嵌入式Linux驱动开发

目录

[第1章 Linux设备与驱动模型 2](#_Toc23828)

**[1](#_Toc18257)**  [3](#_Toc18257)

[1.1 Bus Core Drivers 3](#_Toc11778)

[1.2 总线驱动 3](#_Toc12707)

[1.3 Bus Controller Drivers 5](#_Toc18386)

[1.4 总线控制器驱动 5](#_Toc2429)

[1.5 Device Drivers 5](#_Toc16775)

[1.6 设备驱动 5](#_Toc17718)

[1.7 Introduction to the Device Tree 7](#_Toc3572)

[1.8 设备树 7](#_Toc29383)

[第2章 最简驱动程序 11](#_Toc6104)

[1.1 许可证 13](#_Toc15351)

[实验3.1: “hello world”模块 13](#_Toc11225)

[1.2 helloworld\_imx.ko演示 15](#_Toc17633)

[实验3.2: "带参数的helloworld"模块 15](#_Toc1903)

[1.3 helloworld\_imx\_with\_parameters.ko演示 16](#_Toc32709)

[实验3.3: "helloworld timing"模块 16](#_Toc23541)

[1.4 helloworld\_imx\_with\_timing.ko演示 17](#_Toc10303)

# Linux设备与驱动模型

Understanding the Linux device and driver model is central to developing device drivers in Linux. A unified device model was added in Linux kernel 2.6 to provide a single mechanism for representing devices and describing their topology in the system. The Linux device and driver model is an universal way of organizing devices and drivers into buses. Such a system provides several benefits:

理解Linux设备与驱动模型对Linux设备驱动开发至关重要。Linux内核2.6版本引入的统一设备模型为描述设备及其拓扑结构提供了机制。Linux设备和驱动模型以统一的方式将设备与驱动组织到总线上。这样的系统有如下优点：

* Minimization of code duplication.
* 减少代码冗余。
* Clean code organization with the device drivers separated from the controller drivers, the hardware description separated from the drivers themselves, etc.
* 代码结构清晰，设备驱动与控制器驱动相互独立，设备描述从驱动中剥离。
* Capability to determine all the devices in the system, view their status and power state, see what bus they are attached to and determine which driver is responsible for them.
* 能够查看系统中所有设备以及对应的状态和功耗。查看设备连接到哪个总线并决定使用哪个驱动。
* The capability to generate a complete and valid tree of the entire device structure of the system, including all buses and interconnections.
* 能够为系统中的所有设备构造一个完整有效的树状结构，包括所有的总线与互联。
* The capability to link devices to their drivers and vice versa.
* 提供了将设备和驱动互相关联的能力。
* Categorization of devices by their type (classes), such as input devices, without the need to understand the physical device topology.
* 将设备从具体的拓扑结构中抽象出来，根据类型来划分，比如输入设备。

The device model involves terms like "device", "driver", and "bus":

设备模型的涉及以下device、driver、bus等术语：

* **device**: a physical or virtual object which attaches to a bus.
* **设备**：连接到总线上的物理或者虚拟对象。
* **driver**: a software entity which may probe for and be bound to devices, and which can perform certain management functions.
* **驱动**：负责探测并驱动设备的代码实体，也可以执行部分管理功能。
* **bus**: a device which serves as an attachment point for other devices.
* **总线**：为其他设备提供接入点的设备。

The device model is organized around three main data structures:

设备模型围绕3个主要的数据结构组织：

1. The struct bus\_type structure, which represent one type of bus (e.g.; USB, PCI, I2C).
2. struct bus\_type用来描述一种类型的总线(比如USB， PCI， I2C)。
3. The struct device\_driver structure, which represents one driver capable of handling certain devices on a certain bus.
4. struct device\_driver描述了能够处理特定总线上的特定设备的驱动程序。
5. The struct device structure, which represents one device connected to a bus.
6. struct device描述连接在总线上的一个具体设备。

## Bus Core Drivers

## 总线驱动

For each bus supported by the kernel there is a generic bus core driver. A bus is a channel between the processor and one or more devices. For the purposes of the device model, all devices are connected via a bus, even if it is an internal, virtual, "platform" bus.

