FreeBSD and NetBSD on APM86290 System on Chip

Zbigniew Bodek
zbb@semihalf.com

EuroBSDCon 2012, Warsaw

SEMIHALF
EMBEDDED SYSTEMS

Presentation outline

- Introduction
  - Market requirements
- Hardware overview
  - Features summary
  - Message Passing Architecture
- Porting
- Testing and debugging
- Current state and future work

Introduction

- What is an embedded system?
- Market requirements
  - Hardware
    - Low energy consumption
    - More packet processing power
    - Extra features:
      - Packet classification
      - Security extensions
  - Software
    - Time-to-market
    - Reliability
    - License and availability
    - Support for the chip's extra features

The APM86290

- Incorporates two PowerPC 465 processors in the single package
  - Book-E compliant
- Number of peripherals integrated in the chip, including:

  Gigabit Ethernet
  PCIe
  NAND
  SATA
  USB
  I2C

- On-chip processors are assisted by a rich set of configurable hardware accelerators focused on:
  - Packet classification
  - Scheduling
  - Packet/data manipulation
  - Security extensions

Message Passing Architecture

Message Passing Architecture

- Queue Manager (QM)
  - Allows the most efficient moving of data and packets between the processors and integrated peripherals
  - Communication interface offloads software from the routing of the packets and transaction synchronization.
  - Can be used to reduce communication overhead between software and hardware
- Queue Manager Interface (QMI)
  - Located in each subsystem that can use QM
  - Monitors the queue and prefetch buffers' status
  - Determines what action the subsystem should take for a certain queue status

Message Passing Architecture

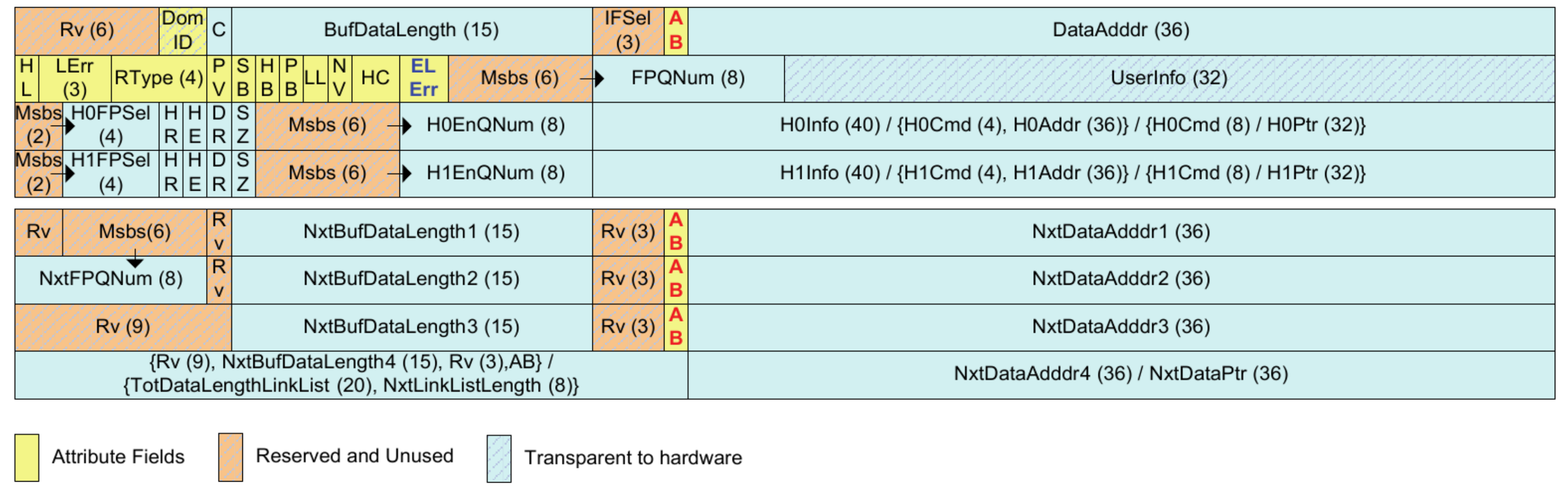
- Data transfers can be organized in queues
- QM allows systems nodes to communicate with each other through the preprogrammed queuing points
- The mechanism distinguishes three main abstractions:
  1) Queue
  2) Message
  3) Buffer

