U-Boot „Falcon“ Mode

Minimizing boot times using U-Boot "Falcon" mode

Stefano Babic / Wolfgang Denk

July 2012

Overview

- Requirements for Boot Loaders
- Frequently Asked For Optimizations: Boot Time
- Hardware Influence and Considerations
- Software Optimizations
- Changes Imposed by Recent Hardware
- SPL – a Little Gem for Multiple Use
- Example: Boot into Linux/Qt quickly
- Things to be done
- Questions...

Overview

- Requirements for Boot Loaders
- Frequently Asked For Optimizations: Boot Time
- Hardware Influence and Considerations
- Software Optimizations
- Changes Imposed by Recent Hardware
- SPL – a Little Gem for Multiple Use
- Example: Boot into Linux/Qt quickly
- Things to be done
- Questions...

Many Different Requirements

- **End User:**
  - almost never interacts with the boot loader at all
    - Boot OS as quickly as possible
- **Application Engineer:**
  - flexible environment for varying software configurations
    - Boot from any available storage devices
    - Easy software installation, reliable software updates
- **BSP Engineer:**
  - efficient development environment
    - Easy to port
    - Easy to extend
    - Easy to debug
- **Hardware Engineer:**
  - powerful test environment
    - Help with board bringup
    - Production tests
    - Service tools to diagnose hardware problems

Boot Time Optimization

- Time from Power-On to “Operational Mode”
- includes:

– Boot Loader	0.3 s
– OS Initialization	3 s
– Application Startup	30 s

- Focus here: Boot Loader
  - but remember Donald Knuth: “Premature Optimization”
  - see also: Röder/Zundel: “Linux FastBoot”
    http://www.denx.de/en/Documents/Presentations

General Optimization Rules

- Avoid anything you don't really need:

  Code is fastest if not executed at all!

  Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away.- Antoine de Saint-Exupery

  - Initialize hardware only if it is really used by U-Boot itself, and only then.
- Make it run fast
  - Caches on?   Burst Mode accesses enabled?
  - Fastest hardware? Maximum bus bandwidth?
  - Fixed configuration vs. probing / bus scans (USB, I2C, PCI, …)?

Other Optimizations

- Maximize resource utilization:
  - Don't busy-wait for long running operations (“Fire and Forget”)
  - Run several tasks in parallel
  - Bus bandwidth vs. CPU performance: compression ?
- Interpret Requirements Intelligently
  - Run tests at end of boot cycle, i. e. before power-down

Areas for Optimization

- Hardware
  - CPU Speed, Bus Bandwidth
  - Boot device (memory or storage?)
  - Boot method (execute user code or immutable boot ROM?)
- Software Design
  - Trade Security for Speed: switch of checksum tests
  - Trade Costs for Speed: use uncompressed images [may not help on fast CPUs]
- Implementation
  - Compute checksums while copying/uncompressing images
  - Avoid copy operations: make Linux accept ramdisk in NOR flash

Hardware Considerations

- Memory Devices: ROM, EEPROM, NOR Flash
  - CPU can directly address individual cells in some ramge of addresses
  - can directly provide code and data, allows XIP

  → limited capacity, expensive → fast, reliable

- Storage Devices: NAND Flash, SDCard, USB, ...
  - controller interface; data need to be read into memory buffer before they can be accessed
  - SoC limitations: only small buffer space available
  - Boot ROM limitations: read only first block (2 ... 128 KiB)

  → huge capacities, cheap → slow, unreliable

Hardware Considerations

- In the olden days:

  Reset → CPU starts executing boot loader in ROM at reset entry point → loads OS

  - Optimizations:
    - Fast CPU, fast ROM (usually NOR flash)
    - Maximize bus bandwidth (32 / 64 bit bus interface)
    - Enable caches
    - Enable Burst Mode Accesses

