Device Driver Abstraction for Multithreaded Sensor Network Operating Systems

Haksoo Choi, Chanmin Yoon, and Hojung Cha

Department of Computer Science, Yonsei University
Shinchon-dong 134, Seodaemun-gu, Seoul 120-749, Korea
{haksoo,cmyoon,hjcha}@cs.yonsei.ac.kr

Abstract. To support the increasing number of sensor devices with various characteristics and requirements, sensor network operating systems should provide an appropriate device driver model that can cover a wide range of device types. Unfortunately, current sensor network operating systems force the user to build complex drivers for even simple devices, provide restricted interfaces, or do not provide any mechanisms. We present a device driver model that is flexible enough to support both simple devices with simple drivers, and complex devices with portable and high-performance device drivers. Users can write a device driver for simple devices with only a few lines of code using the user-mode device driver. Devices that need highly efficient code or portability can be supported by a single-layer or 2-layer kernel-mode device driver. Moreover, shared access and power management can easily be included in the device driver using the device manager. We also provide guidelines for choosing a proper device driver model with concrete examples of real-world devices and support our claims through the evaluation of the device driver model using the RETOS kernel.

Keywords: Multithreaded sensor network operating systems, device driver abstraction.

1 Introduction

Recent research in wireless sensor networks (WSN) technology has made various WSN applications possible, due to advances in sensor hardware and improvements in operating systems for WSN. We have developed RETOS [1], a multi-threaded operating system for WSN. RETOS features a secure operation of the kernel from malformed applications, support for multithreaded applications, and easy development of user applications. In order to support this last, RETOS provides various system calls, flexible modules, and an error detection service for user applications. However, it is still hard to guarantee flexibility in supporting various sensor devices through these techniques, and the operating system faces the non-trivial problem of developing hardware device drivers. Unlike in the PC environment, a standard interface for external hardware devices does not exist in WSN, and each sensor device has unique interfaces tailored to the purpose of the device. Therefore, supporting those sensor devices is complicated for sensor network operating systems. We have focused on two aspects of this problem. First, what is a device driver architecture that enables users to easily write their own device driver in support of operating systems? Second, what is a flexible device driver model that can support the various characteristics and needs of sensor devices? Our experience with previous research into RETOS has been highly useful in finding an answer to these problems, and the proposed device driver model exploits the dual-mode operation—user-mode and kernel-mode—of the RETOS kernel.

The proposed device driver model is designed to be flexible enough to meet the various requirements of the sensor hardware and support the diverse characteristics of the devices. Our model has three sub-models from which users can choose to write the most suitable device driver for their own hardware devices. For simple sensor devices that do not require complex operation and high performance, the user-mode device driver model can be used for secure operation and for writing a device driver. The 2-layer kernel-mode device driver model provides better performance and portability, with the separation of hardware-dependent and -independent code. The single-layer kernel-mode device driver has the highest performance of the three and is intended for devices that require high efficiency in device driver code. In addition, our model provides operating system level service for shared access management among multithreads and power management through the device manager.

This paper is structured as follows: In Section 2, we discuss previous research into device driver models and the characteristics of the RETOS kernel. The following section introduces the details of the proposed device driver model with its three sub-models, the device manager, and how to choose the proper device abstraction. In Section 4, we discuss concrete examples of building a device driver using our architecture. We evaluate our device driver model using RETOS in Section 5. Finally, we conclude the paper in Section 6.

2 Related Work

We have analyzed the previously proposed device driver models for several general-purpose, embedded, and WSN operating systems. Linux [2], NetBSD [3] and Windows Mobile [4] are general-purpose operating systems that have mature device driver architectures supporting a wide range of devices. However, their architectures are heavy and generalized for the requirements of WSN, due to the nature of their general-purpose operating systems. TinyOS [5] is a widely used WSN operating system that has a three-tiered device driver architecture [6] for greater portability and flexibility. However, it is hard to build a layered device driver for simple sensors, and our device driver model addresses this issue. SOS [7] and Contiki [8] do not provide any device driver model that enables users to write their own device driver. The device driver model for MOS [9] follows the POSIX model with its interfaces constrained to only four system calls: *dev_read()*, *dev_write()*, *dev_ioctl()*, and *dev_mode()*. This restriction prevents flexibility in the device driver for WSN hardware devices, which have various characteristics and requirements.