Queue

- Queues are organized as circular buffers and stored off-chip (in DRAM)
- The contents of a queue are prefetched on chip as needed
- Queue state is maintained on-chip for each queue
    - Pointer to head and tail
    - Occupancy level
    - Other parameters
- Queue configuration modes
    - Free Pool
    - Working Queue

Message

- Messages contain information about the corresponding buffers
- Main types:
  - Standard – 32 KB
  - Expanded – 64 KB

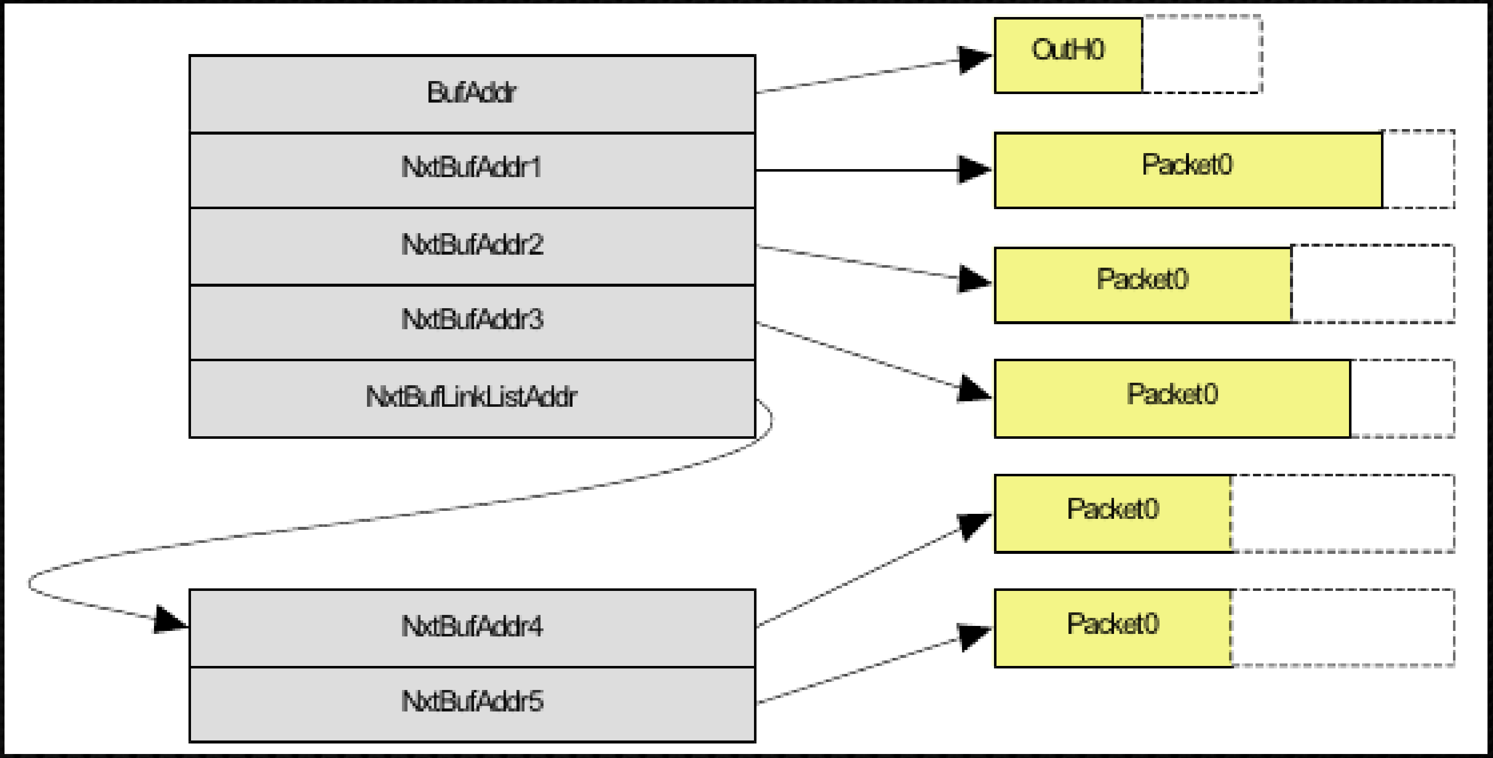


Rv (6)	Dom ID	C	BufDataLength (15)	IFSel (3)	AB	DataAdddr (36)							
HL	LErr (3)	RType (4)	PV	SB	HB	PB	LL	NV	HC	EL Err	Msbs (6) →	FPQNum (8)	UserInfo (32)
Msbs (2) →	H0FPSel (4)	HR	HE	DR	SZ	Msbs (6) →	H0EnQNum (8)	H0Info (40) / {H0Cmd (4), H0Addr (36)} / {H0Cmd (8) / H0Ptr (32)}					
Msbs (2) →	H1FPSel (4)	HR	HE	DR	SZ	Msbs (6) →	H1EnQNum (8)	H1Info (40) / {H1Cmd (4), H1Addr (36)} / {H1Cmd (8) / H1Ptr (32)}					
Rv	Msbs(6) ↓	Rv	NxtBufDataLength1 (15)	Rv (3)	AB	NxtDataAdddr1 (36)							
NxtFPQNum (8)	Rv	NxtBufDataLength2 (15)	Rv (3)	AB	NxtDataAdddr2 (36)								
Rv (9)	NxtBufDataLength3 (15)	Rv (3)	AB	NxtDataAdddr3 (36)									
{Rv (9), NxtBufDataLength4 (15), Rv (3),AB} / {TotDataLengthLinkList (20), NxtLinkListLength (8)}	NxtDataAdddr4 (36) / NxtDataPtr (36)												

🟨 Attribute Fields 🟧 Reserved and Unused ▨ Transparent to hardware

Message contents (Source: APM86290 User's Manual)

Message



Expanded Message usage (Source: APM86290 User's Manual)

