A driver-based approach to protocol stack design

By Curt Schwaderer
Director of Network Technologies
Microware Systems

The addition of network connectivity has the potential to open up new markets and enable new and exciting uses for many products. As a result, network connections of one sort or another are becoming standard requirements for a wide variety of embedded systems. However, making a system network-ready isn't easy. A variety of performance, reliability, and interoperability issues must be considered to make a product behave successfully in the field.

In this article I will describe a new approach to protocol stack design. This approach can be called driver-based, to distinguish it from the more traditional task-based approach. I'll describe both approaches to protocol stack design in detail and then compare them with respect to efficiency and real-time response.

The task-based approach

Before we can compare the technical aspects of the two design alternatives, we must first understand each of them. The traditional approach to protocol stack design involves placing the stack above the real-time operating system or kernel. Unfortunately, most commercial RTOSes don't provide any sort of framework for networking, so protocol stack implementers have had to make do with the minimum functionality that is provided by any RTOS: tasks.

Figure 1 illustrates how a set of tasks can be used to implement the layers of a three-layer communications protocol. Each layer is implemented as a separate task, and intertask communication mechanisms are used to send data and control packets up and down the protocol stack. It's up to the implementer to define the interfaces between the layers, as well as an API for the application programmers to use to send and receive packets over the network.

Several significant sources of inefficiency exist here. First, as the dotted lines in Figure 1 attempt to show, the underlying operating system is busily performing context switches as each packet is shuffled between the application, the upper layers of the communications protocol, and the device driver for the network interface. Each time the RTOS suspends execution of one of these tasks and resumes execution of another, time is wasted saving and restoring task contexts. Since each packet that is sent or received over the network must pass through all of the layers of the protocol stack, these context switches constitute a large amount of overhead.

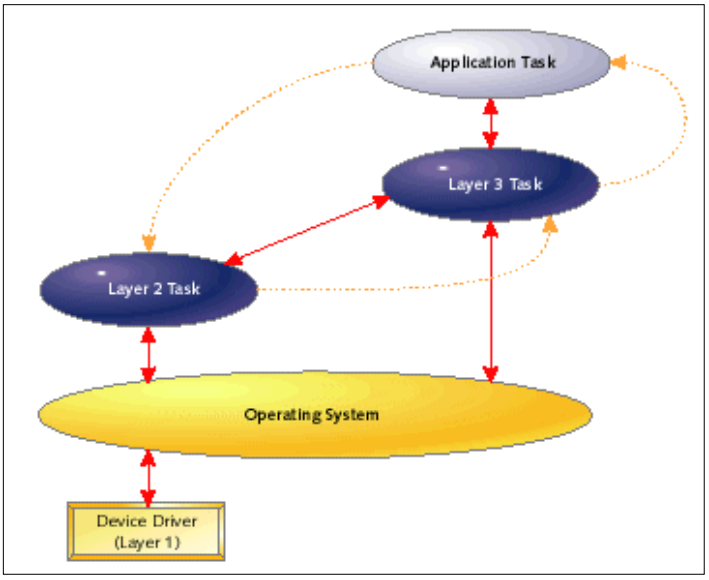

Figure 1

In addition, as data and control information flows between the application task and the network interface, buffers containing that information must be repeatedly added to and removed from intertask communication queues. Unfortunately, the cost of having the operating system move data buffers into and out of intertask communication queues is quite high. This problem is only compounded by the fact that the enqueue and dequeue operations themselves contain critical sections that must be protected from interrupts and context switches. So not only is time wasted putting the data onto and taking it off of a series of queues, but the overall system becomes less responsive to important events like interrupts.

