Where Hardware and Software Meet: The Basics

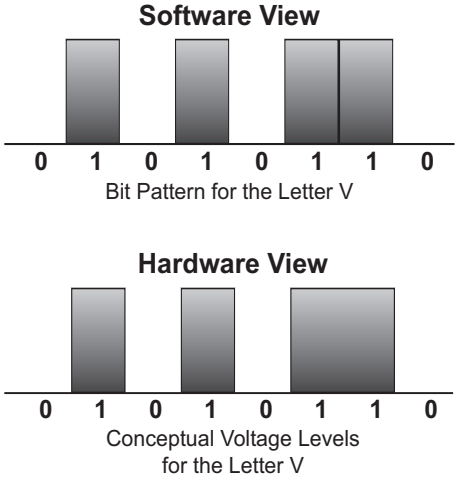
Mike McNair
Science Applications International Corporation

Effective integration of hardware and software requires an understanding of fundamental concepts such as a bit, address, and interrupt. Using these concepts in the context of protocols and applications is what makes an interface useful. This article looks at these fundamental concepts from a software view and then applies them to simple applications where they can be used to expand into other application domains and uses of hardware and software.

From a software perspective, it is important to understand two differing models: one that reflects a software engineer's representation of a system in software, and the devices and processes of the actual system. At its core, software is a model represented in terms of data structures and algorithms. Ideally, these constructs parallel something physical and do so with a high degree of accuracy. The reality is that software not only models a physical construct but also contains its own information (a *meta construct* of sorts) to help manage that physical model. In a simple example, a byte array may represent a message, but in the software model there may be bookkeeping to track the length of the array or a pointer to the body of the message contained within the array. These features are used to help manage manipulation of the byte array but are not a part of the actual stream of bytes as transferred over an interface.

Understanding the interplay between hardware and software requires an understanding of not only the hardware but the model used to represent it. This software model attempts to represent the actual hardware as closely as possible. For the software, the hardware model is represented by target or memory maps, interrupt handlers, addressing schemes, board support packages (BSPs), and other constructs. Throughout this article, these will be discussed in a context that hopefully pulls together a generic view of a hardware model from a software perspective. It is by understanding this view that we understand how hardware and software work together at their lowest levels.

Figure 1: *String of Bits for Letter V*



Basic Concepts

There are some basic hardware features that are simple yet commonly misunderstood by software engineers. If a software developer can grasp the fundamental characteristics of bits, interrupts, and addresses, they will have the building blocks needed to understand how hardware and software interface with each other. The following discussion is generic due to the wide range of hardware that is available, but explains concepts based on common features and approaches to current hardware.

Basic Concept: What Is a Bit?

To a software developer, a bit is a value – usually thought of as 1 or 0 and abstracted as on/off, true/false, or other paradigms. In hardware, a bit is not only a voltage level but is a voltage at a certain time. In the hardware, a value of 1 is assigned a certain voltage, and 0 is assigned another voltage. As the voltage levels change, there is a timing clock that tells the hardware when to sample the signal. Depending on the sensed voltage, the hardware identifies the signal as being either in a 1 state or a 0 state.

Graphically, consider the following for the bit pattern of the letter V (decimal ASCII code 86). In binary, this would be represented as 01010110. This string of bits can be depicted from both a software and hardware view as shown in Figure 1.

In order to transition from the software model to the physical model of actual voltages, the hardware has timing circuitry that *cuts* the voltage stream into bits and identifies those bits. By putting all three forms together, their relation becomes a little clearer.

With this more integrated view, there are other concepts that begin to come into play. For example, the concept of *data rate* is not just how many bits can be transferred per second, but how fast the timing circuit can create pulses so the hardware is able to sense voltage levels and, therefore, discretely identify bits as shown in Figure 2. Grasping the concept of a bit really is the key to understanding most interfacing concepts and issues.

Basic Concept: What Is an Address?

An address, simply put, is the location of something. From a software perspective, addresses represent the placement of various things: executable code, interfaces, interrupt handlers, data, etc. While software typically treats an address as a means of labeling, hardware uses addresses to actually locate things – whether it is where a wire connects to a processor, where data is stored in memory, etc.

