Linux Kernel

Peripheral Devices for Embedded Systems

Rafal Kapela

June 26, 2016

Outline

Outline

Cross-compiling the kernel

When you compile a Linux kernel for another CPU architecture

- Much faster than compiling natively, when the target system is much slower than your GNU/Linux workstation.
- Much easier as development tools for your GNU/Linux workstation are much easier to find.
- To make the difference with a native compiler, cross-compiler executables are prefixed by the name of the target system, architecture and sometimes library. Examples:
  mips-linux-gcc, the prefix is mips-linux-
  arm-linux-gnueabi-gcc, the prefix is arm-linux-gnueabi-

Specifying cross-compilation (1)

The CPU architecture and cross-compiler prefix are defined through the ARCH and CROSS_COMPILE variables in the toplevel Makefile.

- ARCH is the name of the architecture. It is defined by the name of the subdirectory in arch/ in the kernel sources
  - Example: arm if you want to compile a kernel for the arm architecture.
- CROSS_COMPILE is the prefix of the cross compilation tools
  - Example: arm-linux- if your compiler is arm-linux-gcc

Specifying cross-compilation (2)

Two solutions to define ARCH and CROSS_COMPILE:

Pass ARCH and CROSS_COMPILE on the make command line:

make ARCH=arm CROSS_COMPILE=arm-linux- ...

Drawback: it is easy to forget to pass these variables when you run any make command, causing your build and configuration to be screwed up.

Define ARCH and CROSS_COMPILE as environment variables:

export ARCH=arm; export CROSS_COMPILE=arm-linux-

Drawback: it only works inside the current shell or terminal. You could put these settings in a file that you source every time you start working on the project. If you only work on a single architecture with always the same toolchain, you could even put these settings in your /.bashrc file to make them permanent and visible from any terminal.

Specifying cross-compilation (2)

Two solutions to define ARCH and CROSS_COMPILE:

Pass ARCH and CROSS_COMPILE on the make command line:

```
make ARCH=arm CROSS_COMPILE=arm-linux- ...
```

Drawback: it is easy to forget to pass these variables when you run any make command, causing your build and configuration to be screwed up.

Define ARCH and CROSS_COMPILE as environment variables:

```
export ARCH=arm; export CROSS_COMPILE=arm-linux-
```

Drawback: it only works inside the current shell or terminal. You could put these settings in a file that you source every time you start working on the project. If you only work on a single architecture with always the same toolchain, you could even put these settings in your /.bashrc file to make them permanent and visible from any terminal.

Predefined configuration files

- Default configuration files available, per board or per-CPU family
    - They are stored in arch/*arch*/configs/, and are just minimal .config files
    - This is the most common way of configuring a kernel for embedded platforms
- Run make help to find if one is available for your platform
- To load a default configuration file, just run make acme_defconfig
    - This will overwrite your existing .config file!
- To create your own default configuration file
    - make savedefconfig, to create a minimal configuration file
    - mv defconfig arch/*arch*/configs/myown_defconfig

Configuring the kernel

- After loading a default configuration file, you can adjust the configuration to your needs with the normal xconfig, gconfig or menuconfig interfaces
- You can also start the configuration from scratch without loading a default configuration file
- As the architecture is different from your host architecture
  - Some options will be different from the native configuration (processor and architecture specific options, specific drivers, etc.)
  - Many options will be identical (filesystems, network protocols, architecture-independent drivers, etc.)
- Make sure you have the support for the right CPU, the right board and the right device drivers.

Device Tree

- Many embedded architectures have a lot of non-discoverable hardware.
- Depending on the architecture, such hardware is either described using C code directly within the kernel, or using a special hardware description language in a *Device Tree*.
- ARM, PowerPC, OpenRISC, ARC, Microblaze are examples of architectures using the Device Tree.
- A *Device Tree Source*, written by kernel developers, is compiled into a binary *Device Tree Blob*, passed at boot time to the kernel.
    - There is one different Device Tree for each board/platform supported by the kernel, available in arch/arm/boot/dts/*board*.dtb.
- The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.

