SoftStax Porting Guide

Version 3.6

www.radisys.com
World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044

International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620

RadiSys Microware Communications Software Division, Inc.
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Revision A
November 2001

Copyright and publication information

This manual reflects version 3.6 of SoftStax. Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

Table of Contents

Chapter 1: Getting Started

This chapter explains the OS-9 environment and the Input/Output (I/O) capabilities of SPF file manager drivers. The concepts for porting SoftStax components and drivers to your delivery system are also explained.

What is SoftStax

SoftStax handles data received from high-bandwidth, wide-area networks as well as low speed, control channel communications. By using SoftStax, applications can reference specific stacks of required protocol modules and device drivers. The application navigates through these components using an identified data path.

SoftStax consists of the following modules:

- a file manager (SPF)
- debugging library (`dbg_mod.l`)
- ITEM library
- MBUFLIB library (`mbuf.l`)
- per path storage library (`ppstat.l`)
- timer service library (`timer.l`)
- template protocol driver (SPPROTO)
- drivers for various HDLC controllers
- connection oriented or connectionless network emulation driver (SPLOOP)

Introduction to the OS-9 Environment and I/O Capabilities

Applications interact with OS-9 I/O devices by opening a *path*. This is like opening a file handle in the UNIX system. When an application opens a path to a device, OS-9 creates a *path descriptor*. The path descriptor gets its default initialization parameters from a *device descriptor*. A *path identifier* is returned to the application.

When you request operating system services using an `_os_xxx()` I/O system call, the path identifier maps to a path descriptor. This path descriptor is passed to the file manager to process the service request.

Most devices allow many paths open to the same device. Therefore, many applications may be serviced by the same device. The device identifies the individual requests by using the path descriptor.

When the path is closed, OS-9 sends the close request to the I/O system. When the call returns, the path descriptor is deleted. The figures on the following pages are identical from the application perspective. That is, the application uses the same system calls to communicate to a disk device or network device. This is because OS-9 implements a *unified I/O* concept. From the application's perspective, it simply opens a device, gets and receives information, and closes the path to the device. Its operation is totally independent from the type of I/O system used. An application requests the same data I/O using a different I/O system and can not tell the difference.

Assume the application object will not be recompiled. In order for the unified I/O concept to work, the `/disk` descriptor describes the network (SoftStax) environment instead of the random block environment. Because the descriptor still has the same name, the application is object code compatible. However, when `/disk` is opened, it sets up a path through SPF. The I/O functionality is identical from the application's perspective.

Therefore, as shown in the following two figures, the application is binary compatible between both environments. The only things that change in the two systems are the different I/O system below and the `/disk` descriptor which contains different information for each environment.

For More Information

For more information about the I/O systems available for OS-9, contact the Microware Customer Support Department.

Figure 1-1 OS-9 Environment and Usage for a Disk Drive I/O

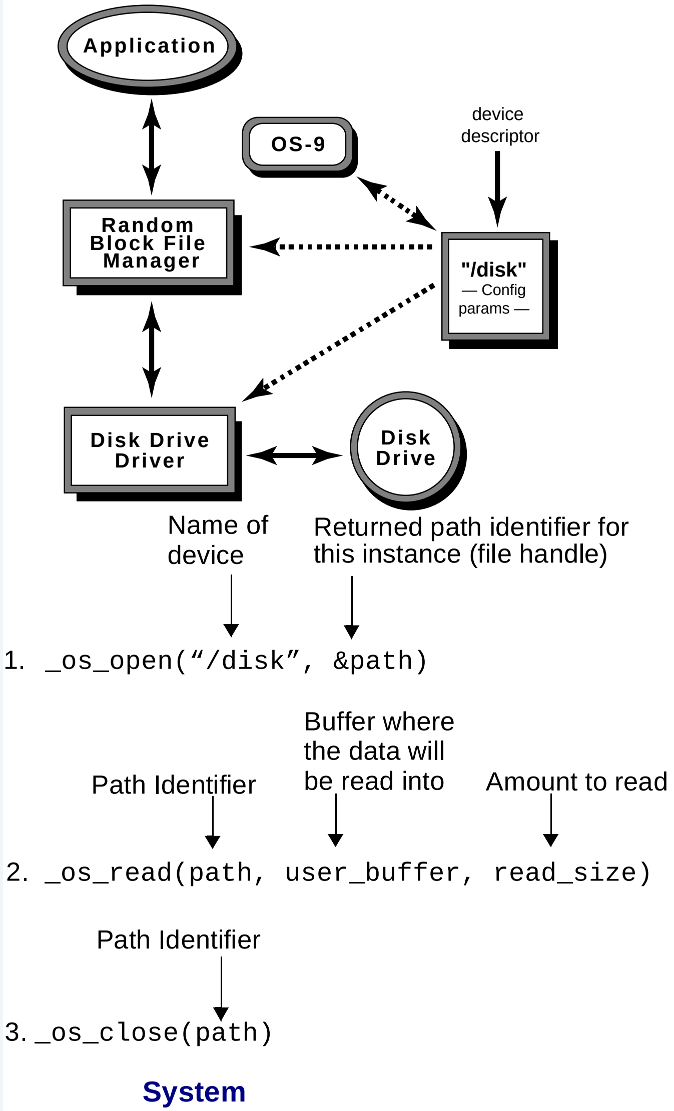
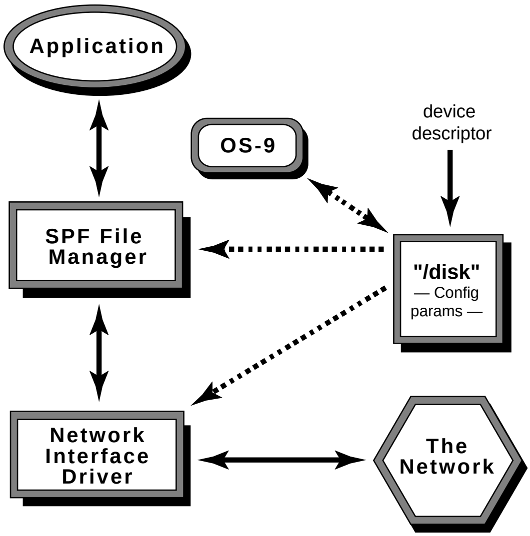


Figure 1-2 OS-9 Environment and Usage for a Networked I/O System



Name of device

Returned path identifier for this instance (file handle)

1. `_os_open("/disk", &path)`

Path Identifier

Buffer where the data will be read into

Amount to read

2. `_os_read(path, user_buffer, read_size)`

Path Identifier

3. `_os_close(path)`

Available I/O Services

Two libraries are available at the application layer, `os_lib.l` and ITEM. These libraries provide five I/O services to the application:

- device manipulation
- path manipulation
- call control
- data I/O
- MPEG program control

Service Calls

The following table lists service calls provided by the two libraries:

Table 1-1 Service Calls Provided by Library

os_lib.l	ITEM	Description
`_os_attach()` `_os_detach()`	`ite_dev_attach()` `ite_dev_detach()`	Use device descriptor only.
`_os_open()` `_os_close()`	`ite_path_open()` `ite_path_close()`	Create/remove a path descriptor instance.
`_os_dup()`	`ite_path_dup()`	Create new path identification (ID) referencing an existing path descriptor.

Table 1-1 Service Calls Provided by Library (continued)

os_lib.l	ITEM	Description
	`ite_path_clone()`	Create different path descriptor with the same connection as path being cloned.
`_os_read()` `_os_write()`	`ite_data_read()` `ite_data_write()`	Read and write data over the path.
`_os_getstat()` `_os_setstat()`		Perform control functions on the path.
`_os_gs_ready()`	`ite_data_ready()`	Get number of bytes available for reading on path.
`_os_ss_sendsig()`		Register for signal when data is available on path.
`_os_ss_relea()`		Remove signal on data ready.
	`ite_data_avail_asgn()` `ite_data_avail_rmv()`	Register to be notified of incoming data available to read and remove registration.

Table 1-1 Service Calls Provided by Library (continued)

os_lib.l	ITEM	Description
	`ite_ctl_addrset()`	Set address.
	`ite_ctl_connstat()`	Get address and connection information.
	`ite_ctl_connect()`	Dial.
	`ite_ctl_disconnect()`	Hang-up.
	`ite_ctl_rcvrasgn()`	Register to receive incoming call notification.
	`ite_ctl_rcvrrmv()`	Remove incoming call registration.
	`ite_ctl_answer()`	Answer incoming call.
	`ite_data_readmbuf()` `ite_data_writembuf()`	No copy, high throughput data delivery and reception.

Porting

In general, the porting process consists of the following steps:

Step 1. Port the Microware OS-9 operating system to your hardware.

Step 2. Port/create the hardware driver and/or protocol drivers.

Step 3. Create device descriptors.

Step 4. Create a boot file.

Porting OS-9

If you have not already done so, the first step is to install OS-9 on your hardware.

For More Information

For more information about porting OS-9, see the ***OS-9 OEM Installation Manual***.

Porting Drivers

Microware and third-party vendors supply protocols for all OS-9 targets. These drivers typically do not need to be ported. However, most hardware drivers, even if available from Microware, usually require at least a descriptor modification for the correct base address, interrupt vector, and interrupt level of the hardware.

To port a driver, you must perform the following steps:

Step 1. Customize each entry point within the driver to fit your hardware configuration.

Step 2. Customize the device descriptors to fit the driver.

Step 3. Use the OS-9 `make` utility to make the driver and descriptors into final object files.

Step 4. Subject the driver to normal debugging, testing, and validation.

Detailed requirements and instructions for installation and configuration of the SoftStax components are provided in this manual. Unless otherwise noted, entry points required to customize SoftStax components are located in the section describing the corresponding subsystem.

Creating Boot Files

Finally, set up a boot file containing an entry for each subsystem.

For More Information

For more information about creating and maintaining boot files, see the ***OS-9 OEM Installation Manual***.

Porting SoftStax

While SoftStax and protocol drivers are hardware independent, the hardware drivers and device descriptors must be ported to your hardware. In some cases, protocol driver descriptors may need to be created.

Before you begin porting, check with Microware to determine if the appropriate protocol driver(s) needed for communication between your equipment and the network are available. If not, you can use the `spproto` driver template to create a protocol driver that communicates with the network your equipment uses.

Your application can use the following Application Programming Interface (API) libraries:

- ITEM (Integrated Telecommunications Environment for Multimedia).
- `socket.l` (Socket library for SoftStax provided with the LAN Communications Pak).

To port SoftStax to your hardware, use the following steps :

Step 1. Port OS-9 to your system (required).

Step 2. Customize the device descriptors for all protocol and hardware drivers you use in the system.

Step 3. Modify the hardware driver if necessary, and recompile for the target OS-9 system.

If the required protocol drivers are not provided by SoftStax, create protocol drivers using the `MWOS/SRC/DPIO/SPF/DRVR/SPPROTO` source as a template and compile for the target OS-9 system.

For More Information

SoftStax device descriptors, driver interfaces, defined values and conventions, and the SPPROTO template are discussed in Chapter 3: SPPROTO Driver, later in this manual.

Sample Application Source Files

Three example applications are provided in SoftStax and can be found in `$(MWOS)/SRC/SPF/EXAMPLES`. The subdirectories under this directory are `EXAMPLE1`, `EXAMPLE2`, and `EXAMPLE3`. Use these examples to become familiar with the SoftStax environment.

