Dr. Dobb's Portal
The World of Software Development

Developing Custom Network Protocols

Although custom protocols are common in the world of real-time embedded systems, creating them can be a problem. For Curtis, the solution is to create protocols that are interoperable with other protocols written for the same framework.

By Curtis Schwaderer, Dr. Dobb's Journal
Jul 22, 2001
URL:http://www.ddj.com/184411047

Curt is director of network technologies at Microware. He can be reached at curts@ microware.com.

Many engineers use custom protocols in their embedded development. However, creating custom protocols can be a challenge -- even when the protocol itself is simple. The lack of a defined software framework for custom protocols makes efficient implementation difficult and interoperability with other standard protocols (such as TCP/IP) nearly impossible. Luckily, there are alternatives. In this article, I present one such alternate approach to creating custom protocols -- one that uses a standard communications software framework for creating custom protocols that are interoperable with other protocols written for the same framework. I'll start by identifying some of the key concepts behind communications frameworks, then turn to the nuts and bolts of how it can be implemented. In the process, I'll present a sampling of the data constructs and calling conventions for applications, as well as the data constructs and interdriver calling conventions of the individual protocol drivers. The code and makefiles for the examples presented here are available electronically from *DDJ* (see "Resource Center," page 5) and from me (send e-mail to curts@microware.com).

Requirements and Concepts

Among other things, efficient communications software frameworks must:

- Maximize network throughput. Ideally, the implementation of the communications framework will push bits through the network pipe at the maximum data rate of the network pipe. For example, if the network pipe is 10 mbits/sec., the throughput of the data you are sending -- plus the messaging overhead of the protocols -- should be as close to 10 mbits/sec. as possible.

- Provide universal interoperability across protocols and vendors. Interoperability is a huge concern for custom-protocol development. For instance, once a network interface is added to a device, the next logical feature is connecting to the device over the Internet. Operation over multiple network topologies also demands interoperability. Protocols used for these network topologies are typically purchased from various protocol vendors. Protocol stack vendors have put together their own custom interfaces, requiring custom protocols to accommodate all communications interfaces involved. This complicates the development.
- Provide a familiar and easy-to-use API and enable applications to be written in a network-independent fashion. Network technologists tend tobe proud of their protocols, exposing every detail to the application programmer. However, the vast majority of applications simply want to connect or disconnect by referring to simple connection type (data, voice, video, and so on).
- Minimize CPU utilization, memory requirements, and development time. The first three requirements may seem contrary to this one, but a discussion of the potential approaches and final implementation will also be analyzed based on this criteria.

Figure 1 illustrates the historical approach to implementing protocols. This approach involves writing protocols as tasks (or processes) written to an operating-system abstraction. Unfortunately, this approach has inherent inefficiencies:

- Because the protocol layers are written as tasks, they're timesliced. As a result, each layer must protect critical sections when passing control or data between task queues. Most implementations use semaphores to protect critical sections.
- An important performance benchmark is not only the context switch time itself, but also the number of context switches as data flows up and down the stack. As data flows from the application through the protocol layer tasks, context switches occur between each task. If many protocol layers are involved, multiple task switches occur before the packet is completely processed.
- Protocols typically implement timers. When the timer expires for a given layer task, another context switch occurs. Many times, context switches to the protocol layer task only for the task to find out there is nothing to do.
- The operating-system abstraction layer is a source of inefficiency because it makes up for the deficiencies of the OS. If the OS supports a service request directly, a wrapper in the OS abstraction layer is still provided, unnecessarily adding to the instruction count. If the service is not supported by the OS, you must implement it.

Figure 2 illustrates an alternate approach to implementing protocols -- one that is driver based. This architecture eliminates the inefficiencies associated with the task-based approach by implementing each protocol layer as an individual driver module that can be dynamically stacked/unstacked at run time by the application. Furthermore, driver-based architectures address the inherent inefficiencies that task-based designs present:

- Protocol layers are implemented in system space. Most operating systems turn timeslicing off when executing in system state. As a result, data processing up and down the protocol stack can be performed as one continuous thread of execution. There is one potential drawback -- if data processing for the protocol stack is intensive, the real-time nature of the OS may be compromised. However, real-world protocols are optimized to be very fast when sending or receiving data. (Studies conducted by Microware, the company I work for, show a 25 percent increase in real-time response over the task-based architecture. This is mostly due to eliminating the critical section code and context switches.)

