Linux clock management framework

Siarhei Yermalayeu, Gert Vervoort, Shankar Mahadevan, Boudewijn Becking
Embedded Linux Conference
November 2 & 3, 2007
Siarhei.Yermalayeu@nxp.com
Outline

- Introduction to HW clock generation on Systems on Chips
- Clock Management SW Framework
- Conclusions
How are the system clocks generated?

Clock generation unit (CGU)
HW mechanisms for clock management

- Clock stopping
  - Disable/Enable every device clock (takes only a few cycles)
  - Base clock can be disconnected from a PLL (takes longer)
    - if allowed for all derived clocks
  - PLLs can be powered down (takes even longer)
    - if allowed for all derived clocks

- Clock scaling
  - Scaling down by using a divider (fast)
  - Scaling up/down by switching to a different PLL (fast unless high to slow clk)
  - Scaling up/down by reprogramming a PLL (takes longer for PLL locking)

- Clock stopping/scaling simple yet effective power saving mechanism
  - Clock trees can consume up to 50% of total IC power
Why SW framework for clock management?

Why SW Framework?
- Modern embedded systems have many (over 100) clocks
- Clocks are to be managed dynamically at run-time
- Structured approach
- Hide complexity of internal clock generation & clock interdependencies
- Reuse across different versions of a system-on-chip
- Reuse across different platforms

Additional requirements
- Clock management from multiple cores
- Use existing OS PM interfaces (if possible)
Linux support for clock management

- Why SW framework for Linux?
  - Linux 2.6 defines API for clock management
  - But no implementation is provided
  - There exist a number of example implementations (TI OMAP, ARM)

- The following API is defined:
  - \( id = \text{clk\_get(string\_name)} \)
  - \( \text{clk\_enable(id)} \)
  - \( \text{clk\_disable(id)} \)
  - \( \text{clk\_set\_rate(id, rate)} \)
  - \( \text{clk\_get\_rate(id)} \)
  - \( \text{clk\_put(id)} \)

- The API is provided in kernel space only (thus for drivers)
Clock management framework SW architecture

User-level Linux Applications

Kernel Device Drivers

/sysfs/clks dump clocks

User space

Kernel space

User space

Kernel space

_clk_get()
_clk_put()
_clk_enable()
_clk_disable()
_clk_set_rate()

Middle-level clock management SW layer
Linux Clock API implementation
(clock reference counting + dependency resolving)

Low-level driver API

Clock API

Linux Kernel

Low-level clock management SW layer
(low-level HW CGU access routines)

ARM SW

HW

HW Clock Generation Unit (CGU)
Clock framework services

- Low-level layer provides CGU HW access/programming routines
  - Platform-dependent layer

- Middle-level layer provides clock usage reference counting
  - on every clock enable/disable request

- Why reference counting?
  - Typically driver serializes requests to a device and the ref. counter = 1
  - If device (i.e. bus bridge) has no SW driver the ref. counter ≥ 1
  - If device is shared between CPUs the ref. counter ≥ 1
  - Safety precaution for future platforms
Clock framework services

- Middle-level layer provides dependency resolving on CGU level
- Clock dependencies on CGU level
  - Parent-child relationship between a clock and its all derivatives
- Why dependency resolving?
  - Enabling one output clock may require several CGU configuration steps
  - Each required CGU clock is automatically enabled and configured
  - Hides clock generation complexity
  - Only CGU output clocks are managed by the framework
    • Clock derivatives internal to devices are to be managed by device drivers
- Clock reference counting with dependency resolving enables automatic power optimization of clock generation
Clock framework internals

- Every clock is described by clk structure
- One type for all clocks
- Major fields/members:
  - Link to parent clock
  - Reference counter
  - Clock HW identifier
  - Clock type
  - Clock output frequency rate
- Internally a clock dependency graph is built
- Dependency links are dynamic and can be reconfigured at run-time
Initialization

- Clock configuration is statically defined by a clock dependency graph
- Each clock description has a frequency rate specified
  - Upon clock enabling a default frequency will be configured
- On HW reset default clock configuration is set up
  - Typically all clocks enabled and derived from an input clock
- During Linux boot clock framework is initialized and the specified clock configuration is programmed to CGU
- On resume from a standby mode a similar procedure but with the clock configuration just before standby
- When booted (resume) some of the clocks are configured by the bootrom and not by the clock framework, thus the framework should resynchronize
Frequency scaling

- Problem 1: Frequency scaling options are limited by the amount and availability of dividers and PLLs
- Problem 2: Dependencies of all kinds should be satisfied
- Solving these problems by the framework requires:
  - formal specification of many clock dependencies
  - Intelligent algorithm for finding the right clock configuration
  - Costs memory, cycles and power
- Our design choice: offload the framework from solving problems 1&2
  - provide a set of clock specific `set_rate()` functions
- Clock rate change is requested using frequency values
  - All dividers settings will be calculated by the framework
Interface to applications