Example 1: Connection Oriented Example

This example creates two processes, `ex1_snd` and `ex1_rcv`. They use the SPF file manager, `sploop` driver with the `loopc0` and `loopc1` (connection oriented) descriptors in SPF. The processes use the ITEM interface library to communicate with SPF.

Example 2: Bidirectional I/O through os_lib.l Example

This example creates an application called `spf_test`. This application can be used with any driver and descriptors as well as protocol driver stacks to test system I/O throughput. Use this example to test the `sp8530` driver on the MVME 147 board. To run the Z8530 driver, this example uses SPF, the `sp8530` driver, and the `sp3` and `sp4` device descriptors.

You can also run this example using the `sploop` driver with the `loopcl5` and `loopcl6` (connectionless) descriptors. This application uses the standard I/O calls found in `os_lib.l`.

Example 3: Loopback test

This example is a `loopback` test where one process establishes a connection to itself and passes data to itself. This driver uses SPF, `sploop` driver, and `loop` and `loop1` device descriptors. The application uses the ITEM interface.

Any of these examples and makefiles can be used as templates to create your own application. If you plan to use the ITEM interface, try starting with the source for Example 1 or Example 3.

For More Information

Refer to ***Appendix A: Examples*** in the ***Using SoftStax*** manual for more information about these examples.

SoftStax Architecture

The following figures show the file directory structure for SoftStax.

Figure 1-3 SoftStax Source File Directory Structure

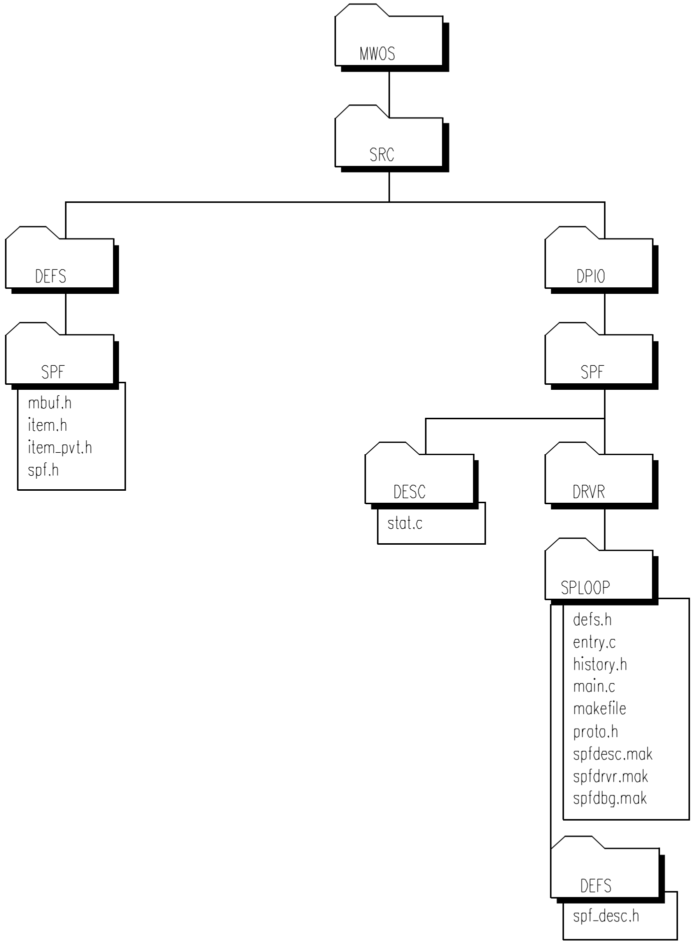


Figure 1-4 SoftStax Source File Directory Structure (continued)

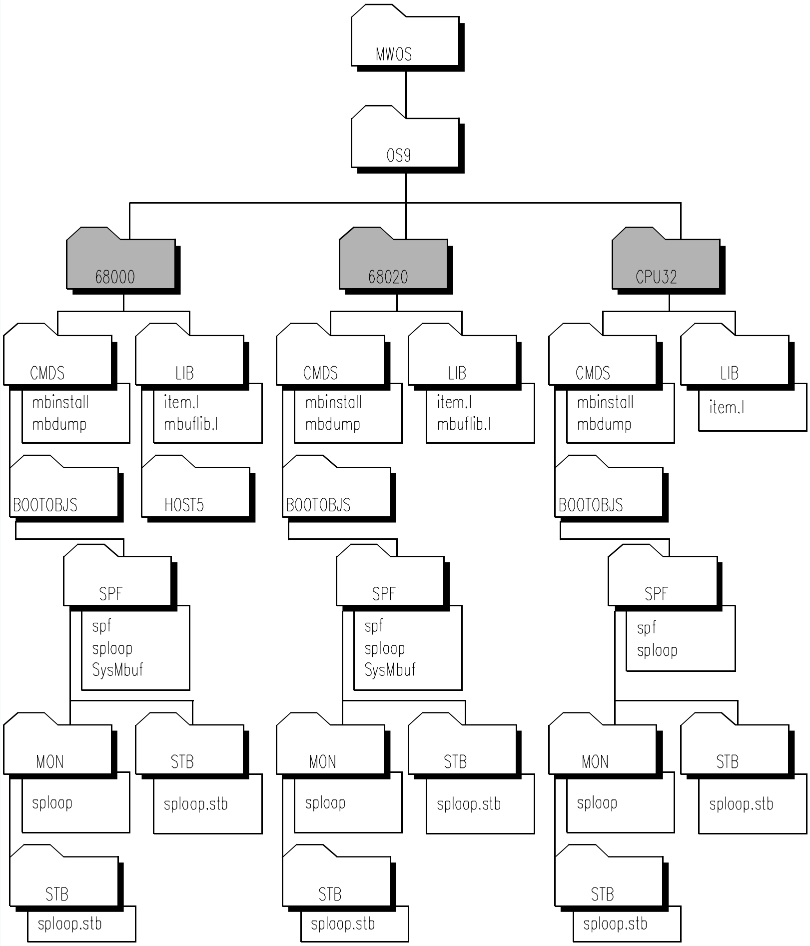
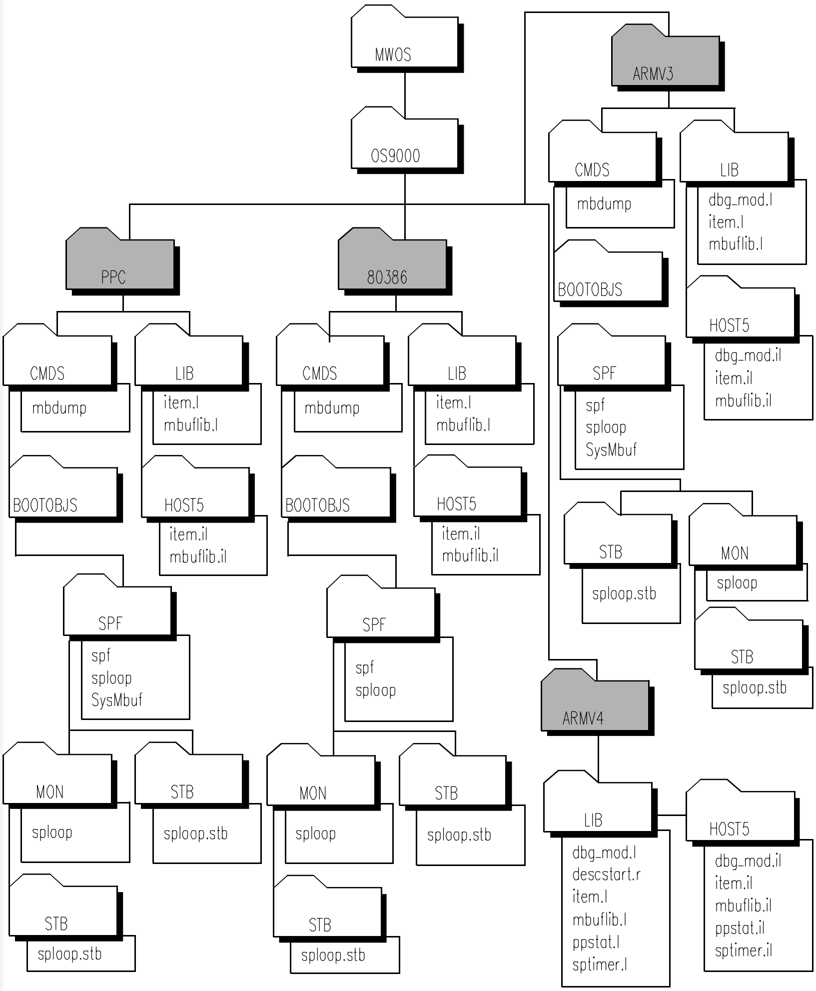


Figure 1-5 SoftStax Source File Directory Structure (continued)



The starting point of the structure is at `MWOS`, or the *Microware OS* directory. From this point, there are three subdirectories: `SRC`, `OS9`, and `OS9000`.

SRC Directory

The `SRC` directory contains source code that can be made for all OS-9 targets. This includes SoftStax, its applications, libraries, drivers, and descriptors.

OS9 Directory

The `OS9` directory contains source code that can only be made for OS-9 targets, as well as the objects for all 68xxx family processors. This is where the objects compiled from the `MWOS/SRC` directory for 68xxx targets (like the SP8530 driver for the MVME147 CPU board) reside.

OS9000 Directory

The `OS9000` directory is a little more complicated because multiple processor families use OS-9000 (including an `80386` subdirectory and a `PPC` subdirectory). These subdirectories contain all the source code that can be compiled for their respective processor families and the objects compiled from the `MWOS/SRC` directory.

Source File Directory Structure

The following rules apply to the source file directory structure:

- All source files for drivers are placed under `MWOS/SRC/DPIO/SPF/DRVR`.
- Protocol driver makefiles and descriptor definition files are placed under `MWOS/SRC/DPIO/SPF/DRVR`.

- Makefiles for a port of any hardware driver are present under the appropriate processor PORTS directory, along with the spf_desc.h file, and the associated objects. Protocol driver makefiles go in the source directory for the protocol driver.
- Protocol objects and descriptors go under <processor>/CMDS/BOOTOBJS/SPF.
- Debugging objects go under BOOTOBJS/SPF/MON (for MONitor object).

Chapter 2: Creating SoftStax Drivers

This chapter provides an in-depth look at network services, driver data structures, path and device descriptors, driver entry points, available facilities, interrupt service routines, and driver types.

The SoftStax Driver

SoftStax drivers fall into one of two categories: protocol drivers or hardware drivers. A hardware driver interfaces directly to hardware registers on some network interface card. The hardware driver is always on the bottom of the protocol stack for a path.

The protocol driver usually does not interface directly to any hardware. Typically, it is a state-machine implementation processing incoming and outgoing data per some protocol specification. Some protocol drivers may interface with hardware. For example, RSA® encryption protocol drivers may use an RSA encryption chip to process the data instead of a software implementation.

Driver Conventions

Driver Names

SoftStax driver names generally start with an `sp` or `rt` prefix. The `sp` denotes an SoftStax driver. Examples in your package are `spx25`, `splapb`, and `sp8530`. The `rt` prefix denotes a special MPEG-2 network device for interactive multimedia systems.

