

Operating systems for embedded systems

- Embedded operating systems
 - How do they differ from desktop operating systems?
- Programming model
 - Process-based
 - Event-based
 - How is concurrency handled?
 - How are resource conflicts managed?
- Programming languages
 - C/C++
 - Java/C#
 - Memory management
 - Atomicity in the presence of interrupts

Embedded Operating Systems

- Features of all operating systems
 - Abstraction of system resources
 - Managing of system resources
 - Concurrency model
 - Launch applications
- Desktop operating systems
 - General-purpose – all features may be needed
 - Large-scale resources – memory, disk, file systems
- Embedded operating systems
 - Application-specific – just use features you need, save memory
 - Small-scale resources – sensors, communication ports

System Resources on Typical Sensor Nodes

- Timers
- Sensors
- Serial port
- Radio communications
- Memory
- Power management

Abstraction of System Resources

- Create virtual components
 - E.g., multiple timers from one timer
- Allow them to be shared by multiple threads of execution
 - E.g., two applications that want to share radio communication
- Device drivers provide interface for resource
 - Encapsulate frequently used functions
 - Save device state (if any)
 - Manage interrupt handling

Very simple device driver

- Turn LED on/off
- Parameters:
 - port pin
- API:
 - on(port_pin) - specifies the port pin (e.g., port D pin 3)
 - off(port_pin)
- Interactions:
 - only if other devices want to use the same port

Simple device driver

- Turning an LED on and off at a fixed rate
- Parameters:
 - port pin
 - rate at which to blink LED
- API:
 - on(port_pin, rate)
 - specifies the port pin (e.g., port D pin 3)
 - specifies the rate to use in setting up the timer (what scale?)
 - off(port_pin)
- Internal state and functions:
 - keep track of state (on or off for a particular pin) of each pin
 - interrupt service routine to handle timer interrupt

Interesting interactions

- What if other devices also need to use timer (e.g., PWM device)?
 - timer interrupts now need to be handled differently depending on which device's alarm is going off
- Benefits of special-purpose output compare peripheral
 - output compare pins used exclusively for one device
 - output compare has a separate interrupt handling routine
- What if we don't have output compare capability or run out of output compare units?

Sharing timers

- Create a new device driver for the timer unit
 - Allow other devices to ask for timer services
 - Manage timer independently so that it can service multiple requests
- Parameters:
 - Time to wait, address to call when timer reaches that value
- API:
 - `set_timer(time_to_wait, call_back_address)`
 - Set `call_back_address` to correspond to `time+time_to_wait`
 - Compute next alarm to sound and set timer
 - Update in interrupt service routine for next alarm
- Internal state and functions:
 - How many alarms can the driver keep track of?
 - How are they organized? FIFO? priority queue?

Concurrency

- Multiple programs interleaved as if parallel
- Each program requests access to devices/services
 - e.g., timers, serial ports, etc.
- Exclusive or concurrent access to devices
 - allow only one program at a time to access a device (e.g., serial port)
 - arbitrate multiple accesses (e.g., timer)
- State and arbitration needed
 - keep track of state of devices and concurrent programs using resource
 - arbitrate their accesses (order, fairness, exclusivity)
 - monitors/locks (supported by primitive operations in ISA - test-and-set)
- Interrupts
 - disabling may effect timing of programs
 - keeping enabled may cause unwanted interactions

Handling concurrency

- Traditional operating system
 - multiple threads or processes
 - file system
 - virtual memory and paging
 - input/output (buffering between CPU, memory, and I/O devices)
 - interrupt handling (mostly with I/O devices)
 - resource allocation and arbitration
 - command interface (execution of programs)
- Embedded operating system
 - lightweight threads
 - input/output
 - interrupt handling
 - real-time guarantees

Embedded operating systems

- Lightweight threads
 - basic locks
 - fast context-switches
- Input/output
 - API for talking to devices
 - buffering
- Interrupt handling (with I/O devices and UI)
 - translate interrupts into events to be handled by user code
 - trigger new tasks to run (reactive)
- Real-time issues
 - guarantee task is called at a certain rate
 - guarantee an interrupt will be handled within a certain time
 - priority or deadline driven scheduling of tasks

