# Porting FreeBSD/ARM to new SoCs and Boards

## Or, getting from power on to printf

Andrew Turner

andrew@fubar.geek.nz

29 September 2013

# Porting FreeBSD/ARM

# Booting

1. Chip firmware
2. Early boot
3. U-Boot
4. ubldr (U-Boot loader)
5. Kernel
6. Userland

# Chip firmware

Performs chip specific initialisation

Hard-wired in the chip

# Early Boot

Performs board specific initialisation
e.g. Set up SDRAM, simple disk drivers

Examples include TI X-Loader, Raspberry-Pi bootcode.bin, U-Boot SPL

# U-Boot

Advanced executable loader

Can load from storage, network, serial (e.g. zmodem), etc

Like a BIOS++

Can be scripted, e.g. change the boot commands

Can load U-Boot executables, some ELF files, raw binaries

# ubldr

Is the FreeBSD loader

Calls back to U-Boot to access disk and network

# Kernel and Userland

Is a standard FreeBSD

# Porting FreeBSD/ARM

# First steps

- Cross-toolchain
- Board kernel config
- Flattened Device Tree config
- UART driver

# Toolchain

For ARMv6 and ARMv7:

```
# make kernel-toolchain TARGET_ARCH=armv6
```

For ARMv4 and ARMv5:

```
# make kernel-toolchain TARGET_ARCH=arm
```

Gets a toolchain that can build the kernel (no userland)
The default compiler is clang/llvm

# Kernel config

- Create the file system layout, `sys/arm/vendor/soc`
- Create standard SoC files
  - files.soc – SoC version of `sys/conf/files`
  - std.soc – Common SoC parts of the kernel config
- Write SoC specific initialisation functions, `initarm_lastaddr`, `initarm_gpio_init`, `initarm_late_init`, `platform_devmap_init`
- Stub out DMA, CPU reset, DELAY, etc. functions
- Provide a `bus_space` struct
- Create a kernel config file in `sys/arm/conf`

# FDT config

- Best case – The board provides usable Device Tree blob
- Second best case – Existing CPU device tree source in the tree
- Worst case – You have to write the entire thing

If you have to write a new DTS (device tree source) file:

- Split out the SoC specific part of the config to a separate file and include it
- Disable devices by default, then enable them in the board file
- The minimal config includes:
  - List of memory ranges (RAM)
  - Chosen stdin/stdout
  - Aliases for the chosen section to use
  - Bus root to add devices be added to

# SoC FDT config

```
/dts-v1/;

/ {
  localbus@20000000 {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    ranges;
    bus-frequency = <0>;

    uart0: uart@20000000 {
      compatible = "ns16550";
      reg = <0x20000000 0x1000>;
      reg-shift = <2>;
      status = "disabled";
    };
  };
```

# Board FDT config

```
/dts-v1/;
/include/ "soc.dtsi";
/ {
  aliases { uart = &uart0; };
  chosen {
    stdin = "uart0";
    stdout = "uart0";
  };
  memory {
    device_type = "memory";
    reg = < 0x40000000 0x08000000 >;
  };
  localbus@20000000 {
    uart0: uart@20000000 {
      status = "okay";
    };
  };
};
```

# UART

- Best case, you have an ns16550 compatible UART
- Worst case, you have to write a custom driver without using the uart framework

If you need to write your own UART driver you will need to:

- Create a uart device class
- Create a uart ops struct with:
  - probe – Just return zero
  - init – Configure the hardware
  - putc – Wait for space in the FIFO, write the character
- Add the device class to the list in `uart_fdt_getdev` in `sys/dev/uart/uart_cpu_fdt.c`

# Porting FreeBSD/ARM

# Symptom

```
 In:   serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
soc# fatload mmc 0 0x40200000 kernel
reading kernel
4598153 bytes read in 261 ms (16.8 MiB/s)
soc# go 0x40200000
## Starting application at 0x40200000 ...
```

And then nothing...

# Tools to find out what is wrong

- JTAG
  - Useful to tell you what state the kernel is in
  - Costs €10s to €10,000s
  - Need to configure software for board, e.g. OpenOCD, DS-5
- UART
  - SoC specific
  - Map the registers into the virtual address space
- GPIO, Buzzer, LED
  - Same setup as UART
  - Limited information on how far the kernel has booted

# Things that can go wrong

- Load the kernel to the wrong location
- Branch to the wrong address
  - An address outside the kernel
  - The start of the kernel load address
  - A random address within the kernel
- Incorrect KERNPHYSADDR, KERNVIRTADDR, or STARTUP_PAGETABLE_ADDR
- No Flattened Device Tree blob
- Incorrect UART driver
- More...

# First Solution

```
 # readelf -h \
 /usr/obj/arm.armv6/src/sys/BOARD_CONFIG/kernel
ELF Header:
 Magic:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
...
 Entry point address:  0xc0200100
```

The physical entry point is:

$$\begin{aligned} entrypoint &= 0xc0200100 - KERNVIRTADDR + KERNPHYSADDR \\ &= 0xc0200100 - 0xc0200000 + 0x40200000 \\ &= 0x40200100 \end{aligned}$$

# Second Problem

Fix by jumping to the correct location

```
 In:   serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
soc# fatload mmc 0 0x40200000 kernel
reading kernel
4598153 bytes read in 261 ms (16.8 MiB/s)
soc# go 0x40200100
## Starting application at 0x40200100 ...
```

And then nothing...