- Protocol layers are implemented as modules (drivers) with defined entry points. As control or data passes through the protocol layers, direct jumps into the protocol layers above or below can be made using the defined entry points. With the driver-based approach, only one context switch from the application to the kernel is made regardless of the number of protocols.
- Timer services are implemented as part of the network infrastructure module. Since the module is executing in system state, timer processing can be implemented more efficiently than if timer services were implemented as part of an OS abstraction layer.
- The defined communications software framework replaces the OS abstraction layer. Services provided by the framework are implemented optimally by the OS. (After all, who knows the OS better than the OS vendor?)
- A significant amount of code overhead is allocation and initialization of memory for the state machine. The protocol layer specifies the amount of data needed and a pointer to the initialized data area within the module. Armed with this information, the network infrastructure module can allocate and initialize memory for any protocol layer. The allocation and initialization code is in one place, as opposed to having each layer containing the same code, decreasing the system's ROM/RAM requirements.

A service helpful to any protocol layer should be implemented by the network infrastructure module or kernel. This way, the network environment incurs the memory cost once, instead of inside every protocol layer. And unlike the operating system abstraction layer, the network infrastructure is tightly coupled with the operating system to allow maximum performance advantage.

Driver-based network architectures are not necessarily new. UNIX System V STREAMS, for instance, provides a classic case of a driver-based framework for nonreal-time operating systems. Although STREAMS architectures have been ported to other real-time operating systems, their implementation is task based, using only the syntax of STREAMS, not the driver-based architecture. In this article, however, I'll use Microware's mwSoftStax network I/O system (provided with the OS-9 real-time operating system) as an example driver-based communications framework to illustrate some of the implementation details behind this approach.

mwSoftStax consists of a network-independent API, a network infrastructure module (the Stacked Protocol File manager, or SPF), protocol layer drivers, and a network-interface hardware driver. The protocol layers are implemented below the kernel in system space. The OS abstraction layer is replaced with the SPF module that interfaces with the kernel to provide a standard protocol stack environment.

Driver-based architectures satisfy the general requirements for software frameworks for custom protocols in the following ways:

- The driver-based architecture eliminates performance inefficiencies associated with the task-based architecture. In performance comparison testing done using the UDP/IP protocol and OS-9, the driver-based architecture was between 22-28 percent faster than the task-based architecture.
- Since the communications protocol framework is fully specified, external protocol interfaces (entry points and data/control passing, for instance) are identical, regardless of the vendor. One added benefit is that OS abstraction layers implemented by the protocol vendor have a direct mapping to the communications protocol framework service definitions for easier porting of commercial protocol stacks into one universally interoperable operating system environment.
- In the case of the mwSoftStax API, the paradigm is based on something everyone is familiar with -- the telephone. The API was designed to be network-independent, enabling simple connectivity with calls that enable applications to do more complicated quality of service configuration through the getting and setting of profiles if required. Profiles are simple tags that imply a complete set of quality-of-service (QoS) parameters for the type of network connection being requested.
- By tightly integrating the communications protocol framework into the operating system, minimal ROM/RAM footprints can be achieved. Also, the performance benefits outlined help minimize the CPU bandwidth required to run the network interface, potentially reducing the cost of the CPU required for the application.

Implementing a Communications Software Framework

Abstracting application-visible aspects of any network is the key to making network independence a reality. Abstractions for the mwSoftStax network device and network addressing were created using structures called *device_type* and *address_type*. The third data structure is the asynchronous notification method called a *no*tify_type structure. This provides a level of operating system independence.

All of the parameters in the *device_type* and *address_type* structures are automatically initialized when the application opens an instance of the protocol stack. Because automatic initialization occurs as an implicit kernel service, applications need not be aware of these two structures. This allows applications to still operate in their simplest form. If required, the API contains calls to get/set all variables within the *device_type* and *address_type* structures.

The *notify_type* structure is used by the application for network event registration/removal. Notification requests can be set for:

- Link down/link up.
- Incoming call.
- Connection active/far end hang-up.
- Data available to be read.
- End of real-time data stream (MPEG/ audio/video).
- Flow control on/off.
- Custom protocol or device-driver network events.

mwSoftStax uses an API called "Integrated Telephony Environment for Multimedia" (ITEM), which was developed using the telephone paradigm for interactive television and video-on-demand set-top box communications application development. There are also options for using the native OS-9 system calls as well as a standard socket interface to control the network.

Listing One shows three different ways an application can invoke a protocol stack consisting of driver A under driver B under driver C. Listing Two shows how simple network-independent sender and receiver applications can be written for a driver-based communications framework. The profile get/set calls are available in the API to set QoS parameters for the protocol stack. The protocol stacks are designed to have a default service profile that can be generically used by an application. However, if the application wants to be specific about specific QoS parameters, the calls in Listing Three can be used to create custom profiles.

