FreeBSD on high performance multi-core embedded PowerPC systems

Rafał Jaworowski
Semihalf, The FreeBSD Project
raj@{semihalf.com, freebsd.org}

Abstract

This paper describes low level design and implementation of the FreeBSD operating system port for the MPC8572 system-on-chip device, a high-end member of the Freescale PowerQUICC III family. It is a modern dual core system, compliant with Book-E definition of the PowerPC architecture, which features a number of peripherals integrated on a single silicon die.

The primary focus of this paper is describing how the multi-core operation was brought forward and full SMP capabilities achieved, but other major components developed in the course of this project, device drivers in particular, are also covered.

1 Introduction

PowerPC is an instruction-set architecture definition, originally developed by Apple, IBM and Motorola[1], based on IBM's POWER technology (RS/6000). It is now maintained by the Power.org group, a not for profit organization, which is shepherding the development of the so called *Power Architecture* (implemented as Cell, PowerPC and POWER processors), maintaining and releasing a unified instruction-set specification and reference platform definitions. All major Power Architecture silicon vendors participate in the Power.org.

Although PowerPC began as a modified subset of the POWER definition, contemporary processors from this line are fully compatible with PowerPC instruction set; all these variations are covered by the term Power Architecture.

With multiple vendors and supporters (AMCC, Freescale, IBM, Xilinx), Power Architecture is a widespread and viable processor design, used from embedded systems, through servers to supercomputers and gaming (all the most popular game consoles are based on the Power Architecture technology). Since this paper is mostly concerned about PowerPC, all further discussion refers to this aspect of the Power Architecture unless explicitly stated otherwise.

From the high level technical perspective PowerPC can be described by the following highlights:

- RISC-like (load-store) architecture
- superscalar
- 32- and 64-bit
- historically big endian (contemporary systems can switch between big and little endian order)

Book-E is the more recent variation of the traditional PowerPC architecture definition, supposed to suit particular needs of the embedded applications, but keeping binary compatibility with the traditional model at the user instruction set level (applications compatibility) [EREF, EREFRM].

There is a number of important differences between Book-E and the traditional spec-

[1] throughout this paper it is called *AIM*, or the *traditional PowerPC* definition

ifications below the application level, which greatly influence how the operating system must be implemented at the machine-specific layer. Among the most distinctive are the following differences [AN2490]:

- Book-E has more flexible approach to memory management (no longer the segmented mode known from AIM; instead there is a page-based memory system with multiple variable-sized pages, pure translation lookaside buffers (TLBs) mechanism. Dedicated registers and machine instructions for handling memory management unit (MMU). The differences in memory management area are by far the most prominent and making Book-E appear like a completely different CPU from this perspective.

- The exceptions model is extended when compared to AIM, additional classes of exceptions are introduced, accompanied by dedicated machine instructions for servicing them.

- Interrupt vector area in Book-E is no longer at a predefined location, handling routines can be more flexibly placed in memory.

It is important to note that Book-E specification leaves some lowest-level items to the discretion of particular vendor's implementation. For example, even though the MMU translation logic is precisely defined, the specification does not impose limits on the number of TLB entries, or details on how exactly to retrieve or store a given TLB entry contents.

The work described in this paper covers FreeBSD support for the MPC8572 system-on-chip (SOC) device, which is built around the dual-core *E500*, a Book-E compliant processor implementation by Freescale Semiconductor, Inc. [E500CORERM, EREF, EREFRM, MPC8572ERM]

At the time this development has begun, initial support for the E500 class machines was available in the public *FreeBSD/powerpc* repository as the outcome of a porting effort completed previously for the single-core E500 implementation [BSDCAN07]. This served as the main starting point for further work, but also some important parts of the common *FreeBSD/powerpc* code were leveraged in this undertaking.

The most important elements, which were used as a base for further extensions and rework, include:

- Booting environment, including firmware support library for PowerPC.

- Kernel *locore* initialization routines.

- Low level MMU support layer (*pmap*[2] interface) for E500.

- On-chip peripheral devices model.

- Basic driver for the integrated Ethernet controller.

- Preliminary support for inter-processor interrupts (IPI).