In many applications, actual addresses are *hidden* through the use of identifiers in source code, virtual addressing, and other schemes; the point being that most application software is not truly concerned with the real address but just requires access to the address. The closer the implementation gets to the hardware, however, the more important it is to know where things are stored and to represent that location with the actual physical address. It is at this level that artifacts like memory maps and BSPs become extremely useful.

Digressing for a moment, a memory map can be thought of as an allocation of memory regions. Throughout the possible range of physical addresses, certain uses of the memory can be assigned to one region or another. The operating system (OS) and application are either constrained or assumed to honor these allocations. A BSP is a special extension of a memory map as it includes not only an identification of the memory regions, but

static data and code used by the underlying OS to manage the application. Things like maximum stack size for function calls or device addresses can be found in a BSP.

Simplistically, memory is conceptually divided into volatile and non-volatile memory. Volatile memory (e.g., random access memory) loses its contents when there is no power while non-volatile memory (disk drives, memory sticks, etc.) has a means of preserving its contents across power cycles. Of course, there are various forms of each kind of memory. When these are all interconnected in a computer system, the overall range of addresses (physical address space) grows.

From a software view, memory has to be looked at as not just the type of memory (with its associated capabilities and constraints), but also how that memory is used. The OS usually manages memory for high-level applications, but at a low level it is important to understand memory type and memory use. In essence, memory management is a manipulation of statically allocated and dynamically allocated memory regions (see Figure 3). Statically allocated memory regions usually contain things like object code, interrupt mapping tables, static data, etc. Dynamically allocated memory regions include data areas created during run time – things like dynamically sized queues, linked lists, and other similar constructs.

When a hardware device is accessed, commands are sent to a specific address associated with that device. The addresses available for this are usually reserved and protected from other uses by segmenting the memory into regions as shown in Figure 3. Physically small microprocessors can have as few as 10-15 pins; more capable, general-purpose processors can have more than 100 pins. Some of these pins are designated as data lines and address lines. In the actual circuitry these lines are physically connected to memory devices, system buses, device controllers, and other system components. It is through these connections and the ability to specify addresses of specific locations that devices can be controlled and monitored.

Most things that can be manipulated by software have an address. Of course, a specific piece of data has an address in memory. Likewise, hardware interfaces are known by the address of their control and data lines. The processor can be manipulated by knowing the address of its registers. Interrupt handlers must be placed at specific addresses. Think of knowing an address as the key to controlling and using a device or function in the computer system.

