Firmware for Embedded Computing

National Chiao Tung University
Chun-Jen Tsai
3/10/2011

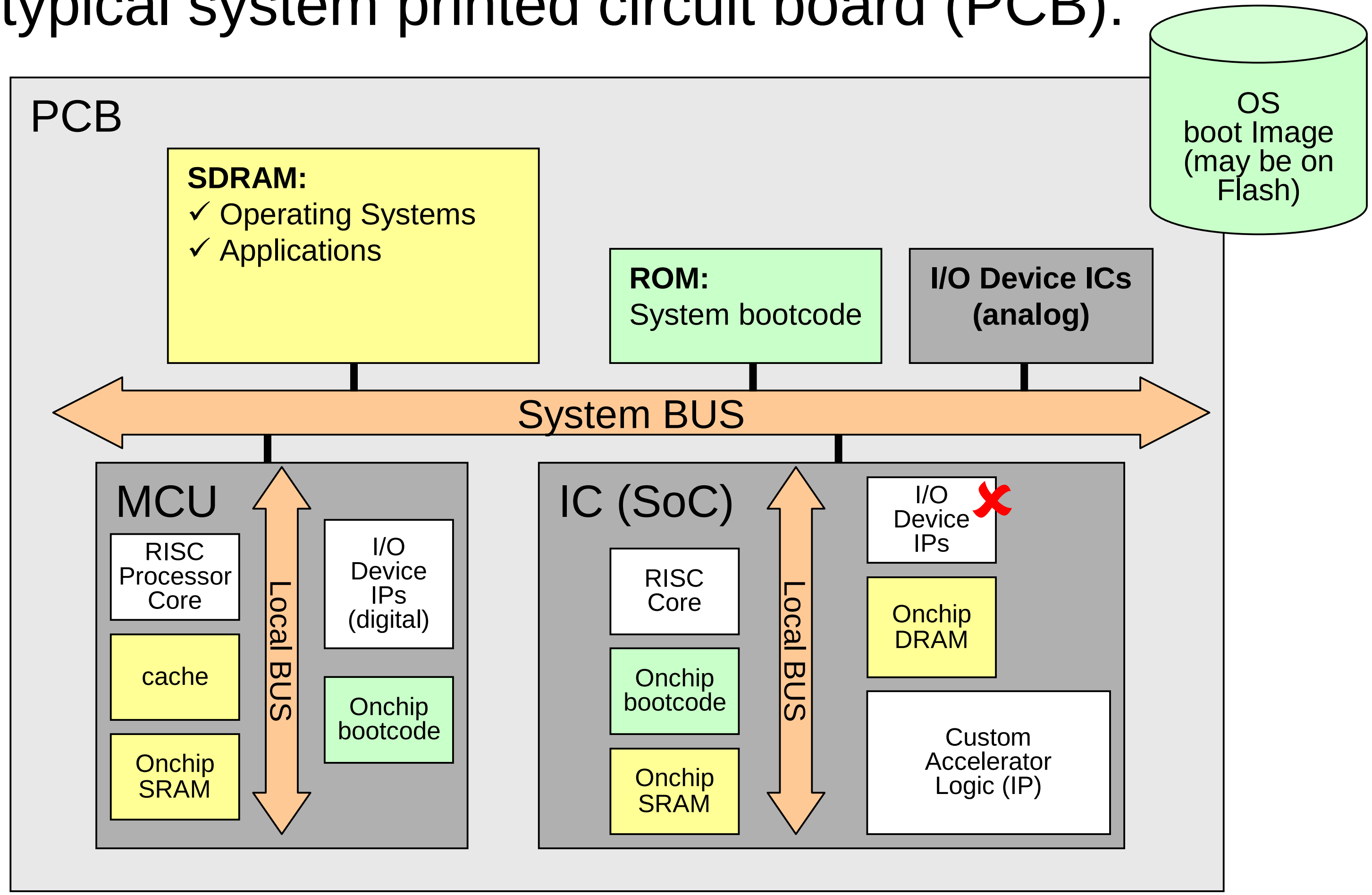
Define “Firmware”

- Firmware is a computer program that is embedded in a hardware device, for example a microcontroller. It can also be provided on flash ROMs or as a binary image file that can be uploaded onto existing hardware by a user†.
- Firmware is stored on non-volatile solid-state memory
- Typical functions of a firmware:
  - Booting and running a system (a board or a chip)
  - Providing basic I/O services
  - Providing debugging services
  - Providing backdoor for system recovery/maintenance

† www.wikipedia.org

Recursive System Layout

- A typical system printed circuit board (PCB):



Board-level vs. Chip-level Firmware

- Typical examples:
  - Board-level: BIOS, bootloader, debug agent, etc.
  - Chip-level: microcontroller codes for an MPEG codec chip, USB controller chip, etc.
- Firmware code size:
  - Board-level: ranges from several KB to several MB
  - Chip-level: as small as possible to reduce cost