2 Porting

The outline of porting the FreeBSD operating system to a new platform includes the following major steps that have to be completed:

- Build environment. It is the first elementary requirement before any real work in the kernel can actually happen: we need to be able to produce proper machine code for the target platform. This may require changes to the compiler, linker, assembler and other utilities, or even the complete port of the build tools for a given architecture, if not already supported.

- Initial, stripped-down kernel build. At first, we will be trying to run something very minimal and it is usually best to eliminate as much code as possible.

[2] machine-specific VM layer, originally derived from Mach implementation and used by all BSD family

- Bootstrapping. Once the minimal kernel is built, there has to be a way to load the image (ELF or pure binary) into memory and pass control to its entry point. In order to minimize dependencies and simplify development set-up, it is usually preferred at the beginning to skip using the FreeBSD last stage loader, and execute the kernel directly. Details of this process however are much dependent on the underlying firmware's specifics and should be considered in a particular context.
- Locore kernel path. Entry point to the kernel, low level routines for CPU initialization have to be crafted in assembly language of the given machine.
- Rudimentary operation of the system. This covers, but is not limited to, getting to work the following elements: console, VM/pmap, time counter, exceptions/interrupts handling, in-kernel debugger (DDB/KDB). Only equipped with such basic instrumentation can we move on towards booting the kernel all the way up through machine-independent layers, until the root file system is mounted and *init* process executed, to finally reach a single user shell prompt.

It should be apparent the above is not a complete or exhaustive list, but only a bird's eye view on the overall scope of work involved into porting the kernel to a new platform.

The porting effort foremost concern is hardware-dependent layer, so using all available debugging facilities is highly recommended, specially the hardware assisted techniques (JTAG tools) are of particular help during early development phases, when no console output is available.

Subsections that follow discuss more details of selected porting stages in the course of the MPC8572 system bring-up.

2.1 Baseline code

The starting point for this development was publicly available source code of the FreeBSD 8-CURRENT branch as of March 2008 timeframe. During the porting duration it was a regular practice to re-sync with up-to-date baseline: the porting process can take longer time to complete, and CURRENT is the development branch, dynamically changing, so it is best not to let the porting work in progress diverge too far from the baseline.

2.2 Build environment

In case of the E500 machines, existing build environment could be fully leveraged. The tool chain bundled with the base FreeBSD (GNU binutils 2.15 and gcc 4.2.1) has had the ability to produce E500 specific machine code for some time already; these tools were previously used with success, so no specific extensions or changes were required in the build tools area for the MPC8572 port.

One note regarding binaries for the FreeBSD E500 support is that the traditional PowerPC ABI (Application Binary Interface) is used [PPCABI], and not the EABI (Embedded ABI), whose features and modifications of the basic model were not particularly required or beneficial for this development.

2.3 Bootstrap

The primary startup environment for the MPC8572 system is U-Boot firmware, an open source boot loader, widely adopted in embedded applications, specially on PowerPC-based systems. U-Boot's role is initialization of the hardware and providing minimal conditions, so that loading an operating system kernel into main memory and passing control to it is possible. U-Boot firmware development was not strict part of this porting effort: it was delivered by the hardware vendor and only slightly adjusted.

FreeBSD regular bootstrap process comprises a variable number of stages, depending on a given platform, but overall there is some form of initial execution performed by firmware (like BIOS or U-Boot), which in turn retrieves and runs some FreeBSD-specific boot code (early stage boot blocks, last stage loader),

which loads and executes the kernel.

With U-Boot based embedded platforms, FreeBSD last stage boot loader is run on top of U-Boot as the so called *standalone application*, it brings in the ELF kernel file to memory and runs it.

The integration between the FreeBSD last stage boot loader and the underlying U-Boot firmware was also not strict part of this porting effort, as all elements[3] were developed previously during the single-core E500 porting project [BSDCAN07]. They proved stable and ready for reuse, with almost no work other than testing and verification.