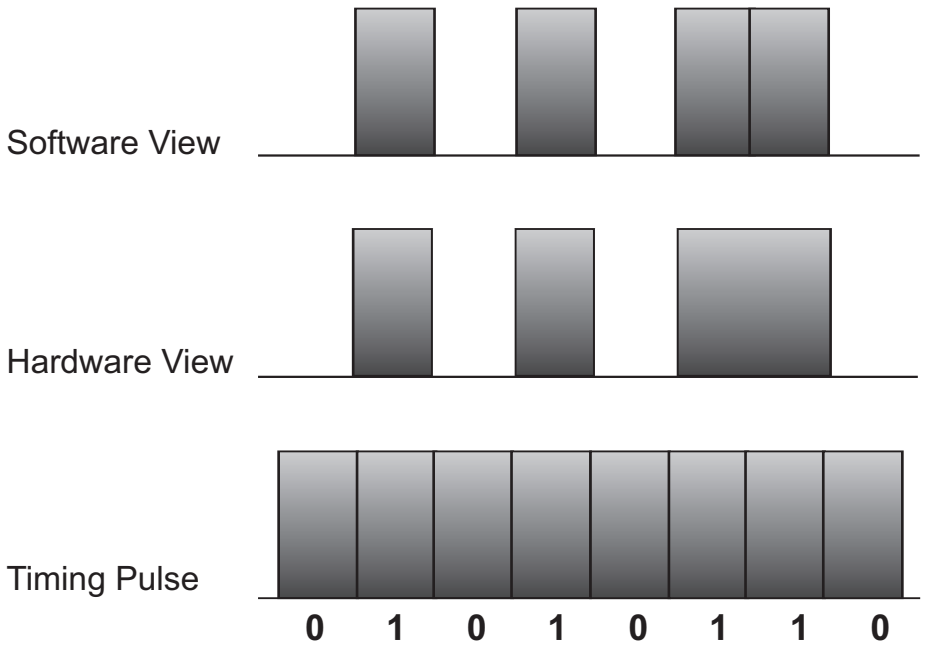


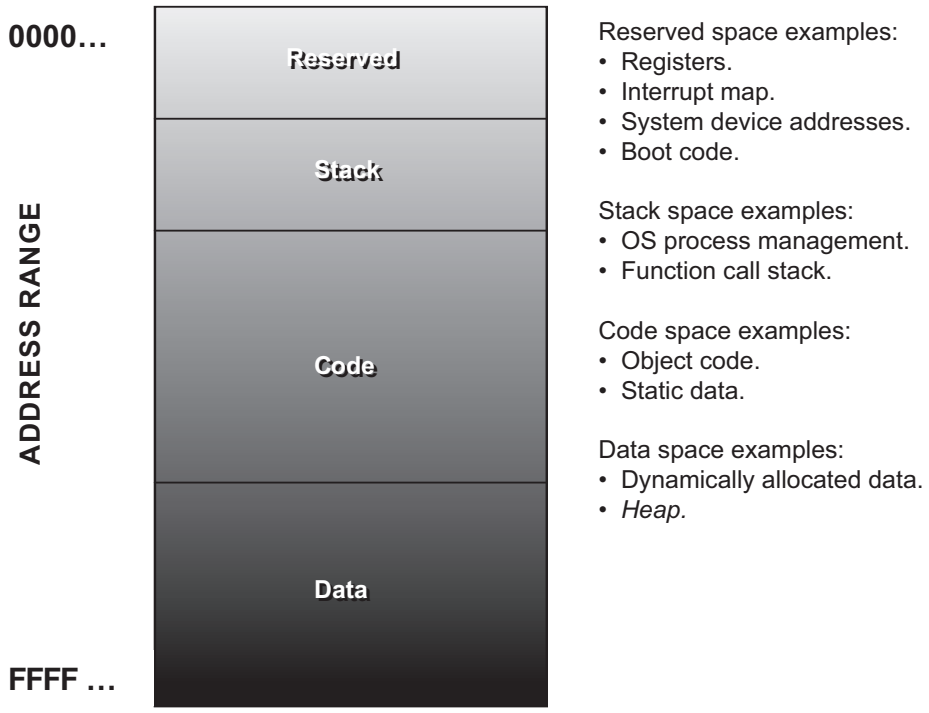
Figure 2: *Software and Hardware Views for the Letter V*

Basic Concept: What Is an Interrupt?

Some interfaces do not transfer data but instead are designed to transfer control signals. Even in a simple RS-232 serial interface, there is more on the cable than just a stream of bits. Some of the individual wires carry control signals. In the hardware, these signals are designated with voltages – just as with bits. The change in voltage on these control wires (transitions) act to signal an event. In order for the hardware to notify the processor that a transition has occurred, it generates what is known as an interrupt. As long as the sensed voltage does not cross whatever the voltage threshold is, there will not be an interrupt.

Using Figure 4 (see page 8) as a reference, consider what happens during the handling of an interrupt. Within the processor, an interrupt is generated when the hardware senses a voltage transition – usually through a control line (1 in Figure 4). When this control signal is sent to the processor, the processor generally interrupts some or all of its processing (depending on the priority of the inter-

