Things to consider with embedded kernel development

Presentation by Tony Lindgren <tony@atomide.com>
at Bossa Conference March 13th 2007

This presentation covers some experiences on:

- Software development strategy for embedded devices
- Embedded platform selection criteria
- Debugging kernel on embedded devices
- Power management for embedded devices

This presentation is available in MagicPoint and .pdf format at:
http://www.muru.com/linux/embedded/

Summary of Tony Lindgren and Linux kernel

- Currently mostly working on Linux kernel for TI OMAP series of processors 1510, 1610, 1710, 2420...
- Worked on initial implementations of dynamic tick or dyntick for Linux for ARM and x86
- Ported Linux for Psion Windermere series of palmtops with other Linux developers (5mx, Revo, Ericsson MC218, Diamond Mako)
- Hacked on various Linux hardware for fun, such as power management for AMD-76x SMP chipsets, fast serial port support for w83627hf chip, PCMCIA socket controller for the Apple PowerBook 1400 and 5300 models, x86_64 VIA chipset IOAPIC, etc. Basically whatever was needed to get own hardware working...

Software development strategy for embedded devices

- There is a hard way of doing things
- There is an easy way of doing things

The hard way of embedded development

- Select latest processor
- Select latest devices
- Do everything from scratch, including processor support
- Do all the device drivers from scratch
- Leave out all debugging ports to save on manufacturing costs
- Maybe leave out the MMU for better latency
- Do this over and over again every year to stay current
- In general, only large companies can afford to do this
- Will potentially provide an advantage in hardware size or speed to competitors

The easy way of embedded development

- Select something that already works at least partially with Linux
- Do initial kernel work on a board with serial and NFS root over Ethernet
- Stay away initially from systems without MMU
- Do drivers as modules over NFS root
- Do custom userspace software first on Linux PC
- Then just shrink down everything into a more limited board

Embedded platform selection criteria

- There are a lot of options for hardware that supports Linux
- We want to follow the easy way of doing embedded development

Obvious things to consider

- Price
  - Decent embedded development boards can be found for US$300
  - Or just use an old PC if possible (Usually bad power consumption)
- Memory
  - 4MB is enough for router, 16MB is enough for a PDA
- Storage size
  - 2MB enough for a router with bootloader, kernel and rootfs
  - 2MB enough for a bootloader and kernel
  - 4MB enough for a bootloader, kernel and initrd
  - 32MB enough to run a minimal Debian distro
- Speed
  - Almost anything will do if you don’t need graphics
  - 32MHz ARM 710T for a PDA (Psion 5mx)
  - Web browsing and multimedia will have much higher needs
  - 200MHz+ for devices with a browser and multimedia (N770, N800)

More obvious things to consider

- Power management
  - Many systems provide only 1 day usage time on batteries
  - One week idle times can be done on batteries (N770, N800)
  - Dyntick + sleep-while-idle can be used for longer idle times
  - Doing good power management for a device can take a lot of time
- Connectivity
  - Serial, Ethernet, WLAN, USB
  - CompactFlash/CardBus/PCMCIA/mini-PCI usually most flexible for development
- Storage format (NOR, NAND, MMC/SD, CompactFlash...)
  - NOR is easiest if you need to write a bootloader
  - CompactFlash support is good in Linux, takes more physical space than MMC/SD
  - NAND support is available in Linux
  - MMC/SD support is available in Linux
  - No SDIO support currently (A closed standard)

Less obvious things to consider

- Try to avoid certain situations
  - Oops, I should have thought of that earlier...
  - Now we have this hardware and we’re stuck with it...
  - How can I connect that device to it...

Access to documentation

- Lot of hardware does not have any public documentation available
- Lot of hardware does have good public documentation!
- Lot of documentation is very bad...
- Try to select hardware with public documentation
- Chances are other people are working on the hardware too
- Other developers can send patches easily
- Usually a search on the web reveals if documentation is available

Debugging ports

- Critical on bringing up new hardware
- Next chapter will deal with kernel debugging a lot more...

MMU (Memory Management Unit)

- Don’t leave it out if possible!
- If things don’t work with MMU, they certainly won’t work without MMU
- Bugs on systems without MMU make things hard to debug

Available GPIOs

- GPIO = General Purpose Input Output
- Can be used as output to power on/off external devices
- Can be used as input for device interrupts
- If you want to control custom hardware, GPIOs are a must
- Some boards offer GPIOs on I2C chips
- Linux I2C support not currently very friendly for embedded systems
- Interrupt handling for I2C GPIO chips can have long latencies as they are run from a timer

Physical things to consider

- Available pins
  - Many SOCs (System On Chip) have a limited number of physical pins
  - Some device combination are limited on SOCs because of lack of pins
  - Debugging pin multiplexing can take a lot of time during driver development
- Package format
  - Many packages are nowadays too small for soldering anything to
  - Can make working on some products nearly impossible
- Environment tolerance
  - Depending on the project, temperature tolerance can be important

