U-Boot bootloader port done right – 2018 edition

Marek Vašut <marek.vasut@gmail.com>

February 4th, 2018
Marek Vasut

- Software engineer
- Versatile Linux kernel hacker
- Custodian at U-Boot bootloader
- OE-core contributor (Yocto...)
- FPGA enthusiast
Structure of the talk

- What is U-Boot bootloader
- News in U-Boot in 2018
- U-Boot basics
- Device Tree and U-Boot
- U-Boot Driver Model
- Barebones U-Boot port 101
- Low-memory system optimization
- Conclusion
U-Boot bootloader

- Boot loader
  - First\(^1\)-ish code that runs on a system
  - Responsible for some HW initialization and starting OS
- Boot monitor
- Debug tool

\(^1\)There are exceptions, ie. Boot ROMs
U-Boot example


CPU: Renesas Electronics CPU rev 1.0
Model: Renesas Salvator-X board based on r8a7795 ES2.0+
Board: Salvator-X
I2C: ready
DRAM: 3.9 GiB
Flash: 64 MiB
MMC: sd@ee100000: 0, sd@ee140000: 1, sd@ee160000: 2
In: serial@e6e88000
Out: serial@e6e88000
Err: serial@e6e88000
Net: eth0: ethernet@e6800000
Hit any key to stop autoboot: 0
=>
=> md 0xe6e88000 4
e6e88000: 00000000 11111111 00300300 00000000 .........0.0.....
=>
U-Boot news and highlights – 2018 edition

- Device Tree control
- Driver Model conversion
- EFI support
- Distro boot command
- DTO application with fitImage
- CI
U-Boot sources

- Git master at:
  http://git.denx.de/?p=u-boot.git;a=summary
- Custodian subtrees at:
  http://git.denx.de/?p=u-boot.git;a=forks
- Available via Git and HTTP protocols
Building the sources

$ git clone git://git.denx.de/u-boot.git
$ cd u-boot
$ export ARCH=plat   # optional, set target architecture
$ export CROSS_COMPILE=plat-none-  # optional, set cross compiler
$ make board_defconfig  # ie. sandbox_defconfig
$ make
U-Boot source tree structure

- arch/ – Architecture specific code
  - arch/nnn/mach-foo – foo CPU specific code
  - arch/nnn/dts – U-Boot device trees
- board/ – Board specific code
- config/ – Per-board U-Boot configurations
- include/ – Header files for all global things
  - include/configs/ – Legacy per-board configs
- drivers/ – U-Boot drivers
- cmd/ – U-Boot commands
- common/ lib/ – Common code
- net/ – Network stack
- fs/ – Filesystems implementation

Kconfig and Makefile permeate the entire structure
Configuring U- Boot

- U-Boot uses Kconfig for configuration
- Some configuration macros still not converted:
  ./scripts/config whitelist.txt
- All new config options are Kconfig-only!
- Example, drivers/net/Kconfig:

```
  source "drivers/net/phy/Kconfig"

  config DM_ETH
  bool "Enable Driver Model for Ethernet drivers"
  depends on DM
  help
    Enable driver model for Ethernet.

  The eth_*( ) interface will be implemented by the UC_ETH class
  This is currently implemented in net/eth.c
  Look in include/net/ h for details.
```

- Example, drivers/net/Makefile:

```
  obj-$(CONFIG_ALTERA_TSE) += altera_tse.o
  obj-$(CONFIG_AG7XXX) += ag7xxx.o
```
Adding configuration option

Adding configuration option means

- Patch matching Kconfig file
- Patch matching Makefile if applicable
- Add code.c sources

unless this is a hardware-related configuration

- Use Device Tree to describe hardware
- Do NOT hardcode hardware topology into U-Boot
Device Tree

- Data structure describing hardware
- Usually passed to OS to provide information about HW topology where it cannot be detected/probed
- Tree, made of named nodes and properties
  - Nodes can contain other nodes and properties
  - Properties are a name-value pair
    - See https://en.wikipedia.org/wiki/Device_tree
- DT can contain cycles by means of phandles
- More on DT at:
  https://www.devicetree.org/
- ePAPR specification of DT:
  https://elinux.org/images/c/cf/Power_ePAPR_APPROVED_v1.1.pdf
include <dt-bindings/power/r8a7795-sysc.h>
/
model = "Renesas Salvator-X board based on r8a7795 ES2.0+";
compatible = "renesas,salvator-x", "renesas,r8a7795";