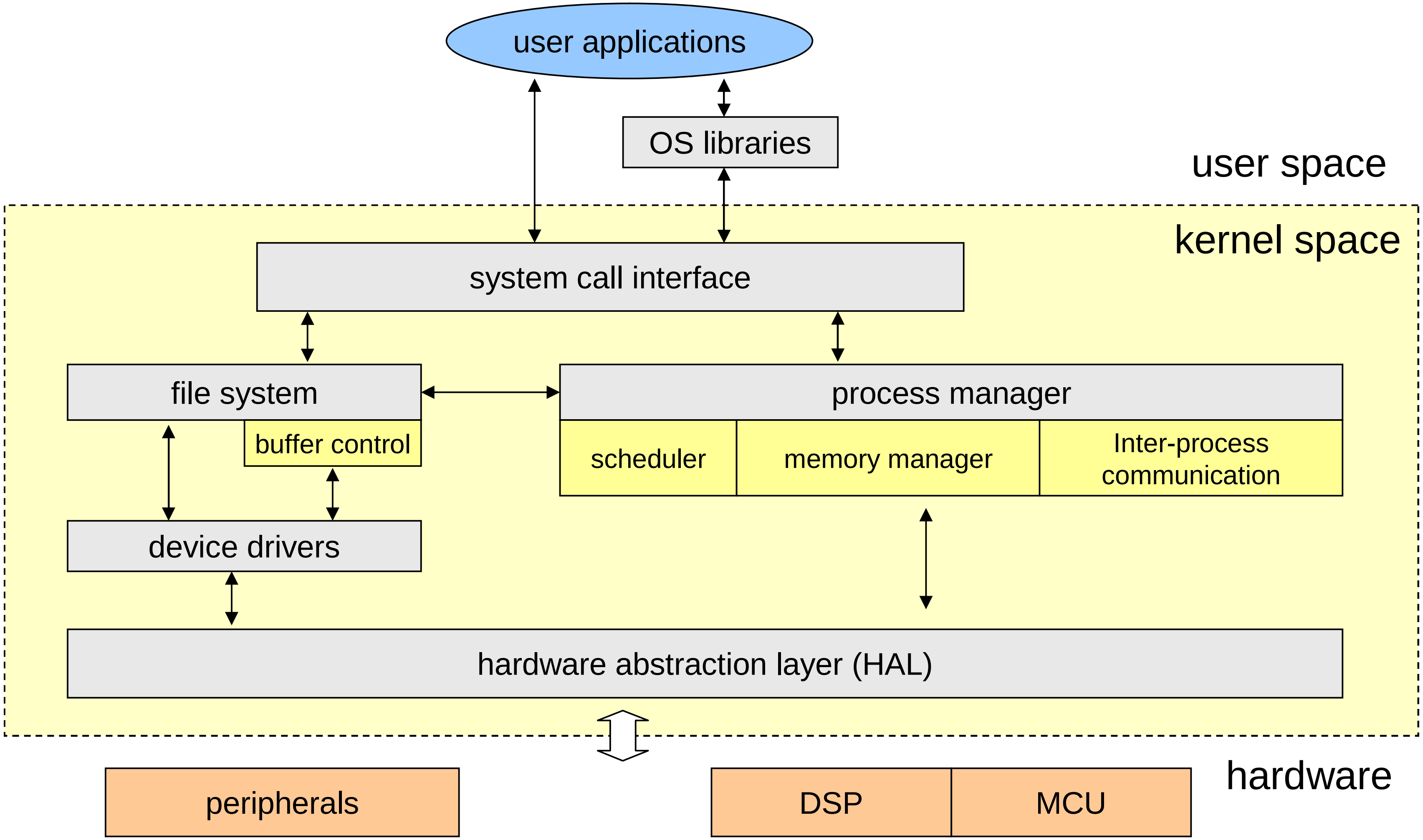
Board-level Firmware and OS

- There is a lot of similarity between board-level firmware and an operating system (OS)
  - Provides I/O interfaces for (application) program
  - Loads and executes program images
  - Helps program development
- In the good old days, the entire OS of a computer can be a firmware (e.g. Apple II)
- Today, more and more consumer electronics have a firmware-based OS (e.g. mobile phones, PDAs)

Operating Systems Components

- Process Management
  - Who gets to use the CPU?
- Memory Management
  - Who gets to use the runtime memory?
- File System
  - How to retrieve/store data?
- I/O (Sub)-system
  - How to talk to the peripherals?
  - Can be part of the file system (e.g. Unix)
- Graphics (Multimedia), Windowing, and Events Subsystem