2.4 Minimal kernel operation

Preliminary steps described so far would let us build and load a skeleton kernel i.e. it would be formally recognized as a kernel image, but it is only a *container*, for which the proper contents need to be provided. There is a great number of conditions that need to be fulfilled before the first tangible signs of its life can be observed, but among the most important are the following:

- Early kernel initialization routine.
- Exceptions and interrupts handling (in particular address translation exceptions).
- MMU support layer: pmap. For Book-E class machines this is a critical area, as the architecture definition does not allow operation of the system with MMU disabled, i.e. MMU is always turned on by design and cannot be switched off.
- Local bus access methods so that CPU is able to issue elementary read/write operations on the bus in appropriate byte order.
- DMA back end layer.
- Integrated peripherals abstraction and individual device drivers.

[3]the U-Boot side API and accompanying support library for the FreeBSD loader

The first three items from the above list are discussed in greater details in their respective, dedicated sections; other elements are summarized below.

For uniform[4] access to the local bus, FreeBSD kernel relies on the so called *bus space* layer, a concept originally introduced in NetBSD, which defines an API exporting bus accessing methods. The foremost consumer of this abstraction are device drivers. Each new architecture or variation of the CPU (if significantly different in this respect) has to provide its own machine-specific implementation of these methods. In case of E500, the already implemented bus space methods for *FreeBSD/powerpc* were used. An extension developed during this port was adding 64-bit width accessors, which were required for the integrated security engine driver to work.

Similar situation is with DMA handling, as there is an abstraction layer called *bus dma*, also derived from NetBSD, and of analogous purpose that the bus space serves. It defines an abstract API for DMA-related operations for the rest of the kernel, with low level methods implemented according to a given architecture's specifics. In case of E500 and the MPC8572 system-on-chip, the existing *FreeBSD/powerpc* bus dma layer could be used without extensions. Like most of PowerPC systems, coherence between DMA, memory and data cache is maintained with hardware help, and there is no need for software assistance (explicit data syncing is not required).

Contemporary system-on-chip devices often integrate a fair number of peripherals in one chip, which is also true for the MPC8572 SOC. In order to be managed and recognized by the kernel, all integrated peripherals need to be modeled into an abstract tree representation, according to the FreeBSD *newbus* paradigm, an object-oriented description of devices hierarchy and dependencies. For this port an existing *ocpbus*[5] code was used as a starting point, and slightly extended. The ocpbus

[4]as seen from higher levels of the kernel

[5]on-chip peripherals bus

driver manages assignment of the built-in hardware resources (like the memory-mapped registers ranges, interrupts) in situations where no heavy weight firmware (like Open Firmware or EFI) is present. In case of U-Boot, the environment is simplistic and there is no feedback from firmware regarding devices' resources assignments available to the kernel, and therefore it needs to do its own management.

2.5 Portability summary

FreeBSD in general appears as a portable software. Thanks to the layered approach and abstractions (bus space, bus dma, newbus hierarchy, pmap), the machine-specific elements are mostly well contained. From this perspective porting work can be planned and then driven by tracking the completion of these interfaces' implementation.

A relatively high degree of reuse of the existing Book-E work was possible, which is natural as the MPC8572 system described in this paper is a more powerful successor within the family, with essentially the same underlying architecture definition and sharing most peripheral devices. Notwithstanding the hardware similarities, it should be noted the code from previous port proved scalable and well designed. Given the differences between older and newer chips, the code stood a good base from the very beginning: with only minor updates and corrections was the existing E500 kernel code able to start executing on the first core of the MPC8572. Much more work was required to bring multiprocessing operation, and this is subject of closer analysis that follows.

3 Multi-core operation bring-up

Before getting to the detailed discussion on bringing full multi-core capabilities, we need to introduce some basic nomenclature.

First, there is the notion of symmetric multi-processor (SMP) architecture, in which all processing units in the system share the memory space and all run a single kernel image instance. Programs, including kernel, have to be explicitly made aware about SMP to take advantage of the parallelism offered by such system. One of alternative approaches is asymmetric multi-processor (AMP) architecture, where each processing unit runs its own instance of the kernel image and has dedicated (separate) memory. With AMP, individual CPUs can as well run completely different operating systems.

In the context of this work we are only concerned and interested in the SMP approach, other schemes like AMP and hybrid solutions are not considered or relevant for further discussion.