The proposed device driver model exploits the dual-mode operation of the RETOS kernel [1]. Dual-mode operation means that the stack for user threads and the kernel are separated so the secure operation of the kernel is guaranteed. Moreover, RETOS is a multi-threaded operating system, so several threads can request access to a device simultaneously, which can possibly lead to a race condition. To address this problem, a device driver model for such operating systems should handle shared access management. TinyOS 2.x [5] includes a dynamic resource management mechanism [10] that arbitrates shared accesses among multiple clients. Because TinyOS is a state-machine-based, event-driven operating system, unlike RETOS, the resource management scheme for TinyOS 2.x is not suitable for multithreaded operating systems. However, our intention for handling the possible race condition and providing proper power management is similar to their work.

3 Flexible Device Driver Model for Dual-Mode Kernel

In our device driver model (Figure 1), there are three different methods for building a driver for a hardware device. These are the user-mode driver, the 2-layer kernel-mode driver, and the single-layer kernel-mode driver. Each method has unique characteristics in terms of performance, portability, and ease of use. The user-mode device driver is simple, easy to build, and provides great portability at the expense of performance. The 2-layer kernel-mode driver has better performance than the user-mode device driver and good portability. The single-layer kernel-mode driver has the best performance but worse portability than the 2-layer kernel-mode driver. Therefore, device driver programmers can select the method that is best for their device. On the other hand, the device manager handles arbitration of shared accesses to specific hardware and performs proper power management based on the information regarding current peripherals in use. In order to achieve shared access management and power management, every device driver should obtain and release the right to use a device through the device manager before it actually uses the device.

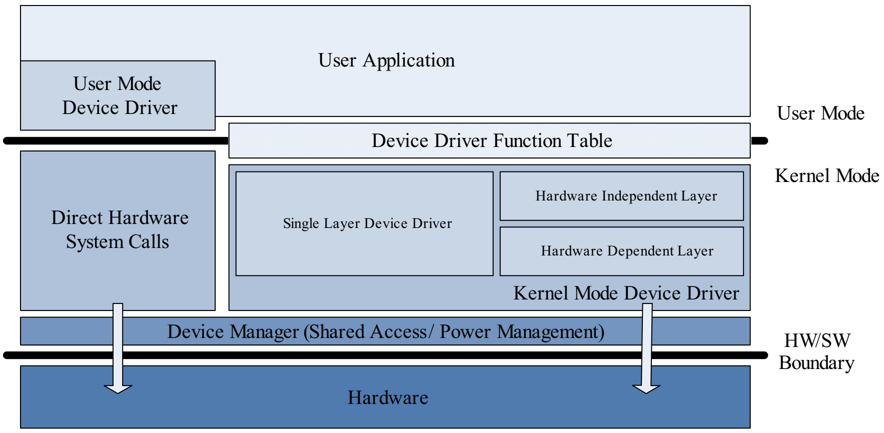
User Application
User Mode Device Driver
User Mode
Device Driver Function Table
Kernel Mode
Direct Hardware System Calls
Single Layer Device Driver
Hardware Independent Layer
Hardware Dependent Layer
Kernel Mode Device Driver
Device Manager (Shared Access/ Power Management)
HW/SW Boundary
Hardware

Fig. 1. The proposed device driver model

3.1 User-Mode Device Driver

The user-mode device driver provides the most stable and easy way to access a user's new hardware. It focuses on the fact that most sensor and actuator devices need a very simple interface with the microcontroller of a sensor node. For example, a device driver for a light sensor consists of simple access to the A/D converter (ADC) of the microcontroller, and possibly a little more code to convert the raw ADC readings into physical values. A device driver for the AC power control board controls the general purpose digital I/O (GPIO) of the microcontroller. Controlling the GPIO requires just one line of firmware code. Therefore, the main purpose of the user-mode device driver is to enable a device driver for simple hardware to be easy to design and stable by providing direct hardware system calls and abstraction of operating system level device management.

The user-mode device driver executes in user mode. The kernel therefore provides a set of primitive system calls, so the user-mode device driver can access the hardware. The primitive system calls are direct hardware system calls, which are a kernel-mode service, as shown in Figure 1. A device driver programmer can easily write a device driver for new hardware using the direct hardware system calls.

Because the user-mode device driver resides in user mode, it has several advantages over traditional kernel-mode drivers. First, the kernel is safe from a malfunctioning user-mode driver because the execution flows of the user and the kernel are separated. Second, a device driver programmer does not need to handle shared access and power management for the device. This occurs in kernel mode, and the direct hardware system calls already handle these issues. Moreover, writing code for a user-mode device driver is very simple and easy. Users only need to know how the device is connected to the microcontroller and write code to make the proper direct hardware system calls.

