Real-time, embedded and the Linux aspect

Iftach Hyams

Goals

- Few definitions, terms and characteristics.
- Going over various issues to clarify requirements.
- Example of systems.
- The Linux aspect.
- **Open discussion**.

What is a R.T. system ?

- It is all about response time to external events.
- Missing (a single) time limit means a functional failure of the system.
- Distinguish from inconvenient or inefficient behavior.
- Not all of the functionalities are time critical.

What is a R.T. system (cont.) ?

- Fastest cycles are not always good enough; fixed time interval may be needed.
- Start up / boot time might be critical as well.
- It is not about software, it is about **system**.
- You can’t study it in school …

What is “soft real-time” ?

- The system bears jittering but for limited cycles / within hysteresis.
- The functionality has to do with humans, so timing is not critical :
  - 24-25 Hz display are enough.
  - Key pressing response takes at least 1/10 second.
- Video is a great example.

What is an embedded system ?

- No direct human interface needed to operate.
- Special hardware / direct HW manipulation.
- Might be limited by resources like : RAM, ROM, FLASH, storage (if at all) CPU speed etc.
- Designed for defined missions, not general purpose.

Cost

- The system needs to be designed carefully. You can’t expect a human to forgive and reboot.
- Comprehensive and complex tests are needed.
- Tailor-made development makes it expensive.

The software aspect

- Must cover all branches and in a limited time & memory (no swap !).
- Helpful if portable and can be tested on ‘host’ (see later).
- ISR are essential.
- No dynamic allocations.

Developing language

- Must be deterministic (!Java).
- Efficiency is an issue (C++ alternatives).
- Control of caching (A.K.A. volatile) and direct address access.
- Safety : Ada vs. C.
- Available (is there a Cobol compiler for ARM processor) ?

Runtime libraries

- These are (precompiled) software components that come with the OS or compiling tools. They implement common functions like mathematical, IPC, I.O. etc.
- Like the rest of the software, these should be deterministic in time and memory, preemptable and reentrant.

Throughput vs. responsiveness

- Throughput has to do with (a lot of) data processed in minimal time.
- Responsiveness : Fixed time to process limited amount of data.
- These often contradict each other.
- IO operation and communication are classic cases where caching is widely used.

Characteristics of G.P./multi-user O.S.

- (Kernel) not preemptable.
- Designed for throughput.
- Does justice in processes CPU time.
- Designed for massive IO.
- Communication oriented applications are classic cases where caching is widely used.
- Mask interrupts in critical sections.

R.T. & embedded system needs

- Clocks and timers.
- A debugging agent (unless J-tag ICE is used).
- Cross development tools ?
- Console ? Shell ?
- ROM / flash / NVRAM etc.
- Experts in both software and hardware.

Solution 1 – no O.S. !

- No overhead or context switch latency.
- No royalty or developer seat fees.
- Direct access to all board and CPU resources.
- Fast power up.
- Full control over every piece of code.
- Simple code design.

Solution 1 – no O.S. ?

- Where do the real-time libraries come from ? What about the debugger ?
- Single task / thread / process.
- What about drivers ? Communication ?
- What if two distinguish software components should share resources ?

Solution 2 – basic kernel/scheduler

- Was home made two decades ago.
- Simple, small, fast, can be part of the code, cheap ?
- Too simple, no MMU, not portable.
- Easy to test and certify.
- Was the only thing available.
- No drivers.

Example – W.R.S. VxWorks

- A very common and mature “RTOS”. Actually it is a highly developed kernel (except version A.E.).
- Has an enormous number of libraries, drivers, tools and documentation.
- Modular – small kernel + needed libraries.
- No MMU (or partial, with some ‘plugins’).
- Partial POSIX compliant.

Uncertainty principle

- While debugging code we change its behavior. Especially, scheduling synchronizations.
- printf is prohibited. Use puts.
- Printing to a serial console might be problematic. Magically, the CPU is free !

Linux and the kernel

- No nested interrupts or priorities.
- Not preemptable (until recently).
- Mask interrupts for critical sections.
- Optimized for average case.
- Depend on files systems.
- The kernel is big.

Why Linux

- Modular and scalable.
- Reliable, safe and stable.
- GPL'd and portable.
- Many drivers are available.
- Partially POSIX compatible.
- Many tools, libraries and apps' are available.
- Host simulation is easy.

Linux solution I

- Activate RT task at interrupt layer.
- Design the rest of the application to minimize interrupts-off periods.

Linux solution II

- Activate a modified kernel with an underlying layer of RT kernel.
- The Linux kernel will be the idle task of the RT one.
- Interrupt layer and mask are simulated by the underlying kernel.
- IPC services are added.
- Int. are forwarded to the regular kernel.

Linux solution III

- Modify the kernel.
- Give a real real-time solution, but with response/latencies worse than native RT OS's.
- Native development environment !
- Source and binary compatible to common Linux applications.

Resources, links and bibliographic

- Notes by Avi Noy, instructor at Haifa university, http://research.haifa.ac.il/~avinoy/
- http://www.rtlinux.org
- http://www.lynuxworks.com
- http://www.embedded.com
- http://www.mvista.com/
- http://www.ittc.ukans.edu/kurt/
- http://www.lynuxworks.com/