There are four main driver data constructs:

- The RTOS-comm framework glue structure: The device entry. This is the cornerstone data structure for a protocol driver. This structure is allocated by the operating system for each instance of an application using this protocol in their protocol stack. Pointers to the driver static and logical unit static are found in the device entry.
- Protocol global structure: The driver static. This is allocated by the operating system once per protocol driver. This structure contains pointers to the executable entry points of the protocol driver. Macros were created as part of the communications framework specification to call drivers above and below in the stack to pass data and control using the entry points in the driver-static storage. If the driver uses any globally defined variables, they are stored in the driver static.
- Protocol per-interface structure: The logical unit static. Many protocols store unique state machine variables on a per- interface basis. To use the same module to run multiple interfaces, the operating system automatically allocates one logical unit for every interface the protocol will be using.
- Protocol per-application invocation structure: The per-path storage. This structure is used to keep information on a per-application-use basis. For each unique path opened by an application that uses the protocol, a unique per-path storage structure is allocated by the protocol. The driver stores the device entries of the protocols above and below it so that data and control can be passed between protocols in a fast and efficient manner.

Interdriver Control/Data Passing

Table 1 shows the data and control passing macros used by a protocol driver. Notice that the macros use the device entries to pass control between drivers. Listing Four is the code behind the simple macros that the drivers use to pass code up and down the stack. The macros sift through the device entry to the driver static of the upper or lower device entry to find the entry point of the driver above in the stack. It sets up the important pointers in the logical unit of the callee driver and performs a direct jump to the executable entry point. This approach provides an easy-to-use and high-performance dynamic environment for protocol stack operation.

Protocol Driver Entry Points

Table 2 lists the protocol driver entry points and parameters. The initialization and termination entry points get called on the first and last invocations of each unique interface; see Listing Five. Protocol drivers typically do nothing in these functions because the storage structures are automatically initialized by the OS. Network device drivers typically initialize and deinitialize registers for each unique interface in these functions.

Protocols and applications pass control through the *dr_setstat()* and *dr_getstat()* entry points. Listing Six shows the *getstat* entry point. Note that the network infrastructure module (SPF) gathers information about the protocol stack using the SPF_GS_UPDATE call. In this way, SPF can make sure the protocol data unit containers passed down can be handled by every protocol in the stack without requiring a copy at any layer, resulting in a much higher performance system. Any custom control commands can be created by defining the primitive value and including it in the *switch* statement of the *getstat* or *setstat* routines.

The *setstat* entry point is identical to the *getstat* entry point in structure, but *setstat* handles any settable or gettable commands, whereas *getstat* handles only gettable commands. *setstat* by default handles protocol stack manipulation such as open, close, push, and pop.

The *dr_downdata()* and *dr_updata()* entry points are called when protocol data units (PDUs) are being sent down and up through the protocol, respectively. Data containers for the OS-9 environment are called "mbufs." The mbuf facility is a fast preallocated memory facility for improved performance. Listing Seven presents the *dr_downdata()* entry-point function code.

DDJ

Listing One

(a)

```
/* The A, B, and C are module names that invoke drivers A, B, and C. */
/* S_IREAD | S_IWRITE are permissions for accessing path, in         */
/*         this case read/write access.                              */
/* path is a handle given back by the OS that identifies protocol    */
/*         stack instance for the app                                */
Ite_path_open("/A/B/C", S_IREAD | S_IWRITE, &path);
(b)
Ite_path_open("/A", S_IREAD | S_IWRITE, &path);
Ite_path_push(path, "/B");
Ite_path_push(path, "/C");
(c)
/* Where /stack is name of a module which contains explicit stack /A/B/C. */
/* How an app invokes whatever protocol stack is required for operation   */
/* in a network independent fashion.                                      */
Ite_path_open("/stack", S_IREAD | S_IWRITE, &path);
```