Temperature

- Hey, with right hardware you can do a lot with just a shell script!
- NetCam at Williams Field, Antarctica, http://thistle.org/

What I would select if I had a choice

- CPU that's supported well under Linux
- CPU that has public documentation
- MMU on the CPU
- At least 32MB of RAM
- At least 32MB of NOR flash or supported NAND flash
- Ethernet connector for NFS root
- Open source bootloader that supports Ethernet booting
- Enough GPIO pins for project
- Power management support depending on project
- Optionally a JTAG connector
- Connectors for custom hardware
- Optional supported USB gadget
- Optional supported USB OHCI host

Cool systems for ARM Linux development

OSK5912 for custom hardware projects
http://omap.spectrumdigital.com/osk5912/

Thecus N2100 for home server and ARM userspace software development
http://www.cyrius.com/debian/iop/n2100/

Often the platform selects you...

- Bought something thinking it runs Linux (Psion 5mx)
- Want to make old computer usable for Mom (Apple Powerbook 5300c)
- Help a friend who is building Antarctic weather stations (NetCam)
- End up working with some company using particular chip (OMAP)

Debugging kernel on embedded devices

Debugging ports

- Serial port + Ethernet usually best combination for development as it allows NFS root (NFS root allows easy insmod/rmmod of kernel modules)
- UART only can do, but makes development slow (Maybe use ppp + ssh)
- JTAG console, also allows loading of kernel images to memory
- USB Ethernet not very handy for low-level debugging
- Custom ports, framebuffer, STI

Serial port

- Most boards have it, and it’s often standard 8250 compatible
- Some boards have custom UARTs
- Usually easy to write a driver
- All time favorite for low level debugging

No serial port on your development board?

- Stay away from it, get something else!
- If you have to use a board without serial port, maybe use a JTAG console
- Hey, just write a minimal debug driver for the framebuffer, all you have to know is the virtual address of the LCD...
- Oh, actually I’m still trying to figure out how I can boot Linux from whateverotheroperatingsystem...

What is JTAG?

- JTAG is acronym for Joint Test Action Group
- IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture
- More info on Wikipedia http://en.wikipedia.org/wiki/JTAG

BDI-2000

- Works with gdb
- Expensive for students/hobbyists

Lauterbach

- Closed source software, but works in Linux
- Expensive for students/hobbyists

Cheap alternatives

- Build it yourself, instructions can be found online for parallel port JTAGS
- Cheap USB JTAG alternatives that work on Linux?

Why is JTAG important?

- Some embedded boards don’t have a serial port :(
- You can load code to memory of the device, such as bootloader or Linux kernel
- And this means you can probably flash a bootloader, kernel and root file system
- You can step through early boot code on a new processor
- It is relatively easy to patch Linux to support a JTAG console, patches are available at least for ARM
- Note that JTAG pins are usually left out of final products

JTAG console

- Can be used like a serial console with some extra patches
- Requires a JTAG connector on board
- Requires a JTAG device (USB, Ethernet, parallel)
- Can be very expensive for a hobbyist (US$ 2000 +)
- Handy for stepping through Linux boot process

Power management for embedded devices

Why is power management hard?

- Suspend and resume are not enough if you want to have the device run for a long time on batteries
- It is relatively easy to get ~1 day of battery life
- To get 10+ days of battery life, everything must be off when idle
- Power management typically has lots of hardware and board specific dependencies
- Implementing power management can take a lot of development time for the architecture and each driver

Power management implementation on OMAP

- 10+ days of uptime on battery
- 4+ hours of usage time with LCD and WLAN on
- Implemented with
  - Good hardware support for power management
  - Clock framework
  - Dyntick (Dynamic tick)
  - Sleep while idle
  - Drivers block retention when needed for latency reasons
- System hits retention with just 32KiHZ timer running in almost every idle

Idle method effect on battery life

- WIF (Wait For Interrupt) total 2+ hours of uptime
- Add hardware specific idle states for total of ~1 day
- Add sleep while idle total of many days
- Add dyntick for total 10+ days

What is still needed for power management

- Voltage framework to go with the clock framework
- Driver specific inactivity timers to suspend devices automatically
- Clean way of doing board specific frequency and voltage scaling

Summary

- Select easy hardware to work with initially
  (MMU + serial + NFS root over Ethernet)
- Do driver development over NFS root as modules
- Try to select hardware with public documentation so others can provide patches easily
- Develop custom software initially on Linux PC
- Once kernel and software are working, you can shrink down everything into a more limited board
- Implementing power management can take a lot of development time
- Long battery life is possible with Linux