Porting the Linux Kernel to a New ARM Platform

Wookey and Tak-Shing, *Aleph One* • *www.aleph1.co.uk*

So, you have some ARM* based hardware that you want to port the Linux kernel to. This is a task that a competent software engineer can undertake assisted by relevant information such as this article, although previous familiarity with the Linux kernel will make it a lot easier. If your platform is a lot like something that has gone before then the port can be relatively simple, but if it's all new then it could be a big job, and you might well be advised to get help from someone experienced in these things, depending on how much of a challenge you want. And of course, if you don't actually know that your hardware works properly (you usually know this if it has already run some OS other than Linux), then again things can get exciting as you may not know if the hardware is broken or your kernel changes are wrong.

This article can't tell you everything you need to know about kernel hacking – it's a huge subject. If you don't know how the kernel works you need to read some relevant documentation. What I will try to cover are the procedures and conventions used in the ARM kernel development community, how the ARM architecture files are set out in the source, and the basics of what you will need to change to port the core of the kernel to your new platform – enough so that it boots and sends serial debug info.

The platform we will use in the examples is *Anakin*, as this isn't too complicated, but is sufficiently unlike other platforms to be an architecture, rather than a sub-architecture, and thus be a non-trivial example. In case you were wondering, Anakin is a vehicle telematics platform designed by Acunia n.v., of Belgium, for which Aleph One did the initial kernel port. It contains an Intel® StrongARM SA-1110 with some memory and an FPGA containing the core devices (memory controller, video controller, serial ports, interrupt controller).

Terminology

Talking about this subject can be confusing with multiple meanings and some overlap of the terms 'device', 'platform', 'machine', and 'architecture', depending on context. For example, a device can be a thing such as a PDA, or a bit of hardware that the kernel must access via a device-driver. Here are the definitions used in this article.

- *architecture:* Either the CPU-type (as in 'ARM architecture', 'x86 architecture'), or category of system design as in 'footbridge architecture' or 'Intel® SA-1100 architecture'. It is sometimes used to mean the same as 'machine' below too;
- *device:* A hardware item that the kernel accesses using a device-driver;
- *machine:* Your particular hardware, as determined by the assigned machine-ID within the kernel;
- *platform:* Same as machine;
- *sub-architecture:* Same as machine.

A machine may be a system architecture in its own right, but is usually a sub-architecture.

Overview of Files

The starting point here is that you have set up your kernel source tree with the ARM patches. This description covers the 2.4 series kernels, specifically 2.4.18. Things will change as kernel development continues. The ARM-specific files are in `linux/arch/arm` (code), and `linux/include/asm-arm` (header files). In a configured kernel tree, the appropriate architecture headers directory (`linux/include/asm-arm` in our case) appears as `linux/include/ asm` so that it can be easily referred to by other files that don't need to know which architecture they are dealing with. Your machine-specific sub-directories go into these directories.

Device drivers for things, even if they are only found on the ARM architecture, or even only on your machine, do not go in these directories, but in the relevant parts of the main tree; usually `linux/drivers/`, but sometimes `linux/fs` or `/linux/net` if there are filesystems, partition types or network protocols specific to your device. Within the ARM-specific directories your new or changed files go in appropriate `linux/arch/arm/mach-XXX` and `linux/include/asm-arm/arch-XXX` directories. e.g. `linux/arch/ arm/mach-anakin` and `linux/include/asm-arm/arch-anakin`. After configuration your headers directory `linux/include/asm-arm/arch-XXX` appears as `linux/include/asm-arm/arch` so that the correct machine header files can be included by using this path.

The other directories in `linux/arch/arm/` contain the generic ARM code.

- *kernel* - core kernel code;
- *mm* - memory management code;
- *lib* - ARM-specific or optimised internal library functions (backtrace, memcpy, io functions, bit-twiddling etc);
- *nwfpe* and *fastfpe* - two different floating-point implementations;
- *boot* - the directory in which the final compiled kernel is left and contains stuff for generating compressed kernels;
- *tools* - has scripts for autogenerating files, such as `mach-types` (see section Registering a Machine ID);
- *def-configs* - contains the default configuration files for each machine.