每个内核支持的总线都有一个对应的通用总线驱动。总线是处理器与设备之间的连接通道。为了统一设备驱动模型，所有的设备都通过总线连接，哪怕是抽象的platform总线。

The bus core driver allocates a struct bus\_type and registers it with the kernel's list of bus types. The struct bus\_type structure defined in include/linux/device.h represents one type of bus (USB, PCI, I2C, etc.). The registration of the bus in a system is done using the bus\_register() function. The struct bus\_type is defined as:

总线驱动会分配一个struct bus\_type结构，然后把这个结构注册到内核的总线类型列表中。struct bus\_type结构体定义在include/linux/device.h中，用来表示一类总线(USB，PCI，I2C等)。把总线注册到系统中是通过调用通过调用bus\_register()函数来完成的。struct bus\_type的定义如下：

（原书P74页代码1）

An example of struct bus\_type instantiation and bus registration is shown in the following code extracted from the platform core driver (drivers/base/platform.c):

下面的代码是一个初始化struct bus\_type并注册的例子，截取自platform驱动(drivers/base/platform.c)：

（原书P74页代码2）

（原书P75页代码1）

One of the struct bus\_type members is a pointer to the struct subsys\_private defined in drivers/base/base.h as:

struct bus\_type结构中的成员变量p指向subsys\_private结构，该结构定义在drivers/base/base.h：

（原书P75页代码2）

The klist\_devices member of the struct subsys\_private is a list of devices in the system that reside on this particular type of bus. This list is updated by the device\_register() function, which is called when the bus is scanned for devices by the bus controller driver (during initialization or when a device is hot plugged).

struct subsys\_private结构中的klist\_devices变量以链表的方式维护了系统中所有关联到这一类型总线上的设备。总线控制器驱动扫描总线上的设备的时候会调用device\_register()函数来更新该链表(在系统初始化或者设备热插入的时候触发)。

The klist\_drivers member of the struct subsys\_private is a list of drivers that can handle devices on that bus. This list is updated by the driver\_register() function, which is called when a driver initializes itself.

struct subsys\_private结构中的klist\_drivers则以链表的方式维护了所有能够处理该总线设备的驱动程序。当驱动初始化的时候通过调用driver\_register()函数来更新该列表。

When a new device is plugged into the system, the bus controller driver detects the device and calls device\_register(). When a device is registered by the bus controller driver, the parent member of the struct device is pointed to the bus controller device to build the physical device list. The list of drivers associated with the bus is iterated over to find out if there are any drivers that can handle the device. The match function provided in the struct bus\_type structure is used to check if a given driver can handle a given device. When a driver is found that can handle the device, the driver member of the struct device is pointed to the corresponding device driver.

当新设备插入系统时，总线控制器驱动会侦测到该设备并调用device\_register()函数。当总线控制器驱动注册一个设备时struct device结构的parent成员被设置为总线控制器设备，用于构造总线上的设备列表。总线上关联的驱动程序会被依次遍历以查找是否有合适的驱动来支持该设备。struct bus\_type结构提供的match函数就是用来检查一个特定的驱动是否能够支持一个给定的设备。当一个能够支持该设备的驱动被找到后，struct device结构的driver成员变量就会被设置为相应的驱动。

When a kernel module is inserted into the kernel and the driver calls driver\_register(), the list of devices associated with the bus is iterated over to find out if there are any devices that the driver can handle using the match function. When a match is found, the device is associated with the device driver and the driver´s probe() function is called, this is what we call **binding**.

当一个内核模块被插入到内核并且相应的驱动调用了driver\_register()函数之后，关联总线上的设备列表会被依次遍历，通过调用match函数来确定是否有设备能够被该驱动所支持。如果查找到这样的匹配，设备就会和该驱动关联，驱动的probe函数也会被调用，这就是我们俗称的绑定。

When does a driver attempt to bind a device?

驱动什么时候会尝试去绑定一个设备呢？

1. When the driver is registered (if the device already exits).
2. 驱动被注册的时候(如果设备已经存在)。
3. When the device is created (if the driver is already registered).
4. 设备被创建的时候(如果驱动已经注册到系统中)。

Summarizing, the bus driver registers a bus in a system and:

总的来说，总线驱动负责在系统中注册总线类型，然后：

1. Allows registration of bus controller drivers, whose role is to detect devices, and configure their resources.
2. 允许总线控制器驱动的注册，该驱动的主要职责包括发现设备，配置资源。
3. Allows registration of device drivers.
4. 允许设备驱动的注册。
5. Matches devices and drivers.
6. 负责设备与驱动的匹配。

## Bus Controller Drivers

## 总线控制器驱动

For a specific bus type there could be many different controllers provided by different vendors. Each of these controllers needs a corresponding bus controller driver. The role of a bus controller driver in maintenance of the device model, is similar to that of any other device driver in that, it registers itself to its bus using the driver\_register() function. In most cases, these bus controller devices are autonomous entities in the system discovered during the kernel initialization calling of\_platform\_populate(), which walks through the DT finding and registering these "platform controller devices" to the platform bus at runtime.

对于一个特定的总线类型，系统中可能存在来自不同供应商提供的多个控制器。这些不同的控制器需要各自对应的总线控制器驱动。和其他驱动一样，总线控制器驱动通过driver\_register函数将自己注册到对应的总线上并负载管理和维护子设备。大多数时候，在系统初始化阶段，通过of\_platform\_populate函数，总线控制器设备会自动被发现。of\_platform\_populate通过遍历设备树发现并注册这些设备到platform总线上。

## Device Drivers

## 设备驱动

Every device driver registers itself with the bus core driver using driver\_register(). After that, the device model core tries to bind it with a device. When a device that can be handled by a particular driver is detected, the probe() member of the driver is called and the device configuration data can be retrieved from the Device Tree.

