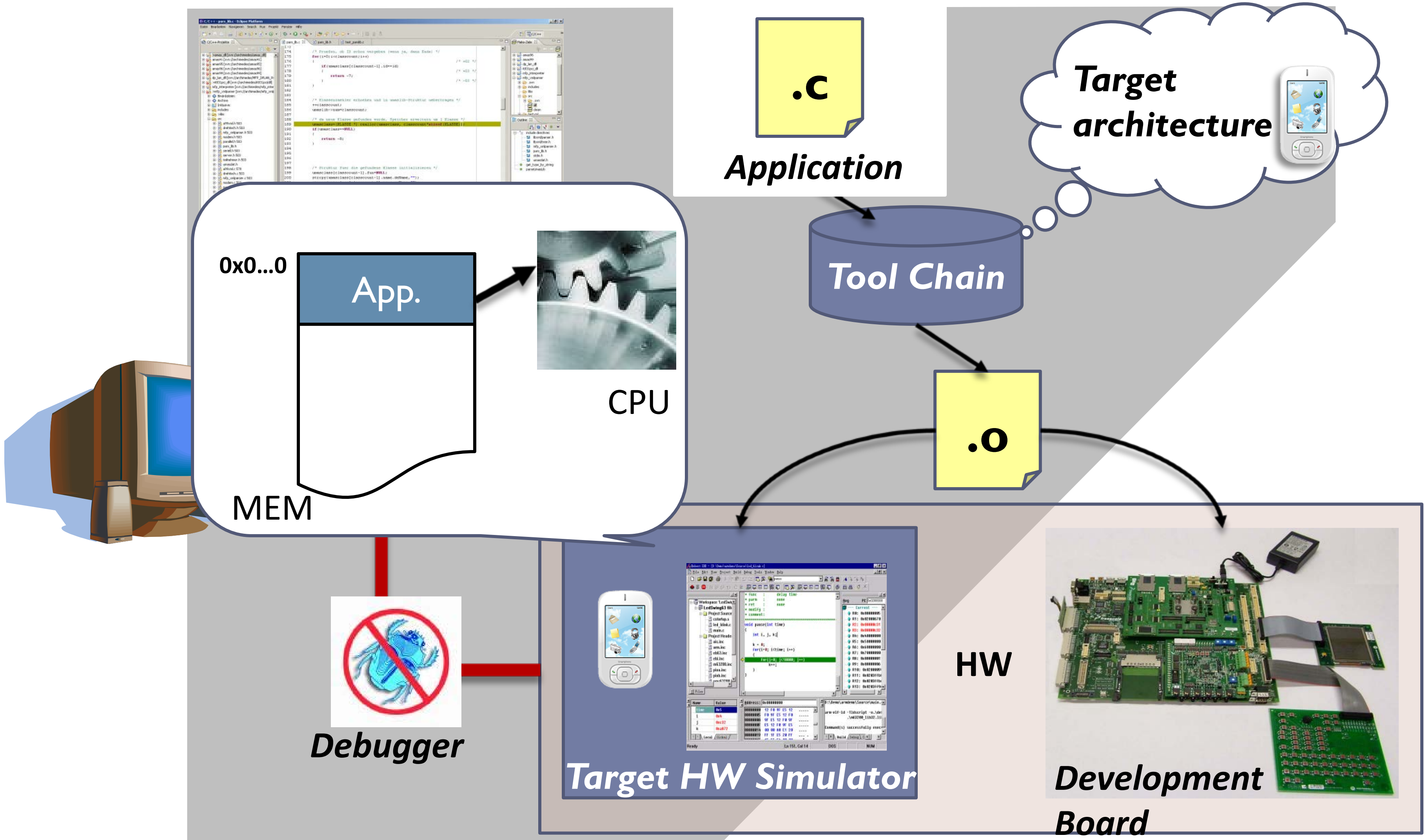
- HW programmed by statically linking API
- Standard / Custom API
- Microkernel

Multimedia SoC

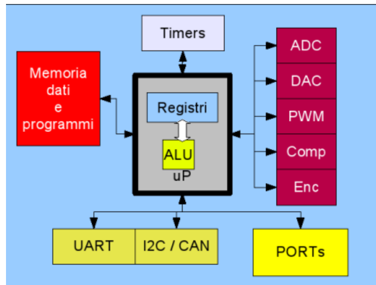
Complex HW

- Programmer needs abstraction layers.
- Embedded O.S.
- Application use HW facilities throught system call & driver