Back to Article

Listing Two

```
/* NETWORK SENDER APPLICATION CODE */
void main(int argc, char **argv)
{
    /* main program variables:
     * ite_path     = path to our DEVICE
     * snd_size     = used to remember the size of our data send packets
     * snd_buffer   = data send buffer
     * stack_size   = used to remember the length of stack string
     * stack_buffer = stack buffer
     * err          = used for error checking
     */
    path_id     ite_path;
    u_int32     snd_size;
    char        snd_buffer[32];
    u_int32     stack_size;
    char        stack_buffer[256];
    error_code  err;

    /* Most apps will need a signal handler due to asynchronous nature of
     * using a network device. Reset any global notification flags to zero!
     */
    if ((err = _os_intercept(sighand, _glob_data)) != SUCCESS) {
     printf("Error %03d:%03d installing signal handler\n", err/256, err%256);
     exit(0);
    }
    /* Shows how to set up protocol stack using driver-based architecture */
    /* Two processes used in this demo: staxsend and staxrecv */
    /* staxrecv should already be running on the target. */
    /* Both processes are using the ITEM Application User Interface */

    /* 1. First, specify a protocol stack */
    printf(" * Type in the desired stack (i.e. /loopcl5): ");
    fflush(stdout);
    stack_size = 255;
    _os_readln(0, stack_buffer, &stack_size);
    stack_buffer[stack_size - 1] = '\0';
    printf(" */\n");

    /* Now invoke the protocol stack. We only need the path to be writeable*/
    if ((err = ite_path_open(stack_buffer, FAM_WRITE, &ite_path,
                                                    NULL)) != SUCCESS) {
        printf("\n\n    FAILURE\n");
        printf("    Error %03d:%03d on ite_path_open()\n", err/256, err%256);
        printf("    Could not open stack [%s]\n", stack_buffer);
        if (err == 215) {
            printf("    Make sure all descriptors in your stack are
                                         valid and in memory\n\n");
        }
        exit(0);
    }
    fflush(stdout);
    snd_size = 31;
    _os_readln(0, snd_buffer, &snd_size);
    snd_buffer[snd_size - 1] = '\0';
    snd_size--;

    /* Write the data down the stack */
    if ((err = ite_data_write(ite_path, snd_buffer, &snd_size)) != SUCCESS) {
       printf("\n\n    FAILURE\n");
       printf("    Error %03d:%03d on ite_data_write()\n", err/256, err%256);
       if (err == 246) {
            printf("    You probably haven't started
                                          the staxrecv application\n");
            printf("    Type [procs -e] and make sure staxrecv is running
before executing staxsend\n\n");
       }
       exit(0);
    }
    /* close the path and exit */
    ite_path_close(ite_path);
    exit(0);
}
/*NETWORK RECEIVER APPLICATION */
void main(int argc, char **argv)
{
    /* main program variables:
     * dev_name     = pointer to the name of our DEVICE
     * ite_path     = path to our DEVICE
     * rcv_size     = used to remember the size of our data receive packets
     * snd_size     = used to remember the size of our data send packets
     * rcv_buffer   = data receive buffer
     * snd_buffer   = data send buffer
     * err          = used for error checking
     */
    char        *dev_name = DEVICE;
    path_id     ite_path;
    u_int32     naptime, exit_flag = 0;
    signal_code sig;
    u_int32     rcv_size;
    u_char      rcv_buffer[32];
    error_code  err;

    /* Most apps will need a signal handler due to asynchronous nature of
     * using a network device. Reset any global notification flags to zero!
     */
    if ((err = _os_intercept(sighand, _glob_data)) != SUCCESS)
    {
        printf("Error %03d:%03d installing signal
                                            handler\n", err/256, err%256);
        exit(0);
    }
    printf("    /* 1. First, a path to the following stack is
                                                    opened. */\n", DEVICE);
    if ((err = ite_path_open(DEVICE, FAM_READ, &ite_path, NULL)) != SUCCESS) {
        printf("FAILURE\n");
        printf("Error %03d:%03d on ite_path_open()\n", err/256, err%256);
       exit(0);
    }

    /* Loop continuously until user terminates application */
    while (exit_flag == 0)  {
        printf("\n");
        printf("        /* 2. Now, wait for data to arrive on the
                                               path we've opened. */\n");
        fflush(stdout);
        naptime = 0;
        sig = 0;
        _os_sigmask(1);
        _os_ss_sendsig(ite_path, 1000);
        _os_sleep(&naptime, &sig);

        /* Ask system how many bytes are available to be read */
        ite_data_ready(ite_path, &rcv_size);

        /* Read the number of data bytes */
        if ((err=ite_data_read(ite_path,rcv_buffer,&rcv_size)) != SUCCESS) {
          printf("Error %03d:%03d on ite_data_read()\n", err/256, err%256);
          exit(0);
        }
        rcv_buffer[rcv_size] = '\0';
        printf("%s", rcv_buffer);
    printf("\n");
    /* Close the path */
    ite_path_close(ite_path);
    exit(0);
}
```

Back to Article

Listing Three

```
/* Get the default profile values for a particular profile */
error_code ite_path_profileget(path_id path, conn_type *conn,
                                   u_int32 *pr_size, void *pr_buffer)
/* Set the values for a particular profile for this path */
error_code ite_path_profileset(path_id path, conn_type *conn,
                                   u_int32 *pr_size, void *pr_buffer)
```