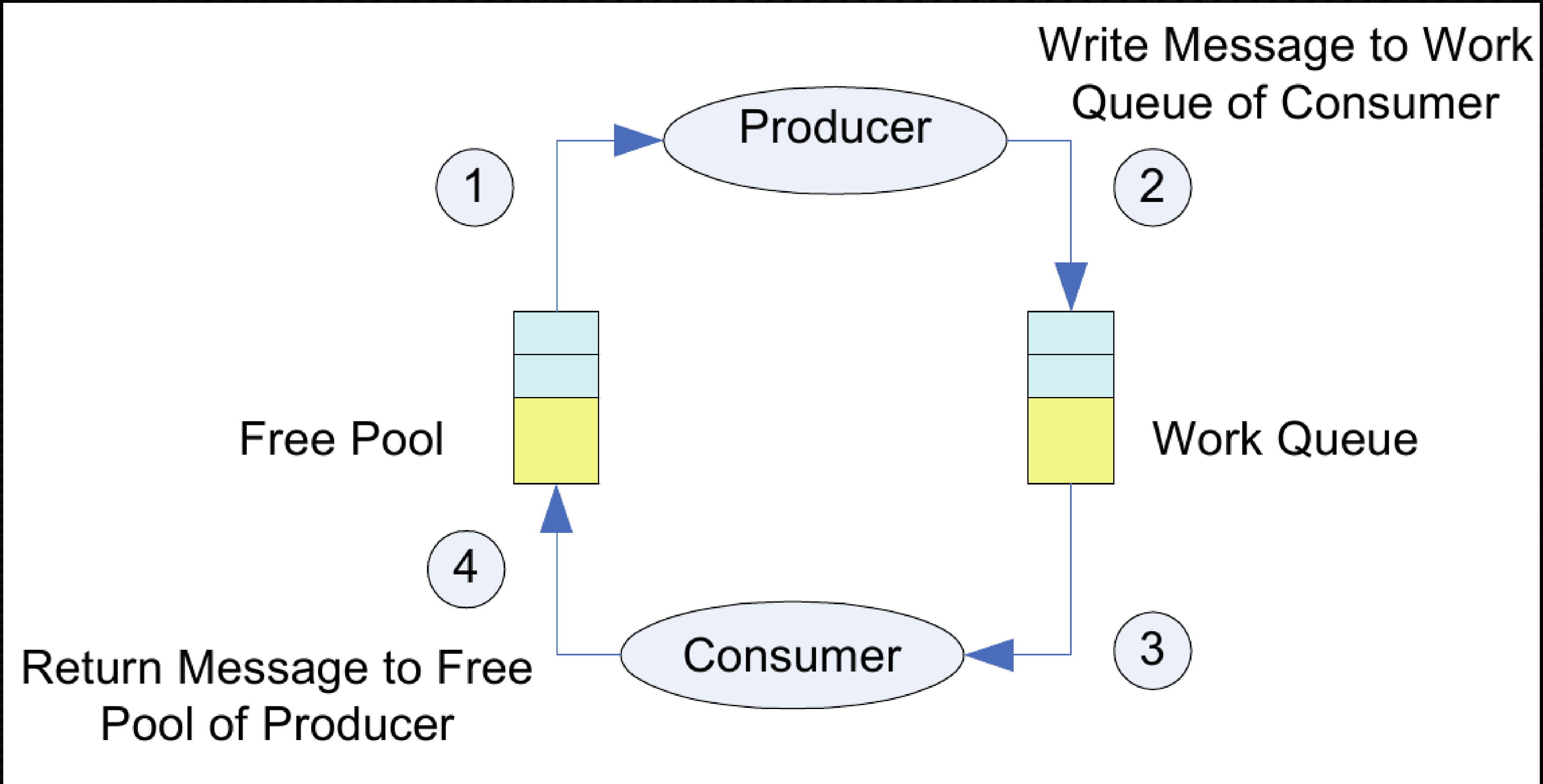
Buffer

- A fixed size memory location that is used to store information such as packet data
- Is kept outside of the chip – in DRAM
- Messages in the Working Queue are assigned to the corresponding buffers in the Main Memory
    - One-to-one assignment