Some Examples

embedded operating systems typically reside in ROM (flash) - changed rarely

- FreeRTOS
 - scale-able real time kernel: preemptive, cooperative or hybrid configurations
 - designed specifically for small embedded systems
 - support for >30 embedded system architectures
 - Designed to be small, simple and easy to use, kernel = 4k to 9k bytes
- Wind River Systems VxWorks
 - one of the most popular embedded OS kernels
 - highly portable to an even wider variety of processors (tiny to huge)
 - modularized even further than the ones above (basic system under 50K)
- TinyOS
 - Open-source development environment specifically for small sensors
 - Simple (and tiny) operating system
 - Scheduler/event model of concurrency
 - Software components for efficient modularity
 - Software encapsulation for resources of sensor networks
 - Programming language and model – nesC

Metrics in Real-Time Systems (1/2)

- **End-to-end latency:**
 - E.g. worst-case, average-case, variance, distribution
 - Can involve multiple hops (across nodes, links, switches and routers)
 - Behavior in the presence or absence of failures
- **Jitter**
- **Throughput:**
 - How many X can be processed?
 - How many messages can be transmitted?
- **Survivability:**
 - How many faults can be tolerated before system failures?
 - What functionality gets compromised?

Metrics in Real-Time Systems (2/2)

- **Security:**
 - Can the system's integrity be compromised?
 - Can violations be detected?
- **Safety:**
 - Is the system "safe"?
 - Can the system get into an 'unsafe' state? Has it been 'certified'?
- **Maintainability:**
 - How does one fix problems?
 - How does the system get upgraded?
- **Dynamism and Adaptability:**
 - What happens when the system mission changes?
 - What happens when individual elements fail?
 - Can the system reconfigure itself dynamically?
 - How does the system behave after re-configuration?

RTOS Considerations

- What processor(s) does it run on?
 - 8-bit, 16-bit, 32-bit, ...
 - Intel Pentium® Processor, PowerPC, Arm, MIPS, SuperH, ...
 - IBM and Intel® Network Processors
- What board(s) does it run on?
 - Complete software package for a particular hardware board is called a BSP (Board Support Package)
- What is the software environment?
 - Compilers and debuggers
 - IDE
 - Cross-compilation + symbolic debugging on target?
 - Profilers (CPU, memory)
 - Test coverage tools
 - Native simulation/emulation support?

Real-Time Operating Systems

- FreeRTOS
- VxWorks from Wind River Systems (www.windriver.com)
- Linux variants
 - Blue Cat Linux (www.linuxworks.com)
 - (Embedded) Red Hat Linux (www.redhat.com)
 - FSM RT-Linux (www.fsmlabs.com)
 - Monta Vista Linux (www.mvista.com)
 - TimeSys Linux (www.timesys.com)
- LynxOS (www.linuxworks.com)
- QNX (www.qnx.com)
- Solaris real-time extensions
- TRON
 - Embedded OS specification in Japan
 - Has multiple profiles for different classes of devices

Common RTOS Features

Utilities

- Bootstrapping support
- “Headless” operation
 - Display not necessary

APIs (Application Programming Interfaces)