Back to Article

Listing Four

```
#define SMCALL(mydeventry,deventry,entrypoint,param)
(   /* update destination fields in the logical unit */
(((Spf_lustat)((deventry)->v_lu_stat))->lu_pathdesc =
                ((Spf_lustat)((mydeventry)->v_lu_stat))->lu_pathdesc),
/* call destination driver entry point */
(*entrypoint)(deventry,param)
)
/* Passing data up/down between drivers */
#define SMCALL_DNDATA(mydeventry,deventry,mb) \
(SMCALL(mydeventry,deventry,
              ((Spf_drstat)(deventry->v_dr_stat))->dr_downdata,mb))
#define SMCALL_UPDATA(mydeventry,deventry,mb) \
(SMCALL(mydeventry,deventry,
              ((Spf_drstat)(deventry->v_dr_stat))->dr_updata,mb))
/* Passing control up/down the stack */
#define SMCALL_SS(mydeventry,deventry,mb) \
(SMCALL(mydeventry,deventry,
              ((Spf_drstat)(deventry->v_dr_stat))->dr_setstat,mb))
#define SMCALL_GS(mydeventry,deventry,mb) \
(SMCALL(mydeventry,deventry,
              ((Spf_drstat)(deventry->v_dr_stat))->dr_getstat,mb))
```

Back to Article

Listing Five

```
/*  Device Driver Initialization Entry point. Any special considerations or
** initialization that the protocol needs to make to the driver static or
** logical unit static areas should be done here. Nothing path-specific:
** That's done in the setstat entry point at the SS_OPEN setstat. The SPF
** File manager will call the dr_iniz entry point of the driver only
** when the logical unit attach count is 1 (i.e. the 1st attach to a
** particular logical unit).
*/
error_code dr_iniz(Dev_list deventry)
{
    Spf_lustat lustat = (Spf_lustat)(deventry->v_lu_stat);
    /* This piece of code creates a debug data module that you can link to
     * and look at when in rombug to aid you in */
/*  troubleshooting your driver. If you'd like debugging on, you define the
 *   DEBUG macro in the spfdrvr.mak */
/* makefile and include debug_mod.l which contains the calls that
 * you'll see throughout this driver source. */
#if defined(DEBUG)
```

```
    if (debug_init(DEBUG,(Dbg_stat*)&lustat->lu_dbg,
                 lustat->lu_dbg_name) != SUCCESS) { lustat->lu_dbg = NULL; }
debug_data(lustat->lu_dbg,"PRIniz     ", (u_int32)deventry);
#endif
    /* I/O is enabled when protocol initialization completes successfully */
    lustat->lu_ioenabled = TRUE;                            /* Enable I/O */
    /* ANY CUSTOM INITIALIZATION CODE FOR THE CUSTOM PROTOCOL GOES HERE */
    return(SUCCESS);
}
/* Device Driver Termination Entry point. This entry point allows the driver
** to clean up before the operating system gets rid of this particular logical
** unit. This entry point is called by the SPF file manager only on
** the last detach from a particular logical unit.
*/
error_code dr_term(Dev_list deventry)
{
#ifdef DEBUG
    Spf_drstat drstat = (Spf_drstat)(deventry->v_dr_stat);
#endif
    Spf_lustat lustat = (Spf_lustat)(deventry->v_lu_stat);
    /* I/O is disabled on a protocol when the end-end protocol terminates */
/* Depending on the protocol, this may be on terminate, or closing of
 * the last path on a particular logical unit */
    lustat->lu_ioenabled = FALSE;                           /* disable I/O */
    DEBUG_DATA(lustat->lu_dbg, "PRTerm   ", deventry);
    DEBUG_4DATA(lustat->lu_dbg, drstat->dr_att_cnt, drstat->dr_use_cnt,
                                     lustat->lu_att_cnt, lustat->lu_use_cnt);
    /* ANY CUSTOM TERMINATION CODE FOR THE CUSTOM PROTOCOL GOES HERE */
    return(SUCCESS);
}
```