Queue usage models

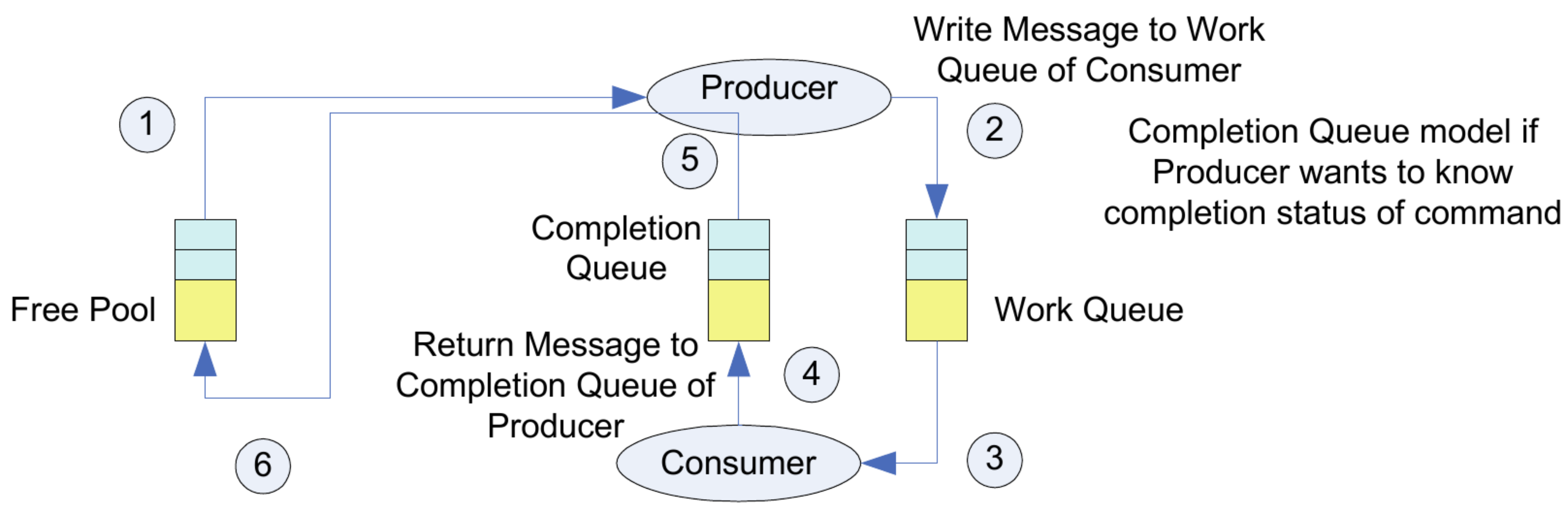
- Basically there are two possible usage models:
  1) One Free Pool and one Working Queue
  2) One Free Pool, one Working Queue and an additional Completion Queue
     - Used when the producer wants to know the completion status of the command