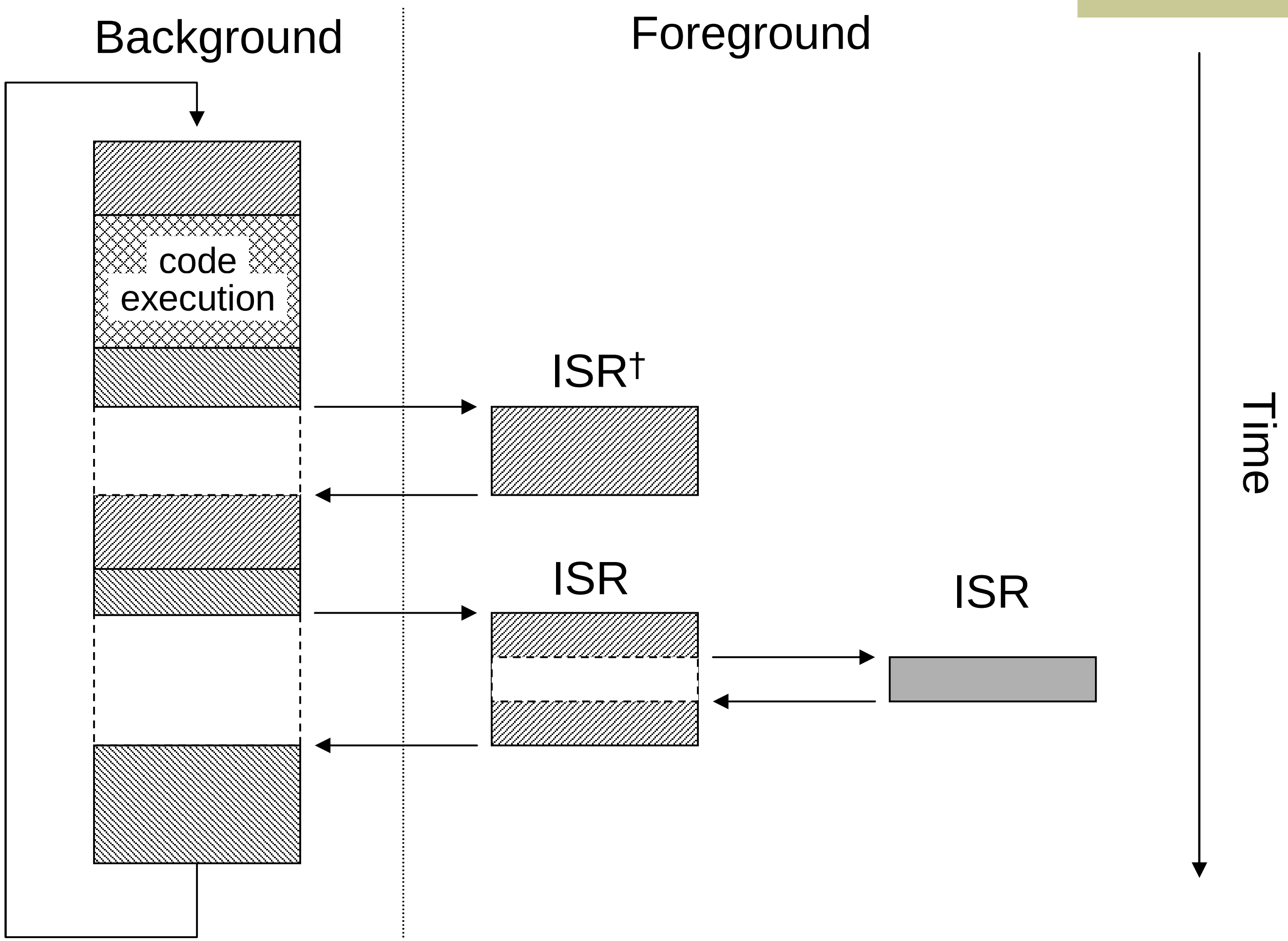
Typical Architecture of an OS



Can We Get away with "OS"?

- There are two kinds of tasks a typical embedded system runs:
  - Background operations:
    - Runs the main application of the embedded system
    - Handling routine (synchronous) tasks
    - Usually called *task level*
  - Foreground operations:
    - Handling asynchronous events
    - Usually called *interrupt level*

Foreground/Background Systems



† ISR stands for Interrupt Service Routine

F/B System vs. Full OS

- Why do we need an OS for embedded systems?
  - Handling complex scheduling problems for background tasks
  - Reduced development cost for a family of product
  - Easy extensibility (for new hardware)
  - Support for third party applications
  - Have someone to blame if things don't work … (e.g. *Microsoft*)
- In general, a firmware-based F/B system should be good enough for most embedded devices

Industrial Firmware Example: AFS

- Firmware are typically custom-designed for each embedded systems
  - However, many firmware codes can be recycled for different applications/projects
- The ARM Firmware Suite (AFS) is a collection of libraries and utilities designed as an aid to application and OS development on ARM-based platforms†

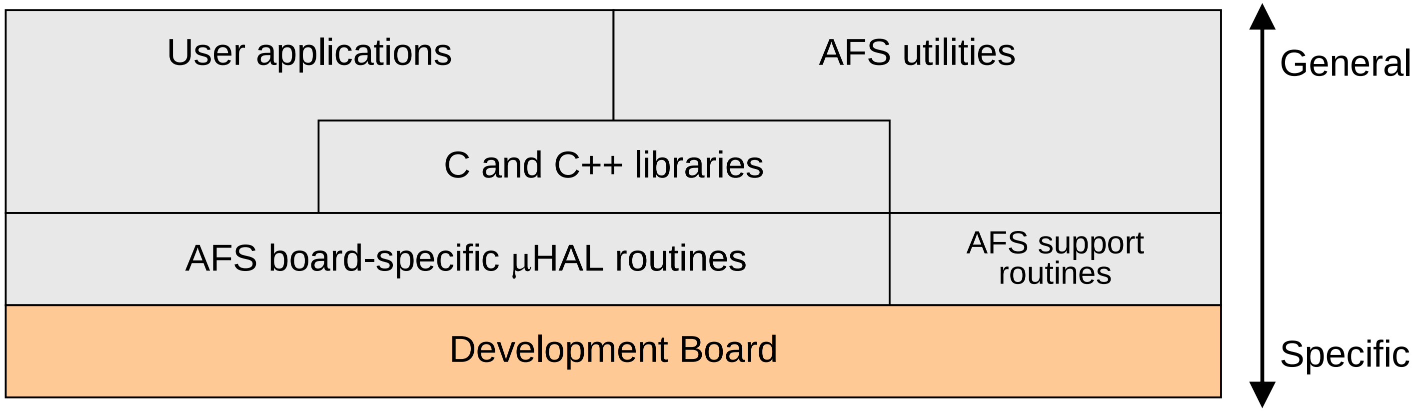
† AFS is free as long as it is used for ARM-based platforms!