The non machine-specific directories in `linux/include/asm-arm` are:

- *arch* - the link to the configured machine headers sub-directory arch-XXX;
- *hardware* - headers for ARM-specific companion chips or devices;
- *mach* - generic interface definitions for things used by many machines (irq, dma, pci) and the machine description macros;
- *proc* - link to proc-armo or proc-armv appropriate for configured machine;
- *proc-armo*, and *proc-armv* - 26 and 32-bit versions of core processor-related headers.

Not every new machine is a whole new architecture, most are only sub-architectures - e.g. all the Intel® SA-1100 and SA-1110 Processor-based devices are grouped together under the sa1100 architecture in `linux/arch/arm/mach-sa1100`. You will need to take a look at the existing machines to see if yours more naturally goes into the tree as a sub-architecture or not. It is useful to do a bit of research to see which other machines are closest to yours in various aspects. Often the easiest way to start is to copy over the files of the nearest machine to yours.

armo and armv

Throughout the ARM architecture core directories you will find both -armv and -armo versions of some files. These indicate variants for the ARM processors 26-bit mode and 32-bit mode. The 26-bit mode is in the process of being phased out of ARM CPU designs and is only used in a few early machines which predate the existence of the 32-bit mode (e.g. the A5000, which has an ARM3 CPU).

The suffixes have the following meaning:
- *armo* is for 26-bit stuff;
- *armv* is for 32-bit stuff.

Registering a Machine ID

Each device is identified in the kernel tree by a *machine ID*. These are allocated by the kernel maintainer to keep the huge number of ARM device variants manageable in the source trees.

The first thing you need to do in your port is register your new machine with the kernel maintainer to get a number for it. This is not actually necessary to begin work, but you'll need to do this eventually so it's best to do it at the beginning and not have to change your machine name or ID later.

You register a new architecture by mailing <rmk@arm.linux.org.uk>, or filling in an on-line form at http://www.armlinux.org.uk/developer/machines/. The on-line version is preferred as this will also set up the password you need to use the patch-system.

If using mail, please give the mail a subject of Register new architecture:

Name: <name of your architecture>
ArchDir: <name of include/asm-arm/arch-* directory>
Type: <MACH_TYPE_* macro name>
Description:
<description of your architecture>

Please follow this format - it is an automated system. You should receive a reply within one day like this:

```
You have successfully registered your architecture!
>
>   The registered architecture name is
>      Anakin
>
>   The architecture number that has been allocated is:
>      57
>
>   This number corresponds to the following macros:
>      machine_is_anakin()
>      MACH_TYPE_ANAKIN
>      CONFIG_ARCH_ANAKIN
>
>   and your architecture-specific include directory is
>      include/asm-arm/arch-anakin
>
> If, in the future, you wish to alter any of these,
> entries, please
> contact rmk@arm.linux.org.uk.
```

Then you need to add the info to linux/arch/arm/tools/mach-types with a line like this:

```
machine_is_xxx          CONFIG_xxx              MACH_TYPE_xxx
machine_ID
lart            SA1100_LART                     LART                    27
anakin          ARCH_ANAKIN             ANAKIN                  57
```

or go to: http://www.arm.linux.org.uk/developer/machines/ where you can download the latest version of mach-types.

The above file is needed so that the script `linux/arch/arm/tools/gen-mach-types` can generate `linux/include/asm-arm/mach-types.h` which sets the defines and macros mentioned above that are used by much of the source to select the appropriate code. You should always use these macros in your code, and not test `machine_arch_type` nor `__machine_arch_type` directly as this will produce less efficient code. The macros are created in such a way that unused code can be efficiently optimised away by the compiler.