# Second Problem

How do we know if we are in the kernel or not?

If we are, where are we?

# Solution

Find some way to output data

e.g. JTAG, UART, LED

# UART Solution

1. Ensure the UART clocks are setup (they most likely will be by U-Boot)
2. Find the UART output register
3. Write a character to the UART register

# UART Solution

For a ns16550 UART with a base address at 0x20000000
The data register, for writing, is at offset 0

Map the memory for the UART registers into the virtual address space.
Add to `mmu_init_table` in `sys/arm/arm/locore.S`
The format is:
`MMU_INIT(virtual address, physical address, pages, flags)`

This gives:
`MMU_INIT(0x20000000, 0x20000000, 1, L1_TYPE_S|L1_S_AP(AP_KRW))`
`L1_TYPE_S` – Use 1MiB sections
`L1_S_AP(AP_KRW)` – Sets the access permission

# Simple putc

Writes data to the serial port, drops data if the FIFO fills up

In C code:

```
void
early_putchar(char ch)
{
  *(unsigned int *)0x2000000 = ch;
}
```

In assembler, may break booting past it:

```
define PUTCHAR(ch)                  \
        ldr r0, =0x2000000;         \
        ldr r1, =ch;                \
        str r1, [r0]
```

# Places to check

In C code:

- Start of `initarm` – The first C code to run
- After the call to `OF_install`, `OF_init`, `fdt_get_mem_regions` – An infinite loop on failure
- Before the call to `setttb` – The call to setup the Translation Table Base for the MMU

If you failed to get into C:

- Start of _start – Make sure you are in the kernel as expected
- After `mmu_done` – Make sure the MMU setup worked
- After `virt_done` – Make sure we are running from a virtual address

# Second Solution

Check if we are in C code:

```
void *
initarm(struct arm_boot_params *abp)
{
  struct mem_region memory_regions[FDT_MEM_REGIONS];
  ...
  int curr;

  early_putchar('A');

  lastaddr = parse_boot_param(abp);
  ...
```

# Third Problem

```
 In:  serial
Out:  serial
Err:  serial
Hit any key to stop autoboot:  0
soc# fatload mmc 0 0x40200000 kernel
reading kernel
4598153 bytes read in 261 ms (16.8 MiB/s)
soc# go 0x40200100
## Starting application at 0x40200100 ...
A
```

And then nothing...

# Triage

We know the kernel is running C code, but failing before it calls printf

Places likely to fail:

- `OF_install` – Fails with a broken kernel config
- `OF_init` – Fails when no FDT blob was provided
- `fdt_get_mem_regions` – Fails when FDT blob it incorrect

# Add more debugging

```
if (OF_install(OFW_FDT, 0) == FALSE)
  while (1);

early_putchar('B');

if (OF_init((void *)dtbp) != 0)
  while (1);

early_putchar('C');

if (fdt_get_mem_regions(memory_regions, &memory_regions_sz,
    &memsize) != 0)
  while(1);

early_putchar('D');
```

# Third Problem

```
 In:  serial
Out:  serial
Err:  serial
Hit any key to stop autoboot:  0
soc# fatload mmc 0 0x40200000 kernel
reading kernel
4598153 bytes read in 261 ms (16.8 MiB/s)
soc# go 0x40200100
## Starting application at 0x40200100 ...
AB
```

And then nothing...

# Triage

We know:

- `OF_install` succeeds
- `OF_init` fails

Possible issues:

- No FDT blob provided – likely, one of two reasons:
  - Not provided by the boot loader
  - Not hard coded in the kernel
- Invalid FDT blob header – unlikely

Using the "go" U-Boot command doesn't provide the FDT blob

# Solution

To get something running add this:

```
options         FDT_DTB_STATIC
makeoptions     FDT_DTS_FILE=board.dts
```

to the kernel config.

It will get something working

Should only be used for development, better for the boot loader provide it

# Booting

```
 In:  serial
Out:  serial
Err:  serial
Hit any key to stop autoboot:  0
soc# fatload mmc 0 0x40200000 kernel
reading kernel
4598153 bytes read in 261 ms (16.8 MiB/s)
soc# go 0x40200100
## Starting application at 0x40200100 ...
ABCDKDB: debugger backends:  ddb
KDB: current backend:  ddb
Copyright (c) 1992-2013 The FreeBSD Project.
```

Followed by more boot messages...

# Conclusion

- Early boot code is fragile
- Easy to make mistakes
- Can be difficult to debug
- Not documented

# Porting FreeBSD/ARM

# Future work

- Early putchar framework
- Location independent kernel
- Documentation

# Questions?

FreeBSD ARM resources:

Email: freebsd-arm@FreeBSD.org
IRC: #bsdmips on EFnet