AFS Components

- ***μ*HAL libraries:**
  The ARM Hardware Abstraction Layer; the basis of AFS
- **Flash library and utilities:**
  Library and utilities for programming on board flash memory
- **Angel:**
  A remote debugging monitor for ADS
- ***μ*C/OS-II:**
  AFS includes a port of the multitasking kernel, μC/OS II, using *μ*HAL API
- **Additional libraries:**
  Libraries for specialized hardware, (e.g. PCI bus, Vector Floating Point), exception handling, and compression
- **Additional components:**
  Including a boot monitor, generic applications, and board-specific applications

Logical Organization of AFS

- AFS also shields the hardware detail from user applications, however, it is much thinner than an OS



μHAL

- The μHAL libraries mask hardware differences between platforms by providing a standard layer of board-dependent functions
- Example of μHAL functions:
  - System (processor, memory, and buses) initialization
  - Serial ports initialization
  - Generic timers
  - Generic LEDs
  - Interrupt control
  - Code/data access in flash memory
  - Memory management (cache and MMU)

Example: Using μHAL (1/2)

- Getting Board Information:

```
infoType platformInfo;

/* who are we? */
uHALr_GetPlatformInfo(&platformInfo);
uHALr_printf("platform Id :0x%08X\n", platformInfo.platformId);
uHALr_printf("memory Size :0x%08X\n", platformInfo.memSize);
uHALr_printf("cpu ID :0x%08X\n", platformInfo.cpuId);
```

- LEDs Control:

```
int idx;
int count = uHALr_InitLEDs(); /* turn off all the LEDs */

for (idx = 0; idx < count; idx++) uHALr_SetLED(idx);
```

Example: Using μHAL (2/2)

- Installing a Timer:

```
static int OSTick = 0;
void TickTimer(unsigned int irq) { OSTick++; }

int main(int argc, char *argv[])
{
    uHALr_InitInterrupts(); /* Install trap handlers */
    uHALr_InitTimers();     /* Initialize the timers */

    uHALr_printf("Timer init\n");
    if (uHALr_RequestSystemTimer(TickTimer, "test") <= 0)
    {
        uHALr_printf("Timer/IRQ busy\n");
    }

    uHALr_InstallSystemTimer(); /* Enable the interrupt */

    /* Get the interval per tick */
    interval = uHALir_GetSystemTimerInterval();
}
```

Firmware Operation Modes

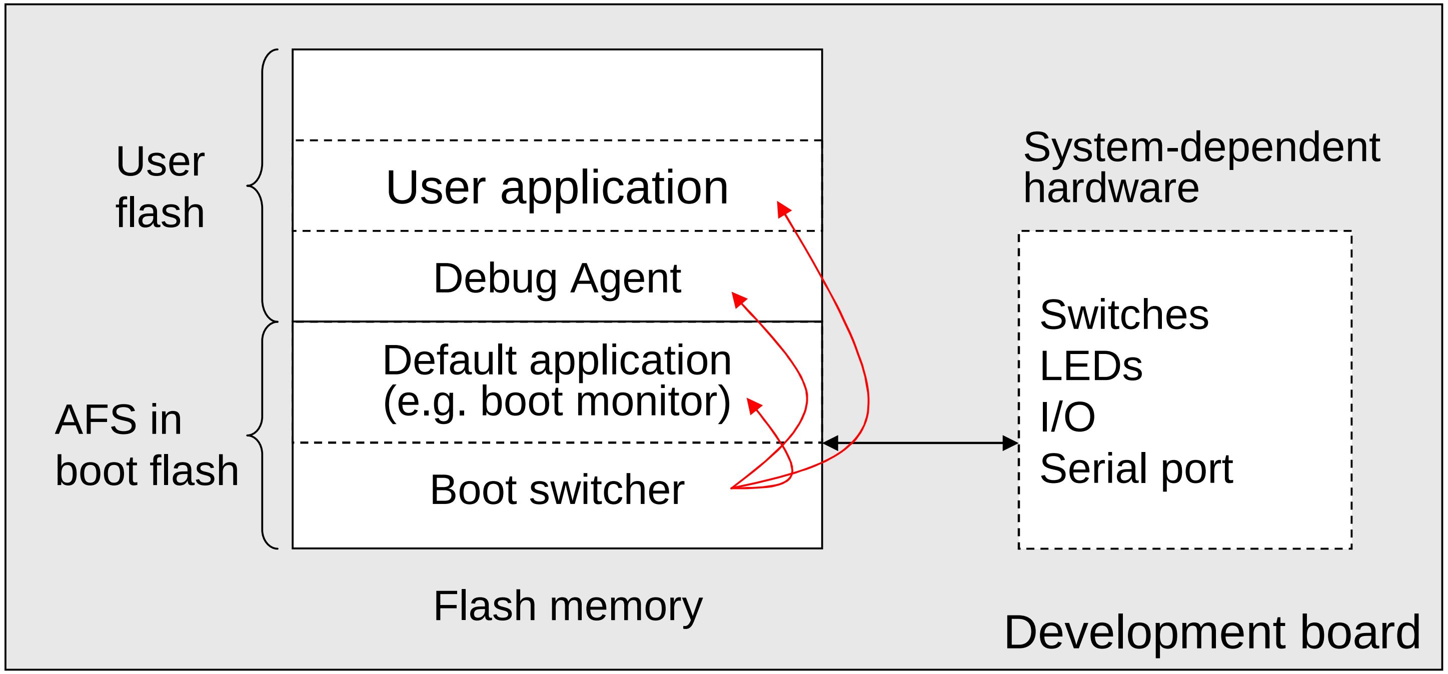
- An embedded system application operates in one of two modes:
  - Standalone:
    A standalone application is one that has complete control of the system from boot time onwards
  - Semihosted:
    A semihosted application is one for which an application or debug agent, such as Angel or Multi-ICE, provides or simulates facilities that do not exist on the target system

```
uHALr_SetLED(0);

#ifdef SEMIHOSTED
    /* All done, give semihosted a chance to break in … */
    uHALr_printf("Press a key to repeat the test.\n");
    uHALr_getchar();
#endif
```