设备驱动通过调用driver\_register和总线关联。然后系统会尝试将新注册的驱动与设备绑定。当一个与驱动相匹配的设备被发现后，驱动的proble函数会被调用，设备相关的配置信息可以通过设备树获取。

Each device driver is responsible for instantiating and registering an instance of the struct device\_driver (defined in include/linux/device.h) with the device model core. The struct device\_driver is defined as:

设备驱动负责初始化和注册device\_driver(定义在include/linux/device.h)到设备子系统。device\_driver的定义如下：

struct device\_driver {

const char \*name;

struct bus\_type \*bus;

struct module \*owner;

const char \*mod\_name;

bool suppress\_bind\_attrs;

const struct of\_device\_id \*of\_match\_table;

const struct acpi\_device\_id \*acpi\_match\_table;

int (\*probe) (struct device \*dev);

int (\*remove) (struct device \*dev);

void (\*shutdown) (struct device \*dev);

int (\*suspend) (struct device \*dev, pm\_message\_t state);

int (\*resume) (struct device \*dev);

const struct attribute\_group \*\*groups;

const struct dev\_pm\_ops \*pm;

struct driver\_private \*p;

};

* The bus member is a pointer to the struct bus\_type to which the device driver is registered.
* bus指向bus\_type结构，用来标记驱动注册到哪个总线上面。
* The probe member is a callback function that is called for each device detected that is supported by the driver. The driver should instantiate itself for each device and initialize the device as well.
* probe是一个回调函数，驱动支持的设备被发现的时候会调用该函数。驱动负责初始化自身以及具体的设备。
* The remove member is a callback function that is called to unbind the driver from the device. This happens when the device is physically removed, when the driver is unloaded, or when the system is shutdown.
* remove函数用来将设备和驱动解绑。在设备移除，驱动卸载或者系统关闭的时候会被调用。

The Linux device model is illustrated in the following figure:



## Introduction to the Device Tree

## 设备树

The "Open Firmware Device Tree", or simply Device Tree (DT), is a data structure and language for describing hardware. More specifically, it is a description of hardware that is readable by an operating system so that the operating system doesn't need to hard code details of the machine.

开放固件设备树，简称设备树，是一种用于描述硬件的语言。操作系统通过对设备树的读取可以避免在代码层面引入过多的硬件细节。

Structurally, the DT is a tree with named **nodes**, and nodes may have an arbitrary number of named **properties** encapsulating arbitrary data. A mechanism also exists to create arbitrary links from one node to another outside of the natural tree structure.

从结构上看，设备树是由带名字的节点组成的，节点可以包含任意多个属性，属性拥有名字并且可以包含任何数据。除了树形结构之外，设备树还提供了为任意两个节点创建链接的机制。

Conceptually, a common set of usage conventions, called "bindings", is defined for how data should appear in the tree to describe typical hardware characteristics including data busses, interrupt lines, GPIO connections, and peripheral devices. As much as possible, hardware is described using existing bindings to maximize use of existing support code, but since property and node names are simply text strings, it is easy to extend existing bindings or create new ones by defining new nodes and properties.

Binding定义了具体的硬件特性的描述规范，包括数据总线，中断，GPIO，外设。对于硬件的描述应该尽可能的通过binding来实现以实现代码重用。属性和名字都是纯文本，可以通过扩展现有的binding来穿件或定义新的节点和属性。

The DT is represented as a set of text files in the Linux kernel source tree. They are located under arch/arm/boot/dts/ and can have two extensions:

设备树在内核代码中通过一组文本文件描述。在arch/arm/boot/dts/中可以找到两种文件类型:

* \***.dtsi** files are device tree source include files. They describe hardware that is common to several platforms which include these files on their \*.dts files.
* \*.dtsi文件是设备树的头文件。用来描述多个平台共用的硬件结构并被包含在相应平台的dts文件中。
* \***.dts** files are device tree source files. They describe one specific platform.
* \*.dts文件是设备树源文件。他们描述了一个具体平台的硬件信息。

Linux uses DT data for three major purposes:

Linux中使用设备树主要有3个目的：

1. **Platform Identification**: the kernel will use data in the DT to identify the specific machine. In a perfect world, the specific platform shouldn't matter to the kernel because all platform details would be described perfectly by the device tree in a consistent and reliable manner. Hardware is not perfect though, and so the kernel must identify the machine during early boot so that it has the opportunity to run machine-specific fixups. In the majority of cases, the machine identity is irrelevant, and the kernel will instead select setup code based on the machine's core CPU or SoC. On ARM, for example, setup\_arch() in arch/arm/kernel/setup.c will call setup\_machine\_fdt() in arch/arm/kernel/devtree.c which searches through the machine\_desc table and selects the machine\_desc which best matches the device tree data. It determines the best match by looking at the compatible property in the root device tree node, and comparing it with the dt\_compat list in struct machine\_desc, which is defined in arch/arm/include/asm/mach/arch.h.
2. 平台区分：内核会通过设备树中的信息来识别机器类型。理想情况下，内核代码应该与具体的平台无关，平台的具体细节统一通过设备树描述。现实情况并没有那么完美，内核必须在启动早期识别出具体的机型，并执行相应机型的特有的代码。大多数时候，机型并不重要，内核根据机型的CPU或者Soc来选择执行相应的代码。拿ARM为例，arch/arm/kernel/setup.c中的setup\_arch函数会调用setup\_machine\_fdt()(位于arch/arm/kernel/devtree.c)。该函数会查找machine\_desc表并选择与设备树最为匹配的machine\_des。通过比较设备树根节点的compatible属性和machine\_desc(定义在arch/arm/include/asm/mach/arch.h)的dt\_compat列表来选择最佳匹配。

The compatible property contains a sorted list of strings starting with the exact name of the machine. For example, the sama5d2.dtsi file under arch/arm/boot/dts folder includes the following compatible property:

compatible属性包含了以机器名开始的一组有序字符串。arch/arm/boot/dts目录下的sama5d2.dtsi包含了如下的compatible属性:

compatible = "atmel,sama5d2";

Again on ARM, for each machine\_desc, the kernel looks to see if any of the dt\_compat list entries appears in the compatible property. If one does, then that machine\_desc is a candidate for driving the machine. See, for example, the sama5\_alt\_dt\_board\_compat[] and DT\_MACHINE\_START declarations in arch/arm/mach-at91/sama5.c. They are used to populate a struct machine\_desc.

在ARM机型上，对于每一个machine\_des，内核会检查其dt\_compat列表中的条目是否包含了compatible属性。如果有的话，对应的machine\_desc就会作为决定目标机型的一个候选。以arch/arm/mach-at91/sama5.c文件中的sama5\_alt\_dt\_board\_compat[]和DT\_MACHINE\_START为例。他们被用来组装成一个machine\_desc结构.

static const char \*const sama5\_alt\_dt\_board\_compat[] \_\_initconst = {

"atmel,sama5d2",

"atmel,sama5d4",

NULL

};

DT\_MACHINE\_START(sama5\_alt\_dt, "Atmel SAMA5")

/\* Maintainer: Atmel \*/

.init\_machine = sama5\_dt\_device\_init,

.dt\_compat = sama5\_alt\_dt\_board\_compat,

.l2c\_aux\_mask = ~0UL,

MACHINE\_END

After searching the entire table of machine\_descs, the setup\_machine\_fdt() function returns the "most compatible" machine\_desc based on which entry in the compatible property each machine\_desc matches against. If no matching machine\_desc is found, then it returns NULL. The function setup\_machine\_fdt() is also responsible for early scanning of the device tree after selecting machine\_desc.

当完整的machine\_descs表被检索完成之后，setup\_machine\_fdt()函数返回一个最佳匹配的machine\_desc。比较的依据在于machine\_desc与compatible属性的哪一个entry相匹配。如果没有匹配的machine\_desc，那么该函数返回NULL。在选择了machine\_desc之后，setup\_machine\_fdt()还负责早期的设备树扫描。

1. **Runtime configuration**: In most cases, a DT will be the sole method of communicating data from firmware to the kernel, so also gets used to pass in runtime configuration data like the kernel parameters string and the location of an initrd image. Most of this data is contained in the /chosen node, and when booting Linux it will look something like this:
2. 运行时配置: 大多数情况下，设备树将作为固件和内核传递数据的唯一方法。所以内核参数，initrd镜像的位置等运行时配置项也可以通过设备树传递。这些数据一般存放在/chosen节点，下面的代码是启动Linux内核时候的一个例子：

chosen {

bootargs = "console=ttyS0,115200 loglevel=8";

initrd-start = <0xc8000000>;

initrd-end = <0xc8200000>;

};

The bootargs property contains the kernel arguments, and the initrd-\* properties define the address and size of an initrd blob. During early boot, the setup\_machine\_fdt() function calls of\_scan\_flat\_dt() several times with different helper callbacks to parse device tree data before paging is setup. The of\_scan\_flat\_dt() code scans through the device tree and uses the helpers to extract information required during early boot. Typically the early\_init\_dt\_scan\_chosen() helper is used to parse the chosen node including kernel parameters, early\_init\_dt\_scan\_root() to initialize the DT address space model, and early\_init\_dt\_scan\_memory() to determine the size and location of usable RAM.