Config files

Add a new config file in `linux/arch/arm/def-configs/` named <machine-name>, containing the default configuration options for your machine. You should also edit `linux/arch/arm/config.in` so that `make config` will support your machine. This file specifies the new `CONFIG_` symbols for your machine and the dependencies of them on other `CONFIG_` symbols.

When you do make <machine-name>_config, e.g. `make anakin_config`, to build a kernel, then the file corresponding to the first part of the parameter is copied out of `linux/arch/arm/def-configs/` to `linux/.config`.

Kernel Basics

There are a number of basic symbols that you need to know the meanings of to understand the kernel sources. Here is a list of the most important ones.

Throughout the code you need to keep in mind the mapping between physical and virtual memory. The kernel deals exclusively in virtual memory once it has started. Your hardware specifications deal in physical memory. One of the fundamental things you need to specify is the mapping between these two. This is contained in the `__virt_to_phys()` macro in `include/asm-arm/arch-XXX/memory.h` (along with corresponding reverse mappings). Normally, this macro is simply:

```
phys = virt - PAGE_OFFSET + PHYS_OFFSET
```

Decompressor Symbols

ZTEXTADDR

Start address of decompressor. As the MMU is off at the time this code is called the addresses are physical. You normally call the kernel at this address to start it booting. This doesn't have to be located in RAM, it can be in flash or other read-only or read-write addressable medium.

ZBSSADDR

Start address of zero-initialised work area for the decompressor. This must be pointing at RAM. The decompressor will zero initialise this for you. Again, the MMU will be off.

ZRELADDR

This is the address where the decompressed kernel will be written, and eventually executed. The following constraint must be valid:

```
__virt_to_phys(TEXTADDR) == ZRELADDR
```

The initial part of the kernel is carefully coded to be position independent.

INITRD_PHYS

Physical address to place the initial RAM disk. Only relevant if you are using the bootpImage stuff (which only works on the older struct param_struct style of passing the kernel boot information).

INITRD_VIRT

Virtual address of the initial RAM disk. The following constraint must be valid:

```
__virt_to_phys(INITRD_VIRT) == INITRD_PHYS
```

PARAMS_PHYS

Physical address of the struct param_struct or tag list, giving the kernel various parameters about its execution environment.

Kernel Symbols

PHYS_OFFSET

Physical start address of the first bank of RAM.

PAGE_OFFSET

Virtual start address of the first bank of RAM. During the kernel boot phase, virtual address PAGE_OFFSET will be mapped to physical address PHYS_OFFSET, along with any other mappings you supply. This should be the same value as TASK_SIZE.

TASK_SIZE

The maximum size of a user process in bytes. Since user space always starts at zero, this is the maximum address that a user process can access+1. The user space stack grows down from this address.

Any virtual address below TASK_SIZE is deemed to be user process area, and therefore managed dynamically on a process by process basis by the kernel. This is referred to as the 'user segment'.

Anything above TASK_SIZE is common to all processes. This is referred to as the 'kernel segment'.

Note that this means that you can't put IO mappings below TASK_SIZE, and hence PAGE_OFFSET.

TEXTADDR

Virtual start address of kernel, normally PAGE_OFFSET + 0x8000. This is where the kernel image ends up. With the latest kernels, it must be located at 32768 bytes into a 128MB region. Previous kernels just required it to be in the first 256MB region.

DATAADDR

Virtual address for the kernel data segment. Must not be defined when using the decompressor.

VMALLOC_START, VMALLOC_END

Virtual addresses bounding the vmalloc() area. There must not be any static mappings in this area; vmalloc will overwrite them. The addresses must also be in the kernel segment (see above). Normally, the vmalloc() area starts VMALLOC_OFFSET bytes above the last virtual RAM address (found using variable high_memory).

VMALLOC_OFFSET

Offset normally incorporated into VMALLOC_START to provide a hole between virtual RAM and the vmalloc area. We do this to allow out of bounds memory accesses (eg, something writing off the end of the mapped memory map) to be caught. Normally set to 8MB.