Typical Firmware Images

- A typical prototype board with AFS



Weakness of Thin Firmware

- A system build upon a thin firmware, such as AFS, has the following weaknesses:
  - Can only implement F/B systems: no multithreading for the applications
  - No device driver model: adding new hardware requires extension of system call API
  - Selection of different applications can only be done via boot switcher

Deeply-Embedded OS

- Sometimes, we need a little extra functions in the firmware to implement a powerful F/B systems
  - Multi-threading for the background task (but still allowing only single process)
  - Installable device drivers (the “installation” may happen before the build time)
  - Componentized model for building a custom system
  - More flexible foreground task management
  - Remains small (otherwise, we can use a full-blown OS)

eCos: A Deeply Embedded OS

- eCos stands for Embedded Configurable Operating System
- eCos is an “application-oriented” OS: The OS can be configured for a specific application/platform combination (e.g. digital security camera)
- eCos is something between a “full” OS and a thin firmware

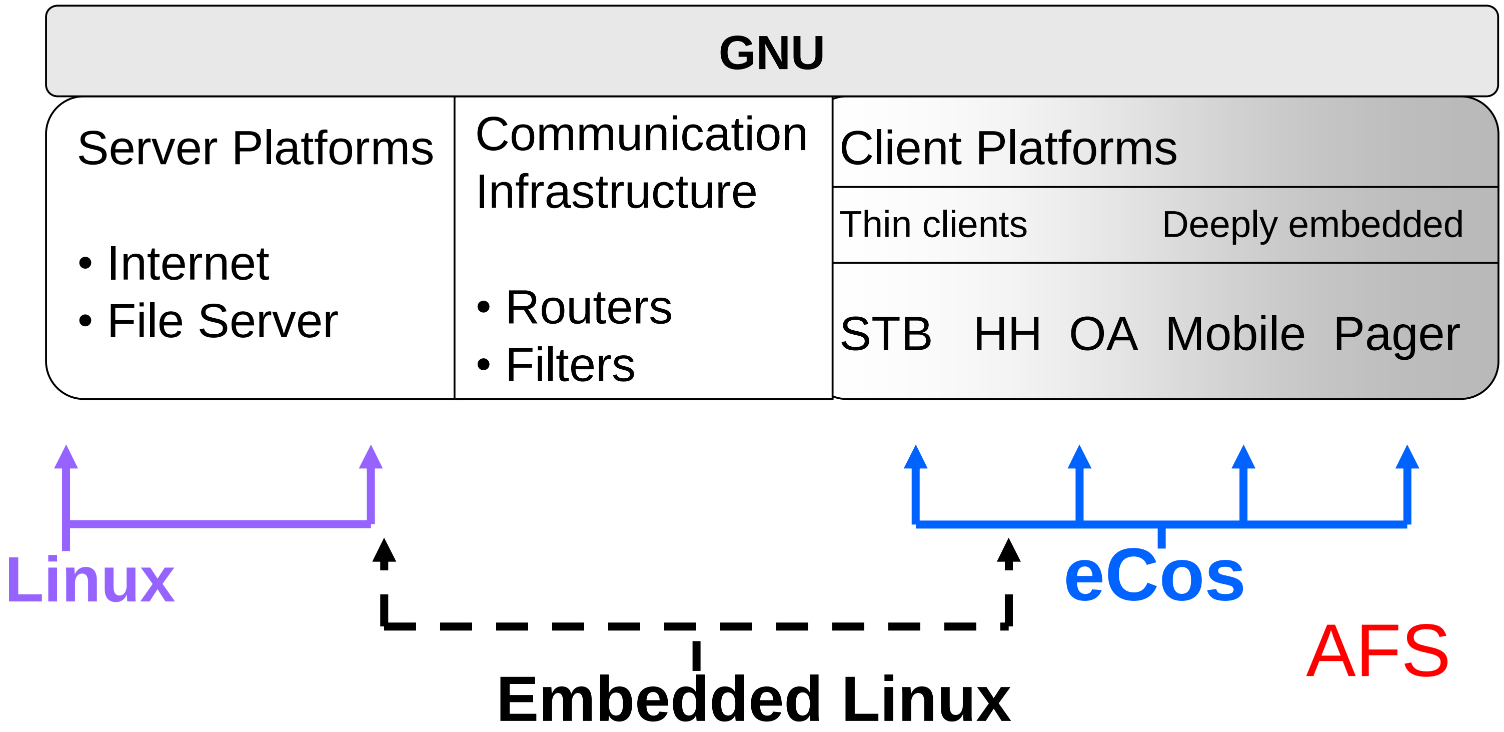
eCos Features (1/2)

- Embedded Configurable Operating System
  - Developed and maintained by RedHat
  - Not Linux-based
  - Under eCos license (similar to GPL)
  - Current version 2.0
  - Minimum footprint : about 50 Kbytes
  - Single address space for all threads
  - Real-time support

eCos Features (2/2)

- eCos is different from other Embedded OS
  - Dynamic memory management is not part of the kernel
  - Device drivers are handled as “packages“ as well
  - eCos kernel is an “optional” package of the OS. It is only required when multi-threading support is required for the application
  - eCos is linked with the user application as a single runtime image!