Within the SMP processors group there always is one to first start executing the kernel code and bring it to the state when all other CPUs are awakened and allowed to run. These are, respectively, the bootstrap processor (BSP) and application processors (AP). Besides BSP's special role during initialization and shutdown phases, the BSP and APs are equal from the consumer (applications) point of view: threads are scheduled on all CPUs in the same way, regardless of their BSP/AP character.

In the MPC8572 dual-core system, the BSP is typically *core0*, and AP is *core1*; this will be the convention used further on.

It is also important to note that each core is equipped with its own memory management unit (MMU) instance, own L1 caches and separate set of other internal resources. The compound, including the core itself is referred to as the *core complex*. Dual-core MPC8572 has therefore two core complexes.

3.1 Initialization

When the system is powered up, the CPU fetches its first instructions to execute from some well defined location. In modern embedded processors like the MPC8572, there's a number of options a designer can choose for the system to boot from, e.g. ROM (FLASH), PCI, I2C, but we will only consider the most typical i.e. FLASH. Note that Book-E class processors do not follow the traditional PowerPC behavior

upon reset exception: there is no reset vector as known from AIM, and the CPU just fetches an instruction from some established address upon reset event.

These out-of-reset steps are the primary responsibility of the code executed at power-up, i.e. the firmware. In general, only core0 of MPC8572 is initially active and executes firmware code. The assumption our SMP kernel takes is that APs are not activated by the firmware i.e. they will only be awakened by the BSP at appropriate time.

3.2 The way of the bootstrap processor

Boot loader running on the BSP puts kernel code and data into memory and passes control to the entry point. The initial routine that starts executing is hand crafted in PowerPC assembly language, and its purpose is low level initialization of the CPU. The code makes certain assumptions about preparation steps which have to be done by the underlying boot loader, before it passes control: memory starts at physical address 0, kernel is loaded at 16-MByte boundary etc. The outline of this kernel initialization code is the following (`sys/powerpc/booke/locore.S`):

- Enable machine-specific features in the CPU (set HID[6] registers).
- Initialize the MMU so that kernel is running with virtual addresses.
- Set up stack.
- Initialize exceptions vector offsets.
- Jump to `e500_init()`, machine-specific higher-level initialization.
- Jump to machine-independent kernel initialization routine, `mi_startup()`, which does not return.

One particular feature of the Book-E specification, with far fetching consequences, is that MMU is always enabled on such processors. In other words, there must be always a valid translation in the TLB for the code executed or data accessed. The TLB translation error exceptions cannot be masked and will always occur upon fetch, load or store operation at a non-translated address.

For this reason, the kernel initialization code which prepares MMU for further work, needs to be very careful with cleaning up undesired translations (left by the firmware): it must not invalidate an entry translating the initialization code itself. To achieve this, a special technique is deployed, with flipping address spaces and setting temporary entries, until the MMU is fully reinitialized, with low level kernel code remaining available (in the translation sense) all of the time.

After the machine-dependent initialization code is finished and the kernel is about to commence the machine-independent part, the CPU and kernel state can be summarized as follows:

- Kernel text, data, possibly debug symbols, internal structures (kernel page tables and others) are covered by TLB translations.
- SOC registers block for the on-chip peripherals is mapped in the virtual space and translated by TLB.
- All other TLB resources are cleared and not used.
- *Decrementer*[7] is configured, so the kernel can reliably count delays and do other time counting, periodic actions etc.
- L1 and L2 caches are enabled.

In case of a uniprocessor system, this mostly concludes the low level initialization of the kernel, but in multiple processor environments the remaining CPUs still need to initialize and become fully available. While traversing machine-independent boot sequence, the BSP reaches the last final stage when the FreeBSD kernel kicks the scheduler and goes up

[6] hardware-implementation dependent

[7] PowerPC internal time counter

into full operation. In SMP system, just before this very last stage APs are awakened.

3.3 Unleashing secondary processor

In general there can be a number of APs, but in the context of this development we only consider one AP (core1), besides the BSP (core0). Overall, AP set-up procedure is very similar to what the BSP does, although some kernel preparation which has already been completed by the BSP, can be taken for granted.