bootargs属性包含了内核参数，initrd开头的属性则定义了initrd数据的起始地址和大小。在启动阶段的早期，分页机制还没有还没有开启，setup\_machine\_fd借助不同的辅助函数来调用of\_scan\_flat\_dt对设备树的数据进行扫描。of\_scan\_flat\_dt扫描整个设备树，利用传入的辅助函数来提取启动阶段早期所需要的信息。回调函数early\_init\_dt\_scan\_chosen主要用来解析包含内核启动参数的chosen节点，early\_init\_dt\_scan\_root则负责初始化设备树的地址空间模型，early\_init\_dt\_scan\_memory的调用决定了可用内存的位置和大小。

1. **Device population**: After the board has been identified, and after the early configuration data has been parsed, then kernel initialization can proceed in the normal way. At some point in this process, unflatten\_device\_tree() is called to convert the data into a more efficient runtime representation. This is also when machinespecific setup hooks will get called, like .init\_early(), .init\_irq() and .init\_machine() hooks on ARM. As can be guessed by the names, .init\_early() is used for any machine-specific setup that needs to be executed early in the boot process, and .init\_irq() is used to set up interrupt handling.

设备填充：在机型识别完成并且早期的配置信息被解析之后，内核的通用初始化代码就可以开始执行了。在这个初始化的过程中，unflatten\_device\_tree函数负责将设备树的数据转化为更为有效的运行时描述方式。在ARM平台，这里也是机型特有的初始化钩子函数被调用的地方，比如.init\_early(), .init\_irq()和.init\_machine()。函数的目的可以从名字看出来，.init\_early负责启动早期特定机型需要执行的代码。.init\_irq()则负责中断相关的设置。

The most interesting hook in the DT context is .init\_machine() which is primarily responsible for populating the Linux device model with data about the platform. The list of devices can be obtained by parsing the DT , and allocating device structures dynamically. For the SAMA5D2 processor .init\_machine() will call sama5\_dt\_device\_init(), which in turn calls of\_platform\_populate() function. See the sama5\_dt\_device\_init() function in arch/arm/mach-at91/sama5.c:

在设备树上下文中最有趣的钩子函数当属.init\_machine()，负责根据platform相关的信息填充Linux设备模型。解析设备树获取设备列表，然后动态的为这些设备分配device结构。对于SAMA5D2处理器，.init\_machine会调用sama5\_dt\_device\_init()，sama5\_dt\_device\_init又会接着调用of\_platform\_populate()函数。arch/arm/mach-at91/sama5.c可以查看sama5\_dt\_device\_init()函数:

static void \_\_init sama5\_dt\_device\_init(void)

{

struct soc\_device \*soc;

struct device \*soc\_dev = NULL;

soc = at91\_soc\_init(sama5\_socs);

if (soc != NULL)

soc\_dev = soc\_device\_to\_device(soc);

of\_platform\_default\_populate(NULL, NULL, soc\_dev);

sama5\_pm\_init();

}

int of\_platform\_default\_populate(struct device\_node \*root,

const struct of\_dev\_auxdata \*lookup,

struct device \*parent)

{

return of\_platform\_populate(root, of\_default\_bus\_match\_table, lookup,

parent);

}

EXPORT\_SYMBOL\_GPL(of\_platform\_default\_populate);

The of\_platform\_populate() function located in drivers/of/platform.c walks through the nodes in the device tree and creates platform devices from it. The second argument to of\_platform\_populate() is an of\_device\_id table, and any node that matches an entry in that table will also get its child nodes registered.

of\_platform\_populate函数位于drivers/of/platform.c遍历设备树的节点并创建相应的platform设备。of\_platform\_populate函数的第二个参数是一个of\_device\_id类型的表，任何一个节点只要和该表中的某一个条目匹配的话，该节点的子节点也会被注册。

const struct of\_device\_id of\_default\_bus\_match\_table[] = {

{ .compatible = "simple-bus", },

{ .compatible = "simple-mfd", },

{ .compatible = "isa", },

#ifdef CONFIG\_ARM\_AMBA

{ .compatible = "arm,amba-bus", },

#endif /\* CONFIG\_ARM\_AMBA \*/

{} /\* Empty terminated list \*/

};

"simple-bus" is defined in the ePAPR 1.0 specification as a property meaning a simple memory mapped bus, so the of\_platform\_populate() code could be written to just assume simple-bus compatible nodes will always be traversed. However, we pass it in as an argument so that board support code can always override the default behaviour.

"simple-bus"定义在ePAPR 1.0规范中，这个属性用来标识一个简易的内存映射总线。of\_platform\_populate的实现可以假定simple-bus节点一定会被遍历到。实际上，通过传参，机型相关的代码可以改变这一默认行为。

# 最简驱动程序

A key concept in the design of the Linux embedded system is the separation of user applications

from the underlying hardware. User space applications are not allowed to access peripheral

registers, storage media or even RAM memory directly. Instead, the hardware is accessed via

kernel drivers, and RAM memory is managed by the memory management unit (MMU), with

applications operating on virtual addresses.

