RETURN

Writing device specific, RTOS independent, hardware independent device drivers

Prepared for

Embedded Systems Conference, San Francisco

By

Brian Schrom

Abstract

The statement "I/O is everything" captures the essence of an embedded system. Embedded systems often contain the same peripheral device across multiple platforms. It is desirable to write the software that controls the device (device driver) one time and reuse it with minimal or no modifications on any platform, regardless of operating system, RTOS, or architecture. This can be accomplished by defining an interface that abstracts the hardware and RTOS specific functions away from the device.

A common adage is "a thing on a thing, doesn't run as fast as a thing." Embedded systems must respond to stimulus in a known or predictable time period. By carefully selecting the interfaces and communication mechanisms of the interfaces between components, acceptable performance can be achieved, yet provide code that is portable, extensible, and maintainable.

December 13, 2001

Table of Contents

Introduction

Writing, debugging, and maintaining device drivers is one of the tasks that embedded systems programmers do on a regular basis. Being able to reuse drivers on multiple platforms can save effort by reducing the amount of replicated code. This can be done through having a small subset of platform specific functions. This report looks into some of the issues that are involved when writing platform independent drivers, as well as, some existing solutions.

There are a number of “nontechnical” reasons why it is desirable to write, maintain, and use a single driver. These reasons include: lower development costs because a single driver interface will cover multiple environments, lower support costs because a driver will continue working across operating system revisions and upgrades, more marketing and sales opportunities because once a driver has been developed, it can be easily moved to other environments with very little effort compared to developing a new driver, and a single binary driver release that will work across multiple operating systems running on the same hardware platform. (Linux implementation of UDI)

All of these reasons result in reduced cost or increased money opportunities, hence it seems that it is worthwhile to pursue. The “technical” issues that arise in implementing a platform independent device driver fall into three categories. The first issue is that the application using the device is probably not at the same operating level as the level that interfaces to the device, or the layer where the device is accessed. This results in there being a minimum of two logical layers, meaning that an application cannot easily customize a device for its specific use. The second issue is that devices are physically accessed differently on different platforms. One system may be big-endian, another one may be little-endian. There needs to be a hardware abstraction layer (HAL) that removes these hardware differences from the device driver. The third issue is that a device driver may need to interact with the kernel by locking a resource or sending notification that an event occurred, or extend the kernel in some way.

Issues

Device drivers interact with three elements of a system. The first element is the “interface” layer of the system. This may be an independent module or implied by the application that is using the device. The second piece is the HAL with the relevant part being made up of device driver helper functions. The last element is how the device interacts with the kernel. The simplest case has no interaction and the most complicated actually extends the kernel through a registered service.

The application that is using the device is probably not at the layer where the device driver is implemented

This statement alludes to the interface layer which may be an independent module, or may be handled by the user of the device.

An embedded application may wish to use a serial port for connection with a serial mouse or remote display. The code that reads and writes raw data to the serial port is deep inside the kernel library and inaccessible to the application that just wants mouse movement commands. This problem illustrates the fact that an "interface[1]" specification is often missing from embedded systems, but may become a requirement when implementing device independent drivers. This can be the secret that allows writing platform independent device drivers. In effect, the modularity created by having separate application, interface, and driver interface layers can remove binding handcuffs and allow for optimization to be done on a larger scale.[2]

The device driver concerns itself with only those properties that are specific to the device. The interpretation of raw data is left to the "interface" that uses the device. This enables any device to be used with potentially, any protocol and also allows devices to be "dumb" enough to only depend on a set of device helper functions that abstract the system enough for the device to operate.

The fundamental problem to reusing the code is that the application code, the interface code, the message handling, the device handling, and the physical device are all at different logical levels. Intermixing these levels takes many small, simple problems, combines them, and makes one big problem out of them. This "may" result in an optimized solution, but also results in code that is not easily extended or maintained. By keeping all of the small pieces independent, a programmer is free to quickly rearrange and reoptimize for a specific solution.[3]

Devices are physically accessed differently on different platforms

How a device is physically accessed primarily depends on the topology of the device and the topology of the bus that the system is on. Accessing a device can be as simple as a memory move instruction or as complicated as an I/O port access, in multiple bus cycles, that must be byte swapped and reassembled before returning the value to the caller. In theory, devices are matched to the bus that they are on, ISA devices with ISA busses, PCI devices with PCI busses, etc. In reality, it is usually a mix and match of devices with busses, so the device helper functions resemble the more complicated device access scenario mentioned above. Factors to consider that contribute to how device access functions work are described below.