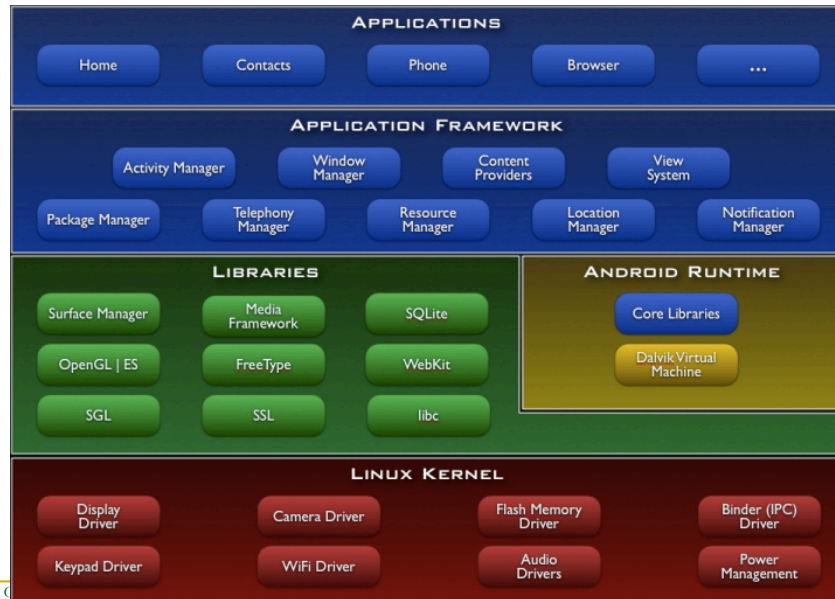
- Multiple threads and/or processes
 - Fixed priority scheduling is most popular
- Mutex/semaphore support likely with priority inheritance support
- Inter-process communications
 - Message queues
- Timers/clock
- Graphics support
- Device drivers
- Network protocol stack

Emerging RTOS Requirements

- Full-featured operating system
- Support for new processors and devices
- Support for Internet protocols and standards
- Support for Multimedia protocols and standards
- Support for File Systems
- Memory protection
- Resource protection, security
- Development tools and libraries
- GUI Environment

Do this with low and predictable overheads.

Future?? Android Layer Cake



Introduction

What is Android?

- A software platform and operating system for mobile devices
- Based on the Linux kernel
- Developed by Google and later the Open Handset Alliance (OHA)
- Allows writing managed code in the Java language

Unveiling of the Android platform was announced on 5 November 2007 with the founding of OHA

Early History of Android

- Google acquired the startup company Android Inc. in 2005 to start the development of the Android Platform. The key players at Android Inc. included Andy Rubin, Rich Miner, Nick Sears, and Chris White.
- In late 2007, a group of industry leaders came together around the Android Platform to form the Open Handset Alliance (<http://www.openhandsetalliance.com>).
- The Android SDK was first issued as an “early look” release in November 2007.
- In September 2008 T-Mobile announced the availability of the T-Mobile G1, the first smartphone based on the Android Platform.
- A few days after that, Google announced the availability of Android SDK Release Candidate 1.0.
- In October 2008, Google made the source code of the Android Platform available under Apache’s open source license.

Early History of Android

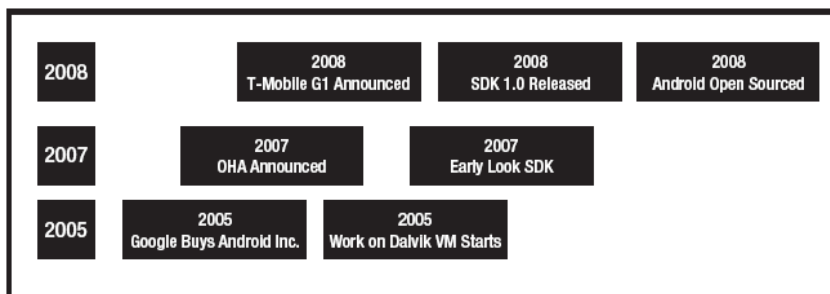


Figure 1-3. Android timeline

Platform

Hardware

Android is not a single piece of hardware; it's a complete, end-to-end software platform that can be adapted to work on any number of hardware configurations. Everything is there, from the bootloader all the way up to the applications.