When MPC8572 powers up, individual cores can be active or non-active. The latter state is referred to as the *holdoff* mode, where the given core is prevented from booting. Decision whether a core is activated or not depends on the system configuration sampled at reset time [AN3542]. For this development it is assumed that only core0 is active (boots the system), and core1 is in holdoff mode until explicitly taken out of this state. Such is the default U-Boot behavior.

Another important aspect when considering system start-up is the *boot page* translation, which is strictly connected with how the E500 core starts after reset. It begins with fetching and execution of the last word[8] of the address space i.e. at effective address `0xFFFF_FFFC`. In order for this to succeed there needs to be a valid TLB translation for the page, in which this initially executed word is located. The default boot page translation after power-up is a 1:1 mapping of the last 4-KByte page of the address space i.e. effective addresses `0xFFFF_F000-0xFFFF_FFFF` translated to the same physical range. In typical scenario the first instruction to execute at `0xFFFF_FFFC` is a branch to the beginning of the boot page, where more initialization code within this 4-KByte area can be found. Figure 1 illustrates this concept.

The boot page topic was not brought forward at the BSP start-up discussion only because it is firmware responsibility to supply boot page when the core0 starts. The AP on the other hand remains in holdoff mode and its

[8] 32-bit width

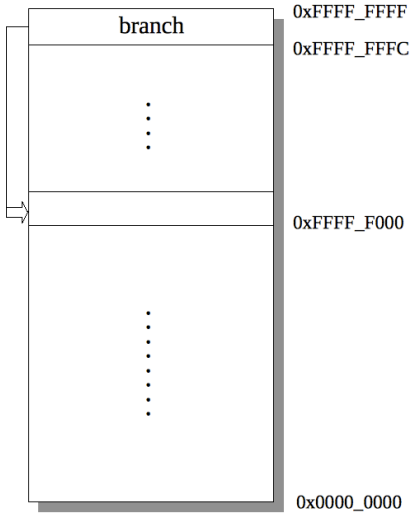

Figure 1: **Boot page**

initialization is supervised entirely by the kernel code (running on the BSP), so we need to take care about boot page contents for core1. Note there can only be one boot page translation in the system at a time (shared by all cores), but once the BSP is up and running it never needs or uses it again.

Based on this is the procedure of bringing the AP out of holdoff mode (steps are executed by the BSP):

- Adjust the boot page translation so that it points to a physical page in memory containing code we want the AP to execute.
- Make the AP run.

The code we "redirect" the AP to boot from has to be crafted in a similar way as the kernel entry point for the BSP. Indeed, its layout is quite similar (note each core complex has own instance of certain units and they need to be initialized separately by the AP):

- Enable machine-specific features in the CPU (set HID registers).
- Initialize the MMU so that kernel is running with virtual addresses.

- Set up stack.
- Initialize exceptions vector offsets.
- Assign per-CPU structures and resources.
- Jump to `pmap_bootstrap_ap()`, finalize MMU set-up.
- Call `cpudep_ap_bootstrap()`, machine-specific SMP initialization.
- Call `machdep_ap_bootstrap()`, machine-independent SMP initialization, which does not return.

At the time the AP performs this last action, its TLB contains some critical entries copied from the BSP settings: translations for kernel and the integrated peripherals registers range. This way both CPUs have the same view of the MPC8572 system.

The last stage of the AP bootstrap and final SMP initialization is driven by the BSP (scenario from the AP perspective):

- Busy wait until the "go" command arrives from the BSP.
- Initialize decrementer and time base registers with BSP-provided settings, so that both cores have the same view of time counting.
- Enable external interrupts.
- Go into scheduling, accept work.

This concludes initialization of the FreeBSD SMP kernel running on the MPC8572.

3.4 Hardware assistance for multiprocessing

When considering SMP kernel, there must be some basic hardware facilities available for the operating system to build its infrastructure on.

The primary requirement, even for uniprocessor systems, are *atomic* operations, and the architecture has to provide low level primitives, which are used to build more complex tools needed in the kernel. PowerPC has always offered elementary mechanisms for this purpose, and in case of this porting work the existing atomic operations implementation for the *FreeBSD/powerpc* were used [AN3441].