# Application Cross-Development



# Embedded Application Programming




Microcontroller

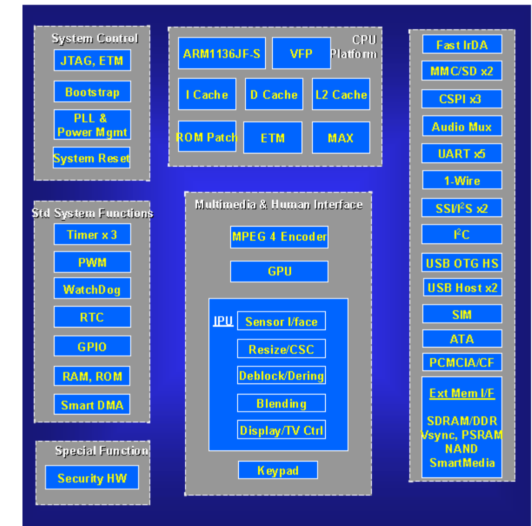
SW & HW Low Complexity

- The application may use directly the HW
- The programmer know the HW and read and write directly internal register

Complexity

Custom middleware

- HW programmed by statically linking API
- Standard / Custom API
- Microkernel




Multimedia SoC

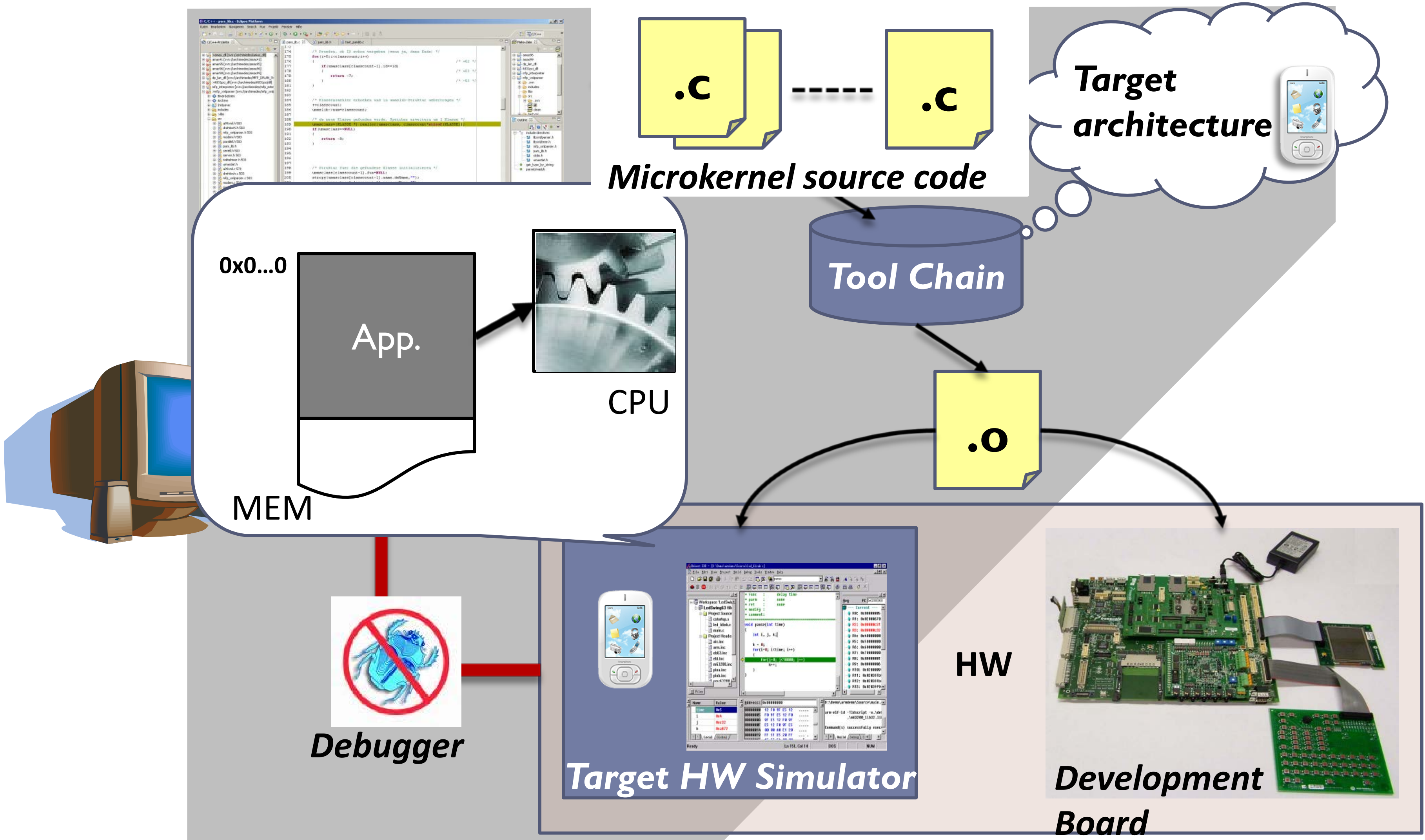
Complex HW

- Programmer needs abstraction layers.
- Embedded O.S.
- Application use HW facilities throught system call & driver

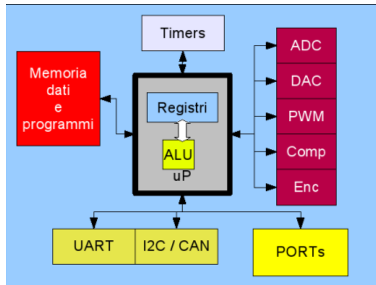
# O.S. Embedded

- Why we need an O.S. on embedded system:
  - Depends on complexity
    - Hardware complexity, programmer needs abstraction layers
    - Multi-process execution => scheduler
    - Memory Managment Unit
    - Store data => File system
- Differents O.S.:
  - Windows CE
  - Linux embedded : based on general purpose linux + BSP
  - ucLinux : Lite version of Linux embedded (No MMU support)
  - RTems : (No File System, No MMU )

# Application Cross-Development



# Embedded Application Programming



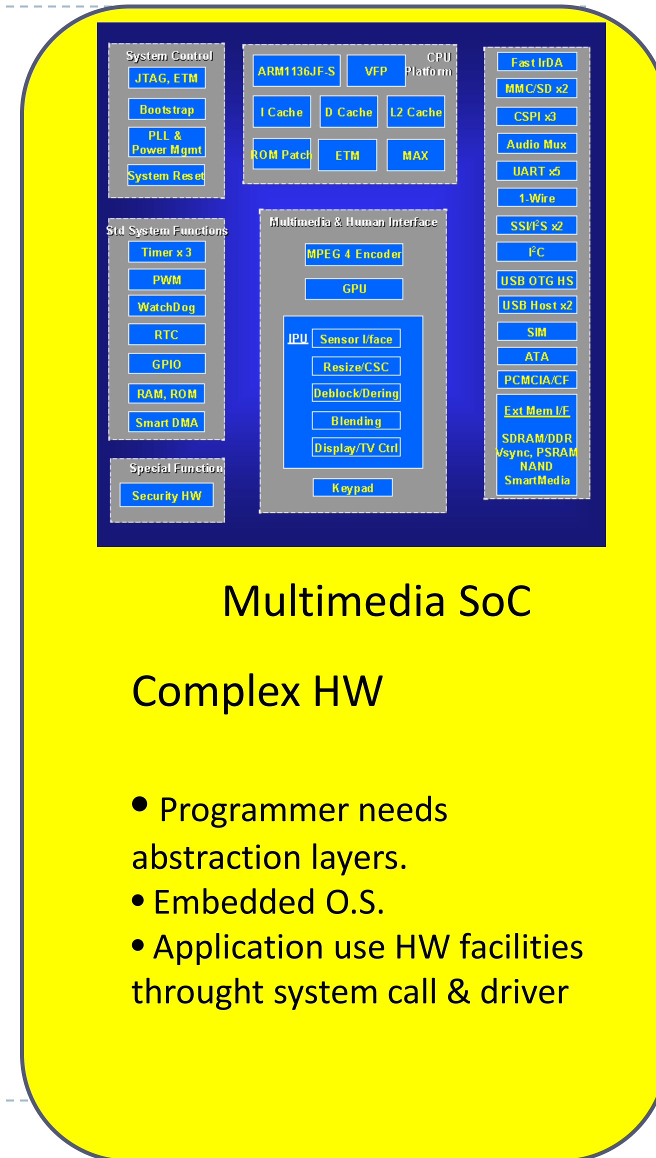

Microcontroller

SW & HW Low Complexity

- The application may use directly the HW
- The programmer know the HW and read and write directly internal register