嵌入式Linux系统设计的一个关键概念在于用户应用与底层硬件的隔离。用户空间应用程序不允许直接访问外设的寄存器，存储媒介甚至内存。硬件通过内核驱动来访问，内存的访问则通过内存管理单元来管理，应用运行在虚拟地址空间。

This separation provides robustness. If it is assumed that the Linux kernel is operating correctly

then allowing only the kernel to interact with underlying hardware keeps applications from

accidentally or maliciously misconfiguring hardware peripherals and placing them in unknown

states.

这样的隔离提供了健壮性保障。假设内核的操作是正确的，应用对硬件设备恶意或者无意的错误配置可能导致硬件设备处于未知状态，只允许内核操作底层硬件可以有效的避免这种情况。

This separation also provides portability. If only the kernel drivers manage the hardware specific

code, only these drivers need to be modified in order to port a system from one hardware platform

to another. Applications access a set of driver APIs that is consistent across hardware platforms,

allowing applications to be moved from one platform to another with little or no modification to

the source code.

这样的隔离也提供了可移植性。如果只有内核驱动执行硬件相关代码，从一个硬件平台移植到另一个平台时只需要修改这些驱动就可以了。应用调用的驱动API在不同的硬件平台是一致的，这就允许应用在平台移植的时候可以几乎不对源代码做任何改动。

Device drivers can be kernel modules or statically built into the kernel image. The default kernel

builds most drivers into the kernel statically, so they are started automatically. A kernel module is

not necessarily a device driver; it is an extension of the kernel. The kernel modules are loaded into

virtual memory of the kernel. Building a device driver as a module makes the development easier

since it can be loaded, tested, and unloaded without rebooting the kernel. The kernel modules are

usually located in /lib/modules/<kernel\_version>/ on the root filesystem.

设备驱动可以是内核模块，也可以静态编译到内核镜像中。默认情况下，大部分驱动会被静态编译进内核，因此他们会自动加载。一个内核模块并不一定是设备驱动，这是对内核的一个扩展。内核模块被加载到内核的虚拟地址空间。将设备驱动编译成模块使得开发更加容易，因为加载，测试以及卸载都可以在不重启内核的情况下进行。内核模块一般存放在根文件系统的/lib/modules/<kernel\_version>/目录。

Every Linux kernel module has an init() and an exit() function. The init() function is called once

when the driver is loaded and the exit() function is called when the driver is removed. The init()

function lets the OS know what the driver is capable of and which of its function must be called

when a certain event takes place (for example, register driver to the bus, register a char device.. ).

The exit() function must free all the resources that were requested by the init() function.

每个内核模块都有一个init()和一个exit()函数。init() 函数在驱动加载的时候执行，exit()函数则在驱动移除的时候被调用。init()函数让操作系统知道驱动的能力以及特定事件（比如，驱动注册到总线，注册一个字符设备等）发生时应该调用驱动的哪个函数。exit()函数必须释放所有init()函数请求的资源。

Macros module\_init() and module\_exit() export the symbols for the init() and exit() functions such that the kernel code that loads your module can identify these entry points.

module\_init()和module\_exit() 宏负责将init()和exit()函数的符号导出，这样内核代码在加载你的模块的时候就能够识别这些入口。

There are a collection of macros used to identify various attributes of a module. These strings

get packaged into the module and can be accessed by various tools. The most important module

description macro is the MODULE\_LICENSE macro. If this macro is not set to some sort of GPL

license tag, then the kernel will become tainted when you load your module. When the kernel

is tainted, it means that it is in a state that is not supported by the community. Most kernel

developers will ignore bug reports involving tainted kernels, and community members may ask

that you correct the tainting condition before they can proceed with diagnosing problems related

to the kernel. In addition, some debugging functionality and API calls may be disabled when the

kernel is tainted.

还有一些宏被用来指定模块的各种属性。这些属性被打包进模块，可以通过各种工具来访问。描述模块的最重要的宏是MODULE\_LICENSE。如果这个宏没有被设置为某种GPL许可证，当你加载模块时内核就会被感染。内核被感染也就意味着处于一种不会被社区支持的状态。大多数内核开发者会忽略涉及被感染内核的bug报告。社区成员在着手分析内核相关的问题之前可能会要求你先修复被感染的内核。另外，当内核被感染时某些调试功能和API调用会被屏蔽。

## 许可证

The Linux kernel is licensed under the GNU General Public License version 2. This license gives

you the right to use, study, modify and share the software freely. However, when the software is

redistributed, modified or unmodified, the GPL requires that you redistribute the software under

the same license, with the source code. If modifications are made to the Linux kernel (for example

to adapt it to your hardware), it is a derivative work of the kernel, and therefore must be released

under GPLv2. However, you're only required to do so at the time the device starts to be distributed

to your customers, not to the entire world.