...]
cpus {
a57_0: cpu@0 {
compatible = "arm,cortex-a57", "arm,armv8";
reg = <0x0>;
device_type = "cpu";
power-domains = <&sysc R8A7795_PD_CA57_CPU0>;
next-level-cache = <&L2_CA57>;
enable-method = "psci";
}
...]
soc: soc {
pmu_a57 {
compatible = "arm,cortex-a57-pmu";
interrupts = <GIC_SPI 72 IRQ_TYPE_LEVEL_HIGH>,
<GIC_SPI 73 IRQ_TYPE_LEVEL_HIGH>,
<GIC_SPI 74 IRQ_TYPE_LEVEL_HIGH>,
<GIC_SPI 75 IRQ_TYPE_LEVEL_HIGH>;
interrupt-affinity = <&a57_0>, <&a57_1>, <&a57_2>, <&a57_3>;
Device Tree in U-Boot

Two ways U-Boot uses DT:

- Patch DT and pass it to kernel
- Understand HW topology
  - CONFIG_OF_CONTROL
  - U-Boot needs early access to DT!
U-Boot early stages

- Platform-specific reset vector code
- `crt0.S`
- `common/board_f.c`
  - U-Boot running from FLASH
    - First item is `fdtdec_setup()`
- `common/board_r.c`
  - U-Boot running from RAM
- Hint: `lib/initcall.c` is nice debug aid
U-Boot DT access

- fdt_*( ) functions in include/fdt_support.h
  Very rudimentary
- fdtdec_*( ) functions in include/fdtdec.h
  Convenience wrappers around fdt_() functions
- dev_read_*( ) functions in include/dm/read.h
  DM-specific DT access functions
- Parsing DT by hand can be useful in early stages, but later we use DM
U-Boot Driver Model

- Harbinger of order within all the ifdef chaos
- Consists of:
  - Classes – Groups of devices which operate the same, ie. GPIO uclass, I2C controller uclass...
  - Drivers – Code which talks to device and presents standard higher-level interface for Class
  - Devices – Each device with a fitting driver gets an instance
U-Boot DM core

- Responsible for handling device life-cycle
- Inherently lazy to reduce boot time
- Upon init, creates root driver
- Everything else is under the root driver
### U-Boot DM example

```bash
dm tree
```