Custom middleware

- HW programmed by statically linking API
- Standard / Custom API
- Microkernel

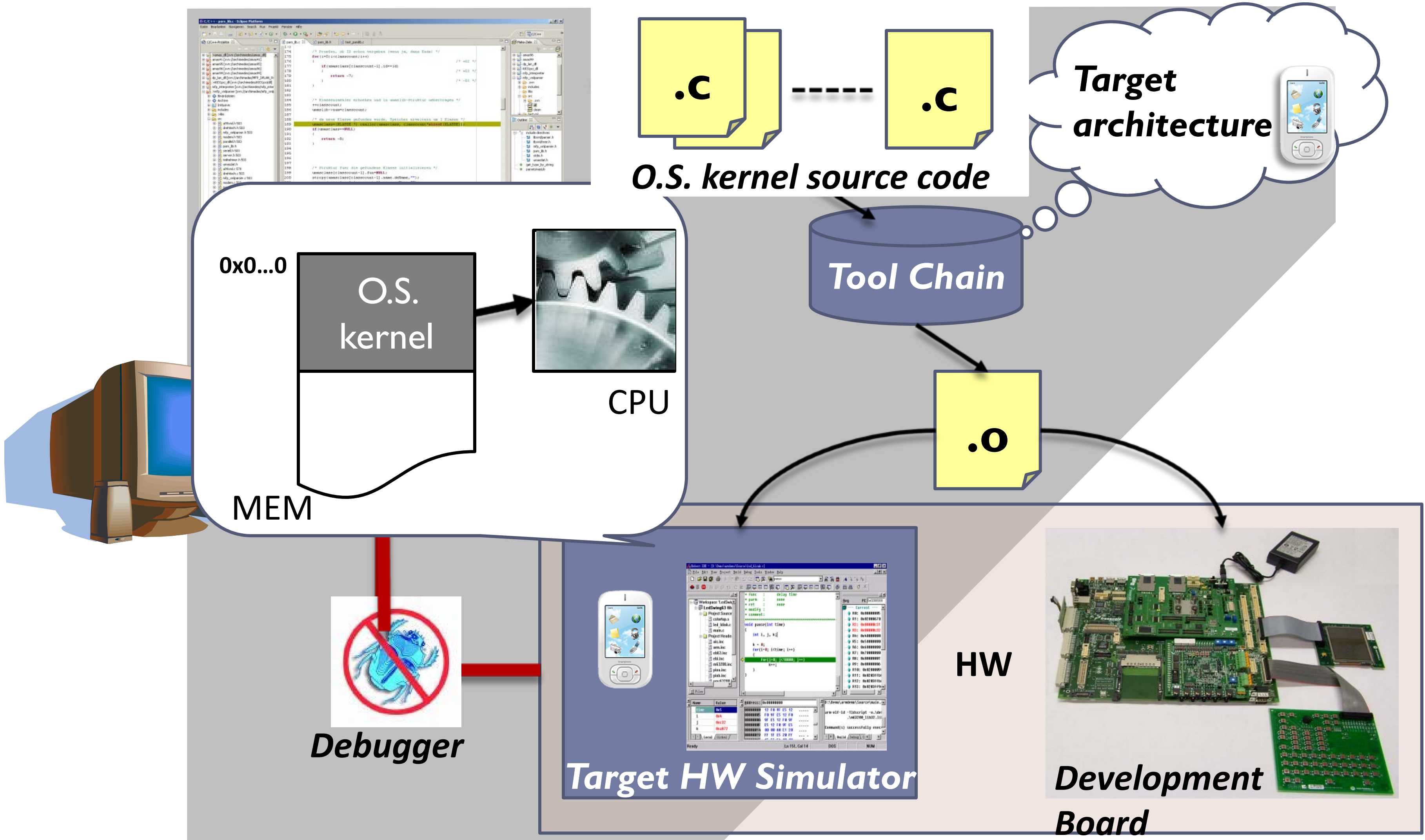



Multimedia SoC