Building and installing the kernel

- Run make
- Copy the final kernel image to the target storage
  - can be uImage, zImage, vmlinux, bzImage in arch/*arch*/boot
  - copying the Device Tree Blob might be necessary as well, they are available in arch/*arch*/boot/dts
- make install is rarely used in embedded development, as the kernel image is a single file, easy to handle
  - It is however possible to customize the make install behaviour in arch/*arch*/boot/install.sh
- make modules_install is used even in embedded development, as it installs many modules and description files
  - make INSTALL_MOD_PATH=*dir*/ modules_install
  - The INSTALL_MOD_PATH variable is needed to install the modules in the target root filesystem instead of your host root filesystem.

Booting with U-Boot

- U-Boot requires a special kernel image format: uImage
  - uImage is generated from zImage using the mkimage tool. It is done automatically by the kernel make uImage target.
  - On some ARM platforms, make uImage requires passing a LOADADDR environment variable, which indicates at which physical memory address the kernel will be executed.
- In addition to the kernel image, U-Boot can also pass a *Device Tree Blob* to the kernel.
- The typical boot process is therefore:
  1. Load uImage at address X in memory
  2. Load *board*.dtb at address Y in memory
  3. Start the kernel with bootm X - Y (the - in the middle indicates no *initramfs*)

Kernel command line

- In addition to the compile time configuration, the kernel behaviour can be adjusted with no recompilation using the **kernel command line**
- The kernel command line is a string that defines various arguments to the kernel
  - It is very important for system configuration
  - root= for the root filesystem (covered later)
  - console= for the destination of kernel messages
  - and many more, documented in kernel-parameters.txt in the kernel sources
- This kernel command line is either
  - Passed by the bootloader. In U-Boot, the contents of the bootargs environment variable is automatically passed to the kernel
  - Built into the kernel, using the CONFIG_CMDLINE option.

Outline

Advantages of modules

- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).
- Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later.
- Caution: once loaded, have full control and privileges in the system. No particular protection. That's why only the root user can load and unload modules.

Module dependencies

- Some kernel modules can depend on other modules, which need to be loaded first.
- Example: the usb-storage module depends on the scsi_mod, libusual and usbcore modules.
- Dependencies are described in /lib/modules/*kernel-version*/modules.dep
  This file is generated when you run make modules_install.

Kernel log

When a new module is loaded, related information is available in the kernel log.

- The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)
- Kernel log messages are available through the dmesg command (**d**iagnostic **mes**sa**g**e)
- Kernel log messages are also displayed in the system console (console messages can be filtered by level using the loglevel kernel parameter, or completely disabled with the quiet parameter).
- Note that you can write to the kernel log from userspace too:
  echo "Debug info" > /dev/kmsg

Module utilities (1)

- modinfo *module_name*
  modinfo *module_path*.ko
  Gets information about a module: parameters, license, description and dependencies.
  Very useful before deciding to load a module or not.
- sudo insmod *module_path*.ko
  Tries to load the given module. The full path to the module object file must be given.

Understanding module loading issues

- When loading a module fails, insmod often doesn't give you enough details!
- Details are often available in the kernel log.
- Example:

```
$ sudo insmod ./intr_monitor.ko
insmod: error inserting './intr_monitor.ko': -1 Device or resource busy
$ dmesg
[17549774.552000] Failed to register handler for irq channel 2
```

Module utilities (2)

- sudo modprobe *module_name*
  Most common usage of modprobe: tries to load all the modules the given module depends on, and then this module. Lots of other options are available. modprobe automatically looks in /lib/modules/*version*/ for the object file corresponding to the given module name.
- lsmod
  Displays the list of loaded modules
  Compare its output with the contents of /proc/modules!