Architecture Specific Macros

BOOT_MEM(pram,pio,vio)

`pram' specifies the physical start address of RAM. Must always be present, and should be the same as PHYS_OFFSET.

`pio' is the physical address of an 8MB region containing IO for use with the debugging macros in arch/arm/kernel/debug-armv.S.

`vio' is the virtual address of the 8MB debugging region.

It is expected that the debugging region will be re-initialized by the architecture specific code later in the code (via the MAPIO function).

BOOT_PARAMS

Same as, and see PARAMS_PHYS.

FIXUP(func)

Machine specific fixups, run before memory subsystems have been initialized.

MAPIO(func)

Machine specific function to map IO areas (including the debug region above).

INITIRQ(func)

Machine specific function to initialize interrupts.

Kernel Porting

Finally we get to the meat of the task. Here we list the most important files, and describe their purpose and the sort of things you should put in them. It looks daunting to start with but most of what is required is just a matter of filling in the numbers appropriate to your hardware. Now that so many different machines are supported it is rare that you have to write much new code - nearly everything can be taken from a suitable donor machine. This is easier to do if you know which machines have a similar architecture to your own.

Throughout this list XXX represents your machine name - 'anakin' in these examples

Files

`arch/arm/Makefile`

Insert the following to this file (replace **XXX** with your machine name):

```
ifeq ((CONFIG_ARCH_XXX),y)
MACHINE                 = xxx
endif
```

`arch/arm/boot/Makefile`

Here you specify ZTEXTADDR, the start address of the kernel decompressor. This should have the same value as the address to which Linux is uploaded in your boot loader. This is normally 32K (0x8000) above the base of RAM. The space between the start of RAM and the kernel is used for page tables.

```
ifeq ((CONFIG_ARCH_XXX),y)
ZTEXTADDR       = 0xXXXX8000
endif
```

`arch/arm/kernel/entry-armv.S`

Machine-specific IRQ functions. You provide the assembly macros disable_fiq, get_irqnr_and_base, and irq_prio_table here. disable_fiq and irq_prio_table is usually empty, but get_irqnr_and_base must be implemented carefully: you should use the zero flag to indicate the presence of interrupts, and put the correct IRQ number in irqnr.

`arch/arm/kernel/debug-armv.S`

These are the low-level debug functions, which talk to a serial port without relying on interrupts or any other kernel functionality. You'll need to use these functions if it won't boot. The functions you need to implement are addruart, senduart and waituart, using ARM assembly. They give you the address of the debug UART, send a byte to the debug UART, and wait for the debug UART, respectively.

`arch/arm/mach-XXX/Makefile`

You need to add a target for your machine, listing the object files in this directory. That will be at least the following:

```
+O_TARGET               := MACHINE.o
+obj-y                  := arch.o irq.o mm.o
```

`arch/arm/mach-XXX/arch.c`

This should contain the architecture-specific fix ups and IO initialisations. (The latter used to go in `arch/arm/mach-XXX/mm.c` but that file is now deprecated). For the meaning of the symbols, refer to *Kernel Basics*.

The setup for your machine is done with a set of macros, starting with MACHINE_START. The parameters you give are filled in to a data structure machine_desc describing the machine. One of the items is the fixup function which, if specified, will be called to fill in or adjust entries dynamically at boot time. This is useful for detecting optional items needed at boot-time (e.g. VRAM in a Risc PC). Note that it should not be used to do main memory size detection, which is is the job of the boot-loader.

```
static void  __init
fixup_XXX(struct machine_desc *desc, struct param_
struct *params,char **cmdline, struct meminfo *mi)
{
    ROOT_DEV = MKDEV(RAMDISK_MAJOR, 0);
    setup_ramdisk(1, 0, 0, CONFIG_BLK_DEV_RAM_SIZE);
    setup_initrd(0xc0800000, X * 1024 * 1024);
}

MACHINE_START(ANAKIN, "XXX")
    MAINTAINER("Acunia N.V.")
    BOOT_MEM(XXX, YYY, ZZZ)
    VIDEO(VVV, WWW)
    FIXUP(fixup_XXX)
    MAPIO(XXX_map_io)
    INITIRQ(genarch_init_irq)
MACHINE_END

static struct map_desc XXX_io_desc[]   __initdata =
  { IO BASE, IO START, IO SIZE, DOMAIN IO, 0,1,0,0,
    LAST_DESC
 };

void   __init
XXX_map_io(void)
{
    iotable_init(XXX_io_desc);
}
```

