Linux iNsOMniA..

SANDIPAN PATRA

11. How to port Linux to a raw ARM board

This document provides a summary of the steps when porting Linux to a new ARM platform or a new processor. In this page we assume that the reader has a knowledge of C and assembly programming and is familiar with ARM and operating systems concepts such as interrupt handling, system calls and memory management.

Some useful information can be found in the kernel documentation section on how to configure the kernel itself. It is probably best to familiarize with Linux on x86 before if you are not familiar with embedded platforms. In addition to Linux kernel source documentation, some notes maintained by community members are available on booting and setting up Linux for ARM platforms, you can consult the following kernel document and Vincent Sander's documentation for additional information.

Porting steps

- Install a cross-development environment.
- Setup the board and ensure that the serial port is working so we can print data through the serial port.
- Download and install the Linux kernel, most of the porting work will be done at this level.
- Add board specific code into the kernel tree.
- Build a kernel image to run on the board
- Test that early kernel printk is working
- Get the real printk working with the serial console.
- For a new board, a new board-specific directory should be added as well as support for interrupt handling, kernel timer services and mapping for memory areas.
- Ethernet drivers are usually the next drivers to focus on as they enable setup of NFS root file system to get access to user utilities and applications.
- Filesystem can be provided in different forms which are listed on LinuxFilesystem

Development environment

Development for embedded platforms is usually done using cross-development tools. In the case of Linux, the host platform is often a Linux host computer connected to the target board using Ethernet, serial and/or ICE/JTAG.

The GNU compiler is required to build the Linux kernel. There is more information on downloading and installing GNU tools is available on LinuxDevTools. Other compilers compliant with the new ABI for the ARM Architecture can also be used to compile user-space applications and some libraries.

Install and build the Linux kernel

Download the latest stable release of the Linux kernel sources from kernel.org and install the source in your local directory. Working from the latest kernel source base is preferred as older versions will not be supported by the community. Later kernel versions usually contain recent bug fixes and enhancements. Older kernels such as 2.4 kernels are no more actively supported.

Adding board specific code

Support code for a new board consist in processor support, board specific code including where devices are mapped etc. and device drivers. Source code for device drivers is based in the *drivers/* directory and *net/* directories. Board specific code should be placed into the *arch/arm/* directory.

The following sections will use ARM Versatile PB926EJ-S as an example and assume that the processor support code already exist. The board intialisation and definition files are located in the *arch/arm/mach-versatile* directory. ARM Versatile board comes in two different form factors, a compact version (Application Baseboard) and a expandable version (Platform Baseboard). This directory contains the following files:

- clock.[c,h] – Functions handling clock settings
- core.[c,h] – IRQ initialisation, definition of initial IO mappings and low-level platform initialisation
- dma.[c,h] – DMA support code and initialisation
- pci.c – PCI setup
- versatile_[ab,pb].c – board initialisation with functions such as versatile_init(), etc.

The *MACHINE_START()* macro defined at the end of *versatile_pb.c* contains the base definition for this platform, using initialisation functions defined in other platform files in this directory.

Header files are located in a similar directory: *include/asm-arm/*. In the case of versatile files are located in *include/asm-arm/arch-versatile/*.

- platform.h – contains register definitions for peripherals and controllers on the board
- debug-macro.S – contains definition of the "addruart" macro called by the *printk()* function to print data on the serial port
- irqs.h – IRQ register definitions
- system.h – contains idle() and reset() functions.
- vmalloc.h – defines constants such as VMALLOC_END.
- …

Connecting it to the main build system

Create a *Kconfig* file for the board in *arch/arm/mach-versatile/* to link this board to the overall build system so that it can be selected when running make *config. Refer to *Documentation/kbuild/kconfig-language.txt* for a description of the syntax.

A Makefile entry need to exist to enable the systen to recurse into the platform sub-directories. Use Makefiles from existing boards as starting points and modify as appropriate.

Platform ID
Finally, add your machine group and type ID. A unique identifier can be obtained by registering your machine at http://www.arm.linux.org.uk/developer/machines/. You could initially assign a local ID in *arch/arm/tools/mach-types* while developing your BSP locally and then register it to get an official ID. You must be careful when changing kernel version as new machine are regularly added and make sure this doesn't conflict with the ID you have chosen.

This ID is used very early in the boot process to determine the platform type and pull in the correct initialisation functions. It is used by the boot loader and stored in banked registers (i.e. *r1*) before the bootloader hands over to the kernel.

Device drivers

New device driver should be added in the appropriate directories, *net/* for network drivers or *drivers/* for others. Makefiles and Kconfig files also need to be edited (or created) so they are linked to the rest of the system. The following book provide a good guide on how to develop new device drivers with Linux.

If the board contains controllers or devices for which Linux already has support for, we only need to include these in Kconfig files and pass the correct parameters to these drivers when initialisation is done.