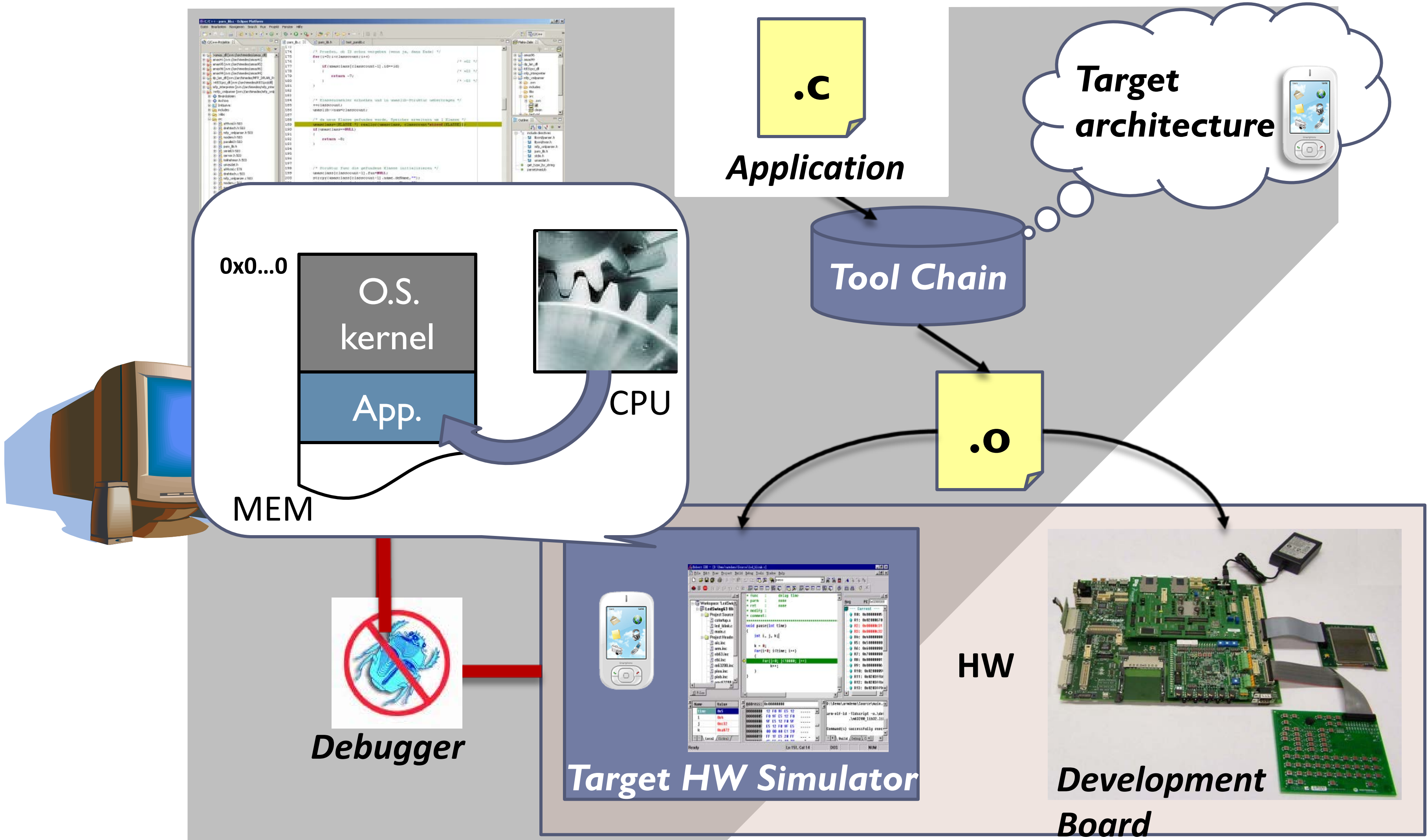
Complex HW

- Programmer needs abstraction layers.
- Embedded O.S.
- Application use HW facilities throught system call & driver