Back to Article

Listing Six

```
/* Device Driver Get-Stat Entry point. This entry point should handle any
** protocol specific getstats. The SPF file manager calls this entry
** point whenever an unknown getstat code is received. Likewise, if
** you're protocol driver doesn't understand the getstat, pass it down to
** the next lower driver in the stack.
*/
error_code dr_getstat(Dev_list deventry, Spf_ss_pb pb)
{
    Spf_lustat lustat = (Spf_lustat)(deventry->v_lu_stat);
    error_code err;
    switch (pb->code) {
        case SPF_GS_PROTID: {   /* Someone requesting my protocol ID value */
pb->param = (void *)SPF_PR_SPPROTO;
return(SUCCESS);
```

```
                           }
               case SPF_GS_UPDATE: {   /* Attempting to get update statistics  */
            Spf_update_pb   upb = (Spf_update_pb)pb;
            Spf_ppstat ppentry;

            /* Provided library call that finds the right per path storage */
            if ((err = pps_find_entry(lustat, lustat->lu_pathdesc,
                                             &ppentry)) != SUCCESS) { return err; }
            if (ppentry->pp_dndrvr) {
    if ((err = SMCALL_GS(deventry,
                            ppentry->pp_dndrvr, pb != SUCCESS) { return(err); }
                /* Now Update per my protocol status */
                /*  The smallest MTU */
                if (upb->stk_txsize > lustat->lu_txsize) { upb->stk_txsize =
                                                          lustat->lu_txsize; }
                /* If we need, We keep the header and trailer sizes below */
                ppentry->pp_stk_txoffset  += upb->stk_txoffset;
                ppentry->pp_stk_txtrailer += upb->stk_txtrailer;
                /* If there are bytes that need to be allowed the header, add
                 * them to what came from below */
                upb->stk_txoffset = upb->stk_txoffset + lustat->lu_txoffset;

                /* Send up IO DISABLED if disabled, Otherwise, send up
                 * whatever the lower one put in there */
                if (lustat->lu_ioenabled == DRVR_IODIS)
                    upb->stk_ioenabled = DRVR_IODIS;
            } else {    /* We're the lowest driver on the stack */
            /* If no one's below us, copy our stats into parameter
             * block and return */
                    upb->stk_txsize = lustat->lu_txsize;
                    upb->stk_txoffset = lustat->lu_txoffset;
                    upb->stk_ioenabled = lustat->lu_ioenabled;
            }
     return(SUCCESS);
            } /* End SPF_GS_UPDATE */
           default: {
    Spf_ppstat ppentry;
            if ((err = pps_find_entry(lustat, lustat->lu_pathdesc,
                                           &ppentry)) != SUCCESS) { return err; }
           if (pb->updir == SPB_GOINGDWN) {
            if (ppentry->pp_dndrvr) { return(SMCALL_GS(deventry,
                                                      ppentry->pp_dndrvr, pb)); }
            } else { return(SMCALL_GS(deventry, ppentry->pp_updrvr, pb)); }
            } /* End default */
    } /* End switch */
    return(EOS_UNKSVC);     /* If we get this far, return unknown service   */
    }
```

Back to Article

Listing Seven

```
/*
** Device Driver Downgoing-Data Entry point. This is the place where data
** being transmitted through the protocol is encapsulated with
** any headers used by the protocol, then sent down to the next lower driver
** in the stack. Note that the lu_txoffset field tells SPF
** how many bytes to leave at the front of the packet for the protocol
** header. Therefore, you can back up the pointer in the mbuf to add
** on the header information, saving excessive copies. The SPF file manager
** calls this whenever transmit (write) data is
** sent from the application. (or higher layer drivers).
*/
error_code dr_downdata(Dev_list deventry, Mbuf mb)
{
    Spf_lustat  lustat = (Spf_lustat)(deventry->v_lu_stat);
    Spf_ppstat  ppentry;
    error_code  err;

    if (mb) {   DEBUG_DATA(lustat->lu_dbg, "PRDnMbData", mb);
        DEBUG_MBUF_DOWN(lustat->lu_dbg, mb);
    } else {    DEBUG_DATA(lustat->lu_dbg, "PRDnMbEpty",0); }

    if ((err = pps_find_entry(lustat, lustat->lu_pathdesc,
                                                &ppentry)) != SUCCESS) {
        m_free_p(mb);           /* sorry, drop the packet */
        return err;
    }
    DEBUG_DATA(lustat->lu_dbg, "PRPpStat", ppentry);

#if 1   /* This code segment is used in a demonstration environment to
          ** show how a protocol driver can be written. It changes the case
          ** of any alphabetic character from upper to lower and vice versa.
          */
{
    u_int16 size = mb->m_size;
    u_int32 temp = 0;
    u_char  *buffer = mtod(mb,u_char*);

    while (temp < size) {
        if ((buffer[temp] >= 'a') && (buffer[temp] <= 'z')) {
            buffer[temp] -= 32;
        } else if ((buffer[temp] >= 'A') && (buffer[temp] <= 'Z')) {
            buffer[temp] += 32;
        }
        temp++;
    }
}
#endif
    if (ppentry->pp_dndrvr != NULL) {
       /* just send the data down to lower protocols */
        return(SMCALL_DNDATA(deventry, ppentry->pp_dndrvr, mb));
    } else {
        return(EOS_NODNDRVR);
    }
}
```