[1] Interface in this context means the binding of a protocol to a device. This is the level of software that interprets raw data from a device, before forwarding the data to the application.
[2] Optimization on a global scale will result in an over-all optimized solution. Often this is the desirable outcome, but careful attention must be made that local areas become too slow.
[3] This is the Keep It Simple Stupid (KISS) rule

Bus Operation
Busses operate in a few access modes. Common busses use fixed width or dynamic bus sizing. Aligned accesses may or may not be supported.

Bus Size
8-bit, 16-bit, 32-bit, 64-bit, etc.

Device Size (register size)
8-bit, 16-bit, 32-bit, 64-bit, etc.

Endianness
Busses may be big endian or little endian.
Devices may be big endian or little endian.

Memory Map
I/O may be mapped to "i/o" space, memory mapped area, or even require a different (privileged) access mode.

Cache
Device access operations, especially through memory mapped devices must not be cached.

MMUs
Devices need to communicate with applications and may need access to shared memory, or some other mechanism (memory copies). If paging is used, then a way to allocate nonswappable memory is also required.

DMA
Devices may be bus mastering (if you are lucky), polled, require external DMA controller, use block-i/o, shared memory, or some combination there of.

Device Types
Devices come in many configurations, character oriented, variable-block oriented, fixed-block oriented.

Interrupts
Interrupts can be edge or level triggered, by a falling or rising edge, and be active-high, or active-low. Often there are interrupt chips that "help" process interrupts, and occasionally, there are a pyramid of interrupt chips that "help" process interrupts. Sometimes interrupts are routed directly to the CPU with different priority levels. Each of these chips requires their own unique processing, independent of the device.

Each of these categories have one or more device helper functions to facilitate accessing the device. These functions form a subset of the HAL.

Device driver may need to interact with, or extend, the kernel

The device driver and the interface layer ensure that there are at least two layers of operation. This implies that there exists a communication mechanism between the device driver and the interface handling the device. The form of this communication may be a simple function call (a callback handler), or a more sophisticated inter-process communication (IPC) mechanism, such as a semaphore, a mutex, or a message.

When a device is extending the kernel, it is usually required to register itself with the kernel as a service. Services usually require some method for accessing "kernel" memory or a shared memory pool. These are accessible by one or more of the following: application, service, kernel, and device driver. If the system has a memory management unit, then provisions for allocating "unswappable" memory must exist. Other kernel call types that may be required are: LogMessage, interfaceSendMsg (that is interrupt safe), RegisterService and lockResource.

A device may need to allocate memory or signal a task from within the driver. Mechanisms to do this will be different for each kernel. A key to writing Kernel independent drivers, is to keep what a driver has to do to a minimum, and pass control to an upper layer with more intelligence to handle the data or events.

Addressing the issues

The solution to writing platform independent device drivers relies on the notion of writing a generic abstraction layer for a device that an "interface" layer uses to accomplish its task. Whether the interface layer is implied by the driver, the application, or is its own entity does not matter. What does matter, is the presence of this abstraction and the "contractual" agreement that the device driver must comply to it. This allows the upper interface, the interface between the device driver and the rest of the system to be defined. This definition then becomes the basis for defining the lower interface, the interface between the device driver and the hardware or platform abstraction layer.

Device driver helper functions deal with all of the RTOS and platform specifics that exist for target platforms. This functionality may be as simple as a set of macros, as complicated as a virtual I/O system, or some combination of the two.

What factors and issues that are applicable to the HAL, depends on the definition of the phrase "portable across platforms." Some questions that may help determine the requirements are: Are all platforms based on the same architecture? If yes, then binary configuration may be possible. If no, then source distribution is almost a mandatory requirement. Are modules dynamically loadable/unloadable? Is remote firmware upgrade a requirement?