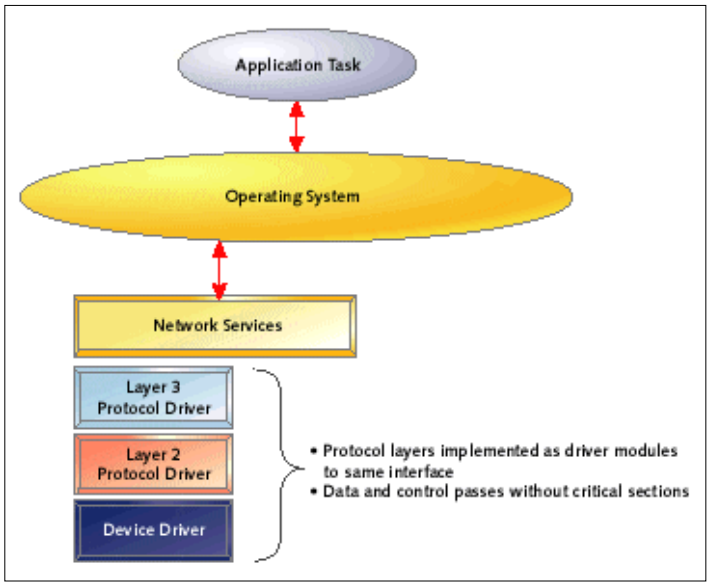

Figure 2

The driver-based approach

An alternative to the task-based approach is to relocate the layers of the protocol stack within or below the real-time operating system. **Figure 2** shows what the same three-layer communications protocol would look like if it were re-implemented in this way. The crucial difference here is that the individual protocol layers are implemented as driver modules instead of tasks. They are layered directly above the device driver for the associated network interface and become, essentially, a part of that driver.

One additional change is that a network services module is added above the protocol stack. The purpose of this module is to abstract those features of the network that are protocol-independent. In other words, it's a way to standardise the API that the application programmer uses to send and receive data over the network. The main advantage of this approach is that it makes it easy to have more than one type of protocol stack in the system. For example, your embedded system might support both PPP over a modem interface to connect to remote computers and an

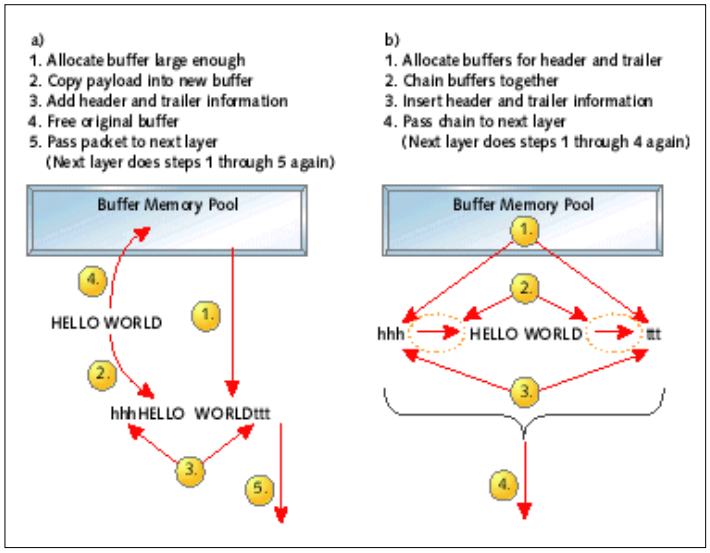

Figure 3

IrDA interface for communication with local computers. But the application programmer need not program differently for each of these cases. She simply uses the API provided by the network services module to communicate with some other computer. The only difference is through which network interface the application is communicating.

A noteworthy feature of the driver-based approach is that the number of context switches is a function only of the number of controlling applications in the system, rather than the number of protocol layers. The advantage here is that you minimise the number of times the RTOS must save and restore task contexts, allowing the extra time to be spent doing more useful things, like executing application code.