Back to Article

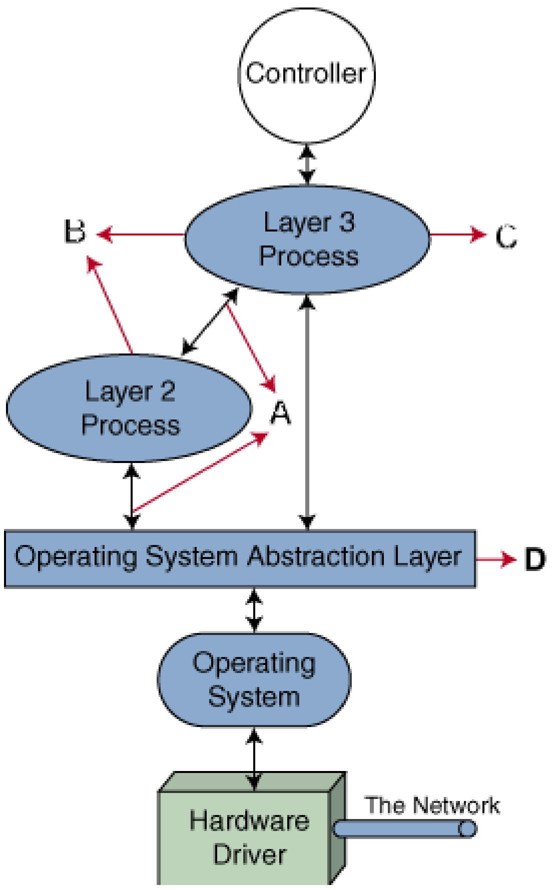


Figure 1: The historical approach to implementing protocols.

