Free and Open-Source Software—An Analog Devices Perspective

By Michael Hennerich and Robin Getz

As a System Designer, Why Should I Care About *Free and Open-Source Software*?

The rapid increase in use of free and open-source software (FOSS) represents the most significant, all-encompassing, and long-term trend that the embedded industry has seen since the early 1980s.[1] FOSS software licenses make source code available and grant developers the right to study, change, and improve the software design.[2] FOSS is already playing, or will eventually play, a role in the life cycle of every major software category, influencing everything from 64-bit servers to 8-bit microcontrollers. FOSS will fundamentally change the value proposition of software for all users and developers.

So, for most embedded developers, if FOSS is not in your design today, most likely it will be soon.

What Is FOSS?

The main difference between *free software* and *open-source software* is in the philosophy of their inherent freedoms. A "free software" license is one that respects the end users' four essential freedoms:

1. The freedom to run it.
2. The freedom to study and change it.
3. The freedom to redistribute copies.
4. The freedom to improve the program and release those improvements.

Being free to do these things means (among other things) that you do not have to ask (or pay) for permission to do so. This is a matter of freedom, not commerce, so think of "free speech, not free beer."[3] Also note that the freedoms are for the *end user*, *not* the developer, *nor* the person who distributes the software.

On the other hand, *open-source software* does not always provide the end user the same freedoms, but it does provide *developers* such rights as access to the source.[4] Various open-source licenses allow developers to create proprietary, closed-source software, which includes no requirement to distribute the source code for the end work. The BSD (Berkeley Software Distribution) license is an example of those that allow binary redistributions without source code.[5]

The key real-world difference between FOSS and closed-source, or proprietary, software is the mass collaborative nature of development—the large number of people working independently on individual projects; here any user can become a developer—reporting and fixing bugs or adding new features.

The popularity of FOSS in the embedded markets is dominated by simple economic motivation[6]—it lowers software costs and hastens time to market. It turns "roll-your-own" developers into system-level integrators who can focus on adding value and differentiating features of their products rather than reproducing the same base infrastructure over and over again. It is the only proven methodology to reel in out-of-control software development costs. Irrespective of the organization, one can almost always find one of the five stages of open-source adoption taking place (our apologies to the late Dr. Kübler-Ross).[7]

Five Stages of FOSS Adoption[8]

State	Symptom of Progression
Denial: that FOSS is already in use	• No recent audits of custom software • Low awareness of popular FOSS components • No official company policy for FOSS usage
Anger: over surprise loss of control	• Software in use with no record of adoption • Management looks to assign accountability • Developers practice "don't ask, don't tell"
Bargaining: to re-establish existing controls and processes	• Crash program to identify total exposure • Program put in place to remove existing FOSS • Lawyers spend hours meeting with development teams
Depression: on realizing the point of no return has been reached	• Realization that extracting open source would bring development to a halt • Recognition that the effort involved in removing FOSS would be prohibitive
Acceptance: can't fight it, might as well prepare for it	• Implementation of a formal FOSS strategy • Adjustments to policies and procedures • An attitude to shift from tolerance to extracting value

While many people equate FOSS with the prominent Linux® kernel, or a Linux-based distribution, the use of FOSS beyond Linux in embedded development is pervasive; it is used by almost three-quarters of organizations and spans hundreds of thousands of projects. However, with the increasing popularity of embedded Linux-based systems, the interest in finding Linux drivers for embedded peripherals (ADCs, DACs, audio codecs, accelerometers, touch screen controllers, etc.) becomes increasingly compelling.