Device Descriptor Names

Device descriptors for hardware drivers are typically `spX`, where `X` is a number. The descriptors in the package for the sp8530 chip are labelled `sp0`, `sp3`, and `sp4`. Device descriptors for MPEG drivers are typically labelled `rtX` where `X` is a number.

Device descriptors for protocol drivers are slightly different. They are typically labelled by just the suffix of the protocol driver they describe and, optionally, a number or letter suffix for more than one descriptor if needed. This makes the protocol stacks easier to read.

For example, the descriptors for spx25 are labelled `x25`, `x25a`, and `x25b`. The `a` and `b` suffixes are used on the `x25` descriptor because the protocol ends in a number and it makes the descriptor name a little easier to read. The descriptors for `splapb` are labelled `lapb`, `lapb0`, `lapb3`, and `lapb4`. Because this protocol ends in a letter, numbers are appended to the end of the protocol name.

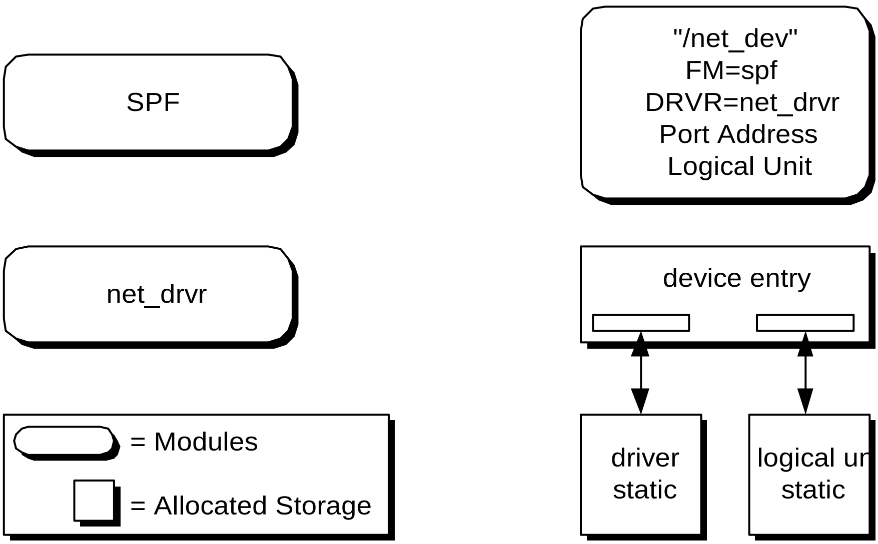
Driver Data Structures

When writing an SoftStax driver, it is important to know what modules are doing at various points of the Input/Output (I/O) process. It is also important to know the structures being allocated automatically for the driver. The following diagram outlines what is occurring at attach time.

At attach time, OS-9 allocates and initializes the following structures for the driver:

`dev_list`	Device list entry (`io.h`)
`spf_drstat`	Driver static storage (`spf.h`)
`spf_lustat`	Logical unit static storage (`spf.h`)

Figure 2-1 Architecture at Attach Time



Driver Static Storage

The driver static storage contains the following items:

- entry points for the driver
- attach count
- port-specific variables
- driver common variables
- driver specific variables defined by a particular driver

The driver static structure is defined in:

```
$(MWOS)/SRC/DEFS/SPF/spf.h.
```

Perform the following edit command to look specifically at the driver static structure for SPF:

```
(umacs -v spf.h, search for “spf_drstat {“)
```

The OS initializes the driver static by using the structure defined in `main.c` of the driver source directory. This initialized structure is compiled directly into the driver object. If there are different device descriptors in memory for the same driver, OS-9 uses the following rules when allocating new driver static.

If two descriptors are using the same driver and the port address is the same, the same driver static storage is used. If the port addresses are not the same, OS-9 allocates and initializes a new version of the driver static for the new descriptor being initialized.

Logical Unit Static Storage

The logical unit static storage contains the following items:

- a pointer to the path descriptor
- up and down driver pointers for this path’s stack relative to this driver
- other variables common to all drivers
- specific variables defined by a particular driver

The logical unit static storage structure is defined in:

```
$ (MWOS)/SRC/DEFS/spf.h
```

Perform the following edit command to look specifically at the logical unit structure in SPF:

```
(umacs -v spf.h, search for “spf_lustat {“)
```

The OS initializes the logical unit static by using the values stored in the device descriptor. If there are different device descriptors in memory for the same driver, OS-9 uses the following rules when allocating new logical unit static.

If two descriptors are using the same driver and the port address is the same and the logical unit number (LUN) is the same, the same logical unit static storage is used. If either the port address or the LUN are not the same, OS-9 allocates and initializes a new version of the logical unit static for the new descriptor being initialized.

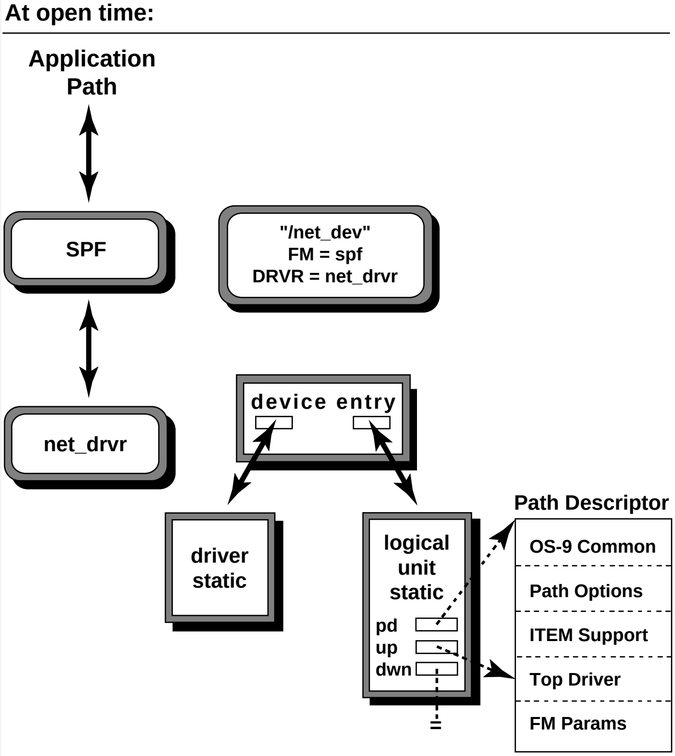
At open time, OS-9 allocates the following structure:

`spf_pdstat` Path descriptor (`spf.h`)

At open time, a path descriptor gets allocated and initialized. A path identifier (32 bit integer) is returned to the application to reference the newly opened path.

The following diagram illustrates the architecture at open time.

Figure 2-2 Architecture at Open Time



Path Descriptor

The following lists attributes of a path descriptor:

- there is one path descriptor for every open path in the system
- a path descriptor allows processes to share paths

Path descriptors contain the following items:

- OS-9 common section for all OS-9 paths
- path options set for a particular network operation
- ITEM support structure

- top driver storage (`deventry`, `read` queue, `drstat`, `lustat` for this path)
- file manager specific variables

The path descriptor structure is defined in the following code:

```
$(MWOS)/SRC/DEFS/spf.h
```

Perform the following edit command to look specifically at the path descriptor structure in SPF:

```
umacs -v spf.h, search for "spf.pdstat {"
```

The file manager initializes the path descriptor by using values stored in the device descriptor. So, not only does the device descriptor contain initialized storage for the logical unit, but also initializes storage for parameters in the path descriptor.

One note about the path descriptor's *Top Driver* section: SoftStax drivers do not store incoming data. Incoming data gets queued on a path descriptor. Therefore, to create the best abstraction for the driver, think of a top driver being assigned to every open path. Protocols at the top of the driver module stack always interface to this top driver when passing up data intended for a particular path. In this way, drivers always have a driver above them. It is the file manager's top driver that is handling the data reception and enqueuing it for the path to read on request. Also, this top driver acts exactly like any other hardware or protocol driver that might exist.

Note

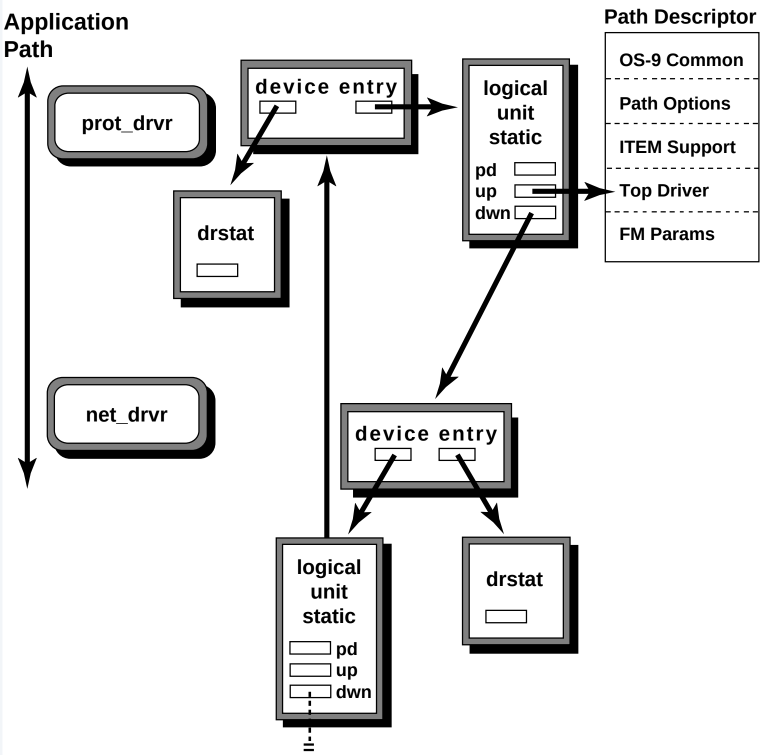
SoftStax is manipulating three variables in the logical unit: `lu_updrvr`, `lu_dndrvr`, and `lu_pathdesc` (noted in the figures as up, dwn, and pd respectively). At open time, pd points to the path descriptor just opened and the updriver pointer points to the top driver section in the path descriptor. The downdriver link is set to `NULL`. It is the `lu_updrvr` and `lu_dndrvr` variables in the logical unit that form the links up and down the protocol stack.

Pushing and Popping Drivers to Paths

After detailing attach and open, one driver is associated with the newly opened path. This section outlines what happens as the application begins pushing more protocol drivers on the path. The following example displays the `prot_drvr` being used on top of `net_drvr`.

Figure 2-3 Pushing a Protocol Driver

Pushing another driver on the stack:



Sequence of Events when Pushing a Protocol Driver

The following process occurs when pushing a protocol driver:

Stage 1. `prot_drvr` called at `dr_iniz()` entry point.

This is the standard initialization of the protocol driver as outlined previously in the Figure 2-1 Architecture at Attach Time discussion.

Stage 2. File manager creates the new updriver/downdriver links in logical unit of `net_drvr`.

The links to the up and down drivers for a path are stored in the logical unit as shown in the diagrams. What these pointers point to is the device entry for the protocols above and below. This way the links up and down also directly have the parameter required to call the driver at its entry points as we'll see later.

Stage 3. `net_drvr` called at `SPF_SS_PUSH` setstat.

The `net_drvr` is being informed a protocol driver is being pushed above it. There may be some housekeeping taking place here. This is discussed later.