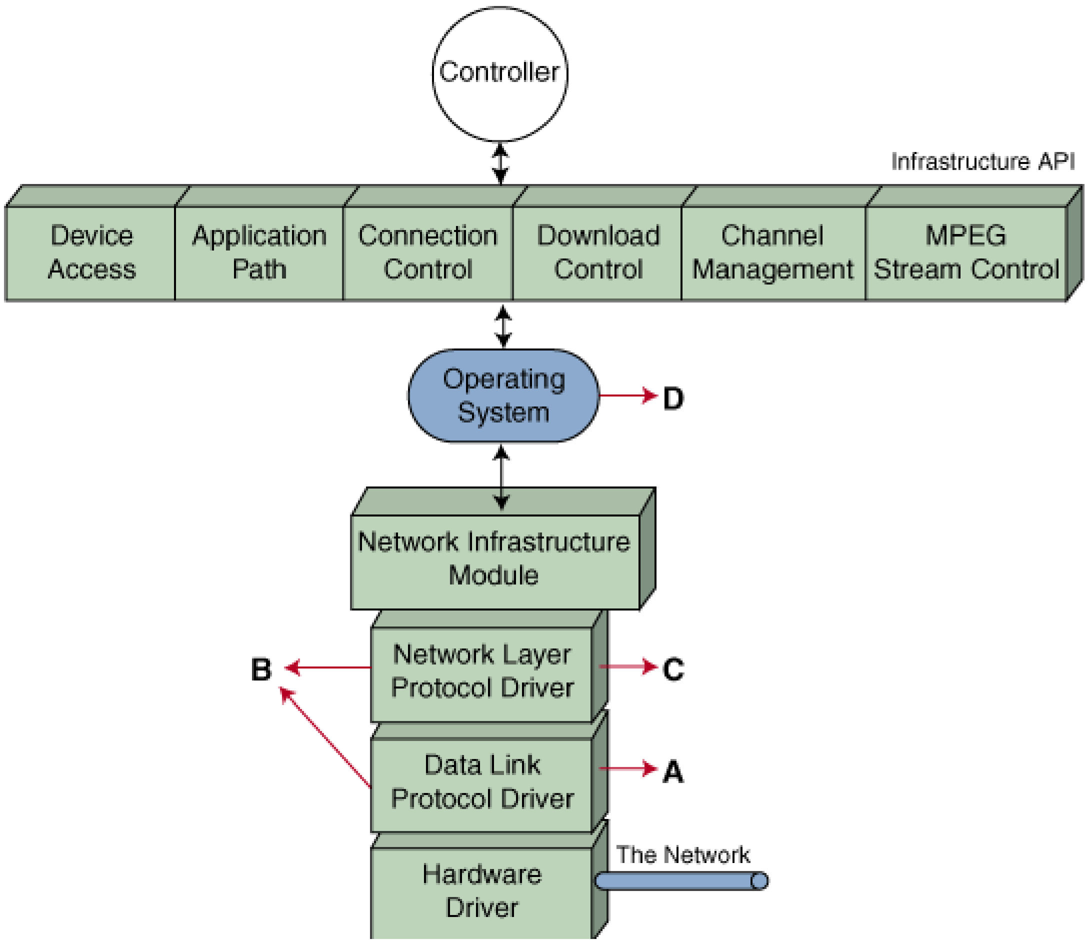
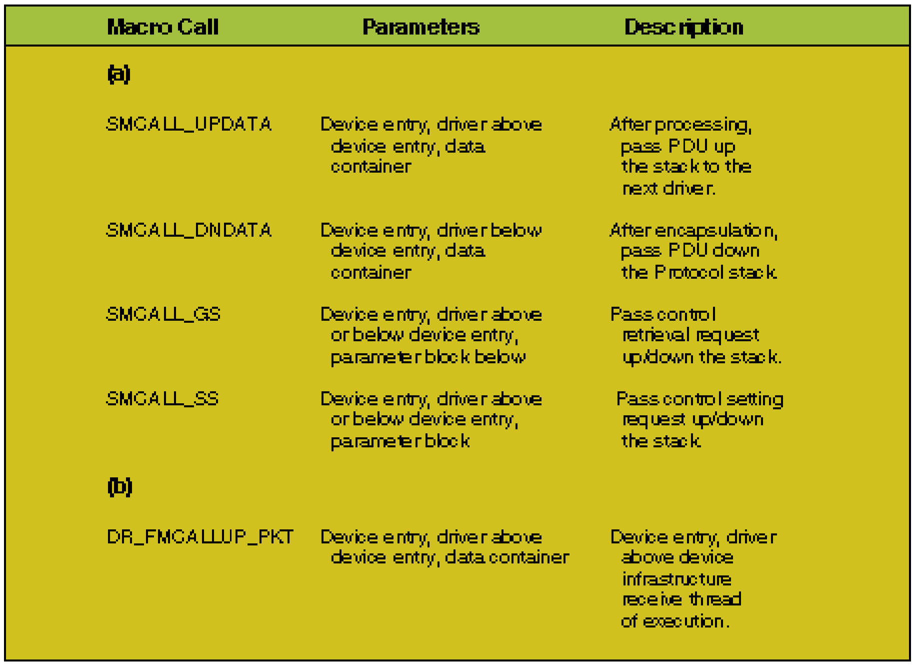
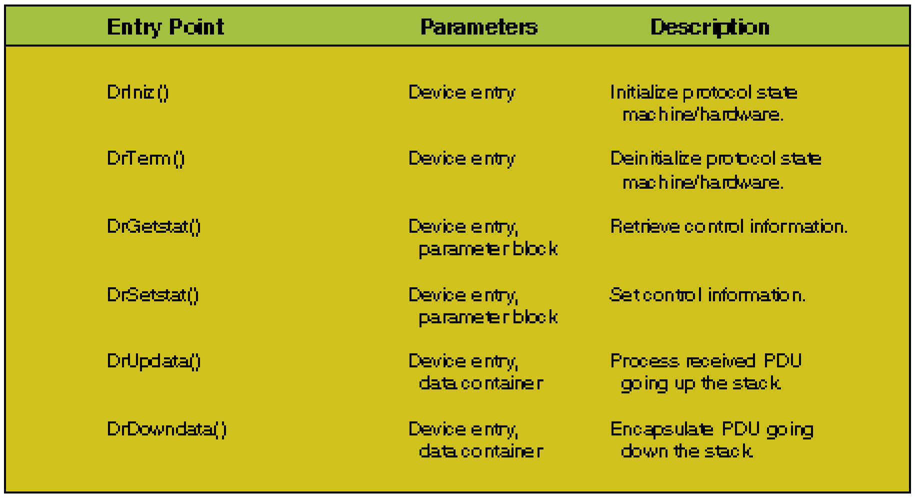


Figure 2: Driver-based protocols.



Macro Call	Parameters	Description
(a)		
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, driver 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 below	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.
(b)		
DR_FMCALLUP_PKT	Device entry, driver above device entry, data container	Device entry, driver above device infrastructure receive thread of execution.

Table 1: Interdriver data/control passing macros; (a) interdriver communication macros; (b) receive interrupt service routine macro.



Entry Point	Parameters	Description
DrIniz()	Device entry	Initialize protocol state machine/hardware.
DrTerm()	Device entry	Deinitialize 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.

Table 2: Protocol driver entry points and parameters.