Hardware Considerations

- Classic U-Boot:
  - reset starts executing code in NOR flash
  - relocation to RAM because execution from RAM typically faster (NOR usually did not support burst mode accesses) and to allow flash programming (other solutions possible)
  - CPU performance versus bus bandwidth:
    - CPU faster: minimize data transfers, use compressed images
    - Bus faster: load uncompressed data

Hardware Considerations

- Today:

  Reset → CPU executes on-chip boot ROM (immutable) → loads X-Loader from Storage → loads U-Boot from Storage → loads OS

- Problems:
  - complicated multi-stage boot procedure
  - inherently slow
  - boot ROM cannot be changed
  - many limitations (buffer size, can read only one block of data)
  - X-Loader duplicates U-Boot, but code (especially drivers) cannot be shared

Software Optimizations

- Step 1: Get rid of X-Loader
  - create SPL (Secondary Program Loader) as separate boot stage that gets loaded and started by the boot ROM
  - small enough to meet hardware restrictions
  - common code base with “normal” U-Boot, shared drivers
  - flexible – allow to use all available hardware resources
  - obsoletes X-Loader, UBL, ...

→ single source, common code base is much better, but not inherently faster

Software Optimizations

- Step 2: Make more flexible
  - SPL basic asks:

    initialize RAM, load U-Boot from storage, start it
  - generalize:

    initialize RAM, load *“some image”* from storage, start it
  - implement a way to select which image to boot;

    for example, test a GPIO (switch, button, jumper)

→ more flexible, but how is this faster?

Optimizing Boot Time using SPL

- 1st image = standard U-Boot
  - All features available as usual
  - suitable for development, production, service, maintenance, software updates, ...

  “Development Mode”

- 2nd image = Linux Kernel
  - When all you want to do is booting an OS, then do not load and run the full U-Boot at all !
  - saves time to load (and run) several 100 KiB of code !

  “Production Mode”

Optimizing Boot Time using SPL

- 1st image = standard U-Boot
  - All features available as usual
  - suitable for development, production, service, maintenance, software updates, ...

  “Development Mode”

- 2nd image = Linux Kernel
  - When all you want to do is booting an OS,
then do not load and run the full U-Boot at all !
  - saves time to load (and run) several 100 KiB of code !

  “Production Mode”