However, user-mode device drivers are not suitable for devices that require a complex interface to the microcontroller. Because the user-mode driver's access to the hardware depends on the direct hardware system calls, it cannot control the hardware in depth. Moreover, the user-mode driver has performance overhead. Because it executes in user mode, every time it requests the direct hardware system calls, kernel/user-mode switching occurs and causes system overhead.

Direct Hardware System Calls. The user-mode device drivers use direct hardware system calls to communicate with hardware devices. This is a set of system calls that provide simple access to the peripherals, such as ADC readings or controlling GPIO. Using these system calls, the user-mode device driver can read the ADC readings or control GPIO and provide device access functions to user applications.

The direct hardware system calls provide all possible interfaces that can be used by the user-mode driver to support a large range of devices. For example, the common microcontroller for the sensor node, MSP430F1611 [12], has several I/O interfaces, such as an analog to digital converter, a digital to analog converter, a general-purpose digital I/O, and a serial communication interface (USART), which can function as an asynchronous UART or synchronous SPI or I^2C interface. New devices can use any of these interfaces, so the direct hardware system calls expose these functionalities to the user-mode device driver.

In addition, the direct hardware system calls handle the operating system-level management of shared access and power. Because shared access and power management is tightly coupled with kernel behavior, the user-mode driver cannot communicate with the device management directly. Moreover, the responsibility of managing shared access and power is too great a burden for the simple user-mode driver; the user-mode driver is not intended for this. Therefore, to write the user-mode device driver easily and simply, the direct hardware system calls handle all the operating system services and let the user-mode device driver be concerned only with accessing its device.

3.2 Kernel-Mode Device Driver

Although the user-mode device driver has several advantages, many devices may still need the kernel-mode device driver due to performance issues or the complexity of the devices. For example, device drivers for radio chips or flash memory that supports a file system are complex and need in-depth hardware control and proper interrupt handling. The user-mode device driver has several drawbacks, such as kernel/user-mode switching overhead and limited ability with regard to controlling hardware. Therefore, the kernel-mode driver is for devices that cannot be handled by the user-mode device driver.

There are two different ways to build the kernel-mode driver to support a wide range of devices, as shown in Figure 1. The 2-layer kernel-mode driver consists of a hardware-dependent layer and a hardware-independent layer to increase portability, whereas the single-layer kernel-mode driver has no separation of hardware-dependent code in order to maximize its performance. Both methods have unique goals; hence, a device driver programmer can choose the model that is suitable for each device.

The device driver function table is necessary for communication between the user application and the kernel-mode driver because the kernel-mode device driver is written as a kernel module so it can be dynamically loaded and unloaded in the kernel. Moreover, the kernel-mode drivers should cooperate with the device manager to manage shared access and power. The behavior of the device manager is discussed in detail in Section 3.3.

2-Layer Kernel-Mode Device Driver. The 2-layer kernel-mode device driver separates the device driver into a hardware-dependent layer (HDL) and a hardware-independent layer (HIL) in order to increase portability. The HDL defines a set of primitive operations that are hardware dependent, and the HIL is implemented in a hardware-independent manner, using the interfaces provided by the HDL. Therefore, a device driver programmer needs to change the HDL part only when a new device driver for different hardware with the same functionality is needed.

The HDL communicates directly with the hardware and creates some level of abstraction to make the HIL part independent of the hardware. For example, the HDL part may include direct access to the memory-mapped I/Os or hardware-specific interrupt handlers. The HDL part then implements the details regarding small, meaningful units of hardware functions and provides them to the HIL part. The HIL part uses interfaces provided by the HDL part, which is an abstraction of the hardware, and implements the actual function of the device driver in a hardware-independent manner. For example, to make a timer device driver that supports virtual timers using one hardware timer, the HIL part implements the virtual timers using the abstraction of the hardware timer provided by the HDL part.

Single-Layer Kernel-Mode Device Driver. Although most devices can be supported by the 2-layer kernel-mode device driver, some devices are highly timing sensitive and require the best possible performance. For example, ultrasound device drivers are timing sensitive, so even 1 ms of error might be a big problem. Flash memory device drivers may need to be implemented in a manner that can reduce their latency as much as possible. To support this type of device, the single-layer device driver model is introduced.

The single-layer kernel-mode driver has no API separating hardware-dependent and -independent code, so it has less code and is more optimized and generally faster than the 2-layer device driver. The single-layer device driver can access the hardware directly at any time to reduce the delay caused by a series of function calls. Moreover, the single-layer driver usually has a smaller code and memory footprint than the identical 2-layer device driver, so it is useful for a resource-constrained sensor node platform.