Linux内核使用GNU GPLv2许可证。这个许可证允许你自由地使用，学习，修改或分享软件。但是，当软件被再分发的时候，不管是修改或者未修改过，GPL许可证要求你以同样的许可证再分发并包含源代码。如果对Linux内核做了修改(比如：针对你的硬件做了调整)，这就是一个内核的衍生品，因此必须再GPLv2下发行。但是，你仅需要在设备被分发到用户手中的时候这样做，而不是整个世界。

The kernel modules provided in this book are released under the GPL license. For more

information on open source software licenses please see <http://opensource.org/licenses.>

本书中提供的内核模块都在GPL许可证下发行。关于开源软件许可证的更多信息可以参考http://opensource.org/licenses。

## 实验3.1: “hello world”模块

In your first kernel module, you will simply send some info to the console every time you load

and unload the module. The hello\_init() and hello\_exit() functions include a pr\_info() function. This is much like the printf syntax you use in user applications, except that pr\_info() is used to print log messages in kernel space. If you look into real kernel code, you will always see something like:

在你的第一个内核模块中，每次加载或者卸载模块时，你将向控制台简单的发送一些信息。hello\_init()和hello\_exit()函数包含了一个pr\_info()函数。pr\_info()函数语法很像你在用户应用中使用的printf，区别就在于pr\_info()是用来在内核空间打印一些日志消息。如果你查看真正的内核代码，你经常会看到这样的代码：

Where KERN\_ERR is one of the eight different log levels defined in include/linux/kern\_levels.h and specifies the severity of the error message. The pr\_\* macros are simple shorthand definitions

located in include/linux/printk.h for their respective printk call and should be used in newer drivers.

KERN\_ERR是定义在include/linux/kern\_levels.h文件中的八个不同的日志级别之一，用于指定错误消息的严重性。位于include/linux/printk.h中的以pr\_开头的宏是相应的printk调用的快捷定义。在新开发的驱动中，应该使用这些宏。

In the Eclipse Configuration for Developing Linux Drivers section in Chapter 1 you created the project my\_modules using the Eclipse IDE. This project will be used to develop all the drivers throughout this book, although you can use your favorite text editor to write the drivers if you do not want to work with Eclipse. The helloworld.c and Makefile files were created and saved in the modules labs directory without writing any code to them. It´s time to write code to these files.

在第一章的Eclipse开发配置中你使用Eclipse集成开发环境创建了my\_modules工程。这个工程将被用于开发本书中的所有驱动，如果你不想使用Eclipse，你可以使用你喜欢的文本编辑器编写驱动。创建并保存在modules实验目录下的helloworld.c和Makefile文件还没有任何代码。是时候往里面写入代码了。

You will repeat the same steps to create the driver´s source file <module name>.c for the rest of the labs. The same Makefile will be reused for all the labs by simply adding the new <module name>.o to the the Makefile variable obj-m.

后续的实验中，你会重复同样的步骤来创建驱动对应的源代码文件<module name>.c。通过简单的把<module name>.o加到Makefile变量obj-m中，所有的实验复用同一个Makefile。

In the Build Targets Tab, it was added all, deploy and clean buttons to compile, clean and deploy all the developed lab modules.

在编译目标标签中，增加了全部，部署，清理三个按钮来编译，清理和部署所有实验开发的模块。

See in the next Listing 3-1 the "helloworld" driver source code (helloworld\_imx.c) for the i.MX7D processor.

在接下来的3-1列表中查看针对i.MX7D处理器的"helloworld" 驱动源代码 (helloworld\_imx.c)

**Note**: The source code for the SAMA5D2 (helloworld\_sam.c) and BCM2837 (helloworld\_rpi.c)

drivers can be downloaded from the GitHub repository of this book.

注意：针对SAMA5D2 (helloworld\_sam.c) 和BCM2837 (helloworld\_rpi.c)的驱动源代码可以从本书的GitHub仓库下载。

列表 3-1: helloworld\_imx.c

（原书P85页代码1）

See in the next Listing 3-2 the Makefile used to compile this first module. The new developed

kernel module names will be added to this Makefile. Secure Copy (SCP) will be added to the Makefile to transfer the modules to the target filesystem, as shown here:

在接下来的列表3-2中查看用于编译第一个模块的Makefile。新开发的内核模块的名字将被加到该Makefile文件中。安全拷贝(SCP)命令将被添加到Makefile中用于把编译好的模块传送到目标文件系统，如下所示：

（原书P85页代码2）

列表 3-2: Makefile

（原书P86页代码1）

## helloworld\_imx.ko演示

（原书P86页代码2）

## 实验3.2: "带参数的helloworld"模块

Many Linux loadable kernel modules (LKMs) have parameters that can be set at load time, boot

time, and sometimes at run-time. In this kernel module you are going to pass a parameter into the

command line that will be set during the module loading. You can also read the parameters via the

sysfs filesystem.