Another advantage of the driver-based architecture is that data and control information passes more easily between the layers. Since all of the layers operate within the same context, the relevant data structures are automatically accessible to the layers above and below. Consequently, you don't need to pass them on intertask queues and you avoid those extra critical sections altogether. Eliminating the critical sections improves the response time of the system to interrupts and high-priority tasks. I'll make quantitative comparison of the efficiency and responsiveness of the two architectures later. Before doing that, I should talk about some other potential sources of inefficiency within a protocol stack and how they can be eliminated within the driver-based architecture.

Buffer copying

The first potential inefficiency involves the allocation, copying, and release of data buffers as data is passed between the layers of the protocol stack. This inefficiency can arise regardless of the stack's architecture, since it has nothing to do with the mechanics of the buffer transfer between layers. It has only to do with the structure of the buffers themselves.

Two common methods for moving packets of data down the protocol stack are shown in Figures 3a and 3b. Both of these methods are flawed. In each case, the application task has some data that it wants to send. This is often called the message or payload. The message is sent to the highest layer of the protocol stack as an array of characters. Since no extra space is left in that buffer for the header and trailer the protocol layer needs to attach to it, the layer must either:

- Allocate a buffer large enough for both the message and its header and trailer, copy the message into it, and add the header and trailer information, or
- Allocate two small buffers, one for the header and one for the trailer, and then chain the three buffers together with pointers

To the layer below, the result of the highest layer's copying or chaining is just a new, larger payload. Each layer adds its own header and trailer to the payload sent by the layer above. So this inefficiency, if present, will be repeated many times for each packet sent over the network. (It is much easier to deal with data that is received over the network, since lower-layer headers and trailers can simply be ignored by the upper layers. The payload gets smaller on the way up the stack, rather than larger.) In both cases, time is spent in the memory allocation routine. The copying technique also wastes time repeatedly copying the contents of the growing payload and freeing older buffers. The chaining approach doesn't incur the extra copy, but does complicate the device driver code that actually transmits the packets. This may be particularly wasteful if the network interface chip supports DMA transfers.

An alternative solution, which fits in well with the driver-based approach, is shown in **Figure 4**. Every time a protocol stack is created or altered, the network service module initiates a protocol stack query to determine the header, trailer, and maximum transmission unit (MTU) requirements for the entire stack. That way, when an application sends a message to the stack, the network services module can simply allocate a pair of buffers that are large enough for the header and trailer requirements of the entire protocol stack, respectively. Each layer simply inserts its header and trailer information into these buffers without memory allocation or copying. This facility alone provides significant performance improvements.

Retransmission buffers

A second potential inefficiency involves protocol layers that provide acknowledgement and retransmission facilities. Implementations of a reliable protocol layer typically involve the layer allocating a second "retransmission buffer" for each packet sent and copying the contents of the sent packet into it. If the peer layer on the remote system acknowledges proper receipt, this retransmission buffer will be destroyed. But if a timeout or "non-acknowledgement" (NACK) occurs instead, the reliable protocol layer sends the contents of that buffer once again (including allocating a new retransmission buffer and copying the resent packet into it).

Packet copies held strictly for potential retransmission can be eliminated if sent packets can be "marked" or "reserved" by the reliable protocol layer. That way no copy is made, but a pointer to the packet is kept. When the device driver finishes transmitting the packet and attempts to free the buffer, the buffering system recognises that the buffer is reserved and instead of freeing it, merely marks it as "transmitted." When the reliable protocol layer receives acknowledgement, it simply "unmarks" the packet and the buffer is returned to the free pool. By building this feature into the network services module, the efficiency of all future protocol

