ADI 公司如何看待自由和开源软件

作者：Michael Hennerich, Robin Getz

系统设计师为什么应当关注“自由和开源软件”？

使用自由和开源软件 (FOSS) 群体的迅速扩增，进一步体现了从 1980 年以来，嵌入式行业最重要的全面长期发展趋势。[1] 获得 FOSS 软件许可，就可以使用源代码，同时还赋予开发人员研究、变更、改进软件设计的权利。[2]在每一类主要软件的生命周期中，FOSS 已经或者必将发挥一定的作用，影响从 64 位服务器到 8 位微控制器的一切平台。FOSS 将从根本上改变所有用户和开发人员对于软件价值主张的看法。

因此，大部分嵌入式开发人员或早或晚都会在设计中使用 FOSS。

什么是 FOSS?

“自由软件”与“开源软件”的主要区别在于其内含的自由概念不同。“自由软件”许可尊重最终用户的四项基本自由：

1. 运行软件的自由
2. 研究和更改软件的自由
3. 再分发副本的自由
4. 改进程序和发布这些改进的自由

可以自由地做这些事情（还有其它事情），意味着您不必征求许可或者支付费用获得许可。这是一个关于自由的问题，而非商业问题，因此应理解为“言论自由，而不是免费啤酒”。[3] 另外还应注意，这些自由是针对“最终用户”而言，而不是开发人员，也不是软件分发者。

另一方面，“开源软件”并非始终赋予最终用户同样的自由，但它赋予“开发人员”访问源代码等权利。[4]各种开源许可都允许开发人员创建专有闭源软件，而不要求分发最终成果的源代码。BSD（伯克利软件发行）许可就是其中一例，它允许以二进制形式再分发软件，无需提供源代码。[5]

在现实世界中，闭源或专有软件与FOSS主要区别在于大众协作开发的性质不同；前者大家都独立开发各自的项目；而后者任何用户都可能成为开发人员，报告并修正缺陷，或者增加新特性。

FOSS 受到嵌入式市场欢迎的原因很简单，主要是经济利益驱动[6]：它能降低软件成本，加快产品上市。FOSS 将“自主开发”的开发人员变为系统集成者，使其能专注于产品增值和与众不同的特性，而不是一次次重复产生相同的基本结构和功能。这是控制软件开发成本的唯一行之有效的方法。无论何种组织机构，总会处于采用开源软件五个阶段中的某一阶段（在此向已故 Kübler-Ross 博士致歉）。[7]

采用 FOSS 的五个阶段[8]

状态	阶段现象
否认： 已在使用 FOSS	• 最近未审查定制软件 • 对流行 FOSS 组件的认知较少 • 公司对于 FOSS 使用无正式规定
愤怒： 对出现失控状态感到吃惊	• 在用软件并无允许使用记录 • 管理层着手分配责任 • 开发人员奉行“不问不说”原则
协商： 重新制定现有相关控管程序	• 采取紧急措施，全面了解情况 • 制定计划，删除现有 FOSS • 律师与开发团队进行冗长的会谈
沮丧： 意识到已经无法回头	• 意识到抽离开源程序将导致开发工作停止 • 认识到删除FOSS所牵涉的工作令人望而生畏
接受： 既然无法对抗，不如做好准备	• 正式实施FOSS战略 • 调整相关政策和程序 • 态度从容忍转变为积极利用

虽然许多人将 FOSS 等同于著名的 Linux® 内核或者基于 Linux 的发行版，但在嵌入式开发中，超出 Linux 范围使用 FOSS 已经非常普遍；几乎四分之三的组织都在使用它，涉及到成千上万个项目。然而，随着基于 Linux 的嵌入式系统越来越受欢迎，为嵌入式外设（ADC、DAC、音频编解码器、加速度计、触摸屏控制器等）提供 Linux 驱动程序的需求变得日益迫切。

在本文中，我们将讨论 ADI 公司对各种 FOSS 项目所做的贡献，重点是 Linux 内核，以及我们的客户如何使用它来降低风险并缩短产品开发时间。我们将探讨几个颇受欢迎的器件，例如数字三轴加速度计 ADXL345，并且说明：