The simplest access to a device register is a function that reads a word from a register on a device, _inWord(DeviceRegister). There are a number of issues that come into play when implementing this function.

- What is the endianness of the data that is returned? Does it need to be swapped before being used?
- Does the access occur in two bus cycles or one? Is it two half-word reads, or one?
- What is the size of the bus? Does it use dynamic bus sizing? Are DeviceRegisters accessed in increments of "bus width size?"
- Is the device in I/O space, or memory mapped space?

In order for a device driver to be independent of its platform, all of these options must be configurable or assumed to be "fixed" for a given implementation.

There are many more factors that can contribute to what it means for a device driver to be "platform independent." Current solutions resemble one of the following: use an open standard definition such as the Uniform Device Interface (UDI) specification, follow a "standard" operating system definition such as Windows or Unix, or a customized "roll-your-own" specification, which is often the case with embedded systems since they are generally very specialized.

If the project has the luxury of having source code, then a set of macros can be constructed that allow all of these conditions to be determined at compile time. If the device is going to be used in library form (binary), then this functionality must be able to be resolved at runtime. The advantage of using macros is that the device access can be determined and resolved at compile time. This results in specialized code for accessing the device that will be smaller and faster. The downside, is that the source code must be available for porting to a new device. This is the approach that Unix has taken, C is the common language, and you recompile on a new platform.

An efficient implementation can still be realized with a binary distribution. This requires careful understanding of requirements, so that as much can be assumed as possible, while retaining the required generality. An advantage of the binary configuration is source level debugging and running on the same architecture without recompiling.

An open standard example following the STREAMs approach uses the Data Link Provider Interface (DLPI) to specify the interface and the UDI to specify the hardware abstraction layer requirements for accessing and implementing a device. Both of these specifications are based on a messaging model and could be well suited to an embedded environment.

The DLPI enables a data link service user to access and use any of a variety of conforming data link service providers without special knowledge of the provider's protocol (See references for DLPI specification).

The UDI is a very "heavy-weight" specification but has the advantage that it is supported by a number of big names in the industry and has reference designs for a number of platforms. This tends to be used on larger systems. A subset may be useful on some embedded systems, depending on the degree of portability that is required. This is a source code specification that is very thorough and complete. It includes device access, dynamic memory, auto detection, auto installation, etc.

If there is a software platform or operating system (i.e Unix or Windows) that is being targeted, then the best choice, possibly the only choice, is to follow the structure that the operating system (OS) requires. OSs often have device driver kits (DDKs) that include sufficient device helper functions or a HAL that makes a device "platform" independent.

If the target is multiple software platforms (operating systems), then some hybrid models insert a "shim" layer that interfaces between the software platform and the device independent layer[4].

The most interesting model is where the programmer gets to define a new device driver abstraction layer… At least, this is the most interesting model for illustrating some of the issues involved in developing a device independent model.

Roll-your-own API – Upper Layer

Assumptions

- An interrupt service routine (ISR) should do the simplest thing that it can, very quickly, very efficiently, and then return to the system.
- ISRs should be handled in-context in real-time systems, otherwise it is not a [traditional] "real-time system."

Starting with a top-down design approach, the device driver must notify the interface when an interesting event occurs. This is usually the result of an interrupt or the result of a polling operation. Of note is the fact that communication is taking place between the ISR and the interface. This implies a communication mechanism; usually the simpler, the better, and most systems use a callback handler, or a simple message or signaling mechanism.

The interface will then respond based on the notification that it receives from the ISR. This response generally falls into reading/writing from/to the device (or queues), controlling the device, or aborting the device (fatal error conditions).

Before the device can respond to read/write/control/etc commands, it must be initialized. Initialization includes configuring the device and enabling it to accept commands, but may also include auto-detection and auto-configuration. The initialization phase may be split into two steps: power-on initialization, and before using the device initialization. The latter is often implemented in an *open* function. If a device needs to be restarted or shut down, then a terminate-like facility is also required. Just like initialization, termination may be split into two functions, often implemented as a *terminate* and a *close* function.