Google Android

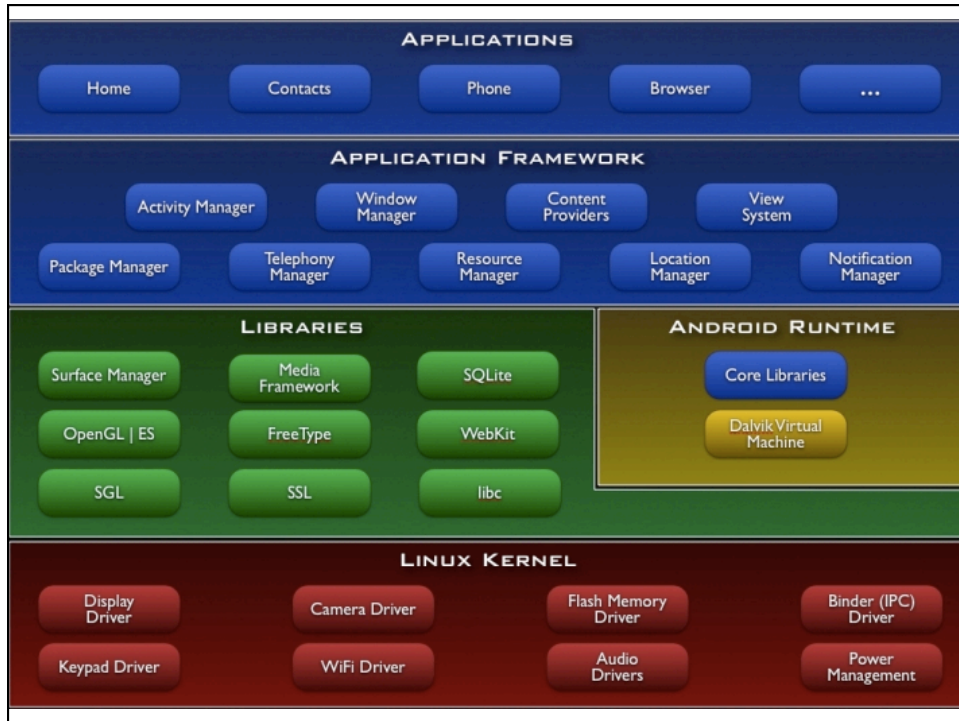
Platform

Network Connectivity

It supports wireless communications using:

- GSM mobile-phone technology
- 3G
- Edge
- 802.11 Wi-Fi networks

Google Android



Linux Kernel



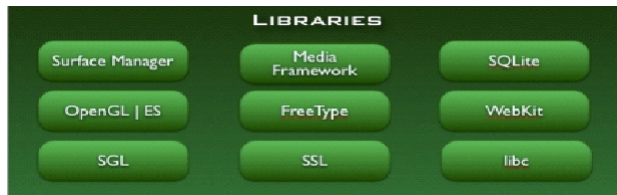
- Works as a HAL
- Device drivers
- Memory management
- Process management
- Networking



Libraries



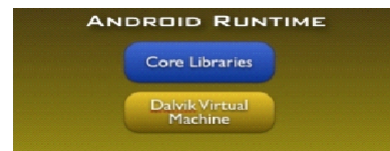
- C/C++ libraries
- Interface through Java
- Surface manager – Handling UI Windows
- 2D and 3D graphics
- Media codecs, SQLite, Browser engine



Android Runtime



- Dalvik VM
 - Dex files
 - Compact and efficient than class files
 - Limited memory and battery power
- Core Libraries
 - Java 5 Std edition
 - Collections, I/O etc...



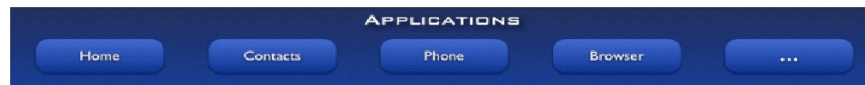
Replaced in Android 5.0 “Lollipop” with **Android Runtime (ART)**

Application Framework



- API interface
- Activity manager – manages application life cycle.

Applications



- Built in and user apps
- Can replace built in apps

Software development

Development requirements

- Java
- Android SDK

Google Android

Software development

IDE and Tools

Android SDK

- Class Library
- Developer Tools
- Emulator and System Images
- Documentation and Sample Code

Eclipse IDE + ADT (Android Development Tools)

- Reduces Development and Testing Time
- Makes User Interface-Creation easier
- Makes Application Description Easier

Android Studio

- Based on IntelliJ IDEA IDE

Google Android