However, the single-layer kernel driver loses its portability because hardware-independent code is mixed with hardware-dependent code. When performance or timing is more important than portability, however, a device driver programmer might want to write code with better performance. In this case, our device driver model can support the programmer.

3.3 Device Manager

In sensor network operating systems, several simultaneous requests to a device often occur at the same time. For example, one thread is trying to read a certain ADC port and waiting for the ADC to finish reading, and another thread can request access to the same ADC port. Without proper access management, one of the threads will not be able to obtain the result, or both threads might fail to read. Moreover, an even more complicated situation can occur. The widely used sensor node, Tmote Sky, has its radio chipset and external flash memory on the same port [11]. Simultaneous access to the shared port is impossible, and it needs proper shared access management. This kind of hardware implementation detail makes the device manager platform-specific.

Management cannot be handled in each application or each device driver because global information on the device access status is required. Therefore, our model has a device manager (Figure 2). By having a separate device management service, neither the user application nor each device driver needs to handle these issues directly. Instead, each device driver should cooperate with the device manager while it directly accesses its hardware, and the user applications are completely blind to those management services.

Moreover, the device manager can perform proper power management because it has global information on the device access status. The device manager can identify which peripherals are currently used. Therefore, the device manager can turn peripherals on or off properly and change the microcontroller's low power mode. Details regarding each device management scheme are discussed in the following sections.

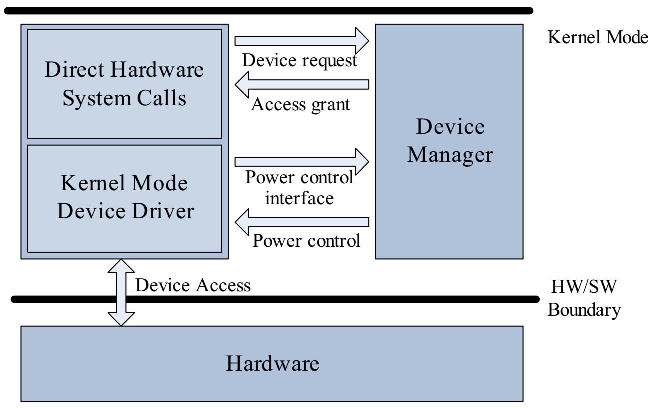

Fig. 2. The device manager

Shared Access Management. Every component of the kernel that accesses hardware devices—the direct hardware system calls and the kernel-mode device driver—cooperate with the device manager. When they need to use a device, they obtain the right to access the device from the device manager. When they have finished using the device, they release the right to use the device. This concept can be considered a binary semaphore. Each device in a sensor node has its own binary semaphore and the shared accesses to the device are managed by requesting and releasing the semaphore.

Several devices connected to the microcontroller on the same port can be managed by the device manager. These devices are accessed when another device on the port is not in use so only one device at a time is used by the device drivers. Because the device manager is aware of the fact that the ports are shared by several devices, each request to one of those devices can be properly granted. This is considered sharing one binary semaphore among several devices on the shared ports.

The following function prototypes comprise the interface for shared access management provided by the device manager. *request_device()* blocks the device driver and waits until the device is available; *request_device_immediate()* does not block and immediately returns with access to the device or error code when the device is not available. *request_device_timeout()* blocks the device driver and waits only for the specified timeout period. *release_device()* is used when the device driver finishes accessing the device.

```
devreq_t request_device(devid_t device);
devreq_t request_device_immediate(devid_t device);
devreq_t request_device_timeout(devid_t device,
                                          time_t timeout);
result_t release_device(devreq_t request_id);
```

Power Management. Because the device manager is aware of which devices are in use through the shared access management, it can determine the best possible low power state of the microcontroller. Moreover, in order to support proper power management, each device driver should provide device-specific power control functions that can be used by the device manager.

The microcontrollers used for the sensor node usually have a set of low power states with different peripherals available. For example, MSP430F1611 [12], used in Tmote Sky [11], has 5 different power modes with different clock sources available in each mode. Therefore, switching to the best possible power mode is necessary to reduce energy consumption, which is very important in an energy-constrained sensor node environment.

In addition to the power management of the microcontroller, device power management is important, as external devices usually consume more energy than the microcontroller. Generally, the devices for small sensor nodes do not have a set of low power modes as in the microcontroller, so the power management scheme is rather simple: just turn it off when it is not in use. However, determining when to turn the devices off or on is not trivial because the operating system should know which devices are currently being used. The device manager in our model tracks the set of currently active and inactive devices because every device driver reports its device use through the shared access management interface. Therefore, the device manager performs power management through the power control interface provided by each device driver.