[4] This is a feature included in UDI

From ideas in the last few paragraphs, the upper interface requirements can be extracted into eight function types:

Upper-level API

Init	This function initializes the device at power on time and puts the device into a state so that it may be opened. This function may acquire kernel resources, enable regions of memory to access the device, initialize or configure busses, configure interrupts, and whatever is needed prior to calling open.
Term	This function is the complement of Init and undoes everything that Init does, presumably, putting the system back into a stable state before power going off. (May not be present on systems that are never intended to have the power turned off. Don't forget about "remote upgrade" or other scenarios that may benefit from a term function.)
Open	This function enables the device for active operation. This may include enabling interrupts, allocating resources, and setting the device up with initial parameters.
Close	This function disables the device from active operation, but leaves the device in a state that may be re opened. This function should release any resources that were acquired in the Open call.
Read	Read data from the device. This may be data that is buffered in a queue, or may be read directly from hardware.
Write	Write data to the device. The device may buffer the data for later writing, possibly via an interrupt, or when a specific command is issued via Ioctl.
Ioctl	The catch-all command. Any command that doesn't fit one of the other functions formats. This function may be used for the read/write/close calls, as well, as ioctl is the generalized command case. Some systems have a function table, rather than an a generic Ioctl command for all of the subfunctions.
Callback	Some way of notifying an upper-layer API that an event has occurred. This would usually be done as an extension of the ISR, but may be invoked from periodic polling, or some other mechanism. This could be implemented as a message to kernel to schedule interrupt handler. E.g. OS/2

One flaw with the above model is that it is lacking support for asynchronous commands that follow more of a request/response nature. The request/response mechanism is especially important when a bus is involved and there are multiple devices sharing the same bus. At a minimum, an Abort call must be added to terminate a that has not yet received a response.

Abort	Cancel a pending request.

Roll-your-own API – Lower Layer

The lower-level API (how the device driver actually communicates with the physical device) is slightly more complicated. Devices need to access registers on the device. These registers are usually 8, 16, or 32 bits in size. Device registers are mapped onto system I/O ports (either memory or I/O space), so some mechanism for translating between these ports is required. Device registers may be in big-endian or little-endian and the system ports may be in big-endian or little-endian. These need to be the same and if they are not, then a swap function must be inserted to perform the operation. A device register may be a different size from a system port, translation for this discrepancy must be taken into account.

Devices are often interrupt driven and need some way to attach an interrupt service routine to a interrupt identifier. If a device must support restart or shutdown, then an unattach function is also required.

Devices need some way to mark a critical section. If device interrupts are handled in-context, then the way to mark a critical section, is by disabling the device interrupt, or alternatively, all interrupts on the CPU depending on what the shared resource is that makes the region a critical one.

Addresses may need to be translated from system memory map onto the device memory map and vise-versa.

Lower-level API (Device Helper Functions)

```
// Pseudo data structures to describe device and system device is on
struct DevReg {
        DevRegId
        register_offset
        register_size
        register_endian
};

struct DevDesc {
        system_endian
        system_bus_size
        system_bus_type
        device_base
        DevReg registers[]
};
```

<u>inDevicePort(*data, DeviceRegId)</u>
Based on the description of the device and the description of the system that the device is on, read a device register, performing any conversion operations that are required.

<u>outDevicePort(*data, DeviceRegId)</u>
Based on the description of the device and the description of the system that the device is on, write to device register, performing any conversion operations that are required.

<u>isrAttach(ISR, InterruptId)</u>
Associate an interrupt service routine (ISR) to a system interrupt.

<u>isrDetach(ISR, InterruptId);</u>
Break the association of an ISR with a system interrupt.

<u>deviceInterruptEnable( )</u>
Enable device interrupts. This is used for leaving a critical code section of a device.

<u>deviceInterruptDisable( );</u>
Disable device interrupts. This is used for entering a critical code section of a device.

<u>systemInterruptEnable( )</u>
Enable system interrupts. This is used for leaving a critical code section among multiple devices.

<u>systemInterruptDisable( );</u>
Disable system interrupts. This is used for entering a critical code section among multiple devices.

<u>systemToIO( )</u>
Convert a system mapped address to an I/O space mapped address.

<u>IOToSystem( )</u>
Convert an I/O space mapped address to a system address.