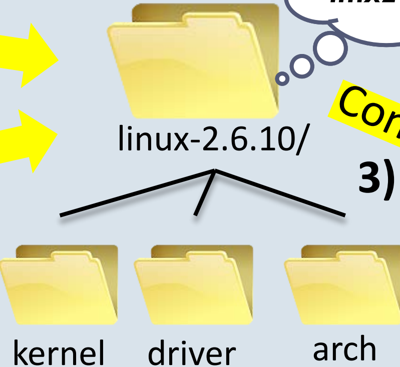
# Application Cross-Development

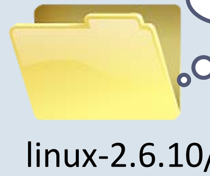


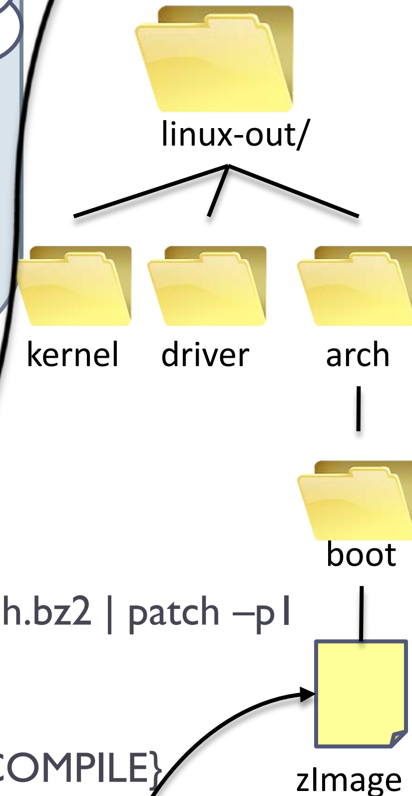
# Application Cross-Development



# Compiling kernel : steps




1. Kernel untar:
   - $ tar -xvf <repository>/linux-2.6.10.tar.bz2
2. Kernel patch:
   - <KernelSrcDir> $ bunzip2 –c <BSP>/linux-2.6.10-mxc-L26_1_17.patch.bz2 | patch –p1
3. Kernel configure:
   - <KernelSrcDir> $ make ARCH=arm CROSS_COMPILE=${CROSS_COMPILE} imx31ads_defconfig
4. Compile kernel :
   - <KernelSrcDir>$ make ARCH=arm CROSS_COMPILE=${CROSS_COMPILE} zImage

# Board Connections

- How to transfer zImage into ADS:
  - JTAG : Flash image into ADS non volatile memory
  - Red-boot:
    - TFTP : Allow to tranfer the zImage trough ethernet port in target SDRAM:
      - load –r –b 0x800000 zImage
      - exec  (start image)
    - Serial Port (Zmodem)
  - File System:
    - cramFS (Read only compressed FS, is dowloaded in target memory)
    - NFS (Remote FS, is physically on Host-PC but virtually on target system)