- ADI 公司创建、修改和维护的驱动程序的各层
- 在何处进行维护（驱动程序的下载位置）
- 接口代码（用于内核的公共代码）——允许在您的平台上使用驱动程序
- 驱动程序开发惯例（哪些文件可以修改或提供，哪些不能）
- 何处可以找到代码——如何提交缺陷和问题报告

Linux 设备驱动程序—架构独立性

多数 Linux 用户都（乐于）不知道 Linux 内核所涉及的底层硬件复杂性和问题，往往会吃惊地发现内核大部分都与运行于其上的硬件无关。事实上，Linux 内核中的多数源代码都与独立于架构的设备驱动程序有关：在 Linux 2.6.32.6 内核的全部 7,934,566[9] 行代码中，有 4,758,810 行代码（60% 以上）都位于 `./drivers`，`./sound` 和 `./firmware` 目录下，比例之高令人震惊。与架构有关的代码只占Linux内核的很小一部分，全部 22 种不同架构仅有 1,501,545 行代码（18.9%）。Linux 内核支持的前 10 大架构是：

架构目录	源代码行数	占内核比例
./arm	302,125	3.81%
./powerpc	188,825	2.38%
./x86	154,379	1.95%
./mips	139,782	1.76%
./m68k	106,392	1.34%
./sparc	88,529	1.12%
./ia64	85,103	1.07%
./sh	77,327	0.97%
./blackfin	74,921	0.94%
./cris	72,432	0.91%

这说明，与架构无关的驱动程序（约占内核源代码的60%）具有举足轻重的作用。

对于每种支持 Linux 的硬件，都有人编写过设备驱动程序。自 2007 年以来，ADI 公司一直位列为 Linux 内核[10]贡献代码最多的 20 家公司之一（总共有 300 多家公司），并设立了专职团队从事Linux设备驱动程序的开发。

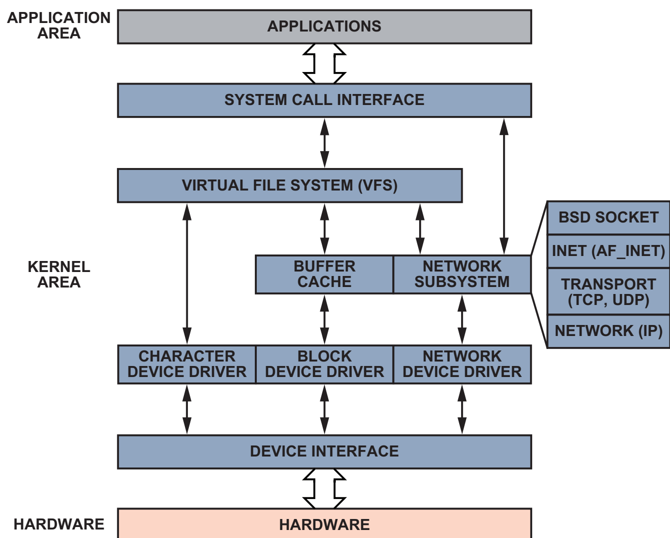
Linux 设备驱动程序的基本知识

设备驱动程序用作硬件与使用硬件的应用程序（用户代码）或内核之间的转译器，它将硬件的工作细节隐藏于幕后，从而起到简化编程的作用。编程人员可以利用一套标准化调用方法（系统调用）编写高级应用程序代码，而不必关心它将控制的特定硬件或运行于其上的处理器。借助定义明确的内部应用程序编程接口（内核 API），应用程序代码便可以通过与软件上层结构或底层硬件无关的标准方式与设备驱动程序实现接口。