`arch/arm/mach-XXX/irq.c`

You should provide the XXX_init_irq function here. This sets up the interrupt controller. Interrupt mask and unmask functions go here too.

`arch/arm/mach-XXX/mm.c`

This file is now deprecated. Its content (IO initialisation) has moved into arch/arm/mach-XXX/arch.c

`include/asm/arch/dma.h`

Defines for DMA channels, and DMA-able areas of memory. For machines without DMA, you can just declare 0 DMA channels as follows:

continued on page 58

continued from page 55

```
#define MAX_DMA_ADDRESS         0xffffffff
#define MAX_DMA_CHANNELS        0
```

`include/asm/arch/hardware.h`

In this file, you need to define the memory addresses, IO addresses, and so on, according to your hardware specifications (memory map and IO map). The _START addresses are the physical addresses, the _BASE addresses are the virtual adresses to which each memory or IO region will be mapped. Refer to other similar machines for examples.

`include/asm/arch/io.h`

Here you define the macros `IO_SPACE_LIMIT` (as `0xffffffff`), `__io(addr)`, `__arch_getw(addr)`, `__arch_putw(data,addr)`, and other related macros, according to your CPU. For CPUs that already have an implementation (for example Intel® SA-1110 Processor and Intel® XScale™ Microarchitecture), you can just copy it across and/or reuse the existing io.h for that CPU.

`include/asm/arch/irq.h`

Here you need to provide the fixup_irq macro. In almost all cases, this is just a direct mapping:

```
#define fixup_irq(i)            i
```

`include/asm/arch/irqs.h`

In this file you will define all your IRQ numbers. For example:

```
#define IRQ_UART0               0
```

`include/asm/arch/keyboard.h`

This file is typically here to cheat the VT driver into thinking that there is a null keyboard. Most ARM devices don't have a real one. If you do this is where to put the keyboard IO defines and structs.

`include/asm/arch/memory.h`

Unless you have an exotic memory-map this is platform-invariant and you can copy this from other implementations.

`include/asm/arch/param.h`

This is included by asm/param.h. Here you can redefine HZ (default 100), NGROUPS (default -1), and MAXHOSTNAMELEN (default 64). If you are okay with the above defaults, you still need to create this file but you can make it an empty file (like the Anakin port).

`include/asm/arch/system.h`

This file is included by `arch/arm/kernel/process.c`. You are required to define `arch_idle()` and `arch_reset()` functions. `arch_idle()` is called whenever the machine has no other process to run - i.e. it's time to sleep. `arch_reset()` is called on system reset. The actual implementation of these functions depends on your specific hardware, and there are some subtleties associated with `arch_idle()`. This function will normally put the hardware of your specific device into a low-power mode and then call the generic cpu function `cpu_do_idle` to do the same thing for the cpu. A typical implementation would be as in the listing below, however in Anakin's case this won't work, because the interrupt controller is in the ASIC, and that is clocked by the processor's mclk. Stopping the CPU stops the ASIC as well, which means that a wake-up interrupt will never get generated, so calling `cpu_do_idle` just hangs forever. You need to know about this sort of hardware detail to get a successful port.

```
static inline void arch_idle(void)
{
        /*
        * Please check include/asm/proc-fns.h,
             include/asm/cpu-*.h
        * and arch/arm/mm/proc-*.S.  In particular,
         cpu_do_idle is
        * a macro expanding into cpu_XXX_do_idle, where
         XXX is the
        * CPU configuration, e.g. arm920, sa110, xs80200,
         etc.
        */
        cpu_do_idle(IDLE_WAIT_SLOW);
}

static inline void arch_reset(char mode)
{
        switch (mode) {
        case 's':
             /* Software reset (jump to address 0) */
             cpu_reset(0);
             break;
         case 'h':
             /* TODO: hardware reset */
        }
}
```