Below is the data structure used by the device manager for proper power management. Each device driver initializes this data structure when the operating system boots. The *dev_access_ctrl* field shows the status of the device usage. The *power_on* and *power_off* fields are initialized by each device driver to provide power control functions for the device manager.

```
typedef struct
{
    devid_t device_id;
    devctrl_t dev_access_ctrl;
    result_t (*power_on) (void);
    result_t (*power_off) (void);
} device_t;
```

3.4 Choosing Proper Device Abstraction

The proposed device driver model has three different sub-models for writing device drivers—the user-mode, the 2-layer kernel-mode, and the single-layer kernel-mode device driver. Each sub-model has unique characteristics to make our device driver model as flexible as possible. Therefore, choosing a proper sub-model becomes important for exploiting the flexibility of our device driver model. In this section, we provide several metrics that can be used to choose a suitable sub-model.

Complexity. To decide between the user-mode and kernel-mode device drivers, the complexity of the device driver should be considered. The user-mode device driver has limited ability to access the hardware because it uses simple direct hardware system calls. On the other hand, the kernel-mode device drivers can access the hardware directly and obtain full control. Therefore, if the device driver is complex and requires full hardware control, the kernel-mode device driver should be used. If not, the simple and more stable user-mode device driver is sufficient.

Performance. The three device driver sub-models have different performance characteristics. The user-mode device driver has the worst performance among the three, due to the kernel/user-mode switching overhead and indirect access to the device. The single-layer device driver has the best performance among the three because the device driver can be fully optimized to reduce any delay in the code. The performance of the 2-layer device driver is worse than the single-layer but better than the user-mode device driver because no frequent kernel/user-mode switching occurs. Therefore, a device driver programmer should carefully consider the performance needs of the device and choose the device driver model accordingly.

Portability. If the device driver needs to be frequently ported to different hardware, a device driver programmer should consider using the 2-layer device driver model. The single-layer device driver mixes hardware-dependent and -independent code to optimize the code; it is not easy to port it to different hardware. On the other hand, only the HDL part needs to be changed if the 2-layer device driver model is used.

Size. In a resource-constrained sensor node platform, the memory and code size of the device driver is an important issue. Usually, the user-mode device driver takes up the smallest amount of memory and code because most of the hardware control and device management is already implemented in the direct hardware system calls and the user-mode device driver simply calls them. The 2-layer device driver model uses the most memory and code of the three, due to its separation of HIL and HDL parts. The single-layer is usually smaller than the 2-layer but larger than the user-mode device driver because it includes code for direct hardware access and management.

4 Application Examples

In this section, we provide examples and guidelines for building device drivers for various types of devices for a typical WSN platform. By using real-world examples, we demonstrate that our device driver model is flexible enough to satisfy the various capabilities of sensor devices and support a wide range of hardware platforms.

4.1 Analog to Digital Converter and Simple Sensors

The analog to digital converter (ADC) is a common device in a typical sensor node platform. From the viewpoint of a user application, its behavior is quite simple. User applications just need to obtain a digitally converted analog value and convert it into a physical value. This process is completed by requesting a direct hardware system call that accesses the ADC of a microcontroller. Therefore, the user-mode device driver will be sufficient for this type of device.

In addition to obtaining the ADC conversion, the user-mode device driver may want to configure the various properties of the ADC, such as reference voltage and different conversion modes. For example, the MSP430F1611 [12] microcontroller provides such capabilities on its ADC12 module. Therefore, the direct hardware system calls also provide such an interface with the user-mode device driver.

Below is an example interface for the direct hardware system calls that can be used to implement the user-mode device driver with simple sensors based on the ADC.

adc_read() returns a conversion value on the specified ADC channel. *adc_read_seq()* uses a sequence conversion mode that converts the series of the ADC channels. *adc_set_refv()* changes the reference voltage of the specified ADC channel.

```
uint16_t adc_read(uint8_t channel);
result_t adc_read_seq(uint8_t *channels, uint8_t num,
                          uint16_t *readings);
result_t adc_set_refv(uint8_t channel,
                          adc_refv_t refv);
```

4.2 General Purpose Digital I/O with Simple Actuators

Typical microcontrollers for a sensor node platform have many pins that generate a digital signal. They are called general purpose digital I/O (GPIO). For example, MSP430F1611 [12] has 6 ports with 8 pins on each port, and ATmega128 has 5 ports with 8 pins and 1 port with 5 pins, which can function as GPIO. Usually, controlling GPIO is quite simple because it just requires setting or clearing a pin signal. Therefore, the device driver for hardware based on GPIO can be built using the user-mode device driver model.