<table>
<thead>
<tr>
<th>Class</th>
<th>Probed</th>
<th>Driver</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>[ + ]</td>
<td>root_drive</td>
<td>root_driver</td>
</tr>
<tr>
<td>clk</td>
<td>[ + ]</td>
<td>fixed_rate</td>
<td></td>
</tr>
<tr>
<td>simple_bus</td>
<td>[ + ]</td>
<td>generic_si</td>
<td></td>
</tr>
<tr>
<td>gpio</td>
<td>[ ]</td>
<td>rcar-gpio</td>
<td></td>
</tr>
<tr>
<td>gpio</td>
<td>[ + ]</td>
<td>rcar-gpio</td>
<td></td>
</tr>
<tr>
<td>pinctrl</td>
<td>[ + ]</td>
<td>sh_pfc_pin</td>
<td></td>
</tr>
<tr>
<td>pinconfig</td>
<td>[ ]</td>
<td>pinconfig</td>
<td></td>
</tr>
<tr>
<td>pinconfig</td>
<td>[ + ]</td>
<td>pinconfig</td>
<td></td>
</tr>
<tr>
<td>pinconfig</td>
<td>[ + ]</td>
<td>pinconfig</td>
<td></td>
</tr>
<tr>
<td>pinconfig</td>
<td>[ ]</td>
<td>pinconfig</td>
<td></td>
</tr>
<tr>
<td>pinconfig</td>
<td>[ ]</td>
<td>pinconfig</td>
<td></td>
</tr>
<tr>
<td>pinconfig</td>
<td>[ ]</td>
<td>pinconfig</td>
<td></td>
</tr>
<tr>
<td>pinconfig</td>
<td>[ ]</td>
<td>pinconfig</td>
<td></td>
</tr>
<tr>
<td>pinconfig</td>
<td>[ ]</td>
<td>pinconfig</td>
<td></td>
</tr>
<tr>
<td>serial</td>
<td>[ ]</td>
<td>serial_sh</td>
<td></td>
</tr>
<tr>
<td>serial</td>
<td>[ + ]</td>
<td>serial_sh</td>
<td></td>
</tr>
<tr>
<td>usb</td>
<td>[ ]</td>
<td>xhci_rcar</td>
<td></td>
</tr>
<tr>
<td>usb</td>
<td>[ ]</td>
<td>ehci_gener</td>
<td>`-- usb@ee0c0100</td>
</tr>
<tr>
<td>regulator</td>
<td>[ ]</td>
<td>fixed_regu</td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td>[ ]</td>
<td>fixed_rate</td>
<td>`-- x23-clock</td>
</tr>
</tbody>
</table>
```
U-Boot Driver life-cycle

- Driver is statically defined by U_BOOT_DRIVER macro
- Upon instantiation, the following are done:
  - (optional) – Preallocation of private data
  - .bind – Bind the driver with DM, device not active
  - .probe – Upon first request, device activated
  - .remove – Counterpart to probe
  - .unbind – Counterpart to bind
Porting U-Boot to a new board 101

- Start small – boot and get serial console
- But serial console is hard, it needs
  - clock – we need clock driver
  - pinmux – we need pinmux driver
  - serial – we need serial driver
- Most parts can be done separately
Starting new port

- Populate arch/foo/mach-bar if applicable
  - Add DTS to arch/foo/dts/
  - Add at least one board
- Populate board/mymfg/myboard if applicable
  - Add configs/mymfg/myboard_defconfig
  - Add include/configs/mymfg/myboard.h
  - Add board/mymfg/myboard/Kconfig
  - Add board/mymfg/myboard/Makefile
  - Add board/mymfg/myboard/myboard.c
  - Add arch/foo/mach-bar/Kconfig entry
- For arch and board specific hooks, see: common/board_f.c and common/board_r.c
- Most of the code should be in drivers/
U-Boot DM serial driver

U_BOOT_DRIVER(serial_sh) = {
    .name = "serial_sh",
    .id = UCLASS_SERIAL,
    .ops = &sh_serial_ops,

    .of_match = of_match_ptr(sh_serial_id),

    .probe = sh_serial_probe,
    .priv_auto_alloc_size = sizeof(struct uart_port),

    .ofdata_to_platdata =
        of_match_ptr(sh_serial_ofdata_to_platdata),
    .platdata_auto_alloc_size =
        sizeof(struct sh_serial_platdata),

    .flags = DM_FLAG_PRE_RELOC,
};
U-Boot DM serial driver II

DT matching is done for you!

```
static const struct udevice_id sh_serial_id[] = {
    {.compatible = "renesas,sci", .data = PORT_SCI},
    {.compatible = "renesas,scif", .data = PORT_SCIF},
    {.compatible = "renesas,scifa", .data = PORT_SCIFA},
    {});

static int sh_serial_ofdata_to_platdata(struct udevice *dev)
{
    struct sh_serial_platdata *plat = dev_get_platdata(dev);
    [...]
    addr = fdtdec_get_addr(gd->fdt_blob, dev_of_offset(dev), "reg");
    if (addr == FDT_ADDR_T_NONE)
        return -EINVAL;
    plat->base = addr;
    [...]
}

U_BOOT_DRIVER(serial_sh) = {
    .of_match = of_match_ptr(sh_serial_id),
    .ofdata_to_platdata = of_match_ptr(sh_serial_ofdata_to_platdata),
    .platdata_auto_alloc_size = sizeof(struct sh_serial_platdata),
};
```
Serial ops, getc, putc, etc...

```c
static int sh_serial_getc(struct udevice *dev) {
    struct uart_port *priv = dev_get_priv(dev);
    return sh_serial_getc_generic(priv);
}

static const struct dm_serial_ops sh_serial_ops = {
    .putc = sh_serial_putc,
    .pending = sh_serial_pending,
    .getc = sh_serial_getc,
    .setbrg = sh_serial_setbrg,
};

U_BOOT_DRIVER(serial_sh) = {
    .ops = &sh_serial_ops,
};
```
Early serial console

Sometimes serial is needed before DM is available:

- Special-purpose code allowing very early prints
- Special-purpose custom print functions:
  - printch(), printascii(), printhex2()...
- CONFIG_DEBUG_UART=y
- Resides in include/debug_uart.h
U-Boot early serial console with DM
See ie. drivers/serial/serial_ar933x.c:

```c
#ifdef CONFIG_DEBUG_UART_AR933X
#include <debug_uart.h>

static inline void _debug_uart_init(void)
{
    [...]
    writel(val, regs + AR933X_UART_CLK_REG);
}

static inline void _debug_uart_putchar(int c)
{
    void __iomem *regs = (void *)CONFIG_DEBUG_UART_BASE;
    u32 data;

    do {
        data = readl(regs + AR933X_UART_DATA_REG);
    } while (!(data & AR933X_UART_DATA_TX_CSR));

    data = (u32)c | AR933X_UART_DATA_TX_CSR;
    writel(data, regs + AR933X_UART_DATA_REG);
}

DEBUG_UART_FUNCS
#endif
```
Clock framework

Clock provider:
- uses UCLASS_CLK
- implements clk_ops to enable/disable/get/set clock
- Resides in include/clk-uclass.h

Clock consumer:
- Uses clk_*( ) clock framework functions
U-Boot clock consumer

SH UART driver consumes clock:

```c
[...]
struct sh_serial_platdata *plat = dev_get_platdata(dev);
struct clk sh_serial_clk;
int ret;
[...]
ret = clk_get_by_name(dev, "fck", &sh_serial_clk);
if (!ret) {
    ret = clk_enable(&sh_serial_clk);
    if (!ret)
        plat->clk = clk_get_rate(&sh_serial_clk);
            "clock", 1);
[...]
```
Pinctrl framework

- One framework handles two roles
- uses UCLASS_PINCTRL
- implements pinctrl_ops to configure pins
- operates per-pin, per-group, per-function
- PINMUX – configures pin multiplexing
- PINCONF – configures pin properties (voltage, pull,...)

Pinctrl consumer:
- Can select pin configuration from multiple options
- DM sets default pin configuration based on DT
- Useful ie. when selecting eMMC IO voltage
U-Boot pinctrl consumer

DT node lists two possible pin configurations:

```c
&sdhi0 {
    pinctrl-0 = <&sdhi0_pins>;
    pinctrl-1 = <&sdhi0_pins_uhs>;
    pinctrl-names = "default", "state_uhs";
}
```

Uniphier SD driver sets IO voltage:

```c
static void uniphier_sd_set_pins(struct udevice *dev) {
    struct uniphier_sd_priv *priv = dev_get_priv(dev);
    struct mmc *mmc = mmc_get_mmc_dev(dev);
    [...]
    if (mmc->signal_voltage == MMC_SIGNAL_VOLTAGE_180)
        pinctrl_select_state(dev, "state_uhs");
    else
        pinctrl_select_state(dev, "default");
}
Other frameworks

- Block layer is fully DM capable
- MTD layer needs DM conversion
- DM can trigger size limits!
Low-memory systems

U-Boot SPL – Secondary Program Loader
▶ Reduced U-Boot build which loads subsequent payload:
U-Boot, Linux (falcon mode), ...
▶ May be very board specific
▶ DM support is optional
▶ DT support is optional
▶ Special CONFIG_SPL_* Kconfig options
  ▶ Allow controlling what goes into SPL vs U-Boot

U-Boot TPL – Tertiary Program Loader
▶ If SPL is too big, TPL loads SPL
▶ Full-on custom solution
▶ Try to avoid this
▶ For TPL, CONFIG_TPL_* also exists
DT compaction

Standard DT blob is too big
- Unused nodes can be removed
  - Done for U-Boot SPL by default
  - Nodes with special DT property are retained:
    - u-boot,dm-pre-reloc
  - LibFDT has fdtgrep tool for this
  - Same marker used for drivers that must be started early
- Driver instantiation with platform data
  - DT nodes replaced with C structures linked with U-Boot
  - Useful in size-limited U-Boot SPL
  - LibFDT support can be dropped from SPL
    (saves quite a bit of .text area)
Conclusion

- Use DT and DM in new U-Boot ports
- Reuse code and drivers as much as possible
- Read the documentation, see doc/
- Submit patches
- Reach out to U-Boot community for help:
  IRC: irc.freenode.net #u-boot
  ML: u-boot@lists.denx.de
Thank you for your attention!

Contact: Marek Vasut <marek.vasut@gmail.com>