Stage 4. File manager creates the new updriver/downdriver links in the logical unit of `prot_drvr`.

The downdriver points to the `net_drvr`'s device entry. The updriver points at the device entry structure in the driver section of the path descriptor (where `net_drvr`'s updriver was pointing before the push).

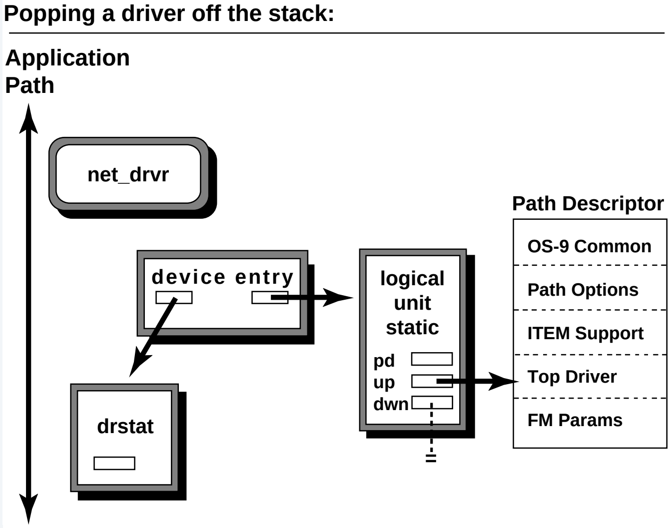
Stage 5. `prot_drvr` called at `SPF_SS_OPEN` setstat.

Now, `proto_drvr` gets a chance to perform its open procedure as outlined previously in the Figure 2-2 Architecture at Open Time discussion.

Note
The protocol drivers are stacked using their device entry structures.

There are now two drivers stacked on the path. The next discussion details popping the top driver off of the path.

Figure 2-4 Popping a Protocol Driver



Sequence of Events when Popping a Protocol Driver

The following process occurs when popping a protocol driver:

Stage 1. `prot_drvr` called at `SPF_SS_CLOSE`.

The protocol should clean up the path related structures and perform any peer-to-peer graceful close messaging for this path.

Stage 2. FM modifies the up and down driver links for `net_drvr`.

Now, the net_drvr updrvr pointer points to a new device entry - the one stored in the path descriptor used by the top driver in SoftStax.

Stage 3. net_drvr called at SPF_SS_POP setstat.

The net_drvr is being informed the protocol driver above it for this path is no longer there. As this point, there may be other housekeeping taking place to store the new updrvr pointer in certain areas the driver allocated on a per open path basis. This is detailed later.

Stage 4. prot_drvr gets detached.

Normal detach is where prot_drvr would deinitialize itself.

Note

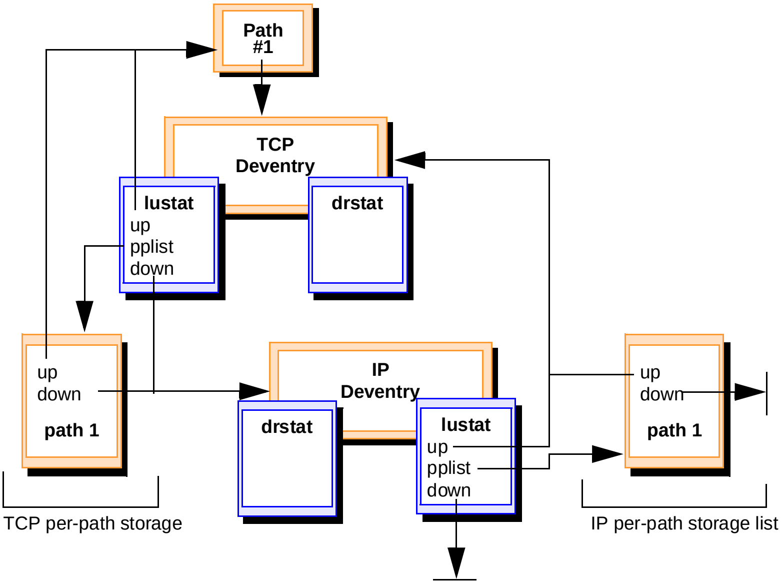
SoftStax does not allow applications to pop the bottom driver off of the stack. If this is attempted, SoftStax returns an EOS_BTMSTK error.

Two Paths Open to One Driver

SoftStax allows multiple paths open to the same driver. However, the logical unit can only hold one updriver/downdriver pair. Two cases are possible to allow two paths open to net_drvr:

- The protocol going on top of the driver is the same protocol. In this case, everything is correct as the updrvr variable in the logical unit gets updated and points to the device entry for the same protocol as the previous path.
- The protocol going on top of the driver is a different protocol. In this case, another structure is required to store the up and down links for each path. This structure is called the *per path storage* or ppstat. Refer to the following example.

Figure 2-5 Path #1 Opens the TCP/IP Protocol Stack



The first part of this scenario involves Path #1 opening the Transmission Control Protocol/Internet Protocol (TCP/IP) stack. The TCP and IP drivers both allocate per-path storage and copy the following from the logical unit to the per path storage area at `SPF_SS_OPEN` time:

- `lu_pathdesc`
- `lu_updrvr`
- `lu_dndrvr`

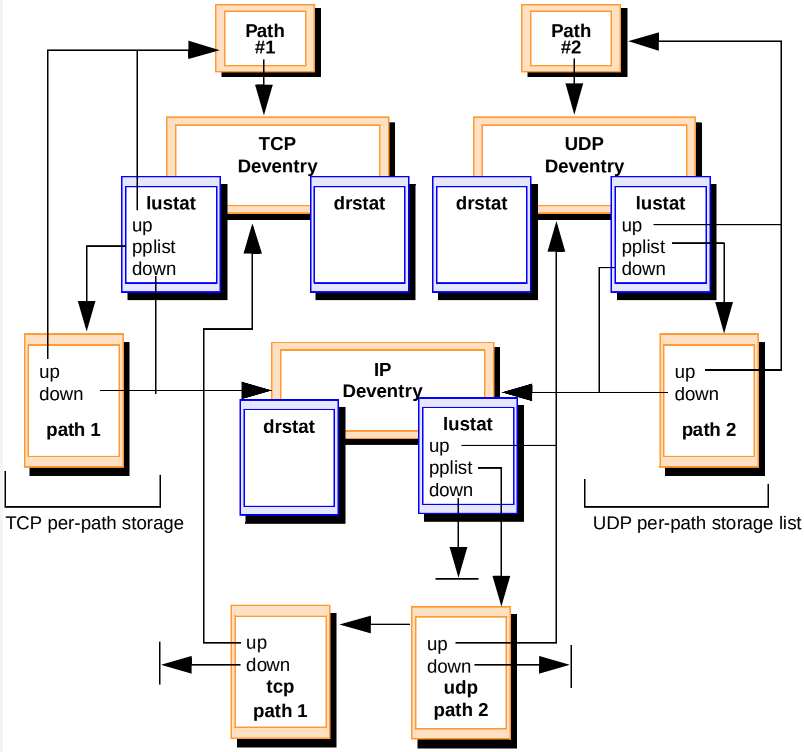
Note

With SoftStax 2.2 the `pps_add_entry()` call is in `ppstat.l` made at `SPF_SS_OPEN` time completing the processing described above. The source to the per path storage library can be found in `MWOS/SRC/DPIO/SPF/LIB/PPSTAT`.

The per path storage variables also map incoming data packets to a particular path or driver above.

For instance, IP uses the protocol field in the IP header to determine which up driver should receive the incoming packet. IP performs an `SPF_GS_PROTID` at `SPF_SS_PUSH` time to secure the protocol ID of the protocol above. IP then stores this in the per path storage structure so it can correctly map incoming data packets to the correct path.

Figure 2-6 Path #1 Opens the TCP/IP Protocol Stack



When the second path is opened and stack created, notice the IPs logical unit `updrvr` now points to UDP instead of TCP. IP lost the link to TCP for Path #1 in its logical unit. However, since IP creates per path storage and has saved the link to TCP in Path #1 per path storage, the system is correctly configured. IP can route incoming packets to TCP and UDP by matching the incoming protocol ID field in the IP header to the correct per path storage and pass the data up using the `ppstat updrvr` variable.

When data goes down the stack, IP looks in the `lu_pathdesc` and finds the correct per path storage matching that `lu_pathdesc` pointer. Once found, it uses the parameters in that per path storage to encapsulate the packet and send it down to lower drivers. The `lu_pathdesc` pointer in the logical unit is always correct at every entry point except `dr_updata()`. This is the incoming data entry point.

Note

For SoftStax 2.2, the driver should use the `pps_findentry()` call in `ppstat.l` to find the per path storage associated with the path making the call.

When data comes up the stack (`dr_updata`), IP specifically searches the per path list for a matching protocol type (`TCP/IP/...`). Once found, it passes the packet up using the `updrvr` pointer in that per path storage. Generically, for a given protocol, the protocol uses an indicator in that protocol's incoming data encapsulation to map this incoming packet to a particular per path storage element.

Note

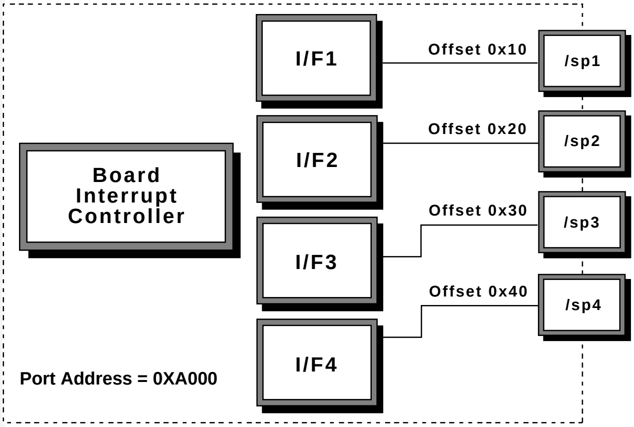
For SoftStax 2.2, the driver should use the `pps_find_enty_by_offset()` call in `ppstat.l` to find the per path storage associated with the addressing in the incoming data packet.

Logical Units

Typically, protocol drivers may only have one descriptor (`/ip`). Therefore, there is really no difference between the logical unit static and the driver static. Both would be allocated only once when the descriptor is initialized/opened.

However, the logical unit is extremely useful for hardware drivers.

Figure 2-7 4-Port Network I/F Board



The above figure shows a 4-port network interface circuit board. The board has four on-board interface chips and one interrupt controller. The interrupt controller has a register set used by the driver to query and service interrupts for the entire circuit board. Each interface chip has a register set for control and data reception/transmission.

This environment maps exactly to the OS-9 driver environment. Create a defined volatile structure of the interrupt controller register set and store a pointer to this structure in the driver static specific section. Also, create a defined volatile structure of the interface chip register set and store a pointer to this structure in the logical unit specific section. Then, create four device descriptors: `/sp1`, `/sp2`, `/sp3`, and `/sp4`.

The idea here is the driver static is allocated only once for the board and contains the interrupt controller register access point and pointers to all logical units for the interfaces currently being used. Logical units are uniquely allocated for each interface chip used.

Each descriptor has a different Logical Unit Number (`LUN`). In fact, you could make the logical unit number the offset from the base address where the interface chip register set can be found. For instance, from the previous figure `/sp1 LUN` might equal 0x10 and `/sp2 LUN` might equal 0x20.