A common example device of using GPIO is an AC power control board. It consists of several relays that can connect or disconnect the AC power line, and each relay is controlled by digital signals from the microcontroller's GPIO. Therefore, a user application can control various AC-powered home appliances by controlling GPIO. The device driver for this AC power control board can be implemented with the user-mode device driver model using the direct hardware system calls related to the GPIO functions.

An example interface for the direct hardware system calls related to the GPIO is shown below. *gpio_mode()* changes the pin behavior, input or output. *gpio_set()* makes a high signal and *gpio_clr()* makes a low signal on the specified pin. *gpio_read()* reads the signal on the specified pin.

```
result_t gpio_mode(uint8_t port, uint8_t pin,
                      gpio_mode_t mode);
result_t gpio_set(uint8_t port, uint8_t pin);
result_t gpio_clr(uint8_t port, uint8_t pin);
uint8_t gpio_read(uint8_t port, uint8_t pin);
```

4.3 Serial Communications

Serial communication is a common method for a microcontroller to communicate with various external devices. There are many types of serial communication interfaces, such as UART, SPI, and I^2C. These are standard interfaces defining how the microcontroller and the external device should be connected and how they can communicate with each other. Then the actual communication protocol is usually to send commands to and receive data from the device. However, all devices that use a serial interface have their own protocols. Because there is no general communication protocol for these serial interfaces, it is hard to generalize them into the direct hardware system calls. In addition, a device driver for this type of device should communicate with the device frequently to obtain data or configure the device. If the driver is implemented with the user-mode driver model, an excessive amount of kernel/user-mode switching will occur, which is a lot of overhead. Therefore, the kernel-mode device driver is suitable for serial communication devices. Moreover, a simple temperature/humidity sensor does not require high performance and optimization of the device driver code, so using a 2-layer model is acceptable to increase portability.

Let's look at an example. SHT11 [13] is a temperature/humidity sensor from Sensirion Inc., which communicates with a microcontroller via a 2-line serial interface. The actual communication protocol includes sending a command, receiving a measurement, resetting the sensor, and reading and writing the status register. Each communication is initiated by sending a pre-defined code to the serial interface. We can define the hardware independent layer (HIL), which is exposed to user applications, and the hardware dependent layer (HDL), which contains hardware-specific code, as shown below. The HDL contains code for generating a proper clock signal for serial communication and sending and receiving a pre-defined code. In addition, converting the sensor readings into a physical value is completely dependent on the sensor device, so it is included in the HDL. The HIL includes functions for a user to obtain physical values. The HIL part is implemented using the provided HDL functions in a hardware-independent manner. If a new temperature/humidity sensor is introduced, only the HDL part needs to be changed.

```
//HIL interface
uint16_t get_temp();
uint16_t get_humid();

//HDL interface
uint16_t convert_physical_temp(uint16_t reading);
uint16_t convert_physical_humid(uint16_t reading);
uint16_t send_temp_read_cmd();
uint16_t send_humid_read_cmd();
result_t reset_sensor();
```

4.4 External Flash Memory

It is common for a sensor node platform to have an external flash memory for logging sensor readings. To reduce energy consumption, the read/write operation on the external flash memory usually occurs in chunks of bytes. The read/write latency is also significant, and the communication of large chunks of bytes on the communication interface between a microcontroller and an external flash is timing sensitive. Moreover, longer operation time on those energy-consuming devices means a shorter lifetime for a sensor node platform. Therefore, the device driver code must be optimized for high performance and efficiency. Because the energy consumption of the external flash device is not negligible, power-down mechanisms are usually provided by the hardware.

The above characteristics of the external flash memory require that the device driver be highly optimized. Therefore, the single-layer kernel-mode driver model is ideal for such devices. Because the single-layer driver model has no separation of hardware-independent and -dependent code, it can be fully optimized to reduce any possible latency caused by following a layered architecture. Compared to the user-mode driver model, it has less kernel/user-mode switching overhead. In addition, having hardware-specific power control functions enables the device manager to properly manage the energy consumption.