<u>ISendMsg( )</u>
Notify an "Interface" that an event occurred. Could be implemented as a message, a callback handler, or a schedule to another task.

These functions are used as helper functions to develop a "generic" inDevicePort abstraction.

inSystemPort8(Port)
inSystemPort16(Port)
inSystemPort32(Port)
	System functions that read 8-bits, 16-bits, or 32-bits from a system I/O address space map.

outSystemPort8(Port,value)
outSystemPort16(Port,value)
outSystemPort32(Port,value)
	System functions that write 8-bits, 16-bits, or 32-bits from a system I/O address space map.

endianSwap8( )
endianSwap16( )
endianSwap32( )
	System functions that swap the endianness of a fundamental data type

Conclusion

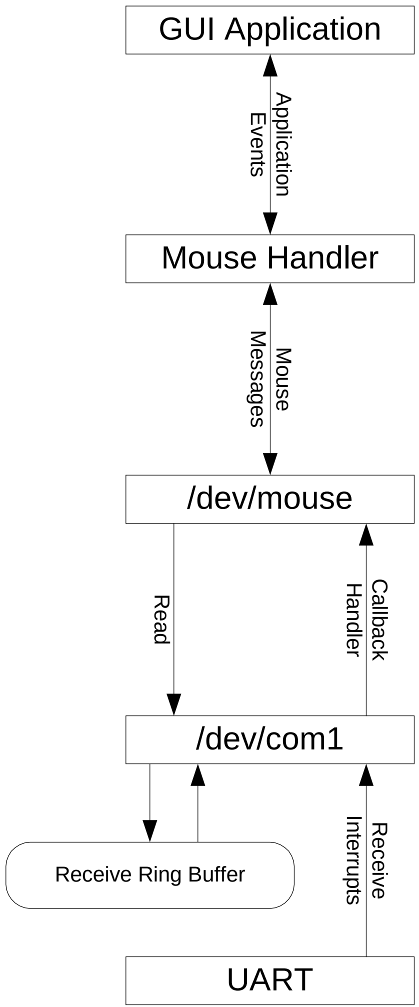
Summary of issues

- Application "device processing" is not at the same level as "device handling" (Interface to device)
- Devices are accessed differently on different platforms, even though they are the same hardware device (Hardware Abstraction Layer)
- Device may need to interact with Kernel, i.e. extend the kernel and work with applications

Summary of solutions

- Lower level API
    - Callback handlers
    - Device helper functions
    - ISRs and Interrupt handlers

- Interface Layer
- Upper level API
    - Init/Term, Open/Close, Read/Write, Ioctl, Callback, Abort



A fictitious mini-case-study

A common problem among embedded applications is that an application is developed, and then the platform is changed. The changes range from minor, where an updated device is added, to major, where a complete redesign, involving new hardware, new CPU, and even a new target customer, is involved. Time-to-market requirements demand that these changes reuse as much as possible from the old design in the new. This is possible by using software layering, which forms the basis for the rest of this paper.

Software layering is a process where a problem is broken apart into separate logical layers that communicate through a well defined interface. This has the advantage that the interface remains constant and independent of the underlying implementation. This means that if there are two components (layers) communicating with each other, then changes in one component do not directly affect the other, provided the interface does not change.

In the example shown in **figure 1**, a GUI application has been written that requires input from a pointing device. For this example, a mouse handler has been developed that handles mouse messages from a mouse driver (/dev/mouse). The mouse driver is acting as an interface that processes serial characters (from /dev/com1), interprets them as a serial mouse driver protocol stream, and translates the results to messages that are sent to the mouse handler. The com1 interface simply provides a hardware abstraction of the UART chip.

This is a very simplified application, in that the mouse driver is an input only device, but representative of a real-world environment.

Figure 1 -- Simple layered application using a serial mouse

Now our manager is coming back and says, "Great new idea! Our application has to support PPP at 230kbps, bi-directional transfer. The in-house networking GURU already has PPP added, now it is up to you to add the device. We added another UART, just like the first one so it will be easy for you to add support for it, right?. By the way, we need it by next week and it can't exceed our already crammed footprint." How should we respond to this "great news?"