For a structure allocation point of view, when an application opens `/sp1`, driver static and logical unit static would be allocated. A pointer to logical unit static would also be stored in the driver static. Now, the application opens `/sp2`. `/sp2` has the same port address as `/sp1` so the same driver static would be used. However, the `LUN` is different between `/sp1` and `/sp2`. Therefore, a new logical unit would be allocated for interface #2 and stored in the driver static.

When the interrupt service routine is written, it gets as its parameter the driver static. The main ISR body checks the board interrupt controller. If interface #2 generated the interrupt, the main ISR passes `/sp2` logical unit to the handler for the interface. Since the processing for each interface is identical and the only difference is what logical unit you are using, the interrupt service routine can be written very efficiently.

Driver Entry Points

Driver entry points include:

```
dr_iniz(dev_list *deventry)
dr_term(dev_list *deventry)
dr_getstat(dev_list *deventry, spf_ss_pb *pb)
dr_setstat(dev_list *deventry, spf_ss_pb *pb)
dr_downdata(dev_list *deventry, mbuf *mb)
dr_updata(dev_list *deventry, mbuf *mb)
```

dr_iniz()

Rule: This entry point is called by SPF if this is the first attach (open) of a particular logical unit.

For example, there are two descriptors in memory, `/sp1` and `/sp2`. `/sp1` uses `LUN 1` and `/sp2` uses `LUN 2`.

Stage 1. Application one opens `/sp1`.

Stage 2. The first entry opens to `/sp1` and creates a `deventry`, `drstat`, and `lustat` for the network driver. Since this is the first attach to logical unit #1, SPF calls `dr_iniz` of the network driver.

```
dr_att_cnt = 1, lu_att_cnt = 1
```

Stage 3. Hardware registers may need to be initialized the first time the driver is called (like the interrupt controller explained previously):

```
if (drstat->dr_att_cnt == 1) {
   /* Initialize those hardware registers */
   /* register the interrupt service routine */
}
```

Stage 4. The driver knows this is the first attach to this particular logical unit because it is the rule for `dr_iniz` being called. The driver performs the necessary actions for initializing the registers or protocol state machine associated with the logical unit.

dr_term()

Rule: SPF only calls `dr_term` on the last detach of the logical unit.

This entry point simply undoes all actions performed by `dr_iniz()`. Typically, if `dr_att_cnt = 0`, de-install the ISR and turn off the hardware.

dr_getstat()

There may be many custom getstats your driver may need to handle. Mandatory getstats to be implemented by any driver are discussed in this section and are already implemented in the spproto template.

SPF_GS_UPDATE

Update the protocol stack statistics. Every time the protocol stack on a path changes, SPF issues an `SPF_GS_UPDATE` getstat. The parameter block passes `spf_ss_update` parameter block (`spf.h`).

First, the driver allows the protocol to pass all the way to the bottom of the stack. When it returns, it updates the parameter block as required for the statistics. The following parameters are found in the `spf_ss_update` parameter block.

`stk_ioenabled` If the I/O is disabled (you can not talk to your peer), set equal to `DRVR_IODIS`.

If you are at the bottom of the stack, set equal to `DRVR_IOEN` or `DRVR_IODIS` depending on whether the I/O is enabled or not. Otherwise, leave the variable alone.

`stk_txsize`	Maximum Transmission Unit (MTU) for this driver. If your MTU is smaller than the MTU currently in the variable, replace it. If your protocol performs fragmentation and reassembly, you can store this variable in your per path storage and pass up `0xFFFF`. When data packets come down the stack, your driver can then fragment the packet based on the MTU stored in the per path storage.
`stk_txoffset`	If you require a header area to perform your data encapsulation for transmit data, add your requirements to the current value in this variable.
`stk_txtrailer`	If you require a trailer area to perform your data encapsulation for transmit data, add your requirements to the current value in this variable. The path stores these values for the stack in the top driver's area. When the application writes data, SPF uses the `txoffset` and `txtrailer` variables for the path to leave room before and after the user data in the mbuf. This is used so protocols don't have to allocate their own mbufs and chain them to add headers and trailers to the outgoing packet.
`stk_reliable`	If your protocol provides reliable data transfer to protocols above, set to `STK_RELIABLE`. This is used for things that may not have to calculate CRCs at higher layers if it is aware the lower protocols are providing reliability.
`stk_hold_on_close`	If your protocol initiates messaging to close the peer-peer protocol communication when the application executes a `close()` call, add `1` to this variable. Otherwise, do not write to this field.

For More Information

Detailed information concerning this function is located in the Driver Considerations section of this manual.

SPF_GS_PROTID

Return your protocol ID value (`spf.h` or `prot_ids.h`) in the parameter field (`param`) of the `spf_ss_pb` passed to you in the `getstat`.

Note

If you are creating a commercially available protocol stack for SoftStax, contact RadiSys Microware Communications Software Division, Inc. for assignment of your protocol IDs.

dr_setstat()

There may be many custom setstats and Integrated Telecommunications Environment for Multimedia (ITEM) setstats your driver can handle. The mandatory setstats implemented by any driver are discussed next.

SPF_SS_OPEN

RESULT OF: `ite_path_open()` or `_os_open()`

`SPF_SS_OPEN` is called every time a path is opening to this driver. If your driver needs per path storage, it would be allocated and initialized at this point.

SPF_SS_CLOSE

RESULT OF: `ite_path_close()` or `_os_close()`

This setstat gets called every time a path using this driver closes. If your driver needs per path storage it would be found for this path and deallocated.

SPF_SS_PUSH

RESULT OF: opening a protocol stack or calling `ite_path_push()`

`SPF_SS_PUSH` gets called when a new protocol is being pushed on top of your driver for this path. This means your `lu_updrvr` changed for this path. Therefore, set your per path storage to the new `lu_updrvr` in the logical unit. As described later, the addressing may also change when protocols get pushed on top. If this is the case, your driver may have to store the current addressing from the ITEM address section in the path descriptor to the per path storage area if the ITEM addressing changes because of the protocol being pushed.

SPF_SS_POP

RESULT OF: closing a stack or calling `ite_path_pop()`

This setstat gets called when the driver on top of your driver is getting popped off of this path. This means your `lu_updrvr` changed for this path. Therefore, you should set your per path storage to the new `lu_updrvr` in the logical unit. Also since this driver is now on top of the driver stack again, you may need to restore your stored addressing in the ITEM section of the path descriptor.

ITE_SET_CONN

This setstat is used to set local and remote addressing for a device. If the addressing being set is the same as the ITEM addressing in the path descriptor, SPF handles the address automatically without driver processing. If the addressing does not match what is in the path descriptor, SPF sends the `ite_conn_pb` found in `item_pvt.h` down to the drivers.

This enables the application to control the addressing of all protocols on the stack, even if they are using different addressing schemes further down the stack.

dr_downdata()

This entry point is called when there is a data packet attempting to be transmitted to the remote peer. The file manager acquires how many bytes to leave at the front of the packet using the `stk_txoffset` field from the `SPF_GS_UPDATE` getstat. The protocol simply adjusts the `m_offset` and `m_size` fields in the mbuf accordingly to include its added header (and trailer), encapsulates the data, and sends the packet down the stack using the `SMCALL_DNDATA()` macro.

dr_updata()

This entry point is called when an incoming packet is available to be processed by the driver. Drivers at the bottom of protocol stacks typically have `dr_updata()` entry points returning `EOS_UNKSVC`. This is because the bottom driver uses the receive interrupt service routine to receive data, not the `dr_updata` entry point.

The protocol driver should de-encapsulate the data, send any acknowledgments, and update retransmission queues. If any data in the packet needs to go further up the stack, the driver uses the `SMCALL_UPDATA()` macro to send the mbuf up to the protocol driver above.

Interrupt Service Routines

SoftStax hardware drivers typically have Interrupt Service Routines (ISR). When the `downdata` entry point gets called in a hardware driver, it enqueues the data on the transmit queue, enables transmit interrupts, and returns. The transmit interrupt service routine then takes care of sending the data out the network interface. The hardware driver also enables itself to receive data, usually at `SPF_SS_OPEN` time. When incoming data comes in, the receive interrupt service routine reads the data into an `mbuf` and sends the data up via the `FMCALLUP_PKT()` macro.

Driver Interrupt Service Routine Conventions

This section describes situations to watch for when writing an interrupt service routine.

Note

The sp82525 driver has these items coded in the source provided.

Writing and Installing the Interrupt Service Routine

The following code segment shows an example function prototype for the interrupt service routine and a conditionalized definition. The function prototype is found in `proto.h`. The conditionalized definition is found in `defs.h`. These files are covered in more detail later.

```
error_code hw_isr(Dev_list dev_entry);

#if defined(_OS9000)
    #define HW_ISR hw_isr
   /* OS-9000 interrupt service routine */
#elif defined(_OSK)
    extern void hw_isr_os9();
    #define HW_ISR hw_isr_os9
   /* OS-9 interrupt service routine */
#endif
```

Defining a Macro as an Interrupt Service Routine

Defining a macro as an ISR allows your driver to be source code compatible across processors. Notice the real name of the interrupt service routine is `hw_isr()`. However, when running under OS-9 for 68K processor family, there is assembly language code converting OS-9 for 68K interrupt service routine conventions to the OS-9 interrupt service routine conventions. This code is labeled `hw_isr_os9`.

As shown in the previous segment, when `_OS9000` is defined, the name of the ISR is `hw_isr()` only because the `hw_isr_os9` code segment is not needed. (The compiler automatically defines `_OS9000` when compiling for all processors except 68XXX and the OS-9000 operating system. _OSK is defined by the compiler when compiling for the 68XXX family.)

Installing the ISR

This section describes the interrupt service routine installation found in the `dr_iniz()` entry point.

```
if ((err = _os_irq(lustat->lu_vector,
   lustat->lu_priority, HW_ISR, dev_entry)) != SUCCESS)
{
      return(err);
}
```

The interrupt vector and priority can be found in the logical unit structure and are initialized by the device descriptor. The `HW_ISR` macro is in the `_os_irq()` call. The correct name gets resolved at compile time, so there is portability across processors. The last parameter in the `_os_irq()` call is `device_entry`. This parameter gets passed to the protocol driver's interrupt service routine when it runs. However, depending on the driver, you could pass pointers to other structures. Pass whatever is most useful for the interrupt service routine to have when executing.

For More Information

For more information on the `_os_irq()` call, see the ***Ultra C Library Reference Manual***. This manual says `_os_irq()` is only an OS-9000 call. The `conv_lib.l` has created a binding to make this call valid for OS-9 also.

OS-9 Interrupt Service Routine Glue Code

This section describes the OS-9 for 68K interrupt service routine glue code. Glue code is code inserted into the driver to ensure compatibility between OS-9 for 68K and OS-9 systems so the driver can run in either or both operating systems without any further source code changes.

The following example is the interrupt-service routine glue code included for OS-9. It ensures the `hw_isr()` routine is properly called. Do not modify this call unless you are using other labels for your functions.

```
#if defined(_OSK)
/* interrupt service routine glue-code for OS-9 */
_asm("hw_isr_os9:
   move.l a2,d0;/* put Dev_list in a2 into d0/*
   bsr hw_is;/* call interrupt service/*
      /* routine/*
   tst.l d0;/*see if SUCCESS returned/*
   beq.s hw_isr_os9_exit;
      /*if so, return/*
   ori #Carry,ccr;/* else E_NOTME returned/*
      /* --set carry bit/*
hw_isr_os9_exit
   rts
");
#endif
```

Note

This code segment has been included in the hardware driver examples available with SoftStax. If you are using one of these drivers as a template for your hardware driver, the code is located in the interrupt service routine source file.