We will discuss here the contributions of Analog Devices, Inc. (ADI), to various FOSS projects, focusing on the Linux kernel and how they are being used by our customers to reduce risk and decrease product development time. We will take a look at a few popular devices, for example, the ADXL345 digital 3-axis accelerometer, and describe:

- layers of the driver created, modified, and maintained by ADI
- where things are maintained (location of driver download)
- interface code (common code for the kernel)—allowing you to use the driver on your platform
- common practice for driver development (which files can be changed or contributed, and which cannot)
- where the code can be found—how to log bug- and problem reports

Linux Device Drivers—Architecture Independence

The majority of Linux users are (happily) unaware of the underlying hardware complexity and issues involved in the Linux kernel, and are surprised to find out how much of the kernel is independent from the hardware on which it runs. In fact, the vast majority of source code in the Linux kernel is related to architecture-independent device drivers: Of the entire 7,934,566[9] lines in the Linux 2.6.32.6 kernel, an overwhelming 4,758,810 lines—over 60%—is in `./drivers`, `./sound`, and `./firmware`. Architecture-dependent code is a very small

fraction of the Linux kernel—1,501,545 lines (18.9%) for *all 22* different architectures. The top 10 architectures that the kernel supports:

Architecture Directory	Lines of Source	Fraction of the Kernel
./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%

This makes clear that architecture-*in*dependent drivers (~60% of the kernel source) play a very important role.

For each piece of Linux-supported hardware, someone has written a device driver. Since 2007, Analog Devices has ranked within the top 20 companies (from over 300+) contributing code to the Linux kernel[10]—and has a full-time team working on Linux device drivers.

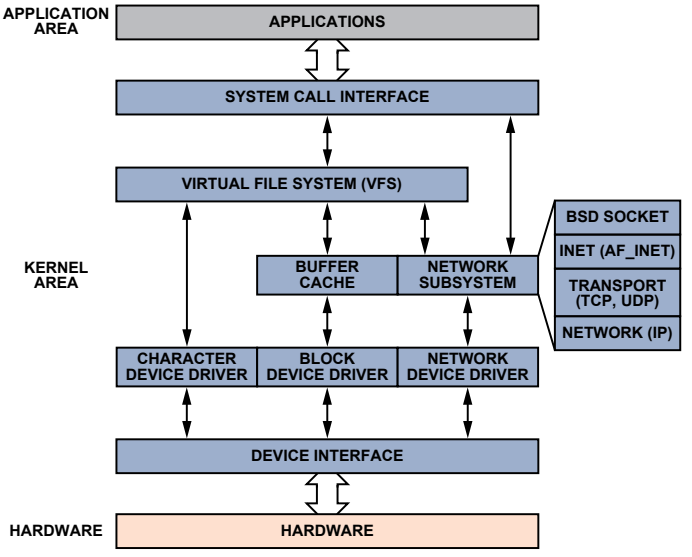
Basics of Linux Device Drivers

A device driver simplifies programming by acting as a translator between the hardware and the application (user code), or the kernel that uses it, hiding the details of how the hardware works. Programmers can write the higher-level application code using a set of standardized calls (system calls)—independent of the specific hardware device it will control or the processor it will run on. Using a well-defined internal *application programming interface* (kernel API) the application code and device driver can interface in a standard way regardless of the software superstructure or underlying hardware.

Operating systems (OS) handle the details of hardware operation for a specific processor platform. Kernel (OS) internal *hardware abstraction layers* (HALs) and processor-specific peripheral drivers (such as $I^2C^®$ or SPI bus drivers) allow even a typical device driver to be processor platform independent. This approach allows a device driver—for the AD7879 *touch screen digitizer*, for example—to be used without modification on any processor platform running Linux, with any *graphical user interface* (GUI) package and suitable application running on top of the Linux kernel. If the hardware designer decides to change to the AD7877 touch-screen controller, (s)he can do so without input from their software team. Drivers are available for both devices; and while they differ and can connect differently (the AD7877 is SPI only, and the AD7879 is SPI or I^2C)—and they both have different register maps—the kernel API that is exposed to user code for touch screens is exactly the same. This puts control of the hardware back into the hands of the hardware architect.

Different types of device drivers in the Linux kernel provide different levels of abstraction. They are generally and historically classified into three categories.[11]

1. **Char (character) devices:** Handle byte streams. Serial port or *input device* drivers (keyboard, mouse, touch screen, joystick, etc.) typically implement the character device type.
2. **Block data devices:** Handle 512-byte or higher power-of-two blocks of data in single operations. Storage-device drivers typically implement this type of block device.
3. **Networking interface:** Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts.