Queue usage model 1


Write Message to Work Queue of Consumer
Producer
1
2
Free Pool
Work Queue
4
3
Consumer
Return Message to Free Pool of Producer

Queue usage model 1 (Source: APM86290 User's Manual)

Queue usage model 2


Write Message to Work Queue of Consumer
Producer
1
2
5
Completion Queue model if Producer wants to know completion status of command
Completion Queue
Free Pool
Work Queue
Return Message to Completion Queue of Producer
4
6
Consumer
3

Queue usage model 2 (Source: APM86290 User's Manual)

Porting

- The general phases of the porting:
  1) Baseline code selection
  2) Cross-build environment preparation
  3) System bootstrap
  4) Early kernel initialization in `locore.s`
  5) Platform initialization
  6) Low-level memory management support
  7) Device drivers along with support for chip's special features
  8) Testing and debugging

Porting - baseline

FreeBSD® 8.1 / PPC460EX → 9.0 / APM86290

NetBSD® 5.99 / MPC85XX → 5.99 / APM86290

Porting - `locore.s`

- First code executed in the kernel
- Assumptions:
  - Basic SoC initialization is done by the firmware (U-boot)
  - Initial mappings are present in the TLB
- Written in the assembly language
  - Capability to be executed from any place
- Goals to achieve:
  1) Remap the kernel in virtual space
  2) Setup temporary stack

Porting - `locore.S`

FreeBSD

- Hook up to the existing locore.S (Book-E)
- Set up the exception vector regs.
- Remap the kernel
  - Create the temporary mapping and switch to it.
  - Create final kernel mapping and switch to it.
  - Invalidate the rest (cut off from u-boot translations)
- Set up stack and go to platform init.

NetBSD

- New start code for each platform (in sys/arch/evbppc/)
- Remap the kernel
  - Create the temporary mapping and switch to it.
  - Create final kernel mapping and switch to it.
  - Invalidate the rest (cut off from u-boot translations)
- Go to the generic locore.S

Poring - platform initialization

- C code
- Main goals to achieve:
  1) Create static mapping for the SoC registers
  2) CPUs initialization (timers, per-cpu structures, caches, etc.)
  3) Message buffer and console initialization
  4) Virtual memory subsystem bootstrap

Porting - platform initialization

FreeBSD®

- Hook up to the existing machdep.c (Book-E)
- Extract the common part for Book-E and platform dependent machdep.c
- Map SoC registers
- Apply minor changes to UART & set up the console
- Set up FDT framework

NetBSD®

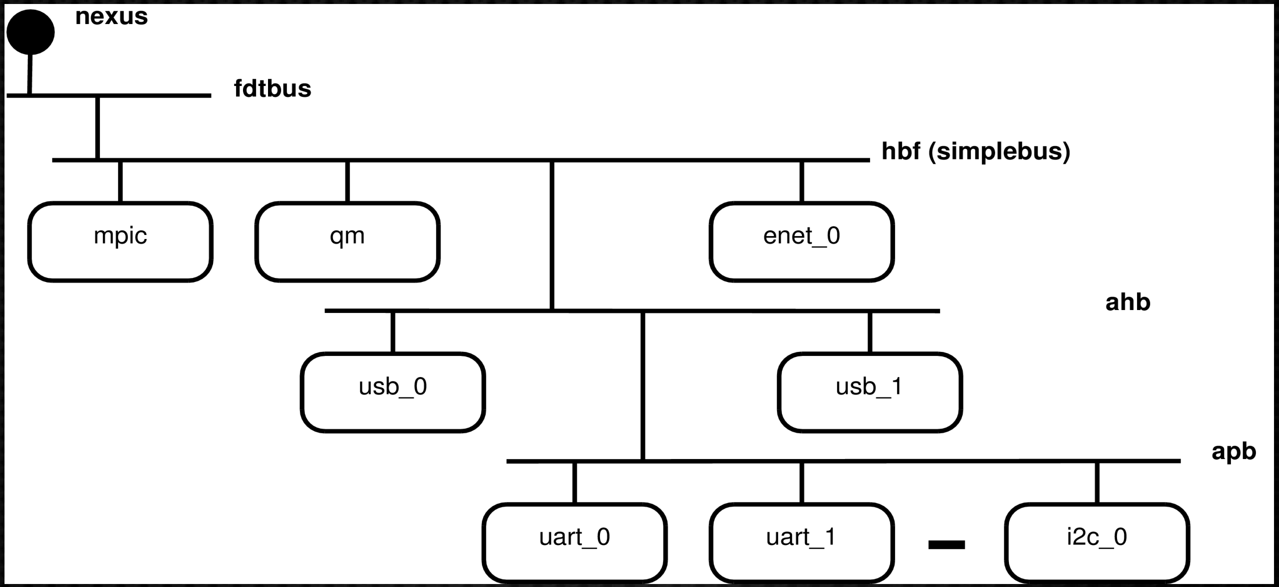
- New machdep for each platform (in sys/arch/evbppc/)
- Map the SoC registers
- Adjust UART driver
- Fill the stub functions for the Book-E exception management
- Set up the exception vector regs.
- Configure system timers