“Open” Embedded OS Spectrum†



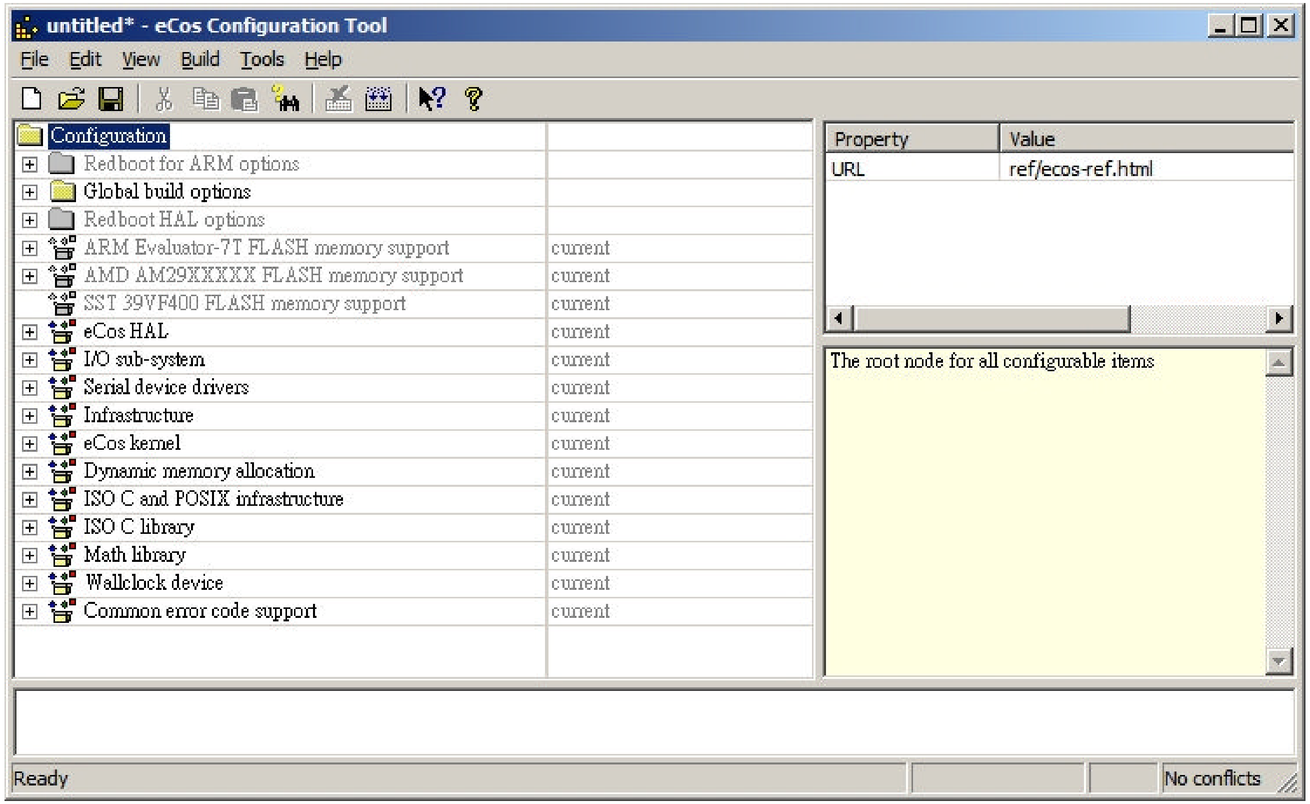
† http://sources.redhat.com/elix/presentation/esc-west99

eCos Supported Hardware

- Architectures:
  - ARM, CalmRISC, FR-V, H8, IA32, M68K, Matsushita AM3x, MIPS, NEC V8xx, PowerPC, SPARC (Leon), SuperH
- Devices:
  - Flash: AMD, ATMEL, Intel, Sharp
  - Ethernet: AMD, Cirrus Logic, Intel, Motorola, Natioanl Semiconductor, SMSC, VIA
  - Seriel: Motorola, 1655x, 8250
  - USB: Intel SA-11x0 on-chip, NEC uPD985xx on-chip
  - RTC: DS-1742
- See http://sources.redhat.com/ecos/hardware.html for a complete list of the supported platforms