针对特定处理器平台，操作系统 (OS) 处理硬件操作的细节。利用内核 (OS) 内部硬件抽象层 (HAL) 和处理器专用外设驱动程序（例如 I²C® 或 SPI 总线驱动程序），通常的设备驱动程序甚至也能独立于处理器平台。这种方法允许一个设备驱动程序（例如触摸屏数字化仪 AD7879 的驱动程序）可以不加修改地用在任何运行 Linux 的处理器平台上，Linux 内核之上运行任何图形用户界面 (GUI) 包和适当的应用程序。如果硬件设计人员决定转而使用触摸屏控制器 AD7877，他（她）将无需软件团队提供信息。两款器件均可用驱动程序；虽然器件不同，连接方式可能不同（AD7877 仅提供 SPI，AD7879 则有 SPI 或 I²C），并且寄存器图也不相同，但相对于触摸屏用户代码的内核 API 完全相同。这样，对硬件的控制权便又回到硬件架构师手中。

Linux 内核中的不同类型设备驱动程序提供不同的抽象层次，传统上一般将其分为以下三类。[11]

1. **字符设备：**处理字节流。串行端口或输入设备驱动程序（键盘、鼠标、触摸屏、游戏操纵杆等）通常实现字符设备类型。
2. **块数据设备：**单次操作处理 512 字节或更多的二次幂块数据。存储设备驱动程序通常实现此类块设备。
3. **网络接口：**任何网络事务都通过接口完成，接口指能够与其它主机交换数据的设备。



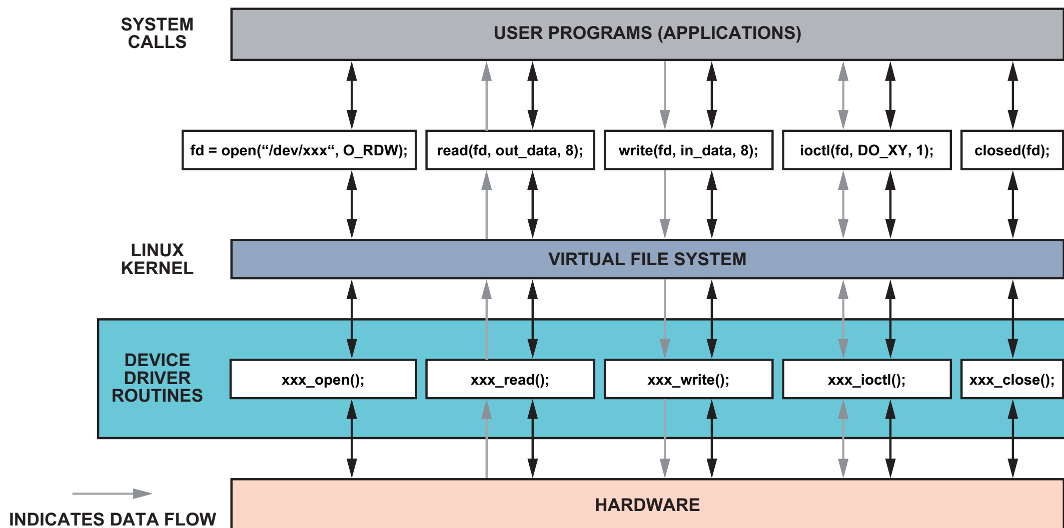
在 Linux 内核中，各特殊类别都可能有多个独立的设备核心层，以帮助开发人员实现提供标准用途的驱动程序，如视频、音频、网络、输入设备或背光处理等。通常，每个子系统在 Linux 内核源代码树中都有其自己的目录。这种“设备驱动程序核心方法”消除了特定类别所有设备驱动程序的公共代码，为上层构建了一个标准接口。每类设备或总线设备核心驱动程序通常会将一个函数集导出至其子类。驱动程序利用这种核心驱动程序注册，并使用核心驱动程序所导出的 API，而不是注册其自己的字符/块/网络驱动程序。这通常包括支持和处理多个实例以及在层间分配数据的方式。绝大部分系统无意了解设备如何连接，但需要知道何种设备可用。Linux 设备模型也包括一种将设备指派给特定类别的机制，如 input（输入）、RTC（实时时钟）、net（网络）或 GPIO（通用输入/输出）等。这些类名在更高的功能层次描述设备，使之能在用户空间中被发现。

特定硬件可能有多个设备驱动程序子系统与之相关。多功能芯片，例如带 I/O 扩展器的背光驱动器 ADP5520，会同时利用 Linux 背光、LED、GPIO 和输入子系统来实现其键盘功能。