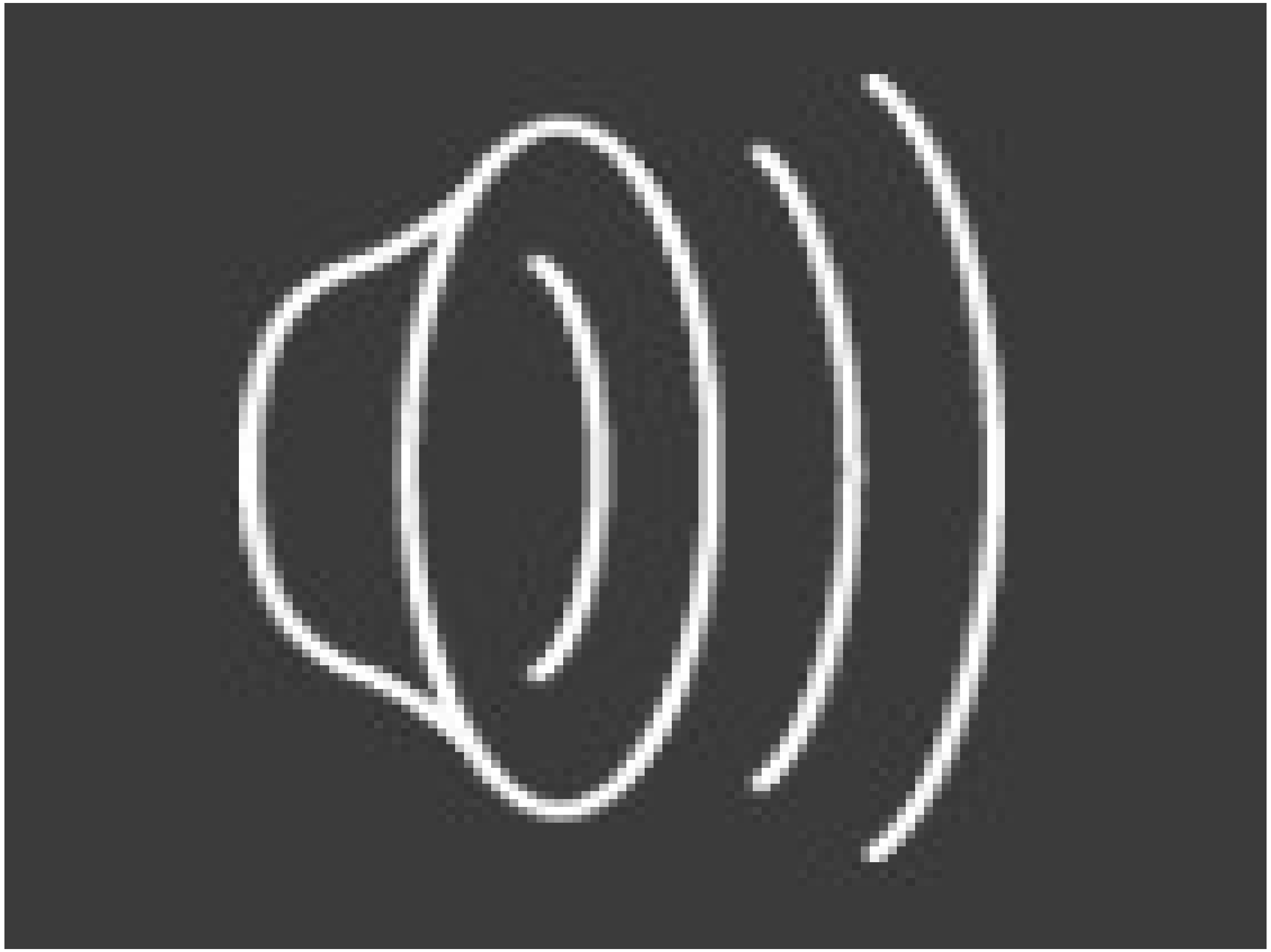
Test Case: Twister Board

- Test setup on “Twister Board”:
  - TI AM3517 CPU at 600 MHz
  - 256 MiB DDR2 RAM
  - 512 MiB NAND flash
- added jumper to select boot mode
- Direct boot of Linux Kernel
- Root FS = 24 MiB UBIFS in NAND (ELDK 5.1 QtE) with slide-show based on “fbi” as application

  See also:

  http://www.denx.de/wiki/U-Bootdoc/FalconBootTwister

“Falcon” Boot on AM3517 Twister Board

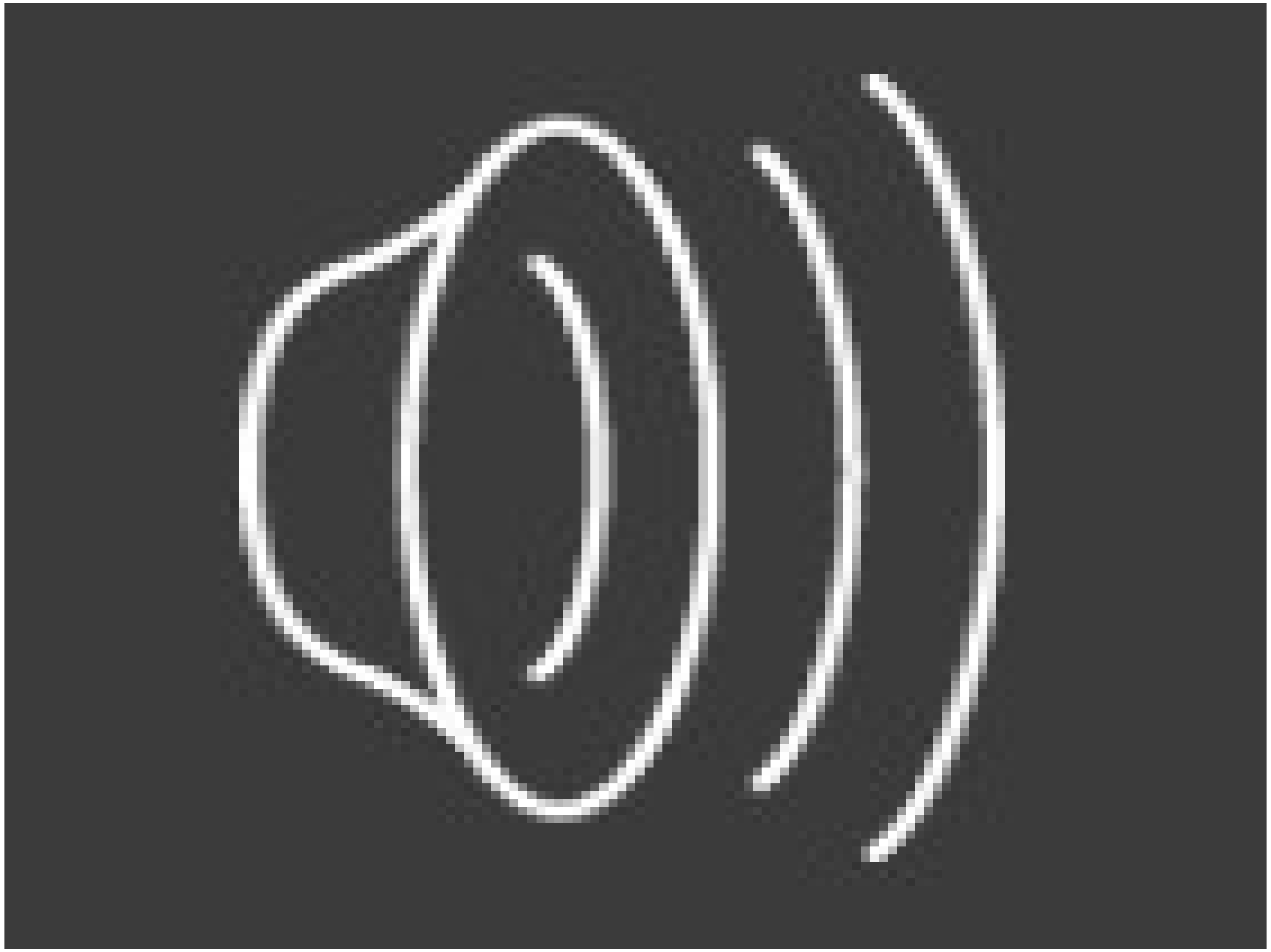


- ROM loads SPL from NAND
- GPIO (Jumper) used to select boot mode

=> “Falcon” mode (shown here):
- Linux kernel loaded from NAND
- UBIFS in NAND mounted as root file system
  -> fast

=> Service mode:
- U-Boot loaded from NAND
- U-Boot can boot Linux, or ...
  -> all features available

Slow Motion x10



Seconds after Video Start:

3.00 = +0.00 Power on
(see red LED)

3.44 = +0.44 Backlight on

5.36 = +2.36 Linux Penguin

5.88 = +2.88 Penguin off

6.08 = +3.08 Qt App running

How does it work?

- All relevant parts of the code are already in mainline
- How can we handle boot arguments or device tree updates normally done in U-Boot ?

  → Setup of the system is in two stages:

  – Using normal U-Boot, we prepare a static ("frozen") parameter block image

  – In "Falcon Mode", the SPL just passes this parameter block to the payload
- So can I use this, too?

  → Yes, if your board uses SPL.

Faster, Faster, Faster ...