`include/asm/arch/time.h`

Here you have to supply your timer interrupt handler and related functions. See the template below:

```
/*
    * XXX_gettimeoffset is not mandatory.  For example,
    anakin has
    * not yet implemented it.  dummy_gettimeoffset (defined
    in
    * arch/arm/kernel/time.c) is the default handler, if
you omit
    * XXX_gettimeoffset.
    */
    static unsigned long XXX_gettimeoffset(void)
    {
        /* Return number of microseconds since last inter-
rupt */
    }

    static void XXX_timer_interrupt(int irq, void *dev_id,
    struct
        pt_regs *regs)
    {
```

```
        /* Add hardware specific stuffs, if applicable */
        do_timer(regs);
        do_profile (regs);
    }

    extern inline void setup_timer(void)
    {
        gettimeoffset = XXX_gettimeoffset;
        timer_irq.handler = XXX_timer_interrupt;
        setup_arm_irq(IRQ_XXX, &timer_irq);
        /* Other hardware specific setups */
    }
```

The do_profile() function is there to allow kernel profiling.

include/asm/arch/timex.h

This file is included by include/asm/timex.h, which is in turn included by include/linux/timex.h. Basically you need to define your clock rate here. For example, for Anakin it is 1/8ms:

```
#define CLOCK_TICK_RATE                 1000 / 8
```

include/asm/arch/uncompress.h

This file is included by arch/arm/boot/compressed/misc.c (which, among other things, outputs the Uncompressing Linux message). You are required to provide two functions, arch_decomp_setup to setup the UART, and puts for outputting the decompression message to the UART:

```
    static void_puts(char char *s)

        /*
        * Hardware-specific routine to put a string to the
          debug
        * UART, converting "\n" to "\n\r" on the way.
        */

    static inline void arch_decomp_setup(void)

        /*
        * Hardware-specific routine to put a string to
          setup the
        * UART mentioned above.
        */

    /* Watchdog is not used for most ARM Linux implemen-
    tations */
    #define arch_decomp_wdog()
```

include/asm/arch/vmalloc.h

This file is largely invariant across platforms, so you can just copy it from other ARM architectures without worrying too much.

If you get all the above right then you should have a bootable compressed kernel for your architecture that can output debug messages though the debug functions. However, this isn't actually much use without some support for the devices in your system. In Anakin's case this is the frame buffer, UARTs under interrupt control, touchscreen and compact flash interface (ide). Console driver functionality is also required to actually interact with these devices. We'll look at how to add these drivers in a future article.

Further Information

A number of resources are also available both on-line and in book format.
The ARM Linux Project – http://www.arm.linux.org.uk/
Linux Console Project – http://sourceforge.net/projects/linuxconsole/
Linux Framebuffer Driver Writing HOWTO – http://www.linux-fbdev.org/HOWTO/

Bibliography

The ARM Linux Project, Russell King, The ARM Linux Project, 2002.

Linux Device Drivers, Alessandro Rubini and Jonathan Corbet, 2nd Edition, 0596000081, O'Reilly & Associates, 2001.

Linux Framebuffer Driver Writing HOWTO, James Simmons, Linux-fbdev.org, 1999.

Linux Console Project, James Simmons, Sourceforge, 2002.

Background

This article is based on a chapter of the second edition of the Aleph One 'Guide to ARMLinux for Developers' book. The first edition is available now and the second edition will be out later this year. www.aleph1.co.uk.armlinux/thebook.html. ❍