Configure and build a kernel image

Ensure that the cross-toolchain program names are correct and accessible from your execution path (i.e. your PATH environment variable).

To configure the kernel you can use some of the following command:

- make xconfig
- make menuconfig
- make gconfig
- …

These commands assume that you have a cross-compiler accessible on your execution path.

- If support is set to be EXPERIMENTAL, select 'Prompt for development and/or incomplete code/drivers' under 'Code maturity level options'.
- In System Type, select VERSATILE and correct subtype if this is a family of boards.
- Select the right CPU under 'CPU selection' (in the case of Versatile there is one CPU currently for this type of board).
- In the 'Device Drivers' -> 'Character devices' select the appropriate device driver (ARM_AMBA_PL011 in the case of Versatile).
- In 'Boot options' you can also enter your CMDLINE boot parameters. This will only be used as default in case the bootloader doesn't provide any command line.

For other options you can either use the default or select 'No'. The resulting configuration file is stored as *.config* in the top level directory.

We can now type *make dep* and *make* to build a kernel image. If successful, the resulting image is placed in the *arch/arm/boot directory*. You can either build a compressed image (*zImage*) or a raw binary image (*Image*). The ELF file containing debug information (if selected in the Makefile) is called *vmlinux* and is at the top-level directory. When loading a kernel image onto the board using a debugger, you should use either *zImage* or *Image* files. The *vmlinux* image can be loaded to perform symbolic debugging.

Kernel boot

Early signs of a working Linux kernel come from the output of *printk()*, which is routed directly to the first console. Having configured a serial console, we should be able to see some kernel messages on the serial port if it has been setup correctly. Note that often the bootmonitor from the board or the boot loader initialises the serial device, however the kernel should not necessarily rely on this.

The setup of the serial console happens much later during the kernel initialisation process. Chances are your new kernel probably dies even before that which is where early printk are useful. printk() allows you to print text as early as the first line of C code. The early initialisation steps are listed in *init/main.c:start_kernel()*. From this you can see that the console_init() call is done later on after we already had calls to *printk()*.

Early *printk()* is very useful, however the serial driver should be working to enable further debug. For Versatile platforms, the serial driver is in *drivers/serial/amba-pl011.c*.

Debug with kgdb

KGDB enables developers to debug the kernel while running and set breakpoints or do single stepping at source code level. If available, you can use two serial ports on the target, one used for KGDB communication and the other to print messages. You should use a crossover-cable (or null-modem) to connect it to your development host. Doing everything using one serial port is also possible.

You should select the Remote GDB kernel debugging which is listed under Kernel hacking when you configure the kernel and then do a 'make clean' and recompile the kernel to make sure that debugging symbols are compiled into the generated Linux image.

Run the new kernel image. At this stage the kernel will be waiting for a gdb client connection. If there is a cross-over serial cable connected between the serial port on the target and the host, you can then set the appropriate baud rate and start the gdb on your host:

- stty -F /dev/ttyS0 38400
- arm-none-linux-gnueabi-gdb vmlinux

At the gdb prompt, type: *target remote /dev/ttyS0*

If successful, you should be talking to the kernel through KGDB.

Interrupt handling

When porting the kernel to a new board it often hangs in the calibration step. This is normal if the interrupt code is not yet implemented in which case *jiffies* are never updated. Before adding support code for interrupt handling you need to identify interrupt controllers, all interrupt sources and how they are routed.

The following steps are involved to handle interrupts:

- *include/asm-arm/arch-versatile/entry-macro.S*: this file contains a macro called get_irqnr defined used to determine the interrupt number when an interrupt is generated. This function is board specific.

- *arch/arm/kernel/entry-armv.S*: this file contains the definition of the irq_handler() which in turn calls the *asm_do_IRQ()* function which handles all hardware IRQs.

- *arch/arm/kernel/irq.c*: this file contains most of the code handling interrupts including the definition of *asm_do_IRQ()* as well as enabling or disabling specific interrupts numbers. Individual IRQ handlers are invoked by the *__do_irq()* function.

A descriptor structure (see irqdesc structure in *include/asm-arm/mach/irq.h*) is associated with each registered IRQ source. This structure contains pointers to functions declared by device drivers to handle individual interrupts.

System time and timer

Linux relies on a system timer to advance the tick count, *jiffies*. Without this, Linux won't run and will stop in *calibrate_delay()* during the startup process as *jiffies* will never be incremented. Access to a real-time clock (RTC) device is also required to obtain the calendar date and time when the kernel boots up.

ARM Versatile board includes primecell SP804 which provide facilities for periodic timers. One of the timers is used to generate system ticks. This is done at the end of *arch/arm/mach-versatile/core.c* with the definition and initialisation of the versatile_timer structure.

Comments RSS
Trackback are closed Comments (1)

1. gaillRasp
   February 22nd, 2011
   QUOTE
   Nice site ….)