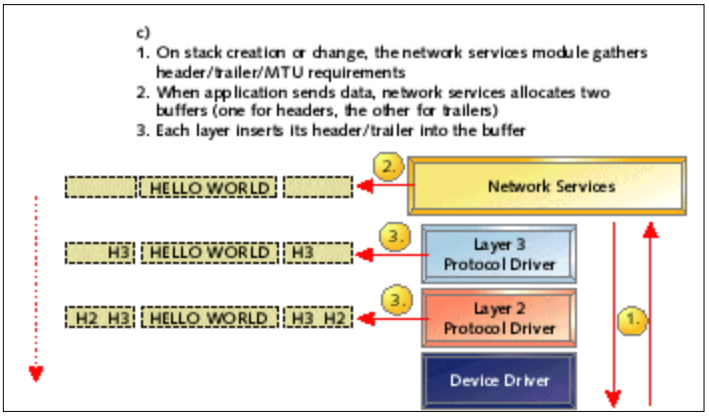

Figure 4

Table 1 Comparison based on instruction count

Section of Code	Task-based	Driver-based	Advantage
Protocol processing TX	473	241	49% fewer instructions
Protocol processing RX	633	401	37% fewer instructions
Critical sections TX/RX	220	241/401	See next two lines
Best-case latency	320	241	25% fewer instructions
Worst-case latency	640	401	37% fewer instructions
Total stack instr. count TX	673	241	64% fewer instructions
Total stack instr. count RX	833	401	52% fewer instructions

Table 1

stack implementations will be improved.

Quantitative comparisons

I will now undertake to compare the traditional task-based approach to protocol stack design with the driver-based approach I've just described in a quantitative manner. For this purpose, I will use the instruction counts for an actual implementation of each. Although instruction count does differ from processor to processor, it still provides reasonable grounds for comparison in this case.

Table 1 contains the instruction counts for a pair of competing implementations of a UDP/IP protocol stack on a PowerPC 860 processor running OS-9. In the case of the driver-based implementation, you'll see that there are a total of 241 PowerPC instructions in the transmission portion of the stack and 401 in the reception portion. These numbers recur throughout the table, because they are the same instructions that are executed in every instance. There is much more variation within the task-based implementation.

Efficiency. Looking only at the first two rows of the table, we see that the driver-based implementation requires fewer instructions both for sending and receiving messages. Fewer instructions executed to get data and control information up and down a protocol stack means two things: data and control packets move faster throughout the system and the overall executable code size is smaller. In terms of data processing through a UDP/IP stack, the driver-based implementation executes almost 50% fewer instructions through the data path. This is to be expected because the context switching and critical section protection overhead has been eliminated altogether. Even the call from the application task to the protocol stack is cheaper; some things change as the system switches from user mode to kernel mode, but this is not a full context switch.

Latency. Now let's focus on the real-time response comparisons in rows three through five. The third row of the table contains the number of instructions within each implementation's critical sections, that is, the number of instructions that must be executed without interruption. These numbers provide a basis for comparing the best and worst-case response time of the system. In the task-based UDP/IP stack, the critical sections are within the enqueue and dequeue operations. Whereas in the driver-based stack, the entire protocol stack is a critical section.

At first glance, the critical section lengths seem to indicate that the task-based UDP/IP stack offers better real-time response (i.e., shorter latency). However, a true measure of determinism must also include any time it takes to switch between the tasks that make up the task-based stack. Because of that, the next two rows of the table tell the real story with respect to latency. The best-case numbers there include the critical section length and semaphore lock/unlock code (100 extra instructions) for a total of 320 instructions. That is the best case, in which the UDP and IP layers are implemented within the same task. The worst-case numbers add to that another 320 instructions for an additional packet transfer plus semaphore lock/unlock. So it turns out that the driver-based implementation also has better worst-case performance than its task-based competitor.

The important point here is that critical section length is not the only relevant factor when determining latency. The need for semaphores (to protect data) and context switches also factor into any latency calculation. Protocol stacks are generally optimised to do as little processing as possible during data transmission and reception. And the results in Table 1 indicate that the overall latency can be reduced if you allow the protocol stack simply to run to completion as opposed to incurring the overhead of context switches.

The driver-based protocol stack imple mentation wins the quantitative comparison hands down.