如前文所述，用户应用程序不能直接与硬件通信，因为那将要求拥有对处理器的管理员权限，例如执行特殊指令或处理中断。使用特定硬件设备的应用程序通常在通过 `/dev` 目录中的节点暴露的内核驱动程序上工作。

设备节点称为伪文件，因为它们看起来像文件；应用程序也可以打开或关闭它们（`open()` 或 `close()`），但在读取或写入这些文件时，数据来自或传递至设备节点相关的驱动程序。这个抽象层次由 Linux 内核中的虚拟文件系统 (VFS) 处理。除了 `read()`、`write()` 或 `poll()` 外，用户应用程序也可以利用 `ioctl()`（输入/输出控制）与设备交互。

除设备节点外，应用程序也可以利用 `/sys` 目录中的文件条目；这是一个 sysfs 虚拟文件系统，可将有关设备和驱动程序的信息，包括父子关系或与特定类、总线的关联，从内核设备模型导出至用户空间。`/sys` 也频繁用于设备配置，特别是当相关驱动程序以一个设备驱动程序核心注册时，此时它只将其标准功能集导出给用户。



设备驱动程序可以注册 /sys “钩子”或“条目”；读取或写入钩子或条目时，将执行设备驱动程序专门注册的回调函数。这些回调函数（在管理员模式下运行）可以接受参数、发起总线传输、调用某种处理、修改特定设备变量，并将整数值或字符串返回给用户。这就为实现其它功能创造了条件；例如，用户空间可以使用触摸屏数字化仪 AD7877 的温度传感器或辅助 ADC。

设备驱动程序既可以静态地构建于内核中，也可以在以后作为可加载模块动态安装。Linux 内核模块 (LKM) 是动态组件，可以在运行时插入和移除。这对于驱动程序开发人员特别有用，因为更快的编译速度可以节省时间，而且测试模块不必重启系统。让硬件驱动程序驻留在可以随时载入内核的模块中，便可以在特定硬件不用时节约 RAM。

加载模块时，也可以赋予其配置参数。对于构建于内核中的模块，参数在内核启动时传送给该模块。例如：

```
root:~> insmod ./sample_module.ko argument=1

root:~> lsmod

Module                    Size Used by

sample_module 1396 0 - Live 0x00653000

root:~> rmmod sample_module
```

驱动程序也可以多次实例化，每次实例化都可以采用不同的设置，目标设备可以有不同的 I^{2}C 从 ID，连接到不同的 SPI 从选择，或者映射到不同的物理存储器地址。所有实例共用同样的代码，以便节省存储器，但具有各自的数据段。

Linux 是一种先占式多任务、多用户操作系统，因此几乎所有设备驱动程序和内核子系统都允许多个进程（可能由不同的用户所有）同时利用设备。常见的例子有 network（网络）、audio（音频）或 input（输入）接口。QWERTY 键盘控制器 ADP5588 的键按下或释放事件会被加上时间戳、排队并发送至所有已打开 input event device（输入事件设备）的进程。这些事件代码在所有架构上都相同，并且与硬件无关。读取 USB 键盘与从用户空间读取 ADP5588 并无区别。事件类型通过代码加以区分。键盘发送键事件 (EV_KEY)、键识别码以及代表按下或释放动作的某种状态值。触摸屏发送绝对坐标事件 (EV_ABS) 以及由 x、y 和触摸压力组成的一个三元组，鼠标则发送相对运动事件 (EV_REL)。加速度计 ADXL346 在发送关于加速度的绝对坐标事件的同时，可以发送关于单振或双振的键事件。