How we respond to the "great news" is very dependent upon how the rest of the system is architected. It is often the case with legacy software that the "layers" shown in **figure 1** below the mouse handler are all crammed together into one "really efficient" implementation. In fact, that probably was the most efficient implementation at the time that the code was originally written, but now the requirements have changed and it is required that two interfaces are be supported by a single device class, the UART device. A place to start with a system like this is to leave the mouse driver code alone, and add completely new PPP driver code. This incurs a lot of redundancy, code that is included twice, in the two drivers that could be using the same code for handling the interrupt, maintaining the ring buffers, etc. A more efficient implementation, would reuse all of this code, and maintain separate data structures for each device.[5]

Assuming the decision has been made to use the layered approach, there are a number of issues that arise that require significant amounts of thought and effort. The remainder of this paper will discuss these issues and present alternative solutions to each of the problems that arise. Hopefully, after we are armed with that information, we will have a good basis for writing device drivers and device driver frameworks. And after our software has been designed with that infrastructure, we will be able to give our boss the "thumbs up" message, and not lose any sleep over it, because as we will see, it is a "no

[5] As a side note, consideration for implementation *should* also take into account future extensibility and maintainability. As a good case in point, look at the pickle that the scenario is in now.

brainer" to add initial PPP UART support, and that should be enough to show our boss that it will work and we can add the more advanced line and signaling features later.[6]

Comments on modularity

There are many benefits to having a modular design including maintainability, flexibility, and testability. The primary argument against "modular" programming is that it isn't as efficient. That may be true, but careful consideration should be taken into account to determine how efficient it must be. Usually a modular solution is efficient enough. Martin Fowler claims that one cannot optimize until it is modular. (Fowler) It is easier to measure, and then optimize, modularized code. This method also makes better usage of a engineer's time, because only the "slow" functions are optimized. "Engineer's are notoriously wrong when picking what should be optimized." In addition, the biggest gains are from more efficient algorithms, not from putting everything together in a monolithic chunk. Modularization allows for reuse so that new algorithms can be coded in a reasonable amount of time.

Once all of the benefits of modularity are realized, careful use of object-oriented languages can add significantly to the usability of devices. For example, instead of needing three differently named functions to access a port, the access function *In* can be overloaded for each of the data types. Devices can be thought of as objects. This can lead to an abstract base class of device…remember to be careful though! (*Effective C++* by Scott Meyers)

Comments on efficiency

At first glance, it may appear that a monolithic approach is the most efficient approach. Traditionally, a "customized" solution is better than a "generalized" solution. However, monolithic systems usually wind up with artificial constraints in them. The constraints are introduced by the design, and the complexity of the design. Instead of winding up with a $O(n_1 + n_2)$, the nesting and interdependencies wind up with a O(log(N)) or even $O(N^2)$ design.

Another factor is that there is overhead associated with processing data. A layered approach is usually data driven rather than code driven. Programmer's often make the mistake that the inefficiency in the data storage (additional data items such as pointers) makes the implementation more inefficient. However, these pointers are usually present in the code, but may be obscured by being in the form of a switch statement. A dereferenced pointer is a direct translation, while a switch statement may be an iterative one…remember to be careful though!

[6] If you decide to use the layered approach, but your current system is implemented in the monolithic style, then there is also a transitioning problem. Martin Fowler presents a number of refactorings that can greatly help the transitioning.

Future of Device Drivers

The future of device drivers is writing ones that are generic. That means that they will most likely follow an industry standard such as UDI. Many manufactures of devices already distribute device drivers that work with Windows and Linux. If operating systems used a model such as UDI, then manufactures could distribute a single driver that would work on any operating environment. Currently, this model requires source distribution of the device driver code.

The next step beyond a single model device driver is binary distribution. This requires a binary interface specification for distribution in addition to a standard API to access the device. Using a technology, such as .NET or Java, could solve this problem.

References:

Refactoring by Martin Fowler
XP Installed – “Embrace change” by Kent Beck.
Sun’s Writing Device Drivers

DLPI Open Group Specification http://www.opengroup.org/onlinepubs/009618899
Uniform Driver Interface http://www.projectudi.org/
  Linux implementation http://www.stg.com/udi_index.html