Module utilities (3)

- sudo rmmod *module_name*
  Tries to remove the given module.
  Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)
- sudo modprobe -r *module_name*
  Tries to remove the given module and all dependent modules (which are no longer needed after removing the module)

Passing parameters to modules

- Find available parameters:
  modinfo snd-intel8x0m
- Through insmod:
  sudo insmod ./snd-intel8x0m.ko index=-2
- Through modprobe:
  Set parameters in /etc/modprobe.conf or in any file in /etc/modprobe.d/:
  options snd-intel8x0m index=-2
- Through the kernel command line, when the driver is built statically into the kernel:
  snd-intel8x0m.index=-2
  - snd-intel8x0m is the *driver name*
  - index is the *driver parameter name*
  - -2 is the *driver parameter value*

Outline

Hello Module 1/3

```
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
  pr_alert("Good morrow");
  pr_alert("to this fair assembly.\n");
  return 0;
}

...
```

Hello Module 2/3

```
...

static void __exit hello_exit(void)
{
  pr_alert("Alas, poor world, what treasure");
  pr_alert("hast thou lost!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR("William Shakespeare");
```

Hello Module 3/3

- __init
  - removed after initialization (static kernel or module.)
- __exit
  - discarded when module compiled statically into the kernel.

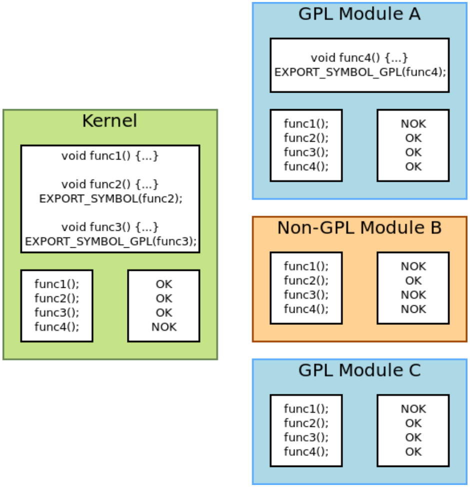
Hello Module Explanations

- Headers specific to the Linux kernel: linux/xxx.h
  - No access to the usual C library, we're doing kernel programming
- An initialization function
  - Called when the module is loaded, returns an error code (0 on success, negative value on failure)
  - Declared by the module_init macro: the name of the function doesn't matter, even though *modulename*_init() is a convention.
- A cleanup function
  - Called when the module is unloaded
  - Declared by the module_exit macro.
- Metadata information declared using MODULE_LICENSE, MODULE_DESCRIPTION and MODULE_AUTHOR

Symbols Exported to Modules 1/2

- From a kernel module, only a limited number of kernel functions can be called
- Functions and variables have to be explicitly exported by the kernel to be visible from a kernel module
- Two macros are used in the kernel to export functions and variables:
  - EXPORT_SYMBOL(symbolname), which exports a function or variable to all modules
  - EXPORT_SYMBOL_GPL(symbolname), which exports a function or variable only to GPL modules
- A normal driver should not need any non-exported function.

Symbols exported to modules 2/2


Kernel
void func1() {...}
void func2() {...}
EXPORT_SYMBOL(func2);
void func3() {...}
EXPORT_SYMBOL_GPL(func3);
func1(); func2(); func3(); func4();
OK OK OK NOK
GPL Module A
void func4() {...}
EXPORT_SYMBOL_GPL(func4);
func1(); func2(); func3(); func4();
NOK OK OK OK
Non-GPL Module B
func1(); func2(); func3(); func4();
NOK OK NOK NOK
GPL Module C
func1(); func2(); func3(); func4();
NOK OK OK OK

Module License

- Several usages
  - Used to restrict the kernel functions that the module can use if it isn't a GPL licensed module
    - Difference between EXPORT_SYMBOL and EXPORT_SYMBOL_GPL
  - Used by kernel developers to identify issues coming from proprietary drivers, which they can't do anything about ("Tainted" kernel notice in kernel crashes and oopses).
  - Useful for users to check that their system is 100% free (check /proc/sys/kernel/tainted)
- Values
  - GPL compatible (see include/linux/license.h: GPL, GPL v2, GPL and additional rights, Dual MIT/GPL, Dual BSD/GPL, Dual MPL/GPL)
  - Proprietary