Driver Callup/Calldown Macros

The following macros are used for driver callup/calldown:

```
#define  FMCALLUP_TIMER_RESTART (mydeventry, tpb)
#define  FMCALLUP_TIMER_START (mydeventry, tpb)
#define  FMCALLUP_TIMER_STOP (mydeventry, tpb)
#define  SMCALL_DNDATA (mydeventry, dst_deventry, mb)
#define  SMCALL_UPDATA (mydeventry, dst_deventry, mb)
#define  SMCALL_GS (mydeventry, dst_deventry, pb)
#define  SMCALL_SS (mydeventry, dst_deventry, pb)
#define  DR_FMCALLUP_PKT (my_deventry, dst_deventry, mb)
#define  DR_FMCALLUP_CLOSE (mydeventry, pathdesc)
#define  DR_FMCALLUP_NTFY (mydeventry, npb)
#define  DR_FMCALLUP_UPDATE (my_deventry, pathdesc)
```

Parameters for these macros include:

`mydeventy`	is the device entry of the current driver. It always gets passed to the driver at every entry point.
`dst_deventry`	corresponds to the `updrvr/downdrvr` pointers stored in the per path storage or logical unit structure. If the call is intended for the driver above, use the `updrvr` pointer. If the call is going down, use the `dndrvr`.
`tpb`	is the timer parameter block. It is of type `timer_pb`, which is found in `timer.h`.
`mb`	is a pointer to an mbuf structure. It is either allocated locally or passed to the driver in the `dr_updata()` or `dr_downdata()` entry points.
`npb`	is the notify parameter block. It is of type `notify_type` which is found in `item.h`.
`pathdesc`	is a pointer to the path descriptor.

pb — is a parameter block for the up/down setstats and getstats. The parameter block must either be or start with the `spf_ss_pb` structure found in `spf.h`.

For More Information

Refer to the ***Using SoftStax*** manual for more information about the `spf_ss_pb` structure.

FMCALLUP_TIMER_RESTART

Restart the timer. This will restart a cyclic timer, or push out the time at which a one-shot timer will time out.

FMCALLUP_TIMER_START

Initiate a registered timer.

FMCALLUP_TIMER_STOP

Stop the timer and delete it from the list.

SMCALL_DNDATA()

Use `SMCALL_DNDATA` to pass packets down the stack.

SMCALL_UPDATA()

Use `SMCALL_UPDATA` to pass packets up the stack.

SMCALL_GS()

Use SMCALL_GS to send a getstat up or down the stack.

SMCALL_SS()

Use SMCALL_SS to send a setstat up or down the stack.

DR_FMCALLUP_PKT()

Use DR_FMCALLUP_PKT to enqueue a packet received by the hardware driver in the interrupt context on the SPF receive thread queue.

DR_FMCALLUP_CLOSE()

DR_FMCALLUP_CLOSE is used by the driver using hold-on-close to notify the file manager the protocol has executed a graceful close thus providing messaging when the application has closed the path.

DR_FMCALLUP_NTFY()

DR_FMCALLUP_NTFY provides an easy way for protocols to tell SPF to send the notification the application may have registered for.

DR_FMCALLUP_UPDATE()

DR_FMCALLUP_UPDATE causes SPF to generate an SPF_GS_UPDATE getstat down the path. It is used when drivers pull stacks underneath for a path.

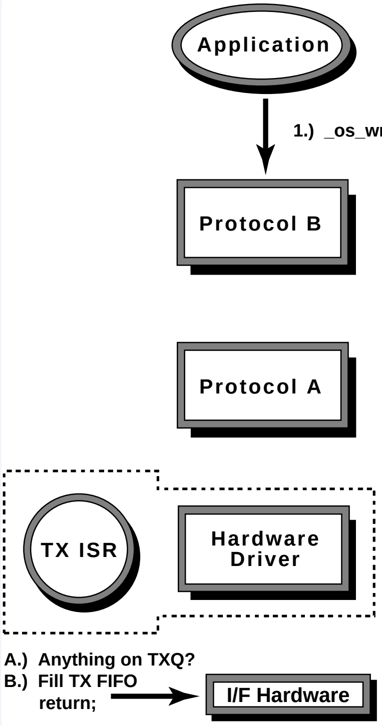
Outgoing Data Processing

Figure 2-8 Sequence of Events for Writing Data shows the threads of execution for outgoing data packets. Writes happen as one autonomous step on the application's thread. When the _os_write call is executed, the protocols encapsulate the data and send it down to be enqueued on the `_os_write()` thread. Once the data is enqueued, `_os_write()` returns control to the application.

The hardware driver's transmit ISR sends the data out asynchronously through the network interface hardware.

As displayed in the figure below, steps occur sequentially as a result of the `_os_write()` call. Steps A and B occur asynchronously as a result of the driver's interrupt service routine.

Figure 2-8 Sequence of Events for Writing Data



Incoming Data Processing

Figure 2-9 Protocol Stack Data Processing for Incoming Packets shows the threads of execution for processing incoming data packets. The receive interrupt service routine gets a complete packet. It determines the device entry of the protocol directly above it, and makes the `FMCALLUP_PKT` call.

The FMCALLUP_PKT call enqueues the data on the SPF receive thread process receive queue and sets an event to wakeup the process. At this point, you are now out of interrupt context.

When the receive thread (spf_rx) is scheduled to run, it takes the packet on the receive queue and calls protocol A's dr_updata entry point with the received packet. Protocols process the incoming data on the SPF_RX thread, *not* interrupt service routine context.

Receive data is queued by the SPF top driver on the path's receive queue. It waits there until the application does an _os_read().

Two important notes concern received data:

Note

RX ISR must use FMCALLUP_PKT to send data up.

There are two options for sending data to higher drivers for processing: SMCALL_UPDATA and FMCALLUP_PKT. FMCALLUP_PKT uses the receive thread context to send the data up the stack. If the ISR uses SMCALL_UPDATA, the packets processed by all protocols in interrupt service routine context. This could cause receive FIFO overflow problems.

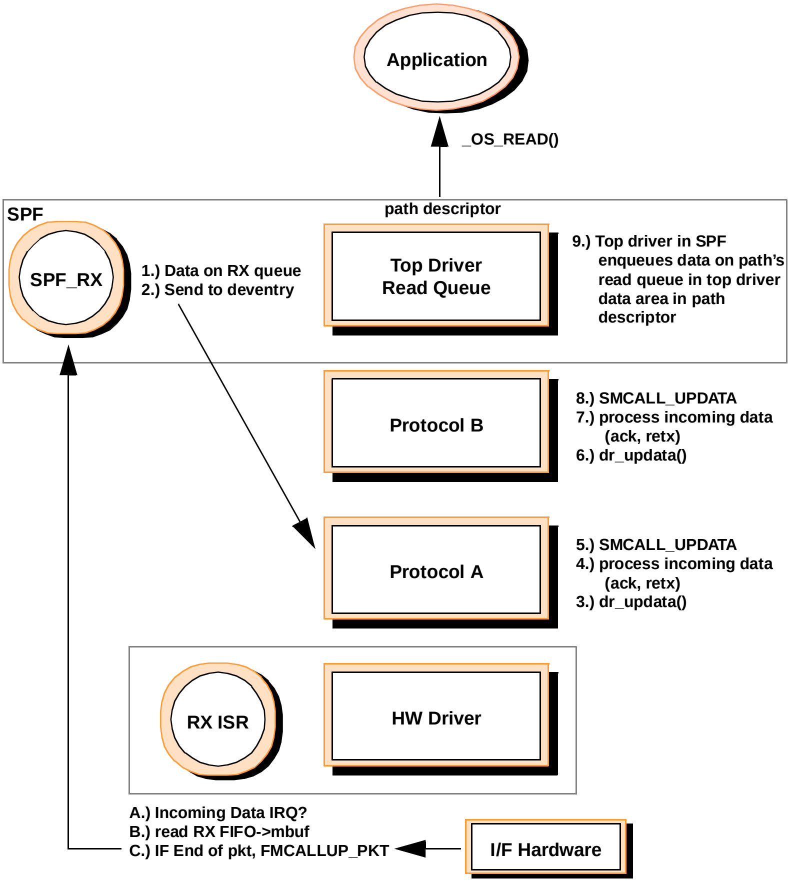
Recall that the second parameter in the FMCALLUP_PKT call is the device entry to send the packet to. This device entry pointer is stored in the mbuf immediately after the mbuf header. When the receive thread processes the mbuf, it uses this pointer to determine which driver to pass the mbuf to. Hardware drivers need to perform a get_mbuf() call instead of an m_get() call for this reason. The get_mbuf() call leaves room for this device entry pointer immediately after the mbuf header. The m_get() call does not.

Note

Protocols never sleep in dr_updata().

Notice the only context that received data is processed in receive thread context. Therefore, if a driver gets called at `dr_updata()` in receive thread context, sends a message, and then sleeps waiting for the response, the system stops forever. Once you put the receive thread to sleep, there is no way for more data to come up any stack, the response never comes, and no more received packets are processed because it's impossible to context switch the receive thread since it is in system state.

Figure 2-9 Protocol Stack Data Processing for Incoming Packets



Driver Data Structures (spf.h)

Driver writers need to familiarize themselves with three main header files: `spf.h`, `item.h`, and `item_pvt.h`. `spf.h` contains all of the OS-9 system level structures and definitions for SoftStax. The `item.h` file is an application oriented header file exposing the structures and functions available to the application. The `item_pvt.h` file contains all setstat / getstat codes and parameter blocks used by ITEM to implement the ITEM API.

The `spf.h` header file is broken into three sections:

- application oriented (nothing predefined)
- driver oriented (`#define SPF_DRVR`)
- file manager oriented (`#define SPF_FM`)

Drivers should define `SPF_DRVR` before including `spf.h` so they include all structures for driver use as well as application use. This is already done in the spproto template and hardware drivers.

The following data structures provide these functions:

Table 2-1 Structures

Structure	Function
`spf_popts`	Path Descriptor Options
`spf_desc`	Device Descriptors
`spf_drstat`	Driver Static
`spf_lustat`	Logical Unit Static

spf_popts

Path Descriptor Options

Declaration

The path options structure `spf_popts` is declared in the file `spf.h` as follows:

```
struct spf_popts {
   #if defined(_OSK)
      u_int8      pd_devtype;
      u_int8      pd_buf1;
   #elif defined(_OS9000)
      u_int16     pd_devtype;
   #endif
   u_int16        pd_devclass;
   u_int8         pd_version;
   u_int8         pd_ioenabled;
   u_int8         pd_ioasync;
      #define IO_SYNC              0
      #define IO_WRITE_ASYNC       1
      #define IO_READ_ASYNC        2
      #define IO_ASYNC             3
   u_int8         pd_iopacket;
      #define IO_CHAR              0
      #define IO_PACKET            1
      #define IO_DGRAM_TOSS        2
      #define IO_NEXTPKT_ONLY      4
      #define IO_PACKET_TRUNC      6
   u_int32        pd_optsize;
   u_int32        pd_iotime;
   u_int32        pd_readsz;
   u_int32        pd_writesz;
   u_int16        pd_rsv;
   u_int16        pd_txsize;
   u_int16        pd_txoffset;
   u_int16        pd_txtrailer;
   u_int16        pd_txmsgtype;
      #define TXMSG_CONF           0x8000
   u_int8         pd_reliable;
   u_int8         pd_rsv2[93];
};
```

Description

The path descriptor options are structures that applications can request and get information on how the path they are using is configured. The application can also set certain path options for different operations. This section discusses the fields in the path options structure `spf_popts`.

