Linux on ARM

Gernot Kvas (gernot.kvas@fh-joanneum.at)

April 19, 2008

Table of contents

The ARM architecture

- 32-bit RISC: ARM offers IP cores of machines with interesting features
    - Different instruction sets: Thumb, Jazelle
    - Separate data/instruction busses
    - DSP-Style vector operations
- Due to the licensing model, there exists a multitude of different SoCs implementing ARM cores.
- Good support by gcc/gdb and other Open-Source tools
- ARM CPUs offer a good performance/power consumption trade-off
- Who would not want to run Linux on these?

Booting the system

PC

BIOS

Does basic hardware initialsation

(e.g.) GRUB

Passes control to the kernel

Kernel

Does the obvious

ARM

Firmware/Bootloader

On ARM systems, replaces both the BIOS and the Bootloader, brings up the hardware to a state where Linux can take over

Kernel

Does the obvious

Booting on ARM

- Linux on ARM requires the firmware/bootloader to set up the hardware. See: `Documentation/arm/Booting`
- The following steps are required:
  1. Set up and initialise the RAM (M)
  2. Initialise one serial port (R)
  3. Detect the machine type (M)
  4. Set up the kernel tagged list (M)
  5. Call the kernel image (M)
- Before calling the Kernel:
  - Switch off D-Cache, MMU, DMA
  - Switch off Interrupts
  - Get ARM in Supervisor Mode
  - Set `R0` to 0, `R1` to Machine Type and `R3` to &(ATAGS)

Das U-Boot - The Universal Bootloader

- Das U-Boot (`http://www.denx.de/wiki/UBoot`) is a bootloader that amongst others boots ARM
- Essentially does what is required by the previously mentioned boot process
- Supports various ARM cores
- Offers hardware support to boot from different storage devices
- It is relatively easy to add your board:
  1. Create a config.h file for your board
  2. Write an assembler file that sets up RAM
  3. Write a C-file that does the high level init
- U-Boot deals with the booting requirements

Machine Registration

- Each individual machine (= embedded system) is assigned a number
- This is the number passed in `R2`

```
# http://www.arm.linux.org.uk/developer/machines/?action=new
#
# Last update: Fri May 11 19:53:41 2007
#
# machine_is_xxx        CONFIG_xxxx         MACH_TYPE_xxx        number
#
ebsa110                 ARCH_EBSA110        EBSA110              0
riscpc                  ARCH_RPC            RISCPC               1
nexuspci                ARCH_NEXUSPCI       NEXUSPCI             3
ebsa285                 ARCH_EBSA285        EBSA285              4

csb726                  MACH_CSB726         CSB726               1359
tik27                   MACH_TIK27          TIK27                1360
mx_uc7420               MACH_MX_UC7420      MX_UC7420            1361
```

ARM relevant bits in the kernel

- Relevant directories - everything below `arch/arm`:
  - `mm/lib/kernel/tools`: You rarely have to deal with those
  - `arch/arm/mm/proc-*` shows the supported ARM CPUs:

proc-arm1020e.S	proc-arm740.S	proc-arm940.S	proc-syms.c
proc-arm1020.S	proc-arm7tdmi.S	proc-arm946.S	proc-v6.S
proc-arm1022.S	proc-arm920.S	proc-arm9tdmi.S	proc-v7.S
proc-arm1026.S	proc-arm922.S	proc-macros.S	proc-xsc3.S
proc-arm6_7.S	proc-arm925.S	proc-sa1100.S	proc-xscale.S
proc-arm720.S	proc-arm926.S	proc-sa110.S	

  - Important for the implementer: `arch/arm/arch-*`, `include/asm-arm/mach-*`

Adding a SoC

If you start supporting a totally new SoC:

1. Requires some assembler code in `include/asm-arm/mach-YOURSOC/`
   - `entry-macro.S`: Initial low level handling of interrupts.
   - `debug-macro.S`: Some routines to get early debug messages