For example, M25P80 [14], from STMicroelectronics Inc., is a 1Mb external flash memory with a high-speed SPI serial interface. The SPI interface supports a clock rate of up to 75MHz so it can operate at very high speed. The read operation occurs in chunks of 1 to infinite bytes with a single instruction, and the write operation can occur in chunks of 1 to 256 bytes. To support the high-speed SPI serial communication and reading and writing chunks of bytes, a fully optimized device driver with high performance is necessary. Moreover, the M25P80 supports a “deep power-down” mode, which typically consumes 1μA current. Therefore, a proper power control interface should be provided by the device driver so the device manager can control the power of the flash memory. An example of the interface to user applications provided by the M25P80 single-layer kernel-mode device driver is as follows:

```
result_t power_on();
result_t power_off();
result_t write_enable();
result_t write_disable();
result_t read_bytes(uint8_t *buf, uint8_t size,
                    uint32_t addr);
result_t read_bytes_fast(uint8_t *buf, uint8_t size,
                         uint32_t addr);
result_t write_bytes(uint8_t *buf, uint8_t size,
                     uint32_t addr);
result_t sector_erase(uint32_t addr);
result_t bulk_erase();
```

5 Evaluation

In this section, we evaluate the proposed device driver model using the RETOS operating system. We focus on two important points. First, we show how the user-mode device driver effectively reduces the amount of code to be written. Second, we compare the performance and portability of a single- and 2-layer device driver for the same device. The device drivers are implemented on a Tmote Sky [11] platform with an MSP430F1611 [12] microcontroller. A user-mode device driver for an acoustic sensor that uses ADC is implemented, and the amount of code is compared to the ADC device driver of the current RETOS kernel. In addition, two UART device drivers are implemented using the single- and 2-layer device driver models for comparison. Table 1 summarizes the devices and the applied model used for the evaluation of the proposed device driver model.

Table 1. The devices chosen for evaluation

Applied model	Device
User-mode device driver	Acoustic sensor
Single- /2-layer kernel-mode device driver	UART

The comparison between the driver in the current RETOS kernel and the new device driver model is as follows. Table 2 shows the code and RAM size comparison for the current RETOS kernel and the new kernel with the proposed device driver model. The measurements include the actual device driver codes mentioned in Table 1. On the MSP430 platform, the new device driver model introduces about 17% of the code size and 50% of the RAM size overhead to the current RETOS kernel. The overheads seem to be relatively big in terms of ratio but the absolute sizes are still small enough for typical sensor node platforms.

Table 2. Code and RAM size comparison

	Current RETOS	New model	Ratio
Code size	20.7 kB	24.3 kB	1.17
RAM size	1.13 kB	1.7 kB	1.50

The following code listing shows the amount of code for the acoustic sensor device driver. We see that only a few lines of code are needed for building a device driver for an acoustic sensor when the user-mode device driver is used. Without the user-mode device driver model, the programmer should know how to control complicated hardware registers to obtain a single ADC reading. However, the direct hardware system call makes writing a user-mode device driver simple and easy.

The current RETOS device driver for ADC

```
uint8_t adc_get(uint8_t owner, uint8_t channel) {
  if (adc_busy == TRUE) return FAIL;
  adc_busy = TRUE;
  adc_owner = owner;
  ADC12CTL0 = ADC12ON | REFON | REF2_5V | SHT0_6;
  ADC12MCTL0 = channel + SREF_1;
  ADC12IE = 0x01;
  ADC12CTL1 = SHP | ADC12SSEL_3;
  ADC12CTL0 |= ENC | ADC12SC; // start the conversion
  return SUCCESS;
}

interrupt(14) __attribute((naked)) adc_intr() {
  ...
  uint16_t data = ADC12MEM0;
  ...
}
```

The user-mode device driver for ADC

```
#define MIC_ADC_CHANNEL    2

uint16_t read_mic() {
  //direct hardware system call
  return adc_read(MIC_ADC_CHANNEL);
}
```