All path option sections are 128 bytes long.

Fields

`pd_devtype`	is always defined as `DT_SPF`.
`pd_buf1`	is the long word alignment.
`pd_devclass`	is always defined as `DC_SEQ`.
`pd_version`	is the compatability version for SoftStax use.
`pd_ioenabled`	I/O is enabled and disabled on this path.
`pd_ioasync`	defines asynchronous operation.
`IO_SYNC`	uses synchronous read and write operations.
`IO_WRITE_ASYNC`	enable asynchronous write operation. If no buffer is available, an `EWOULDBLOCK` error returns instead of blocking.
`IO_READ_ASYNC`	enable asynchronous read operation. If no data is available, an `EWOULDBLOCK` error returns instead of blocking.
`IO_ASYNC`	enable asynchronous read and write operations.
`pd_iopacket`	defines packet oriented operation.
`IO_CHAR`	`0` = Character oriented (read number of bytes into buffer and return). If the data is not available to be read yet, the application blocks until the entire read request is fulfilled. Non-zero is packet oriented mode. In packet oriented mode, there are other modes of operation.
`IO_PACKET`	return all available packets if the read request is larger than the amount of data in the queue.

`IO_DGRAM_TOSS` is used for UDP datagram operation only. If the amount of data read is smaller than the current packet on the read queue, discard the rest of the packet.

`IO_NEXTPKT_ONLY` indicates the number of requested bytes is really the size of the read buffer passed in by the requestor. Only the next packet is returned in the buffer. For example, if the next packet is 50 bytes long and the application does a read of 1000 bytes, SoftStax returns the 50 byte packet even if there may be more packets waiting on the read queue.

`IO_PACKET_TRUNC` indicates if a portion of the packet being read does not get read (a read of 50 bytes when a 100 byte packet is received), throw the rest of the packet away.

For More Information

Examples of these values and how they are used can be found in the ***Using SoftStax*** manual.

`pd_optsize` is always set to 128 bytes.

`pd_iotime` is the timeout value for the path. If the data is not available to be read at the time of the `read()` call, this is the number of ticks the application is willing to wait for the data. If the read is not fulfilled within the time period, the application returns with what data was available and the number of bytes read into the buffer.

`pd_readsz / pd_writesz`
are flow control parameters set on the path so read and write queues on a given path cannot starve the buffer pool for the whole system. The `pd_readsz`

variable is enforced by SPF. The pd_writesz must be enforced by the driver owning the TX queue.

pd_rsv — is reserved for future use.

pd_txsize, pd_txoffset, pd_txtrailer, and pd_reliable — are stack parameters as gathered by the SPF_GS_UPDATE call.

For More Information

Refer to the SPF_GS_UPDATE section of this manual.

pd_txmsgtype — sets transmit message mode of operation. These flags allow for flexibility of reliable transmission of data. This would be enforceable by the drivers.

TXMSG_CONF — is a way to allow for blocking writes. Typically, when data is transmitted by _ite_data_write(), it is enqueued on a driver transmit queue and the call returns immediately. The driver would have to block on this current write until the data was successfully sent. Then, the application performing the write would wakeup and continue.

pd_rsv2 — is padding at the end which will be used in future releases and therefore should not be used by drivers for any other reason.

spf_desc

Path Descriptors

Declaration

The device descriptor structure `spf_desc` is declared in the file `spf.h` as follows:

```
struct spf_desc {
   dd_com          dd_desccom;
   spf_popts       dd_popts;
   device_type     dd_item;
   u_int32         dd_pmstak;
   u_int8          dd_rsv1[16];
};
```

Description

The device descriptor `spf_desc` contains structures allowing SPF to initialize a newly opened path to a default configuration. In general, the device descriptor initializes the default path options as above, the ITEM structure for the path, and the protocol module stack for the path. The following is a description of `spf_desc`.

The `spf_desc.h` section details how these values are initialized in the descriptor.

Fields

`dd_desccom`	is a common header for all OS-9 device descriptors found in `io.h`.
`dd_popts`	defines default values for the path options section when an application opens a path using this descriptor.
`dd_item`	defines default values for the ITEM `device_type` structure when an application opens a path using this descriptor.

`dd_pmstak`	is the offset to the protocol stack string for this descriptor. This string indicates the protocol drivers used by this path from the bottom up (left to right creates the lowest driver in the stack to highest).
`dd_rsv1`	is reserved for future use.

spf_drstat

Driver Static

Declaration

The driver static structure `spf_drstat` is declared in the file `spf.h` as follows:

Note

The definition section has been omited in the description below. Refer to `spf.h` for the complete declaration.

```
struct spf_drstat {

   u_int32 dr_version;
   error_code(*dr_fmcallup)(
      u_int32          code,
      void*            param1,
      void*            param2);

/* definition section--not shown */

   error_code (*dr_iniz)(Dev_list deventry);
   error_code (*dr_term)(Dev_list deventry);
   error_code(*dr_getstat)(
      Dev_list         deventry,
      Spf_ss_pb        gs_pb);
   error_code (*dr_setstat)(
      Dev_list         deventry,
      Spf_ss_pb        ss_pb);
   error_code (*dr_downdata)(
      Dev_list         deventry,
      Mbuf             mb);
   error_code (*dr_updata)(
      Dev_list         deventry,
      Mbuf             mb);
   u_int32          dr_att_cnt;
   Spf_lustat       dr_lulist;
   u_int8           dr_lumode;
```

```
    #define DR_ALLOC_LU_PERPORT         0
    #define DR_ALLOC_LU_PERPATH         1
  u_int8            dr_rsv1[11];
  u_int32           dr_use_cnt;
  #ifdef SPF_DRSTAT
    SPF_DRSTAT
  #endif
};
```

Fields

`dr_version`	is the current version of SoftStax for future compatibility.
`dr_fmcallup()`	is used by the file manager to inform the driver of the file manager's callup service routine address. SPF fills this in when the device is initialized.
`dr_xxx`	are the entry points of the driver.

For More Information

Refer to Driver Entry Points for more information about these fields.

`dr_att_cnt`	is the number of attaches done to this driver.
`dr_lulist`	is set to `0`. This field is not used.
`dr_lumode`	is set to `0`. This field is not used.
`dr_use_cnt`	is the number of paths using this driver in their protocol stack.
`dr_rsv1`	is reserved for future use.
`SPF_DRSTAT`	This macro is defined so all drivers can place specific variables within the driver static by defining this macro and associated structure in `defs.h` in the driver source code.

spf_lustat

Logical Unit Static

Declaration

The logical unit static structure `spf_lustat` is declared in the file `spf.h` as follows:

Note

The definition section has been omited in the description below. Refer to `spf.h` for the complete declaration.

```
struct spf_lustat {
   /* general logical unit fields */
   u_int32           lu_att_cnt;
   u_int32           lu_use_cnt;
   void*             lu_port;
   u_int16           lu_num;
   /* SPF specific fields */
   u_int8            lu_ioenabled;
      #define DRVR_IOEN             1
      #define DRVR_IODIS            0
   u_int8            lu_reliable;
   u_int32           lu_txsize;
   u_int16           lu_txoffset;
   u_int16           lu_txtrailer;
   void*             lu_attachptr;
   Dev_list          lu_updrvr;
   Dev_list          lu_dndrvr;
   Spf_pdstat        lu_pathdesc;

/* definition section--not shown */

   Spf_lustat        lu_next
   u_int8            lu_hold_on_close;
      #define PATH_NOHOLD           0
      #define PATH_HOLD             1
   u_int8            lu_rsv2;
   u_int16           lu_pps_size;
```

```
   Spf_ppstat         lu_pps_list;
   void               *lu_pps_idata;

   /* Logical Unit Options Fields */
   spf_luopts      lu_luopts;
   #ifdef SPF_LUSTAT
      SPF_LUSTAT
   #endif
};
```

Fields

`lu_att_cnt`	is the current number of attaches to this logical unit.
`lu_use_cnt`	is the number of paths using this driver's logical unit in their protocol stack.
`lu_port`	is the port address for the driver. For protocol drivers, this can be set to `0`.
`lu_num`	is the logical unit number assigned to this logical unit.
`lu_ioenabled`	refer to `SPF_GS_UPDATE` for the individual driver.
`DRVR_IOEN`	IO is enabled.
`DRVR_IODIS`	IO is disabled.
`lu_reliable`	refer to `SPF_GS_UPDATE` for the individual driver.
`lu_txsize`	refer to `SPF_GS_UPDATE` for the individual driver.
`lu_txoffset`	refer to `SPF_GS_UPDATE` for the individual driver.
`lu_trailer`	refer to `SPF_GS_UPDATE` for the individual driver.
`lu_attachptr`	is the location where the device entry returned by the first attach to the driver by the file manager is stored. When the first path is opened, the `lu_attachptr` is initialized by the FM. The last path close to this driver causes the FM to use this to detach the last time to the driver so its memory will be deallocated from memory.

`lu_updrvr`, `lu_dndrvr`, and `lu_pathdesc`	are described previously in the stacking/unstacking example. Macros are available for easy reference to ITEM and notification information in the path by accessing the `lu_pathdesc` of the logical unit.
`lu_next`	allows logical units to be chained.
`lu_hold_on_close`	indication of hold-on-close path (described later).
`PATH_NOHOLD`	indicates don't hold-on-close.
`PATH_HOLD`	indicates hold-on-close.
`lu_rsv2`	is reserved for future use.
`lu_pps_size`	set to the size of your per path storage in bytes.
`lu_pps_list`	points to the per path storage linked list area.
`lu_pps_idata`	points to initialized per path storage data.
`lu_luopts`	contains information on the device class and version.
`SPF_LUSTAT`	is a macro defined so all drivers can place specific variables in the logical unit static by defining this macro and assocated structure in `defs.h` in the driver source code.

ITEM Support

The purpose of ITEM is to provide applications with an Application Programming Interface (API) that is network independent and operating system independent. ITEM provides three levels of abstraction to achieve this goal:

- abstraction of the network device itself using the `device_type` structure
- abstraction of the addressing by using the `addr_type` structure
- abstraction of how notification on asynchronous triggering activity happen by using the `notify_type` structure

First, the `item_pvt.h` file is discussed. The `path_type` structure found in `item_pvt.h` encompasses all three of the abstractions. `item.h` is discussed next. These are the abstractions the application can access and the API calls available to perform network independent control.

item_pvt.h

Referring back to Figure 2-2 Architecture at Open Time, this is the section in the path descriptor labeled ITEM Support. This is the structure `path_type`.