One of the more serious problems an SMP system designer faces, are data coherency issues, and there are a couple of ways they could be resolved [SCHIMMEL]. The best situation from the operating system perspective is when there is hardware-enforced coherency in place. This is the case of MPC8572, which implements mechanisms that help put off the data coherency maintenance burden from the kernel [AN3544].

The hardware *coherency module* allows on-chip caches (L1 and L2) to snoop on local bus (CCB[9]) for transfers affecting potentially cached locations. From the other end, bus masters (DMA engines) must be configured to advise the cache logic when modifying cacheable locations.

Additional mechanism which allows to keep both cores coherent with regards to main memory contents is the M-bit (memory coherence) in the page translation attributes. When set for a page, information is broadcast to the other processor whenever the local CPU modifies data in this page. The cache logic of the other processor can take appropriate action in case the affected contents were cached there.

Besides data coherency enforcement the following elements add to MPC8572 SMP hardware assistance facitlities:

- Cache invalidation instructions broadcast.
- TLB invalidation instructions broadcast.
- Integrated interrupt controller[10] with multi-processor support i.e. Inter Processor Interrupts (IPI).

[9] Core complex bus, internal bus connecting both cores and their resources (MMU, caches)

[10] OpenPIC compliant implementation

Note there is no hardware-enforced coherency for the instruction cache; in case cacheable memory locations containing executable code are altered, there needs to happen explicit synchronization of the instruction cache to avoid incoherency.

4 Memory management challenges

FreeBSD machine-specific layer handling the MMU is called *pmap*, which provides an API for the machine-indpendent VM layer and other kernel subsystems. For the Book-E class of processors, a new pmap module was developed at the time of the single-core MPC85xx port [BSDCAN07]. MMU in dual-core MPC8572 is the same design, with only minor quantitative differences like the number of TLB entries or additional page sizes supported.

E500 core complex consists of two MMU sub-modules (L1 and L2), but only L2 is controlled by software; L1 is managed entirely in hardware and hence pmap does not have to be aware about its existence. From the programming perspective the L2 MMU is built of two separate translation look-aside buffers:

- TLB0, set-associative, fixed 4-KByte page size, 256/512 entries (depending on the core revision).
- TLB1, fully-associative, pages of variable size (4-KByte–1-GByte, or 4-KByte–4-GByte, depending on the core revision), 16 entries.

The Book-E pmap implementation uses TLB1 for the *permanent* translations e.g. kernel code, data and other important areas (peripherals registers, decode windows) are covered by the TLB1 translations. As the name suggests, these entries do not change during system activity. For *dynamic* translations purposes (regular translations, subject to recycling, like the user processes' pages) the TLB0 is used.

Because the Book-E specification is very flexible about how the page tables are laid out in software, the system programmer is free to choose any model. The FreeBSD Book-E pmap implementation uses the most natural approach, a 2-level forward translation table, which is illustrated[11] at figure 2.

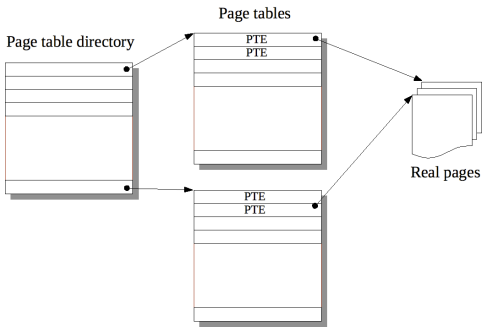

Figure 2: **Page tables layout**

The pmap module originally implemented for single-core MPC85xx systems, was refactored and extended before requirements of the SMP environment were met, and both cores of the MPC8572 could run safely in parallel with the shared code, data and other resources, without corruption. Among the most challenging problems faced during MP-safe conversion of the Book-E pmap were the following:

- Parallel and nested TLB miss exceptions and page faults (deadlock avoidance).
- TLB invalidations synchronization between CPUs.
- MP-safe page tables contents updating.

In order to satisfy the SMP reality the pmap module was extended and adapted, but it should be noted that its overall design was not changed, and proved flexible enough to accommodate to the new conditions. TLB0 handling was simplified and made more robust (redundant keeping of the TLB0 state was eliminated). TLB miss exceptions handlers were extended, so that certain steps while servicing the miss are truly atomic.