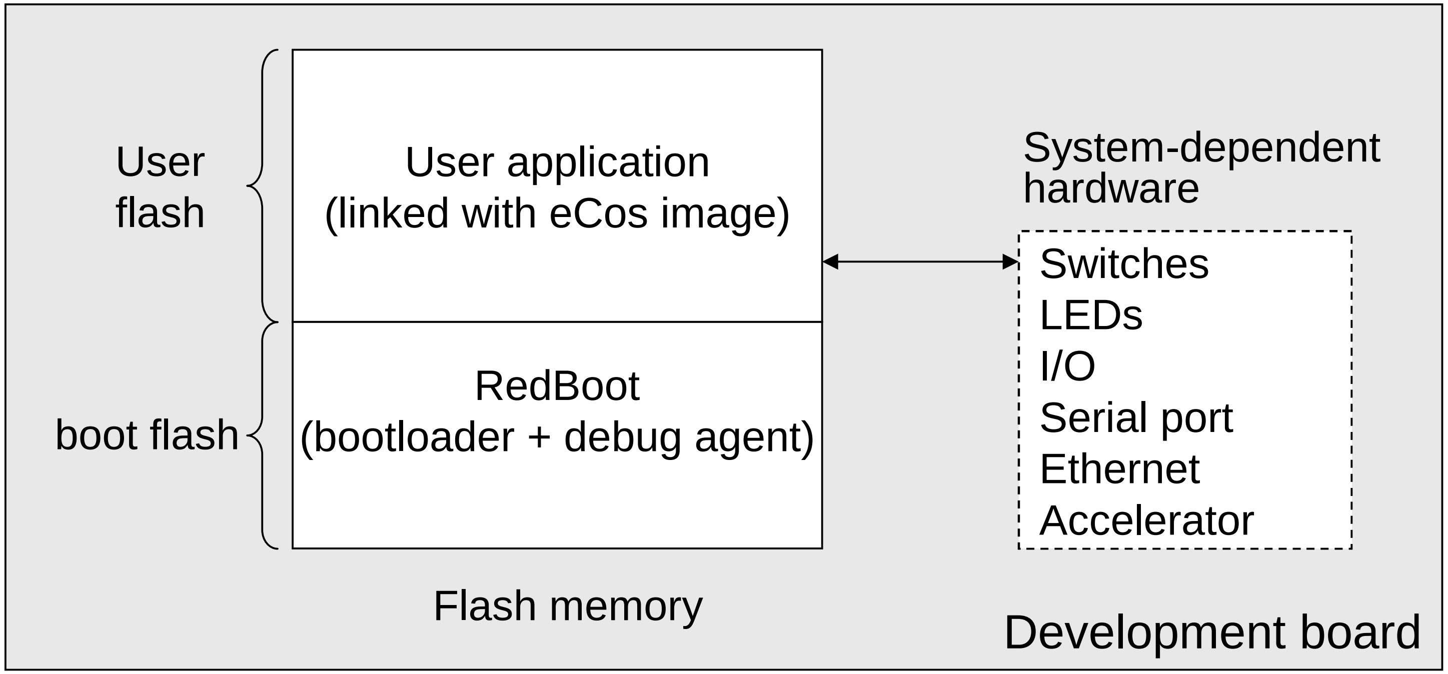
Componentized Build Environment

- eCos components required for a custom build can be selected using a GUI build tool



Typical eCos Usage

- Typical eCos board-level Image (could be for chip-level bootcode as well)



RedBoot – the Boot Loader for eCos

- RedBoot -- RedHat embedded debug and bootstrap loader
  - Based on eCos HAL
  - Support boot scripting
  - Simple command line interface
  - Support flash & network booting of OS
  - Support BOOTP, DHCP
  - Support TFTP, X/Y-modem for program download
  - Support GDB for remote debugging via serial or Ethernet connections
- Source code:
  - http://sources.redhat.com/redboot/

eCos Kernel

- Support multi-threading embedded applications:
  - The ability to create new threads in the system, either during startup or when the system is already running
  - Control over the various threads in the system, for example manipulating their priorities
  - A choice of schedulers, determining which thread should currently be running
  - A range of synchronization primitives, allowing threads to interact and share data safely
  - Integration with the system's support for interrupts and exceptions

eCos Kernel Is Optional

- For simple foreground/background (F/B) systems, the eCos kernel package can be skipped
  - F/B applications have a central polling loop, continually checking all devices and taking appropriate action when I/O occurs
  - RedBoot is one of such eCos applications
    - However, RedBoot with network support includes the kernel since the TCP/IP stack uses multithreading internally

eCos Schedulers

- eCos support two types of task scheduler
  - Bitmap scheduler
  - Multi-level queue (MLQ) scheduler
- The number of priority levels is configurable -- the default is 32
- Low priority thread only runs if all higher priority threads are blocked

Bitmap Scheduler

- Only allows one thread per priority level, so if the system is configured with 32 priority levels then it is limited to only 32 threads
- Bitmaps can be used to keep track of which threads are currently runnable, waiting on a mutex, or other synchronization primitive
- Bitmap scheduler is fast and totally deterministic
- Does not support SMP and priority inversion prevention

MLQ Scheduler

- The MLQ scheduler allows multiple threads to run at the same priority
  - Each priority level maintains a queue of threads
  - Timeslicing is used among threads in same priority queue
  - Timeslicing can be enabled/disabled at kernel build time
- Operation of finding the highest priority queue with runnable thread is expensive
  - Default behavior is LIFO, which does not guarantee highest priority queue will be examined next; in this case MLQ is only used to increase the number of simultaneous threads
  - MLQ scheduler also support strict priority queuing but the system's dispatch latency is worse

eCos Synchronization Primitives

- Mutexes
- Condition variables
- Counting semaphores
- Mail boxes
- Event flags (binary semaphores)

Sync. Support in Device Drivers

- The eCos common HAL package provides its own device driver API which contains some of the above synchronization primitives
- If the configuration includes the eCos kernel package then the driver API routines map directly onto the equivalent kernel routines
- If the kernel package is not included and the application consists of just a simple F/B system then the driver API is implemented entirely within the common HAL