Porting – Low-level MM support

- Most sensible area of the operating system
- *pmap*
  - Manages physical address maps
  - Maintains the page tables
  - Handles memory management hardware
  - TLB
    - `tlbwe, tlbre, tlbsx[.]`
    - `tlb_write(), tlb_read(), tlb_inval_entry() and tid_flush()`

Porting – device drivers

- Flattened Device Tree (FDT)
  - Introduced in FreeBSD 9.0
  - Describes the embedded system's resources in a unified way (DTS file)
  - Same kernel for multiple platforms of the same family
  - fdtbus and simplebus



Simplified look of the device tree for APM86290

Porting – device drivers

- autoconf(9) in NetBSD
  - Direct and indirect configuration
  - Is driven by the table generated from machine description by the config(8)
  - Bus drivers from scratch

Porting – device drivers

- FDT (with minor hacks)
- Ready to use fdtbus
- Ready to use simplebus
- Drivers for the other buses

- Device description in the kernel configuration file
- Bus drivers from scratch

Porting – supported devices

- Support for the following interfaces have been developed:
  - Interrupt controller
  - Gigabit Ethernet along with Queue Manager
  - PCI-Express
  - USB host controllers – EHCI and OHCI
  - UART
  - I2C
  - GPIO
  - RTC

Porting – interrupt controller

- MPIC
    - Compliant with the *OpenPIC Register Interface Specification 1.2*

FreeBSD

- Ready-to-use OpenPIC driver
- Machine dependent interrupt management layer (intr_machdep.c)
- Incoming interrupts scheduled in the similar way to the processes running in the system

NetBSD

- No ready-to-use driver only different flavors of the OpenPIC driver designed for specific usage
- SPL(9)

Porting – Ethernet controller driver

- Cooperates with QM to maximize the performance
- Assigned queues:
  - Rx queue
  - Tx queue
  - Completion Queue
  - Free Pool

Working Queues

Porting – Ethernet controller driver

- Two data paths (ingress and egress)
  - Ingress - Classifier
- `ame_if_start()` - to start the packet sending
  - `ame_handle_tx_completion()` - callback handler informing about the command completion (executed by the QM)
- `ame_handle_rx_msg()` - called to handle send the incoming packet to the network stack
- Extended debugging (DDB utilization)

Testing and Debugging

- JTAG debuggers
- Integrated debug circuits

- Kernel debugging features
- Testing frameworks

Testing and Debugging

- In-kernel debugger (DDB)
  - Can be enabled by adding options to kernel configuration file:

```
options     KDB
options     DDB
```

  - Needs basic console initialization
- Kernel Tracing Facility (KTR)
  - Can be enabled by adding option to kernel configuration:

```
options     KTR
```

  - Logs actions while the kernel is running

Testing and Debugging

- Automated Testing Framework (ATF)
  - Located in `/usr/tests`
  - Running the tests is as simple as typing:

```
# atf-run | atf-report
```

Testing and Debugging

FreeBSD®

- DDB
- KTR

NetBSD®

- DDB
- ATF

Current state and future work

- What would be nice to be done:
  1) SMP
  2) SATA support
  3) L2 cache
  4) Extended QM utilization
  5) Cryptographic engines support
  6) Power management support

Acknowledgements

- Mentors of the project
  - Rafał Jaworowski (Semihalf, The FreeBSD Project)
  - Bartłomiej Sięka (Semihalf)

- Grzegorz Bernacki, Michał Mazur, Marcin Ropa, Łukasz Wójcik, Piotr Zięcik (all Semihalf)

Any questions?