[11] PTE acronym means page table entry

For local core TLB invalidations a new performance-optimized (assembly) routine was introduced, and system-wide invalidations (broadcast within coherency domain), are protected by a dedicated spin lock, as there can be only one system-wide TLB invalidation broadcast on the bus at a time.

Page table management logic was also optimized and lock-protected, so that updating sequences are atomic across CPUs. In order for this to happen a dedicated TLB miss spin lock was introduced, which prevents servicing TLB miss exceptions by the other CPU, while page table contents are being updated by the current processor.

5 Beyond core kernel—integrated peripherals

Above basic kernel support a number of device drivers for the on-chip peripherals were developed in the course of this project. The FreeBSD kernel infrastructure for managing device drivers is an object-oriented framework (newbus), and all MPC8572 peripherals drivers described in this section are compliant with this model. Figure 3 illustrates the concept.

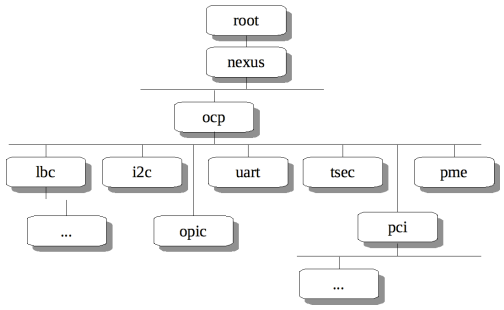

Figure 3: **Device drivers hierarchy**

5.1 Enhanced Three-Speed Ethernet Controller (eTSEC)

TSEC is the 1-Gbit Ethernet engine found in various Freescale parts and the MPC8572 has an enhanced version of the controller. An existing device driver, originally developed for the single-core MPC85xx port [BSDCAN07], was extended and adapted to make use of the advanced functionality offered by the newer hardware. Additional features of the driver introduced during this porting work:

- polling
- interrupt coalescing
- VLAN tagging
- hardware checksum calculation
- jumbo frames

Fine grained locking was also provided as a prerequisite for MP-safe operation.

5.2 Pattern Matching Engine (PME)

PME is a hardware accelerator for matching patterns, specified as regular expressions, in data going through the system bus. Among its main applications is network packets inspection (content filtering), which can be done in real-time and with high performance. The engine also includes a decompression module, so that inspected data can be unpacked on the fly if required.

The device driver for PME was written from scratch, with fine grained locking for MP-safe operation. It can be compiled as a kernel dynamic module.

5.3 Security Engine (SEC)

The device driver for the crypto accelerator was written from scratch. It is compliant with the open crypto (OCF) framework, and supports versions 2.0 and 3.0 of the engine. The following algorithms and schemes are currently supported:

- 3DES, AES, DES
- MD5, SHA1, SHA256, SHA384, SHA512.

The driver is fine grained locked for MP-safe operation and can be compiled as a kernel dynamic module.

5.4 PCI-Express bridge

The existing PCI driver, developed previously for the single-core MPC85xx, was extended to also recognize and set up the PCI-Express bridge found in MPC8572.

5.5 Integrated DMA Engine (IDMA)

This general purpose DMA engine can be used for offloading CPU when transferring data by different bus masters. Various methods are available for specification of the data (via descriptors or explicitly), and there is more than one mode of operation (single transfers, chained mode).

The driver for IDMA was written from scratch, only basic mode of operation (direct) is supported for the moment. As part of this development a sample user-space application, demonstrating basic IDMA operation was also written.

5.6 I^2C controller

Device driver for the I^2C integrated controller was written from scratch. As an add-on the the main controller driver, a helper *i2c* tool was also developed to help diagnose and inspect slave devices on the I^2C bus.

6 Future development

Even though FreeBSD for the MPC85xx systems is in a decent shape, there is a number of areas which would benefit from improvements and new development. Some more interesting topics are highlighted below:

- Currently a single virtual address space is used ($AS = 0$), where the kernel is mapped into the higher $1/4^{th}$ of the 32-bit address space, but the Book-E specification allows for more elaborate set-ups. For example, the kernel could reside in its own 32-bit address space, separate from the user process (which could then also utilize the full 4-GByte address space).