许多Linux可加载内核模块 (LKMs) 都可以在加载时，系统启动时或者系统运行时设置参数。在这个内核模块中，你将通过命令行设置模块加载时的参数。你也可以通过sysfs文件系统读取这些参数。

The sysfs is a virtual filesystem provided by the Linux kernel that exports information about

various kernel subsystems, hardware devices, and associated device drivers from the kernel's

device model to user space through virtual files. In addition to providing information about

various devices and kernel subsystems, exported virtual files are also used for their configuration.

sysfs时Linux内核提供的一个虚拟文件系统。内核通过虚拟文件将内核设备模型中的各种内核子系统，硬件设备以及关联的设备驱动信息导出到用户空间。除了提供关于各种设备以及内核子系统的信息，这些导出的虚拟文件也被用于他们的配置。

The definition of module parameters is done via the macro module\_param().

模块参数的定义通过module\_param()宏实现。

（原书P87页代码1）

The main code sections of the driver will now be described:

该驱动的主要代码段描述如下：

1. After the #include statements, declare a new integer num variable and use the

module\_param() on it:

在#include语句之后，声明一个新的整型变量num，在接下来的module\_param()宏中使用。

（原书P87页代码2）

1. Change the pr\_info statement in the hello\_init() function, as shown below:
2. 修改hello\_init()函数中的pr\_info语句如下：

（原书P87页代码3）

1. Create a new helloword\_with\_parameters.c file in my\_modules project and add helloworld\_with\_parameters.o to your Makefile obj-m variable, then build and deploy the module using Eclipse.
2. 在my\_modules工程中创建一个新文件helloword\_with\_parameters.c。在Makefile中将helloworld\_with\_parameters.o添加到obj-m变量。接下来使用Eclipse构建并部署模块。

（原书P87页代码4）

See in the next Listing 3-3 the "helloworld with parameters" driver source code (helloworld\_imx\_with\_parameters.c) for the i.MX7D processor.

在接下来的3-3列表中查看针对i.MX7D处理器的"带参数的helloworld" 驱动源代码(helloworld\_imx\_with\_parameters.c)。

**Note:** The source code for the SAMA5D2 (helloworld\_sam\_with\_parameters.c) and BCM2837

(helloworld\_rpi\_with\_parameters.c) drivers can be downloaded from the GitHub repository

of this book.

注意：针对SAMA5D2 (helloworld\_sam\_with\_parameters.c) 和BCM2837 (helloworld\_rpi\_with\_parameters.c)的驱动源代码可以从本书的GitHub仓库下载。

列表 3-3: helloworld\_imx\_with\_parameters.c

（原书P88页代码1）

## helloworld\_imx\_with\_parameters.ko演示

（原书P88页代码2）

## 实验3.3: "helloworld timing"模块

This new kernel module, when unloaded, will display the time (in seconds) that has passed since

the driver was loaded.

这个新的内核模块会在卸载时显示从驱动被加载时开始流逝的时间（以秒为单位）。

You will use the do\_gettimeofday() function located in kernel/time/keeping.c to accomplish this task. When called, it fills a struct timeval structure with seconds and microseconds. The struct timeval structure is defined as:

你将使用位于kernel/time/keeping.c文件中的do\_gettimeofday()函数来实现该任务。该函数在被调用时会在struct timeval结构中填充秒和微秒信息。struct timeval结构定义如下：

（原书P89页代码1）

The main code sections of the driver will now be described:

该驱动的主要代码段描述如下：

Include the header file that defines do\_gettimeofday() as a function prototype:

包含定义了do\_gettimeofday()函数原型的头文件：

（原书P89页代码2）

After the #include statements, declare a struct timeval structure where the time will be

stored when the module is loaded and unloaded:

在#include语句之后，声明一个struct timeval结构用于存储模块加载和卸载的时间。

（原书P89页代码3）

When the module is unloaded the time difference is calculated:

当模块被卸载时，计算时间差。

（原书P89页代码4）

See in the next Listing 3-4 the "helloworld timing" driver source code (helloworld\_imx\_with\_timing.c) for the i.MX7D processor.

在接下来的3-4列表中查看针对i.MX7D处理器的"helloworld timing" 驱动源代码(helloworld\_imx\_with\_timing.c)。

Note: The source code for the SAMA5D2 (helloworld\_sam\_with\_timing.c) and BCM2837

(helloworld\_rpi\_with\_timing.c) drivers can be downloaded from the GitHub repository of

this book.

注意：针对SAMA5D2 (helloworld\_sam\_with\_timing.c) 和BCM2837 (helloworld\_rpi\_with\_timing.c)的驱动源代码可以从本书的GitHub仓库下载。

列表 3-4: helloworld\_imx\_with\_timing.c

（原书P89页代码5）

## helloworld\_imx\_with\_timing.ko演示

（原书P90页代码2）

# 字符驱动