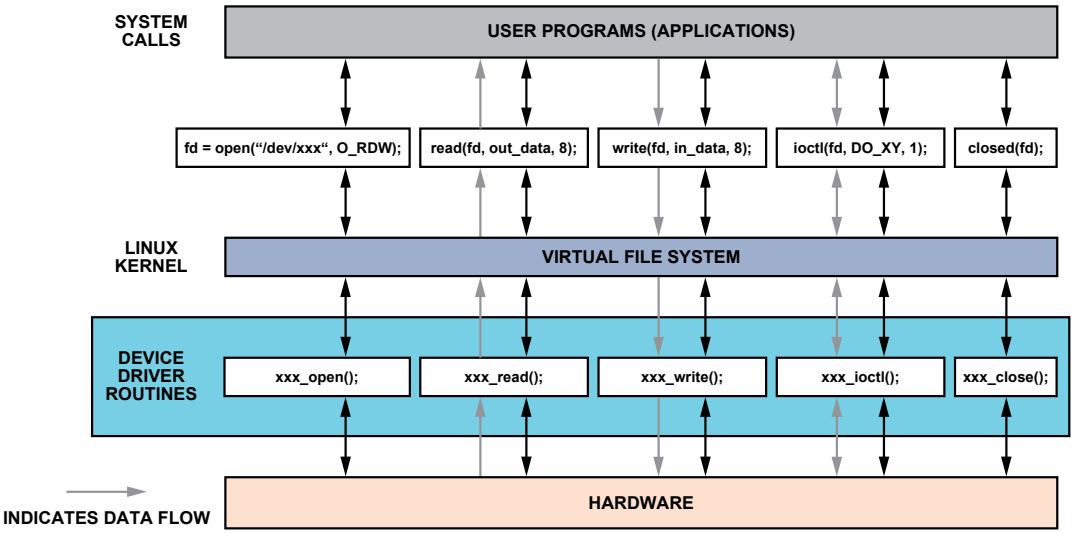
Each particular category may feature several independent device core layers within the Linux kernel, helping developers to implement drivers that serve standardized purposes—such as video, audio, network, input device, or backlight handling. Typically, each one of these subsystems has its own directory in the Linux kernel source tree. This *device driver core approach* removes code that would otherwise be common to all device drivers of a specific class and builds a standardized interface to the upper layer. Each class device, or bus device core driver, typically exports a set of functions to its child. Drivers register with such core drivers and use the API exported by the core driver instead of registering a character/block/network driver of their own. This typically includes support and handling for multiple instances—and the way data is distributed between layers. Huge portions of the system have very little interest in how devices are connected, but they need to know what kinds of devices are available. The Linux device model also includes a mechanism to assign devices to a specific class, such as input, RTC (real-time clock), net (networking), or GPIO (general-purpose input/output). These class names describe such devices at a higher, functional level and allow them to be discovered from user space.

There may be several device-driver subsystems associated with a particular piece of hardware. A multifunction chip, like the ADP5520 backlight driver with I/O expander, concurrently leverages the Linux backlight, LED, GPIO, and input subsystems for its keypad functionality.

As noted earlier, user applications are not allowed to communicate with hardware directly because that would require supervisor privileges on the processor, such as executing special instructions or handling interrupts. Applications utilizing a specific hardware device typically operate on kernel drivers exposed via nodes in the `/dev` directory.

Device nodes are called pseudofiles: they look like files; applications can also `open()` or `close()` them; but when they are read or written, the data comes from or is passed to the device nodes' associated driver. This level of abstraction is handled by the virtual file system (VFS) inside the Linux kernel. Besides `read()`, `write()`, or `poll()`, user applications may also interact with a device using `ioctl()` (input/output control).

In addition to the device nodes, applications may also utilize file entries in `/sys`, a sysfs virtual file system that exports information about devices and drivers, including parent/child relationship or association to a specific class or bus, from the kernel device model to user space. `/sys` is also heavily used for device



configuration, especially when the driver in question registers with a device driver core, which exports only its standardized set of functionality to the user.