- 3 seconds is pretty lame. Company [redacted] claims they support sub-second boot times.
- Yes, but did you try to do the same on your own board?
- Usually such results cannot be re-used:
  - Fine-tuned to specific hardware / boot device (NOR ?)
  - Not all needed code / know-how published
  - Synthetic use case that conflicts with real-life requirements
- This here is different:
  - Only standard technology used
  - Can be used as is in a real project
  - We encourage you to re-use all this !

Limits for Boot Times ?

- So can you do in less than 3 seconds or not?
- Yes, we can !
  - Code is not fully optimized yet: caches are switched off in SPL due to unknown problem (runs slower with caches on)
  - Use hardware that better supports booting fast
  - Optimize other areas, too (size of Linux kernel image, ...)
- But:
  - Each additional step will need an increasing amount of efforts
  - Further optimizations may strip functionality
  - May quickly become highly project-specific

Things to do

- Convert more boards to use SPL
- Spread the word about the new capabilities
- Fix remaining issues (why is booting slower with caches enabled?)
- Push the last few remaining bits upstream

What's in a Name?

Why the name “Falcon” mode?

- All obvious names were already in use: Fast-, Presto-, Quick-, Rapid-, Speed-, Swift-, Turbo-, … You-name-it-Boot
- ***Pergrin Falcon -*** The world's fastest animal:

  The pergrin falcon can fly/dive from

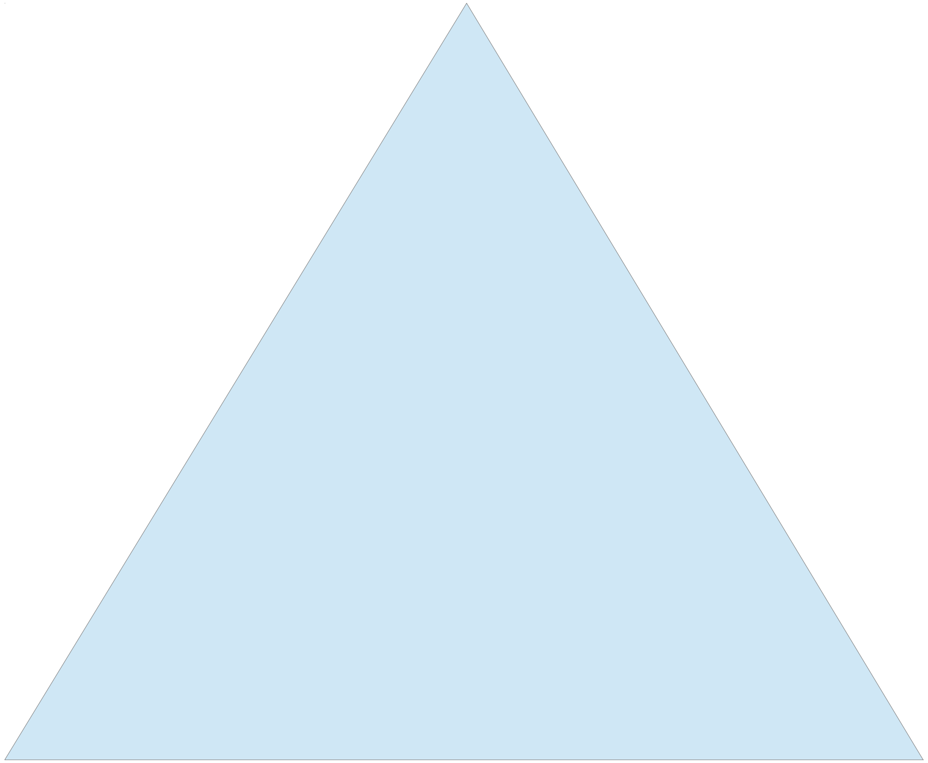
  up to 100 to 175 miles per hour (160...280 km/h).

  See http://wiki.answers.com/Q/What_is_the_fastest_animal_in_the_world
- So *“Falcon Boot”* is just our way to say: hey, we can boot pretty quickly...

Questions ...

- It's your turn now...

Triple Constraint



Good

Fast

Cheap

Pick any two!