- All clock management takes place in device drivers
  - No interface to clock management is exposed to applications

- Why?
  - Device clocking can be complex involving clocks from other devices
  - Device drivers hide clocking complexity from the applications
  - Device drivers provide HW independent interface
  - Many clock dependencies between various devices are handled automatically by device drivers
Even more dependencies

- By design clock framework covers CGU dependencies only
- How about clock dependencies between devices?
  - Functional (bus ↔ peripheral)
- Some devices have no SW driver, but still have to be clock managed
  - bus, bus bridges, L2 cache
- Workarounds:
  - Use a ‘dummy’ driver for clock management only
  - Another driver that requires such device can handle the clock management
Domain interface

- In case of many complex dependencies a concept of domain can be introduced

- Domain
  - Any combination of clocks grouped together by a certain rule
  - Treated as one unit
  - Domain definitions can have intersected clocks
  - Domains can be exposed to both drivers and applications

- Domain interface API is similar to clock API
  - `dmn_clk_enable(dmn_id)`
  - `dmn_clk_disable(dmn_id)`
  - `dmn_clk_set_rate(dmn_id, rate)`
  - `dmn_clk_get_rate(dmn_id)`
Clock framework architecture (with domains)

User-level Linux Applications

Kernel Device Drivers

Linux Kernel

High-level clock management SW layer (translation of domains to system clocks)

Middleware clock management SW layer
Linux Clock API implementation (clock reference counting + dependency resolving)

Low-level clock management SW layer (low-level HW CGU access routines)

Clock API

Low-level driver API

HW CGU IP

/udev/cgu
dmn_clk_on()
dmn_clk_off()
dmn_set_clk_rate()

dmn_clk_on()
dmn_clk_off()
dmn_set_clk_rate()

clk_get()
clk_put()
clk_enable()
clk_disable()
clk_set_rate()

/sysfs/clks
dump clocks

User space

Kernel space

User space

Kernel space

User space

Kernel space

User space

Kernel space
Multiple core clock management

- In multi-core system all cores may need clock management
- Design choice: forward all clock requests to a single core
  - Heterogeneous systems are assumed
- Con’s
  - Longer delay compared to direct HW control
  - Typical IPC latency: 10us BC, 100us WC
- Pro’s
  - Reliable operating in case a device is shared between the cores
  - Reuse of the framework (ref. counting, dependency solving, etc)
Clock framework architecture (multi-core)

- **DSP SW**
  - Clock API Wrapper
  - Interprocessor communication
  - clk_get()
  - clk_put()
  - clk_enable()
  - clk_disable()
  - clk_set_rate()

- **User-level Linux Applications**
  - /sysfs/clks dump clocks

- **Kernel Device Drivers**
  - Interprocessor communication
  - clk_get()
  - clk_put()
  - clk_enable()
  - clk_disable()
  - clk_set_rate()

- **User space**
  - Interprocessor communication
  - Clock API
  - Low-level driver API

- **Kernel space**
  - Clock API
  - Low-level clock management SW layer (low-level HW CGU access routines)

- **Middle-level clock management SW layer**
  - Linux Clock API implementation (clock reference counting + dependency resolving)

- **ARM SW**
  - Linux Kernel

- **HW**
  - HW Clock Generation Unit (CGU)

- **Interprocessor communication**

- **Low-level driver API**
Extensions to framework and clock API

- By default the clock framework saves power aggressively
  - Everything that can be switched off will be switched off

- Each request for a clock can potentially lead to PLL switch on/off
  - PLL locking takes considerable time (0.5 ms)

- Solution 1: Introduce the concept of time to the framework
  - Specify the expected time period for a clock being not used
  - Specify latencies for PLL power modes, etc
  - The framework shuts down CGU clocks and PLLs as far as time allows
  - Extension to Linux clock API is required: `clk_disable (id, time)`

- Solution 2: Defer stopping PLLs
  - If clock enable request arrives within a time-out interval drop an older PLL disable request
  - No changes to Linux clock API are required
Conclusions

- Linux clock API is the first step towards energy-optimized clock management for embedded systems
- Already today a clock management framework can be designed around the clock API only
- We need to work further towards higher level of generalization, such as:
  - Clock description by \textit{clk} structure
  - Intelligence for dependency graph traversing
- We need to agree which dependencies should be resolved by the framework, device drivers, applications
- We have to consider making clock management time aware
  - This would also be important for voltage control

Thanks for your attention!!!

Contact info: siarhei.yermalayeu@nxp.com