Device drivers can register `/sys` *hooks* or *entries*; a specially registered callback function from the device driver gets executed when they are read or written. These callback functions—running in supervisor mode—may accept parameters, initiate bus transfers, invoke some processing, modify device-specific variables, and return integer values or character strings back to the user. This allows additional functionality; for example, the temperature sensor or auxiliary ADCs on the AD7877 touch-screen digitizer can be made available to user space.

Device drivers can be statically built into the kernel, or dynamically installed later as loadable modules. *Linux kernel modules* (LKMs) are dynamic components that can be inserted and removed at run time. This is especially valuable to driver developers since time is saved by quicker compilation and by not having to reboot the system to test the module. By letting the hardware drivers reside in modules that can be loaded into the kernel at any time, it is possible to save RAM when the specific hardware is not in use.

When a module is loaded, it can also be given configuration parameters. For a module that is built into the kernel, parameters are passed to it during the kernel boot. For example:

```
root:~> insmod ./sample_module.ko argument=1
root:~> lsmod
Module                  Size  Used by
sample_module 1396 0 - Live 0x00653000
root:~> rmmod sample_module
```

Drivers can also be instantiated multiple times, with different settings, with the target device sitting on a different I^2C slave ID, connected to a different SPI slave select, or mapped to a different physical memory address. All instances share the same code, which saves memory, but will have individual data sections.

Since Linux is a preemptive multitasking, multiuser operating system, almost all device drivers and kernel subsystems are designed to allow multiple processes (possibly owned by different users) to leverage the devices concurrently. Popular examples are the *network*, *audio*, or *input* interfaces. Key-press or -release events of an ADP5588 QWERTY keypad controller are time-stamped, queued, and sent to all processes that opened the *input event device*. These event codes are the same on all architectures and are hardware independent. There is no difference between reading a USB keyboard and reading the ADP5588 from user space. Event types are differentiated from codes. A keypad sends key-events (`EV_KEY`), together with codes identifying the key and some state value representing the press- or release action. A touch screen sends absolute coordinate events (`EV_ABS`) with a triplet consisting of x, y, and touch pressure, while a mouse sends relative movement events (`EV_REL`). An ADXL346 accelerometer may send key events for tap or double taps while it sends absolute-coordinate events for the acceleration.

In some applications, it could also make sense if the ADXL346 accelerometer generated relative events, or sent a specific key code—very application-specific settings. In general, there are two ways of driver customization: during run time or during compile time.

Device characteristics that are likely to be customized during run time use module parameters or `/sys` entries.

Using an Open-Source Linux Driver—Customization for a Specific Target

For compile time configuration, it's common Linux practice to keep board- and application-specific configuration out of the main driver file, instead putting it into the *board support file*.

For devices on custom boards, as typical of embedded and SoC- (system-on-a-chip) based hardware, Linux uses `platform_data` to point to board-specific structures describing devices and how they are connected to the SoC. This can include available ports, chip variants, preferred modes, default initialization, additional pin roles, and so on. This shrinks the board-support packages (BSPs) and minimizes board and application specific `#ifdefs` in drivers. It is up to the driver's author to decide which tunables go into `platform_data` and which should have access during run time.

Digital accelerometer characteristics are highly application-specific and may differ between boards and models. The following example shows a set of these configuration options. These variables are fully documented in the header file, adxl34x.h (`include/linux/input/adxl34x.h`).

```
#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,
};
```

To attach devices to drivers, the *platform and bus model* eliminates the need for device drivers to contain hard-coded physical addresses or bus IDs of the devices they control. The platform and bus model also prevents resource conflicts, greatly improves portability, and cleanly interfaces with the kernel's power-management features.

With the platform and bus model, device drivers know how to control a device once informed of its physical location and interrupt lines. This information is provided as a data structure passed to the driver during probing.

Unlike PCI or USB devices, I^2C or SPI devices are not enumerated at the hardware level. Instead, the software must know which devices are connected on each I^2C/SPI bus segment and what address (slave selects) these devices are using. For this reason, the kernel code must instantiate I^2C/SPI devices explicitly. There are different ways to achieve this, depending on the context and requirements. However, the most common method is to declare the I^2C/SPI devices by bus number.