某些应用中，加速度计 ADXL346 产生相对事件或者发送特定键代码（特定应用设置），也很有意义。一般而言，定制驱动程序有两种方式：运行时或编译时。

可能在运行时进行定制的设备特性使用模块参数和或 /sys 条目。

使用开源 Linux 驱动程序—通过定制实现特定目标

对于编译时配置，将特定板和特定应用配置排除在主驱动程序文件之外是 Linux 的惯例，一般将其放入 board support file（板支持文件）中。

对于定制板上的设备（这是嵌入式和基于 SoC 片上系统硬件的典型现象），Linux 使用 platform_data 指向描述设备及其如何连到 SoC 的特定板结构。这可以包括可用端口、不同芯片版本、首选模式、默认初始化、引脚的其它作用等。这将能缩小板支持包 (BSP)，并尽量减少驱动程序中板和应用特定的 #ifdef。至于哪些可调变量进入 platform_data，哪些应当在运行时具有访问权，则由驱动程序的作者决定。

数字加速度计特性与应用具有非常密切的关系，不同的板和型号可能具有不同的特性。下例显示了一组配置选项。这些变量在头文件 adxl34x.h (include/linux/input/adxl34x.h) 中有详细描述。

```
#include <linux/input/adxl34x.h>
static const struct adxl34x_platform_data
adxl34x_info = {
 .x_axis_offset = 0,
 .y_axis_offset = 0,
 .z_axis_offset = 0,
 .tap_threshold = 0x31,
 .tap_duration = 0x10,
 .tap_latency = 0x60,
 .tap_window = 0xF0,
 .tap_axis_control = ADXL_TAP_X_EN | ADXL_TAP_
 Y_EN | ADXL_TAP_Z_EN,
 .act_axis_control = 0xFF,
 .activity_threshold = 5,
 .inactivity_threshold = 3,
 .inactivity_time = 4,
 .free_fall_threshold = 0x7,
 .free_fall_time = 0x20,
 .data_rate = 0x8,
 .data_range = ADXL_FULL_RES,

 .ev_type = EV_ABS,
 .ev_code_x = ABS_X,            /* EV_REL */
 .ev_code_y = ABS_Y,            /* EV_REL */
 .ev_code_z = ABS_Z,            /* EV_REL */

 .ev_code_tap = {BTN_TOUCH, BTN_TOUCH, BTN_
 TOUCH}, /* EV_KEY x,y,z */

 .ev_code_ff = KEY_F,           /* EV_KEY */
 .ev_code_act_inactivity = KEY_A,/* EV_KEY */
 .power_mode = ADXL_AUTO_SLEEP | ADXL_LINK,
 .fifo_mode = ADXL_FIFO_STREAM,
 };
```

为将设备与驱动程序相关联，“平台和总线模型”无需设备驱动程序来包含其所控制设备的硬编码物理地址或总线 ID。平台和总线模型还能防止资源冲突，大大改善便携性，并与内核的电源管理特性干净利落地接口。

利用平台和总线模型，设备驱动程序一旦获得设备的物理位置和中断线路，便知道如何控制设备。该信息在探测期间作为一个数据结构传递给驱动程序。

与 PCI 或 USB 设备不同，I²C 或 SPI 设备不会在硬件层次上进行枚举。相反，软件必须知道每个 I²C/SPI 总线段上连接了哪些设备，以及这些设备使用什么地址（从选择）。因此，内核代码必须明确实例化 I²C/SPI 设备。这可以通过多种不同方法实现，具体取决于上下文和要求。不过，最常用的方法是通过总线号码声明 I²C/SPI 设备。