- The E500 core has 36-bit physical address space, but current FreeBSD support is oblivious to this potential. In order for the kernel to make us of it (very big RAM, more I/O devices etc.), there would be needed a feature somewhat equivalent to the PAE (Physical Address Extension) support, known from the *FreeBSD/i386*.

- MPC8572 features an integrated Table Lookup Unit (TLU), an engine accelerating complex table lookups with hardware assistance. Among applications that would benefit from the [missing] TLU driver are firewall (rules lookup), database and similar.

- The pattern matching (PME) driver has to be integrated with the FreeBSD firewall software (*ipfw*, *pf*), so that the packet content analysis capabilities are utilized by the kernel.

- eTSEC Ethernet controller has built-in support for the IEEE 1588 Precision Time Protocol (PTP), but the driver code needs to be extended in order to use it. This task could be generalized towards developing a generic PTP infrastructure for the FreeBSD kernel, to which capable network interfaces would plug in, and the PTP handling would be a generic code.

- The security engine (SEC) driver should be extended with the RC4 stream cipher and RNG support.

- The general purpose DMA engine driver is currently only able to demo a basic operation of the device, but cannot be used by any real consumers. A bigger project around this would be to bring up a generic DMA API to the FreeBSD kernel and userland, which would utilize such all-purpose DMA engines instead of CPU calling copy routines. This too, should be a scalable framework with machine-dependent backend plugged into and a generic layer.

7 Acknowledgments

I would like to thank the following people:

Alan L. Cox (The FreeBSD Project), for conversations on pmap interface in the SMP context.

Mark J. Douglas (Freescale), for all the help, support and assistance.

Marcel Moolenaar (The FreeBSD Project), for groundwork on SMP for the AIM PowerPC and IPI support implementation in the OpenPIC driver.

Grzegorz Bernacki, Rafał Czubak, Michał Hajduk, Jan Sięka, Piotr Zięcik (all Semihalf), for all the great work on this project.

Work on this paper was sponsored by Semihalf.

8 Availability

The code described in this paper is, or will soon be, available from the FreeBSD Project Subversion repository, 8-CURRENT (HEAD) branch. It is expected to be part of the FreeBSD 8.0-RELEASE.

References

[AN2490] Jerry Young, Freescale Semiconductor, Inc., *MPC603e and e500 Register Model Comparison*, AN2490/D Rev. 0, 7/2003

[AN2665] Freescale Semiconductor, Inc., *e500 Software Optimization Guide (eSOG)*, AN2665 Rev. 0, 04/2005

[AN3441] Freescale Semiconductor, Inc., *Coherency and Synchronization Requirements for PowerQUICC^{TM} III*, AN3441 Rev. 1, 12/2007

[AN3542] Ted Peters, Freescale Semiconductor, Inc., *SMP Boot Process for Dual E500 Cores*, AN3542 Rev. 0, 1/2008

[AN3544] Gary Segal and David Smith, Freescale Semiconductor, Inc., *PowerQUICC^{TM}Data Cache Coherency*, AN3554 Rev. 0, 12/2007

[BSDCAN07] Rafał Jaworowski, *Embedding FreeBSD/powerpc*, BSDCan 2007, Ottawa

[E500CORERM] Freescale Semiconductor, Inc., *PowerPC^{TM}e500 Core Family Reference Manual*, Rev. 1, 4/2005

[EREF] Freescale Semiconductor, Inc., *EREF: A Reference for Freescale Book E and the e500 Core*, Rev. 2.0, 01/2004

[EREFRM] Freescale Semiconductor, Inc., *EREF: A Programmer's Reference Manual for Freescale Embedded Processors*, Rev. 1, 12/2007

[MPC8572ERM] Freescale Semiconductor, Inc., *MPC8572E PowerQUICC^{TM} III Integrated Host Processor Family Reference Manual*, Rev. 2, 05/2008,

[PPCABI] Sun Microsystems and IBM, *System V Application Binary Interface. PowerPC Processor Supplement*, 09/1995

[SCHIMMEL] Curt Schimmel, *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*, Addison-Wesley Professional Computing Series, 1994

[VAHALIA] Uresh Vahalia, *UNIX Internals: The New Frontiers*, Prentice Hall, 1995