This method is appropriate when the I^2C/SPI bus is a system bus, as in many embedded systems, wherein each I^2C/SPI bus has a number which is known in advance. It is thus possible to pre-declare the I^2C/SPI devices that inhabit this bus. This is done with an array of `struct i2c_board_info / spi_board_info`, which is registered by calling `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));
 (...)
}
```

So, to enable such a driver one need only edit the board support file by adding an appropriate entry to `i2c_board_info` (`spi_board_info`).

It has also been noted that the driver needs to be selected during kernel configuration. Drivers are sorted by subsystems they belong to. The ADXL34x driver can be found under:

```
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
```

Selected drivers will be compiled automatically once the user has started the kernel build process.

The above code declares four devices on I^2C Bus 0, including their respective addresses, IRQ, and custom platform_data needed by their drivers. When the I^2C bus in question is registered, the I^2C devices will be instantiated automatically by the `i2c-core` kernel subsystem.

```
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);
```

At some point during kernel startup, or at any time later, a device driver named "adxl34x" may register itself, using struct i2c _ driver—which is registered by calling i2c _ add _ driver(). Members of struct i2c _ driver are set with pointers to ADXL34x driver functions, connecting the driver with its bus master core. (The module _ init() macro defines which function (adxl34x _ i2c _ init()) is to be called at module insertion time.)

If the name of the driver that is filed matches the name given with the I2C_BOARD_INFO macro, the i2c-core bus model implementation invokes the driver's probe() function, passing it the associated platform _ data and irq from the board support file to the driver. This only happens in cases without recourse conflicts, such as when a previously instantiated device uses the same I^2C slave address.

The adxl34x _ i2c _ probe() function then starts to do what its name implies. It checks if either an ADXL345 or ADXL346 device is present and functional by reading the manufacturer and device ID. If this succeeds, the driver's probe function allocates device-specific data structures, requests the interrupt, and initializes the accelerometer.

It then allocates a new input device structure using input _ allocate _ device() and sets up input bit fields. In this way, the device driver tells the other parts of the input systems what it is and what events can be generated by this new input device. Finally the ADXL34x driver registers the input device structure by calling input _ register _ device().

This adds the new input device structure to linked lists of the input driver—and calls device-handler modules' connect functions to tell them a new input device has appeared. From this point on, the device may generate interrupts. The interrupt service routine, once executed, reads the status registers and event FIFOs from the accelerometer and sends appropriate events back to the input subsystem using input _ event().

Drivers Maintained by Analog Devices

A complete list of Linux drivers maintained by Analog Devices can be found in the mainline Linux kernel (at kernel.org) or within ADI's own Linux distribution website at https://docs.blackfin.uclinux.org/doku.php?id=linux-kernel:drivers. It includes a wide variety of drivers, from audio, digital potentiometers, touch-screen controllers, and digital accelerometers to ADCs and DACs.

To get help with these drivers, in the standard open-source fashion, Analog Devices sponsors a website that includes web forums and mailing lists at http://blackfin.uclinux.org/gf/—where a full-time ADI team is responsible for answering questions and handling requests about FOSS drivers in a timely manner.

References

(Information on all ADI components can be found at 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 http://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/.

Authors

Michael Hennerich [michael.hennerich@analog.com] joined Analog Devices in 2004. As a systems and applications design engineer he worked on a variety of DSP- and embedded-processor-based applications and reference designs. Michael now works as an open-source systems engineer in Munich. In this role he is the leading device driver and kernel developer for the Blackfin architecture Linux support. He holds an MSC degree in computer-based engineering and Dipl.-Ing. (FH) degree in electronics and information technologies from Reutlingen University.

Robin Getz [robin.getz@analog.com] joined ADI in 1999 as a senior field applications engineer, and he currently leads the Free and Open-Source efforts at Analog Devices, which include an entire Linux distribution for ADI processors, the GNU Toolchain, and platform independent device drivers for a variety of processors and operating systems. Prior to joining ADI, Robin held various positions at other multinational semiconductor manufactures, where he obtained multiple patents. He received a B.Sc. in 1993 from the University of Saskatchewan.