当 I²C/SPI 总线是一条系统总线时，这种方法是合适的；许多嵌入式系统正是这种情况，其中每条 I²C/SPI 总线都有一个事先已知的号码。因此，可以预先声明连到该总线的 I²C/SPI 设备。这可以利用一个结构体 `i2c_board_info / spi_board_info` 阵列来完成，该阵列通过调用以下内容注册 `i2c_register_board_info()/spi_register_board_info()`

```
static struct i2c_board_info __initdata bfin_
i2c_board_info[] = {
#if defined(CONFIG_TOUCHSCREEN_AD7879_I2C) ||
defined(CONFIG_TOUCHSCREEN_AD7879_I2C_MODULE)
 {
   I2C_BOARD_INFO("ad7879", 0x2F),
   .irq = IRQ_PG5,
   .platform_data = (void *)&bfin_ad7879_ts_
   info,
},
#endif
#if defined(CONFIG_KEYBOARD_ADP5588) ||
defined(CONFIG_KEYBOARD_ADP5588_MODULE)
{
  I2C_BOARD_INFO("adp5588-keys", 0x34),
  .irq = IRQ_PG0,
  .platform_data = (void *)&adp5588_kpad_data,
},
#endif
#if defined(CONFIG_PMIC_ADP5520) ||
defined(CONFIG_PMIC_ADP5520_MODULE)
{
   I2C_BOARD_INFO("pmic-adp5520", 0x32),
   .irq = IRQ_PG0,
   .platform_data = (void *)&adp5520_pdev_
   data,
},
#endif
#if defined(CONFIG_INPUT_ADXL34X_I2C) ||
defined(CONFIG_INPUT_ADXL34X_I2C_MODULE)
{
   I2C_BOARD_INFO("adxl34x", 0x53),
   .irq = IRQ_PG0,
   .platform_data = (void *)&adxl34x_info,
},
#endif
};
static void __init blackfin_init(void)
{
(...)
i2c_register_board_info(0, bfin_i2c_board_info,
ARRAY_SIZE(bfin_i2c_board_info));
spi_register_board_info(bfin_spi_board_info,
ARRAY_SIZE(bfin_spi_board_info));
(...)
}
```

因此，为了启用这样一个驱动程序，只需要编辑板支持文件，将适当的条目添加至 `i2c_board_info (spi_board_info)`。

还应注意到，需在内核配置期间选择驱动程序。驱动程序按照所属的子系统分类。可在以下位置查找 ADXL34x 驱动程序：

```
Device Drivers --->
  Input device support --->
   [*] Miscellaneous devices --->
   <M> Analog Devices AD714x Capacitance
   Touch Sensor
       <M> support I2C bus connection
       <M> support SPI bus connection
```

```
  <*> Analog Devices ADXL34x Three-Axis
Digital Accelerometer
        <*> support I2C bus connection
        <*> support SPI bus connection
```

一旦用户开始内核构建过程，就会自动编译所选的驱动程序。

上面的代码声明 I²C 总线 0 上有四个设备，包括其各自的地址、IRQ 以及其驱动程序所需的定制 platform_data。注册相关I²C 总线时，i2c-core 内核子系统会自动实例化这些 I²C 设备。

```
static struct i2c_driver adxl34x_driver = {
 .driver = {
   .name = "adxl34x",
   .owner = THIS_MODULE,
},
 .probe = adxl34x_i2c_probe,
 .remove = __devexit_p(adxl34x_i2c_remove),
 .suspend = adxl34x_suspend,
 .resume = adxl34x_resume,
 .id_table = adxl34x_id,
};
static int __init adxl34x_i2c_init(void)
{
return i2c_add_driver(&adxl34x_driver);
}
module_init(adxl34x_i2c_init);
```

在内核启动的某一时间点，或者在其后的任何时候，一个名为 adxl34x 的设备驱动程序可以利用 struct i2c_driver 注册自己——通过调用 i2c_add_driver() 进行注册。struct i2c_driver 的成员利用指向 ADXL34x 驱动程序函数的指针进行设置，将驱动程序与其总线主控内核相连接。（宏 module_init() 定义模块插入时调用哪个函数 (adxl34x_i2c_init())。）

如果存档的驱动程序名称与宏 I2C_BOARD_INFO 所提供的名称相符，i2c-core 总线模型实现方法将调用驱动程序的 probe() 函数，将相关的 platform_data 和 irq 从板支持文件传递到驱动程序。这仅会在没有追索冲突的情况下发生，例如前一实例化设备使用同一 I²C 从地址时。

adxl34x_i2c_probe() 函数随后开始执行其名称所表示的功能。它通过读取制造商和设备 ID，检查 ADXL345 或 ADXL346 是否存在以及是否正常工作。如果检查成功，驱动程序的探测函数将分配特定设备数据结构，请求中断，并初始化加速度计。