Figure 3: *Memory Regions*



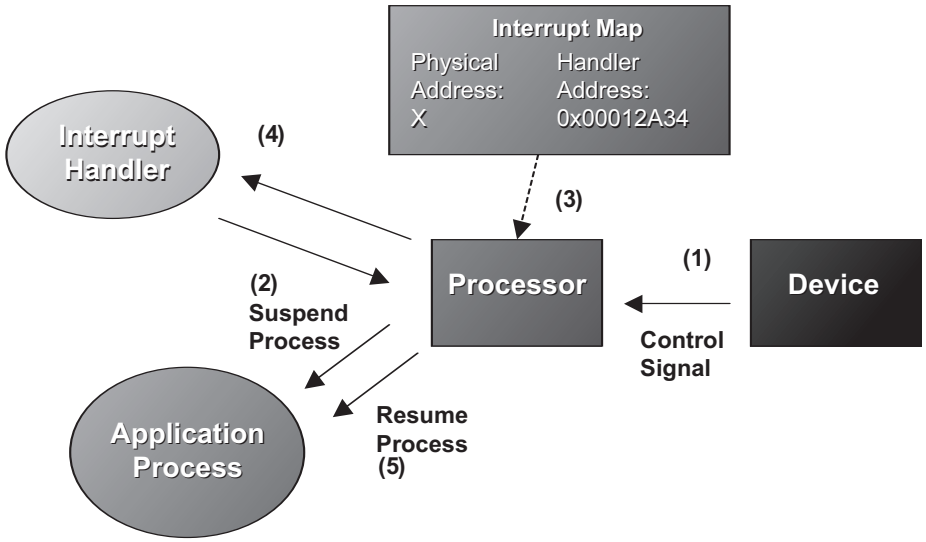


Figure 4: *Handling of an Interrupt*

rupt) in order to do something with that signal (2 in Figure 4). Most OS allow for the concept of an interrupt handler. Since this signal enters the processor at a particular physical address, there must be a means of mapping the physical address of the control signal to the address of an interrupt handler (3 in Figure 4). The OS then handles passing control from whatever code is currently being executed to the interrupt handler by using a table lookup that cross references the physical address of the signal to the start address of the interrupt handler code (4 in Figure 4). When the interrupt handler has completed its execution, the OS resumes execution of the interrupted code (5 in Figure 4).

Because interrupt handlers truly interrupt what a processor is doing, they are usually written to be executed quickly. Generally, an approach of temporarily storing data or state and then exiting is common. The longer an interrupt handler takes to complete, the less time there is for other processing within the computer. At an application level, this can have a tremendous impact on any time critical processing. At a low level, it is possible to have interrupts arrive at a high-enough rate that if a handler takes too long, the next interrupt will not be handled. Should this happen, an event is missed – possibly with corresponding data. In a mission-critical system, either changing critical processing timelines or missing an interrupt can be disastrous.

Putting It All Together

Many real-world systems rely on software controlling some hardware device through an address or interrupt. With the advancement of technology, many variations exist on these themes, but once these basic concepts are understood, it is relatively simple to expand the concept into the new implementation.

As a way of bringing the simple concepts in this article together, briefly consider two examples: turning on a light emitting diode (LED) and turning on a motor. While these may seem uninteresting, it is important to realize that an LED can indicate whether power is applied to a device, a weapon system is ready to fire, or a strobe light effect can be achieved by simply turning the LED on and off. Likewise a motor can be used to control the spin of media in a CD player, the speed of a wheeled vehicle, or the arm of a robotic device.

Example: LED Control

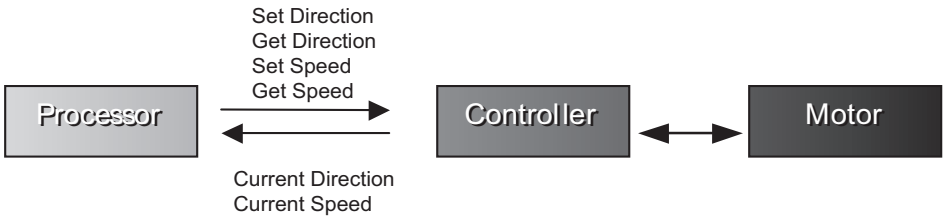
An LED is a simple device that only needs power applied to it to turn it on and power removed from it to turn it off. Since a bit is actually mapped to a voltage, a very simple implementation for controlling an LED would be to wire the electrical interface to a specific address or port on the processor where the controlling code is executing. By writing a 1 to that port, the LED can be turned on; by writing a 0 to that port, the LED can be turned off. The port in this case can be either an actual address or processor register. Either way, the operations are simply achieved by writing a value to a specific location. By including a delay between the ON and OFF writes and placing the code in a loop, it is possible to *blink* the LED at a desired rate.

Example: Simple Motor Control

For software to control a motor, there almost always needs to be a hardware controller that provides a somewhat intelligent interface between the processor executing the software and the motor itself. As a result, the application does not talk directly to the motor but instead talks through a controller. Consider a simple model of a motor that has these operations: set motor speed, set motor direction, get motor speed, and get motor direction. The controller provides the interface for actual control of the motor in response to these simple commands. As with an LED, the controller allows command messages to be written to a specific address and return data (due to the *get* commands) be sent at another address.