   This code is in `include`, because `arch/arm/kernel/entry-common.S` and `arch/arm/kernel/debug.S` pick it up
2. High level stuff is done in `arch/arm/arch-YOURSOC`
   - `irq.c`: Contains the interrupt handling (ACK/MACK/MASK)
3. Your core CPU is already supported, thus requiring only these subtle changes
4. But: You have no drivers yet! These live in the `drivers` directory

Adding a new machine

- Typically requires only changes to Kconfig/Makefile in the respective `arch-*` directory and a single C-file

```
static void __init mach_spectro2_init_machine(void)
{
        ns9xxx_init_machine();

        platform_add_devices(devices, ARRAY_SIZE(devices));

        spi_register_board_info(spi_b_board_info, ARRAY_SIZE(spi_b_board_info));
        spi_register_board_info(spi_a_board_info, ARRAY_SIZE(spi_a_board_info));

        i2c_register_board_info(0, spectro2_i2c_devices, ARRAY_SIZE(spectro2_i2c_devices));
}

unsigned int ns_sys_clock_freq( void )
{
        return 398131200;
}

MACHINE_START(SPECTRO2, "Spectro2")
        .map_io = mach_spectro2_map_io,
        .init_irq = mach_spectro2_init_irq,
        .init_machine = mach_spectro2_init_machine,
        .timer = &ns9xxx_timer,
        .boot_params = 0x100,
MACHINE_END
```

Debugging via UART and JTAG

- As a serial port is strongly recommeded by the bootloader, use it for debugging
  - Uses functions defined in debug-macro.S
    - addruart - Checks for MMU to adjust base address
    - senduart - Sends a byte
    - busyuart - Checks for UART to finish
    - waituart - Waits for CTS
- Other possibilities include the usage of a JTAG device
  - You will need one for initial bootloader development
  - Fortunately, JTAG devices are available for around 100 Euros
  - OpenOCD `http://openocd.berlios.de/` is a good Open-Source package that allows GDB to talk to your CPU via a JTAG device

Buildroot/OpenEmbedded

- Buildroot and OpenEmbedded are good starting points for your userspace applications
- Buildroot is a framework of Makefiles
  - Configured with a Kernel-like ncurses interface
  - Quite easy to add packages
  - Tightly linked to uClibc, a small C library
- OpenEmbedded uses a more powerful concept of packages
  - Used by OpenMoko, Angstrom
- Be prepared to spend some time getting a properly configured system. Once you have it, keep all the configs!

Cross Compiling

- Typically for embedded systems, programs are compiled on the host
- This requires a cross compiler
  - Use higher level tools to configure your compiler, this saves you from trouble
  - Fortunately, both Buildroot and OpenEmbedded do the job for you!
  - Ideally, use the same compiler for all your stuff

Kernel-Wise

- Use the source, Luke!
- Keep in touch with current kernel development: Don't get stuck with an ancient kernel version, you might need new stuff!
- Try to get your serial driver working first
- Don't jump too many kernel versions at once when moving to a more recent version
- Use GIT

Community-Wise

- Watch the relevant mailing lists:
  - linux-arm-kernel - Kernel list
  - linux-arm - General talk
  - linux-arm-toolchain - Toolchain list
  - LKML - Linux kernel mailing list (If you have lots of time)
  - Mailing lists of subsystems (e.g. SPI, MMC)
- Follow the "Release early - Release often" policy
- Don't be afraid to show your code: Peer reviews of your code guarantee quality
- Try to get your stuff into the kernel - out of tree stuff is harder to maintain

Stuff that was discussed after the talk

- Buffalo ARM9-based Linkstations (LS Pro/LS Live) give good eval boards
  - The JTAG header and serial port are labeled on the silkscreen
  - http://buffalo.nas-central.org has a Wiki with all important facts
  - Marvell git-tree: http://git.kernel.org/?p=linux/kernel/git/nico/orion.git
  - Kernel 2.6.25 now has Marvell SoC support
  - You can use the typical u-boot method of loading a new kernel image via tftp, even with the stock u-boot loader
  - Use a recent OpenOCD version with Ferocon support
  - Amontec offers JTAG interfaces for about 30 Euros that work with OpenOCD