Compiling a Module

- Two solutions
  - *Out of tree*
    - When the code is outside of the kernel source tree, in a different directory
    - Advantage: Might be easier to handle than modifications to the kernel itself
    - Drawbacks: Not integrated to the kernel configuration/compilation process, needs to be built separately, the driver cannot be built statically
  - Inside the kernel tree
    - Well integrated into the kernel configuration/compilation process
    - Driver can be built statically if needed

Modules and Kernel Version

- To be compiled, a kernel module needs access to the kernel headers, containing the definitions of functions, types and constants.
- Two solutions
  - Full kernel sources
  - Only kernel headers (linux-headers-* packages in Debian/Ubuntu distributions)
- The sources or headers must be configured
  - Many macros or functions depend on the configuration
- A kernel module compiled against version X of kernel headers will not load in kernel version Y
  - modprobe / insmod will say Invalid module format

New Driver in Kernel Sources 1/2

- To add a new driver to the kernel sources:
    - Add your new source file to the appropriate source directory. Example: drivers/usb/serial/navman.c
    - Single file drivers in the common case, even if the file is several thousand lines of code big. Only really big drivers are split in several files or have their own directory.
    - Describe the configuration interface for your new driver by adding the following lines to the Kconfig file in this directory:

```
config USB_SERIAL_NAVMAN
        tristate "USB Navman GPS device"
        depends on USB_SERIAL
        help
          To compile this driver as a module, choose M
          here: the module will be called navman.
```

New Driver in Kernel Sources 2/2

- Add a line in the Makefile file based on the Kconfig setting: obj-$(CONFIG_USB_SERIAL_NAVMAN) += navman.o
- It tells the kernel build system to build navman.c when the USB_SERIAL_NAVMAN option is enabled. It works both if compiled statically or as a module.
  - Run make xconfig and see your new options!
  - Run make and your new files are compiled!
  - See kbuild/ for details and more elaborate examples like drivers with several source files, or drivers in their own subdirectory, etc.

Hello Module with Parameters 1/2

```
/* hello_param.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>

MODULE_LICENSE("GPL");

/* A couple of parameters that can be passed in:
how many times we say hello, and to whom */
static char *whom = "world";
module_param(whom, charp, 0);
static int howmany = 1;
module_param(howmany, int, 0);
```

Hello Module with Parameters 2/2

```
static int __init hello_init(void)
{
  int i;
  for (i = 0; i < howmany; i++)
    pr_alert("(%d) Hello, %s\n", i, whom);
  return 0;
}
static void __exit hello_exit(void)
{
  pr_alert("Goodbye, cruel %s\n", whom);
}
module_init(hello_init);
module_exit(hello_exit);
```

Declaring a module parameter

```
#include <linux/moduleparam.h>
module_param(
 name, /* name of an already defined variable */
 type, /* either byte, short, ushort, int, uint...
          charp, or bool. (checked at run time!) */
 perm  /* for /sys/module/<module_name>/
    parameters/<param>,
          0: no such module parameter value file */
);
/* Example */
int irq=5;
module_param(irq, int, S_IRUGO);
```

Modules parameter arrays are also possible with module_param_array, but they are less common.

Resources

If you want to gain some knowledge by your own...

- Wikipedia – Embedded system
  http://en.wikipedia.org/wiki/Embedded_system
- Greg Kroah-Hartman, O'Reilly, Linux Kernel in a Nutshell, Dec 2006.
  http://www.kroah.com/lkn/
- Free Electrons - embedded Linux experts
  http://free-electrons.com/

Questions ?

Rafal Kapela
rafal.kapela@put.poznan.pl