Internetworking

It's increasingly common that systems require multiple protocol stacks. Because no common, protocol-independent API has existed, protocol stack vendors have all invented their own. As a result, equipment requiring the use of multiple protocols (e.g., voice over IP (VOIP), TCP/IP over ATM, and WAN access) must spend large amounts of time doing what the industry terms internetworking these protocol stacks. Internetworking involves mapping two or more disjoint protocol interfaces together; it can be difficult, expensive, and performance-degrading.

In addition, almost 30% of developers deal with some kind of proprietary protocol in their designs. As Internet connectivity is added to the feature list of these systems, the custom protocols must be able to interoperate with standard TCP/IP stacks and, sometimes, wide-area networks as well. The task-based approach requires the custom protocol to be re-engineered into the frameworks of each new required protocol. But the driver-based approach may already have structures and interfaces (within the network services module) that allow custom protocols to be developed and immediately interoperable with other, perhaps vendor-supplied, protocols.

Details, details, details

Any reasonably good driver-based protocol stack will contain similar data structures, data and control passing primitives, and module functions (or entry points). So I wanted to give you a

Table 2: Protocol driver entry points and communication calls

Network Infrastructure Driver Entry Points		
Entry Point	**Parameters**	**Description**
DrInit( )	Device entry	Initialised protocol state machine/hardware
DrTerm( )	Device entry	Deinitialised protocol state machine/hardware
DrGetstat( )	Device entry, parameter block	Retrieve control information
DrSetstat( )	Device entry, parameter block	Set control information
DrUpdata( )	Device entry, data container	Process received PDU going up the stack
DrDowndata( )	Device entry data container	Encapsulate PDU going down the stack
Inter-driver Communications Macros		
Macro Call	**Parameters**	**Description**
SMCALL_UPDATA	Device entry, driver above device entry, data container	After processing, pass PDU up the stack to the next driver
SMCALL_DNDATA	Device entry, device below device entry, data container	After encapsulation, pass PDU down the protocol stack
SMCALL_GS	Device entry, driver above or below device entry, parameter block	Pass control retrieval request up/down the stack
SMCALL_SS	Device entry, driver above or below device entry, parameter block	Pass control setting request up/down the stack
Receive Interrupt Service Routine Call-up Macro		
Macro	**Parameters**	**Description**
DR_FMCALLUP_PKT	Device entry, driver above device entry, data container	Put data on receive queue of network infrastructure receive thread of execution

Table 2

feel for what these might be.

Data structures. These are the types of data structures you will need:

- Device entry provides the conduit between the RTOS and a particular protocol layer module
- Driver static is allocated only once for each protocol layer, regardless of the number of network interfaces beneath the protocol. It is the protocol layer's global storage area
- Logical unit static is allocated on a per-interface basis. So, if you have a driver that is controlling two interfaces, there will be two of these, but only one each of the driver static and device entry data structures
- Per path static is allocated once for every application task's invocation of the protocol

As these four types of data structures are defined, protocol-specific data should be placed into the most appropriate type. The particular data structure that's selected should be a direct function of how the variables are used: by the protocol state machine as a whole, on a per-interface basis, or per-application. For example, the base address of a particular network interface chip in memory would be part of the logical unit static.

Functions. Obviously, if you'll be implementing more than one protocol stack, it is helpful to develop a set of function interfaces that is both generic and full-featured. **Table 2** describes the data and control passing primitives and entry points for a driver-based protocol stack framework we developed for use with OS-9. This list should give you a good idea of the types of function interfaces that are needed at the network services level and between the layers of the protocol stack.

Efficient and responsive

As new requirements appear and technological advances occur, we must challenge existing ways of doing things. The driver-based approach to protocol stack design introduced here challenges the traditional method that relies on tasks. This new approach has been shown to be more efficient and more responsive and, if implemented as suggested, is also more adaptable to the requirement of multiple protocol stacks in the same system. Hopefully, the next generation of connected embedded systems will not suffer the same problems as those of the past.