From within the application, the command to set the direction of the motor

Figure 5: *Retrieving a Current Value*



Message Set

Name	ID	Data
Set Direction	01	0 = clockwise spin 1 = counter-clockwise spin
Get Direction	02	no data
Current Direction	03	same as Set Direction
Set Speed	04	0 = stop 1.255 = values for speed
Get Speed	05	no data
Current Speed	06	same a Set Speed

* A message for this motor would be composed of a byte for Message ID and a byte for Message Data.

spin is achieved by sending the correct command message to indicate direction. Likewise, motor speed is achieved by sending a command message with a quantification of speed (maybe 1.255 for a byte data field where 0 = stop). Retrieving the current value of either of these is achieved by sending the proper command message and reading for the returned value from the controller – each usually at two different addresses as shown in Figure 5.

Given these operations, an accelerator pedal on an electric car, for example, can be used conceptually like a joystick to supply speed values to the motor. The controlling software merely reads the amount of pedal depression and converts it to a value that is appropriate for the motor controller. (Changing the vehicle direction from forward to reverse could be a button that the software reads the state of and then sends an appropriate motor command to the motor controller.) For the operator of this vehicle, a separate task can periodically read the value for motor direction and motor speed, scale the speed as needed to map it into the appropriate units (km/hr), and display the result to an operator.

Making It Happen

A key piece to implementing low-level interfaces is the support provided by the processor, OS, and development environment. Processors will vary in terms of address width (8 bit, 16 bit, 32 bit, etc), addressable memory range, number of registers, and number of interrupt lines (etc.), where these are in addition to characteristics like processor speed, cache size, and physical memory. Each of these characteristics must be matched up to overall system performance, number and type of external devices, and other considerations.

In Conclusion: Expanding the Applications

In order to facilitate the management of low-level interfaces to devices, a set of code is written to encapsulate and handle the nuances of the interface and device. This abstraction is referred to as a device driver. To its applications, a device driver presents a set of operations to control and manage the device, but many times actually communicates with the controller of the device and not the device itself. At its lowest levels, a device driver handles the intricacies of handling interrupts, formatting and parsing command messages, providing sequencing as required, and performing other device specific activities. A device driver is simply a device or interface manager that is built on the manipulation of bits, addresses, and interrupts.

The simple concepts of bit, address, and interrupt cover most types of hardware/software interfaces at a low level. Overlaid on these low-level constructs are various protocols (message and communication protocols, Universal Serial Bus, and other device and application specification paradigms) to control more sophisticated coordination on both sides of the interface. Even though the addition of protocols presents complication to the implementation of the interface, the interface itself is fundamentally represented in terms of control (signals and interrupts) and data (streams of bits and addresses).◆

Additional Reading

1. Embedded.com <www.embedded.com>.
2. Programmer's Heaven <www.programmersheaven.com/zone7/index.htm>.
3. Device Software Optimization <https://dso.com>.
4. Heath, Steve. Embedded Systems Design. 2nd edition. Newness, June 2002.

About the Author

Mike McNair is a senior systems engineer for Science Applications International Corporation where he is a part of the chief engineer team for unmanned ground vehicles on contract to the U.S. Army. His background includes more than 20 years of experience as a programmer, technical lead, and software program manager on projects ranging in size, complexity, and target for a variety of customers. He has also served on several process improvement (including Software Engineering Process Group lead) and training initiatives.

Science Applications International Corporation
8303 N Mopac
STE B-450
Austin, TX 78759
Phone: (512) 366-7834
Fax: (512) 366-7860
E-mail: michael.k.mcnair@saic.com

Coming Events

February 5-9
RSA Conference 2007
San Francisco, CA
www.rsaconference.com/2007/US/

February 10-14
13th International Symposium on High-Performance Computer Architecture
Phoenix, AZ
www.ece.arizona.edu/~hpca/

February 13-15
SE 2007 The IASTED International Conference on Software Engineering
Innsbruck, Austria
www. iasted.org/

February 21-22
Warfighter's Vision 2007
Winning the Global Fight
Alexandria, VA
www.afei.org/brochure/7a04/index.cfm

February 26 – March 2
ICCBSS 2007
International Conference on COTS-Based Software Systems
Banff, Alberta, Canada
www.iccbss.org

June 18-21, 2007
2007 Systems and Software Technology Conference

Tampa Bay, FL
www.sstc-online.org

Coming Events: **Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. E-mail announcements to nicole.kentta@hill.af.mil.**