然后利用 input_allocate_device() 分配新的输入设备结构，并设置输入位域。这样，设备驱动程序就告知输入系统的其它部分这是何种设备，以及这种新的输入设备能够产生何种事件。最后，ADXL34x 驱动程序通过调用 input_register_device() 注册该输入设备。

这将把新的输入设备结构添加到输入驱动程序的链接列表中，并调用设备处理程序模块的连接函数，告知其已出现一个新的输入设备。从此时起，该设备可以产生中断。一旦执行中断服务例程，就会从加速度计读取状态寄存器和事件 FIFO，并利用 input_event() 向输入子系统发回适当的事件。

ADI 公司维护的驱动程序

欲查看 ADI 公司维护的 Linux 驱动程序完整清单，请访问 Linux 内核主网站（kernel.org）或 ADI 公司 Linux 分发网站：https://docs.blackfin.uclinux.org/doku.php?id=linux-kernel:drivers。它包括各种驱动程序，从音频、数字电位计、触摸屏控制器、数字加速度计到 ADC、DAC 尽在其中。

如需帮助，请访问 http://blackfin.uclinux.org/gf/；这是 ADI 公司按照标准开源软件方式主办的网站，包括网络论坛和邮件列表。ADI 公司的一个专职团队负责及时回答疑问，并处理有关 FOSS 驱动程序的请求。

参考文献

（有关所有 ADI 公司器件的信息，请访问：www.analog.com。）

[1] IDC study/survey from over 5000 developers in 116 countries. *Open Source in Global Software: Market Impact, Disruption, and Business Models*. 2006.

[2]http://en.wikipedia.org/wiki/Free_and_open_source_software.

[3]www.gnu.org/philosophy/free-sw.html.

[4]www.opensource.org/docs/osd.

[5]www.opensource.org/licenses/bsd-license.php.

[6]Riehle, Dirk. "The Economic Motivation of Open-Source Software: Stakeholder Perspectives." IEEE Computer, vol. 40, no. 4 (April 2007). pp 25–32. http://dirkriehle.com/computer-science/research/2007/computer-2007.pdf.

[7]Kübler-Ross, Dr. Elisabeth E. *On Death and Dying*. Routledge. ISBN 0415040159.

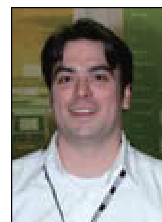
[8]Forrester Research. 1973. http://i.i.com.com/cnwk.1d/i/bto/20090521/Picture_2_610x539.png.

[9]All lines of source measurements were counted with David A. Wheeler's SLOCcount from www.dwheeler.com/sloccount.

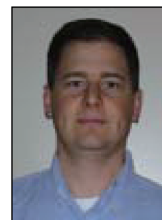
[10]Kroah-Hartman, Greg. SuSE Labs/Novell Inc., Jonathan Corbet, LWN.net, and Amanda McPherson. Linux Foundation; *"Linux Kernel Development: How Fast It Is Going, Who Is Doing It, What They Are Doing, and Who Is Sponsoring It."* www.linuxfoundation.org/publications/whowriteslinux.pdf.

[11]Corbet, Jonathan, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*, Third Edition. http://lwn.net/Kernel/LDD3.

作者简介



Michael Hennerich [michael.hennerich@analog.com] 于 2004 年加入 ADI 公司。他曾担任系统和应用设计工程师，从事过各种基于 DSP 和嵌入式处理器的应用和参考设计。Michael 现在是 ADI 慕尼黑公司的一名开源系统工程师，主管 Blackfin 架构 Linux 设备驱动程序和内核开发工作。他拥有德国罗伊特林根大学计算机工程硕士学位和电子与信息技术工程硕士学位。



Robin Getz [robin.getz@analog.com]于 1999 年作为资深现场应用工程师加入 ADI 公司，目前主管 ADI 公司自由和开源软件工作，包括面向 ADI 处理器的完整 Linux 分发版、GNU 工具链以及支持各种处理器和操作系统、与平台无关的设备驱动程序。加入 ADI 公司之前，Robin 曾在其它跨国半导体制造公司出任多种不同职位，并取得了多项专利。他于 1993 年获得加拿大萨斯喀彻温大学理学学士学位。