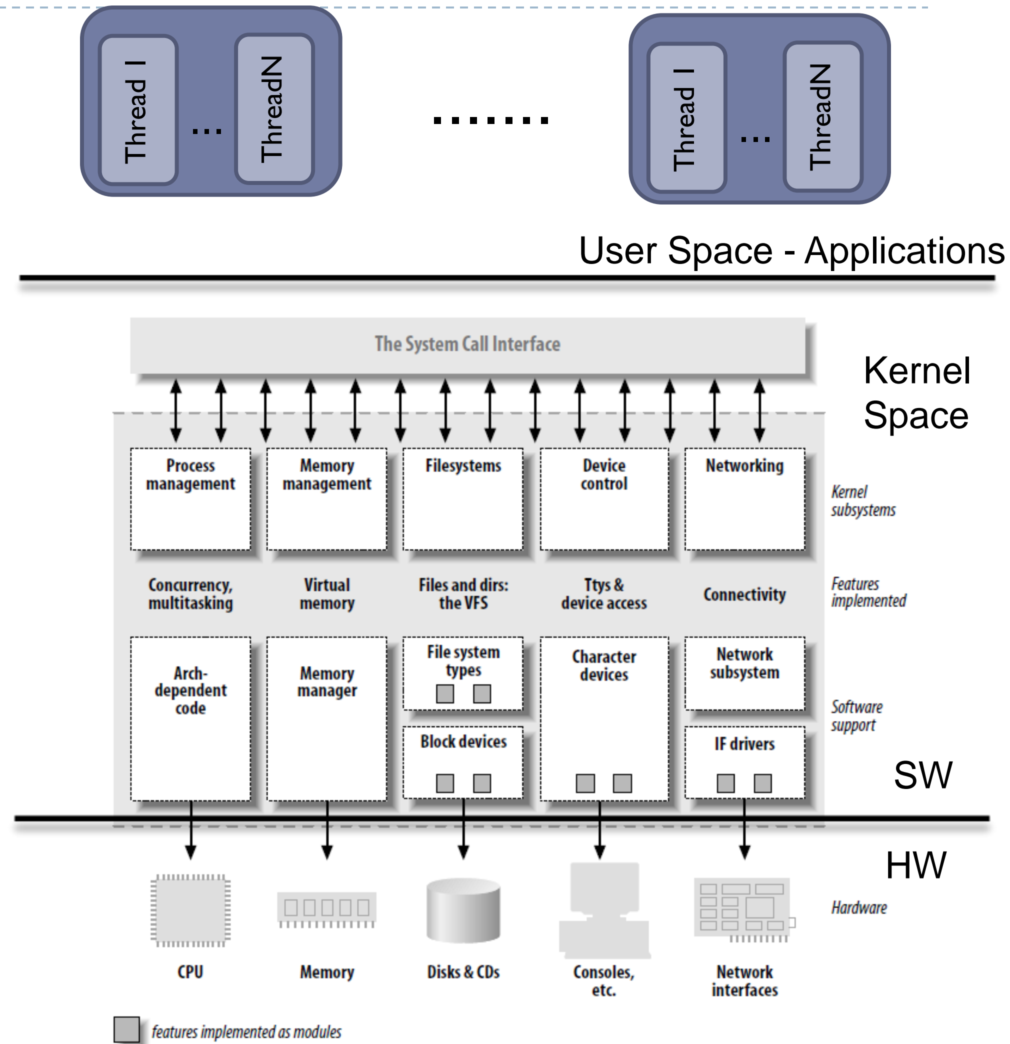
# Debug

- Terminal:
  - Allow the programmer to interact with the hardware
  - Allows inspection of the execution flow of the program through debug prints
  - When using O.S. allows remote shell control
- ICE (In circuit emulation):
  - JTAG protocols
  - Register and memory inspection
- GDB
  - Breakpoints
  - Register and memory inspection

# Operating System

Embedded Operating System - Linux:

- **Abstraction Layers**:
  - User Space - Application
  - Kernel Space
  - Multitasking -Scheduler
  - System Call
  - File system
  - Virtual memory
- **Modularity:**
  - Kernel module
  - Sincronizzation mechanism



# Kernel module

- Application

```
#include <stdio.h>

main ()
{
      printf("Hello, world \n");
}
```

- Kernel Module

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENCE("Dual BSP/GPL");

static int  hello_init(void)
{
      printk(KERNEL_ALERT "Hello,
world\n");
      return 0 ;
}

static void hello_exit(void)
{
     printk(KERNEL_ALERT "Hello,
world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Application