Interrupt Handling

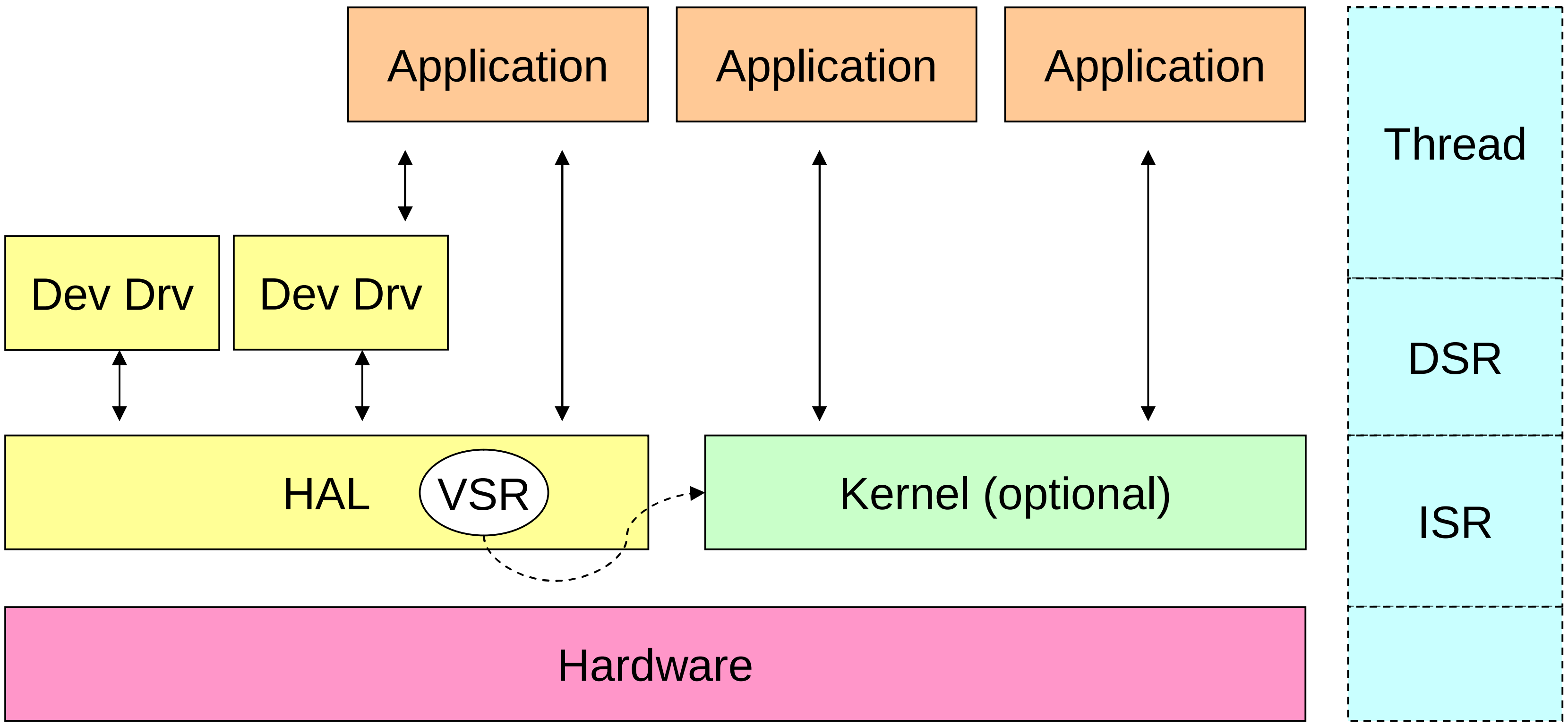
- Kernel uses a two-level approach to interrupt handling:
  - Associated with every interrupt vector is an Interrupt Service Routine or ISR, which will run as quickly as possible
  - However an ISR can make only a small number of kernel calls, and it cannot make any call that would cause a thread to wake up
  - If an ISR detects that an I/O operation has completed it can cause the associated Deferred Service Routine (DSR) to run and make more kernel calls, for example, to signal a condition variable or post to a semaphore

Interrupt Translation†

- eCos translates different interrupt jump table mechanisms to a common approach
- Each hardware vector executes a trampoline code that makes an indirect jump via a table to the actual handler called the Vector Service Routine (VSR)
- The trampoline code performs the absolute minimum processing to identify the exception source, and jump to the VSR
- The VSR is responsible for saving the CPU state and handle the exception or interrupt

† eCos reference manual, p.188

eCos HAL & Kernel



eCos HAL Principles

- eCos kernel itself is largely implemented in C++, but the HAL is implemented in C and assembly to enforce portability
  - All interfaces to the HAL are implemented by CPP macros
- The HAL provides simple, portable mechanisms for dealing with the hardware of a wide range of architectures and platforms

HAL Structure (1/2)

- Common HAL
  - Generic debugging functionality, driver API, eCos/ROM monitor calling interface, and tests.
- Architecture HAL
  - Architecture specific debugger functionality
  - Exception/interrupt vector definitions and handlers
  - Cache definition and control macros
  - Context switching code
  - Assembler functions for early system initialization
  - Configuration options

HAL Structure (2/2)

- Variant HAL
  - Extensions to the architecture code (cache, exception/interrupt)
  - Configuration options
  - Drivers for variant on-core devices
- Platform HAL
  - Early platform initialization code
  - Platform memory layout specification
  - Configuration options (processor speed, compiler options)
  - Diagnostic IO functions
  - Debugger IO functions
  - Platform specific extensions to architecture or variant code (off-core interrupt controller)
- Auxiliary HAL

Discussions

- Thin firmware such as AFS is less and less popular for embedded systems
- If the application platform of a device is based on an open standard (e.g. J2ME or Android), a deeply embedded OS kernel such as eCos is a better choice than Linux, WinCE, BSD Unix, …, etc. for complex embedded systems
  - For multimedia, feature-rich functions, just leave it to the portable system middleware
- Software is the key to high-value consumer electronics