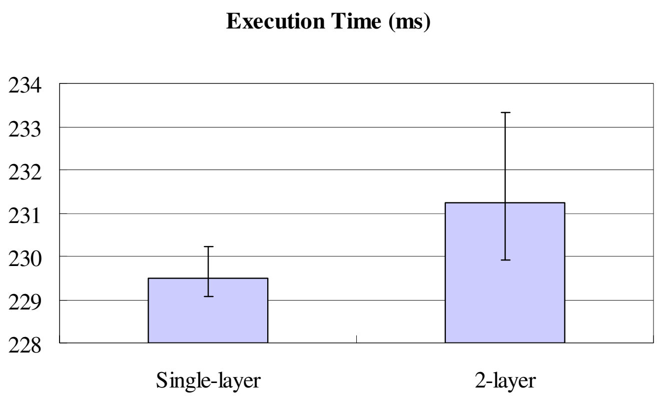

Fig. 3. Comparison of the execution times for UART device drivers

Figure 3 compares the execution time of a single- and 2-layer UART device driver. The total execution time for calling the *serial_send()* system call 1,000 times was measured. The single layer device driver is approximately 1.75 ms faster than the 2-layer device driver. The reason the single-layer device driver is faster is that it does not define an API isolating HIL and HDL, so it has less code and fewer function calls. Therefore, the single-layer device driver model is suitable for devices that need the best performance possible in a given hardware. On the other hand, the 2-layer device driver model has better portability because only the hardware-independent layer (HIL) needs to be changed when, for example, a different UART protocol is needed. A more interesting fact in Figure 3 is that the standard deviation of execution time is quite different between the single- and 2-layer drivers. Clearly, the single-layer device driver has a smaller standard deviation than the 2-layer device driver. It seems that the difference is caused by the fact that the 2-layer device driver uses an API defined by the HDL, so it has more function calls and bottom halves. This means that the 2-layer device driver allows more opportunities for interrupt handlings and bottom halves of other kernel behavior to be invoked because it has shorter critical sections than the single-layer. This indicates that the single-layer device driver should be considered when more stable performance of the device driver is required.

6 Conclusion

The main contribution of this paper is to present a flexible device driver model that can support a wide range of sensor devices that have various characteristics and requirements. Unlike current sensor network device driver models, our solution provides three different approaches from which users can choose the most suitable driver model for their own sensor hardware. Moreover, handling shared access and power management has been made easy with the device manager. The analysis in Section 4 and the evaluation results on a widely used sensor network platform support our device driver model. The user-mode device driver enables a safe and convenient building process for device drivers. The 2-layer kernel-mode device driver provides great portability in device driver code. Lastly, the single-layer kernel-mode device driver allows the best optimization of device driver code for sensor devices that require high performance.

Acknowledgements

This research was supported by the National Research Laboratory (NRL) program of the Korean Science and Engineering Foundation (No. M10500000059-6J0000-05910) and the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Advancement) (IITA-2007-C1090-0701-0015).

References

1. Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J., Yoon, C.: RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks. In: IPSN 2007. Proceedings of the Sixth International Conference on Information Processing in Sensor Networks, pp. 148–157 (April 2007)
2. Mochel, P.: The Linux Kernel Device Model. In: Mochel, P. (ed.) OLS 2002. Proceedings of the Ottawa Linux Symposium, pp. 368–375 (June 2002)
3. The NetBSD project, http://www.netbsd.org
4. The Windows Mobile operating system, http://www.microsoft.com/windowsmobile
5. Levis, P., Gay, D., Handziski, V., Hauer, J.-H., Greenstein, B., Turon, M., Hui, J., Klues, K., Sharp, C., Szewczyk, R., Polastre, J., Buonadonna, P., Nachman, L., Tolle, G., Culler, D., Wolisz, A.: T2: A Second Generation OS For Embedded Sensor Networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin (November 2005)
6. Handziski, V., Polastre, J., Hauer, J.-H., Sharp, C., Wolisz, A., Culler, D.: Flexible Hardware Abstraction for Wireless Sensor Networks. In: EWSN 2005. Proceedings of the Second European Workshop on Wireless Sensor Networks (January 2005)
7. Han, C.-C., Rengaswamy, R.K., Shea, R., Kohler, E., Srivastava, M.: A Dynamic Operating System for Sensor Nodes. In: Mobisys 2005. Proceedings of the Third International Conference on Mobile Systems, Applications, and Services (June 2005)
8. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In: SenSys 2006. Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (November 2006)
9. Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., Han, R.: MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. Ramanathan, P., Govindan, R., Sivalingam, K. (eds.) ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks 10(4), 563–579 (2005)
10. Klues, K., Handziski, V., Culler, D., Gay, D., Levis, P., Lu, C., Wolisz, A.: Dynamic Resource Management in a Static Network Operating System. Technical Report WUCSE-2006-56, Washington University in St. Louis (October, 2006)
11. Tmote Sky sensor node, http://www.moteiv.com
12. MSP430x15x, MSP430x16x, MSP430x161x mixed signal microcontroller, http://www.ti.com
13. SHT11 digital humidity sensor, http://www.sensirion.com
14. M25P80 serial flash memory, http://www.st.com