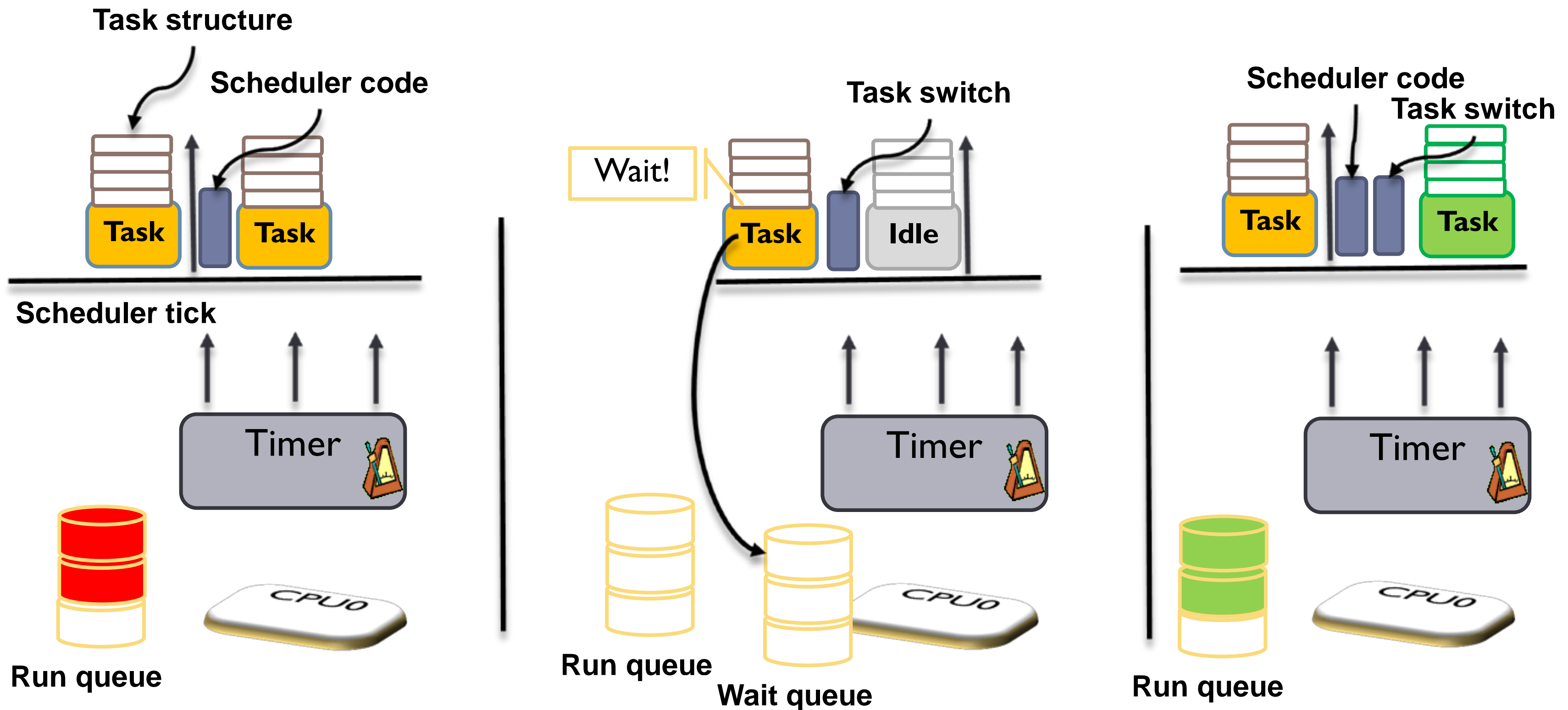
System Call

Kernel

REAL HW – architecture dependent

# Linux Scheduler

- **Single Core**
  - **Different Tasks / Application**



# Synchronization Mechanisms

- User Space - Application:
  - **Time Division Multiplexing**
    - O.S. Scheduler selects which task to run
  - Concurrency – different task can access to the same variables.
    - Semaphore
    - Critical section
    - Atomicity

- Kernel space :
  - Different modules that interacts
  - **HW Interrupt can preempt important kernel tasks**
  - How to avoid kernel corruption after an HW interrupt?
  - How to synchronize different kernel modules ?
  - How to synchronize with HW ?

Interrupt and Timers:

- There are two sources of interrupts in Linux, **synchronous** and **asynchronous**
- **Synchronous interrupts**, better known as **exceptions**, are generated by the CPU control unit
  - Can be generated by software special functions (ex. Timers) (**SW interrupts**)
- **Asynchronous interrupts** (known as interrupts) are **generated by HW resources**, such as serial module, or HW timers Interrupts.
- The address of all the interrupt service routines depends on architecture:
  - X86: programmed into the Interrupt Descriptor Table (IDT). Can be placed enywhere
  - ARM : All FIQ and IRQ are vectored to locations four bytes apart starting at zero

# Synchronization Mechanisms

## Interrupt :

- The nature of an **asynchronous interrupt** is that it happens at any time.
- If **it happens** during a time when the **kernel is busy** performing an **important function**, then the **kernel must do** the following:
  1. **Switch over** and **execute as much** of the **interrupt service routine** as necessary
  2. **Switch back** and **finish** the **remainder of the task** it **was performing before** the interrupt occurred
  3. **Switch back** yet again and **finish the remainder** of the **interrupt service routine**
- The **first half** of the **interrupt service routine** is referred to as the **top half**, while the **second half** is referred to as the **bottom half**

## Timers :

- Easy for device-driver programmers to use
- Used mainly **for detecting device "lockups"**
- But could also be used for other purposes
- The driver-writer merely needs to:
  1. **define** a "customized" **timeout-function**
  2. **allocate** and **initialize a kernel-structure**
  3. **call standard routines** for **timer-control**

```
struct timer_list  my_timer;

int init_module( void )
{
        init_timer( &my_timer );
        my_timer.data = (unsigned
        long)&my_data;
        my_timer.function = my_action;
        my_timer.expires = jiffies + 5 * HZ;
        add_timer( &my_timer );
        return SUCCESS;
}
```