The `path_type` structure consists of:

- notification list
- `device_type` structure

The `item_pvt.h` contains the following:

- the definition for the ITEM structure in the path descriptor found in `item_pvt.h`
- ITEM codes
- parameter blocks used by ITEM to provide the API. These parameter blocks are used by the drivers to realize the API services available through ITEM.

Notification List

When the ITEM notification requests are made, SoftStax logs the request for notification in the notification array for that path. With the exception of `ITE_ON_DATAVAIL`, when the driver detects one of these conditions, it is expected to use the notification entry in the appropriate path descriptor to notify the requestor of the occurrence of the condition by using the `DR_FMCALLUP_NTFY()` macro described earlier.

For example, an application requests to be notified on incoming call. This request is completely processed by SoftStax by logging the request in the notification array for the path. The call control protocol in that path's stack gets an incoming connect request message. The protocol queries the notify list of the path and finds there is a process waiting for this notification by examining the `ntfy_on` field of the `notify_type` structure. It then makes the `DR_FMCALLUP_NTFY()` call by using a pointer to the `ON_INCALL` element of the notify list in the path descriptor.

The following notifications are available to the application:

Table 2-2 Notifications

Notification Type	Setstat / Getstat code used in item.pvt.h
Link down	`ITE_ON_LINKDOWN`
Incoming call	`ITE_ON_INCALL`
Connection established	`ITE_ON_CONN`
Data available	`ITE_ON_DATAVAIL`
End of MPEG-2 program	`ITE_ON_ENDPGM`
Far End hangup	`ITE_ON_FEHANGUP`
End of download procedure	`ITE_ON_DNLDONE`

Table 2-2 Notifications (continued)

Notification Type	Setstat / Getstat code used in item.pvt.h
Message confirmation	`ITE_ON_MSGCONF`
Resource added to session	`ITE_ON_RESADD`
Data link up	`ITE_ON_LINKUP`

item.h

`item.h` is the file applications include when using the ITEM API.

`item.h` has four sections:

- `device_type` declaration
- `addr_type` declaration
- `notify_type` declaration
- function prototypes

The main function of ITEM is to provide a network access abstraction for the application. Therefore, the storage structures and API allow applications to access generic connectivity and notifications. Even though different networks may implement signalling and addressing differently, the end result achieved from the perspective of the application is the same.

The structures defined in `item.h` are generic and protocol drivers are expected to maintain and use the ITEM structures so applications interoperate over any kind of network protocol.

device_type

Device Information

Declaration

The `device_type` structure is declared in the file `SPF/item.h` as follows:

```
typedef struct device_type {
   u_int16           dev_mode;
   u_char            dev_netwk_in, dev_netwk_out;
      #define ITE_NET_NONE         0x00
      #define ITE_NET_CTL          0x01
      #define ITE_NET_DATA         0x02
      #define ITE_NET_MPEG2        0x03
      #define ITE_NET_CHMGR        0x04
      #define ITE_NET_OOB          0x05
      #define ITE_NET_VIPDIR       0x06
      #define ITE_NET_SESCTL       0x07
      #define ITE_NET_X25          0x08
      #define ITE_NET_ANY          0xFF
   u_int16           dev_callstate;
      #define ITE_CS_IDLE          0x0001
      #define ITE_CS_INCALL        0x0002
      #define ITE_CS_CONNEST       0x0004
      #define ITE_CS_ACTIVE        0x0008
      #define ITE_CS_CONNTERM      0x0010
      #define ITE_CS_CONNLESS      0x0020
      #define ITE_CS_SUSPEND       0x0040
   u_char            dev_rcvr_state;
      #define ITE_ASGN_RSVD        0x00
      #define ITE_ASGN_NONE        0x01
      #define ITE_ASGN_THEIRNUM       0x02
      #define ITE_ASGN_ANY         0x03
      #define ITE_ASGN_PROFILE 0x04
   u_char            dev_rsv1;
   u_int32           dev_rsv2
   addr_type         dev_ournum, dev_theirnum;
   char              dev_display[ITE_MAX_DISPLAYSIZE];
} device_type, *Device_type;
```

Description

`device_type` contains general information about network type, current call state, local and remote addressing, and a virtual display area.

Fields

`dev_mode`	characterizes the mode of the device (readable or writable). Legal values are: `FAM_READ`, `FAM_WRITE`, and `FAM_NONSHARE`.
`dev_netwk_in`	allows for independent characterization of the input side of the network device. Refer to ***Using SoftStax*** for legal values of this field.
`dev_netwk_out`	allows for independent characterization of the output side of the network device. Refer to ***Using SoftStax*** for legal values of this field.
`dev_callstate`	shows the current call state for the device. Refer to ***Using SoftStax*** for legal values of this field.
`dev_rcvr_state`	indicates whether anyone is registered to receive notification for an incoming call on this network device. Refer to ***Using SoftStax*** for legal values of this field.
`dev_rsv1`	is reserved for future use.
`dev_rsv2`	is reserved for future use.
`dev_ournum`	is the address information for our-end.
`dev_theirnum`	is the address information for their-end.
`dev_display`	is an array that may be used by signalling protocols for messages such as caller ID. The application can perform an `ite_path_constat()` call and print the display information to a screen or LCD display.

addr_type

Address Information

Declaration

The `addr_type` structure is declared in the file `SPF/item.h` as follows:

```
typedef struct addr_type {
   u_char            addr_class;
      #define ITE_ADCL_NONE         0x00
      #define ITE_ADCL_UNKNOWN 0x01
      #define ITE_ADCL_E164         0x02
      #define ITE_ADCL_INET         0x03
      #define ITE_ADCL_RSV1         0x04
      #define ITE_ADCL_X25          0x05
      #define ITE_ADCL_ATM_ENDSYSTEM 0x06
      #define ITE_ADCL_LPBK         0x07
      #define ITE_ADCL_NSAP         0x08
      #define ITE_ADCL_DTE          0x09
      #define ITE_ADCL_DCE          0x0A
      #define ITE_ADCL_LAPD         0x0B
   u_char            addr_subclass;
      #define ITE_ADSUB_NONE        0x00
      #define ITE_ADSUB_UNKNOWN           0x01
      #define ITE_ADSUB_VC          0x02
      #define ITE_ADSUB_PVC         0x03
      #define ITE_ADSUB_LUN         0x04
      #define ITE_ADSUB_SLINK       0x05
      #define ITE_ADSUB_MLINK       0x06
   u_char            addr_rsv1;
   u_char            addr_size;
   char              addr[32];
} addr_type, *Addr_type;
```

Description

This structure abstracts the specific addressing for the application. The address has an address class and subclass telling the application what kind of addressing is used. Addressing may change down the stack. The address size and array where the actual address is stored follows.

The address type in the `path_type` structure only belongs to the top protocol in the stack. When a protocol with a new addressing type is pushed on, the old top protocol must save the current addressing for their protocol during the `SPF_SS_PUSH`. This addressing information should be copied back into the ITEM path structure during `SPF_SS_POP`.

It is legal for stacks to be created which have different addressing and therefore different ITEM address types at each layer. For example, say you are doing end-to-end signalling using UDP/IP over an Asynchronous Transfer Mode (ATM) interface. The ATM interface itself has its own kind of address type--`ITE_ADCL_ATM_ENDSYSTEM` (refer to `item.h`). Then, the UDP/IP protocols use socket addressing. Furthermore, the top end-to-end signalling may use Network Service Access Point (NSAP) addressing. With any luck, the only addressing exposed to the application would be NSAP, because that's the addressing used in the ITEM structure since that protocol driver is on top. However, if the application needs to change the socket address, it can use the `ite_ctl_addrset()` call. It simply creates an `addr_type` structure with `ITE_ADCL_INET` and sends the new socket address down. Since the SoftStax ITEM structure uses NSAP, it transparently passes down until it gets to UDP, where the new socket address will be stored for that path. Using `ite_ctl_addrset()` call, the application can manipulate addressing at any layer of the stack.

Fields

`addr_class`	indicates the address class. Refer to ***Using SoftStax*** for legal values of this field.
`addr_subclass`	indicates the address sub-class. Refer to ***Using SoftStax*** for legal values of this field.
`addr_rsv1`	is reserved for future use.
`addr_size`	is the number of valid bytes in the addr array.
`addr`	contains the specific address value.

notify_type

Notification Information

Declaration

The `notify_type` structure is declared in the file `SPF/item.h` as follows:

```
typedef struct notify_type {
   struct notify_type      *ntfy_next;
   u_char            ntfy_class;
   u_char            ntfy_on;
   u_char            ntfy_rsv1;
   u_char            ntfy_ctl_type;
   void              *ntfy_ctl;
   u_int32           ntfy_timeout;
   u_int32           ntfy_rsv[2];

   union
   {
      struct {
      u_int32        proc_id;
      u_int32        sig2send;
      } sig;

      struct {
      u_int32        ev_id;
      int32          ev_val;
      } ev;

      struct {
      u_int32        ev_id;
      int32          ev_inc_val;
      } inc_ev;

      struct {
      u_int32        mmbox_handle;
      error_code     (*callback_func)();
      } mmbox;

      struct {
      void           *callbk_param;
      error_code     (*callback_func)();
      } callbk;
```

```
    } notify;
} notify_type, *Notify_type;
```

Description

When asynchronous ITEM requests are issued by the application, this structure is used to allow the application to tell the system in what way they wish to be notified when the triggering activity occurs. Currently, the mechanisms used for notification are the `ITE_NCL_SIGNAL`, `ITE_NCL_EVENT`, and `ITE_NCL_CALLBACK`. You may also use `ITE_NCL_BLOCK`, however, this makes the request synchronous.

Fields

The caller fills in the `ntfy_class`, `ntfy_ctl_type`, `ntfy_ctl`, and `notify` fields of the structure before issuing a call to the driver. The remaining fields are for internal driver use.

The fields set by the caller are described below:

`ntfy_class` specifies whether the request should execute synchronously or asynchronously, and for asynchronous requests the type of notification desired. Valid values are:

`ITE_NCL_BLOCK`	specifies a synchronous request.
`ITE_NCL_SIGNAL`	specifies an asynchronous request with notification via a signal.
`ITE_NCL_EVENT`	specifies an asynchronous request with notification via an event.
`ITE_NCL_CALLBACK`	specifies an asynchronous request with notification via a call-back function.
`ITE_NCL_EVENTINC`	specifies incrementing events notification.
`ITE_NCL_SIGNALINC`	specifies incrementing signal notification.

Note

When the ITE_NCL_CALLBACK ntfy_class is specified, the caller is executing in system state. Typically this ntfy_class is used only by device drivers issuing asynchronous requests to other device drivers.

ntfy_on	specifies what notification in the array the application is interested in.
ntry_rsv1	is reserved for future use.
ntfy_ctl_type	identifies the type of pointer set in the ntfy_ctl field. For asynchronous calls, the ntfy_ctl_type field should be set to NTYPE_RETURN. For synchronous calls, the ntfy_ctl_type field should be set to NTYPE_NONE.
ntfy_ctl	set to point to the caller's return_type structure for asynchronous calls. This structure contains a single field set by the driver to the completion status of the asynchronous call before the notification is issued. The return_type structure is defined in the file SPF/item.h. The return_type structure simply provides a location for an error code to be returned. Thus, this structure must remain allocated and unused by the caller until the asynchronous call has completed. When a caller is notified an asynchronous call has completed, it should check the return_type structure to determine whether or not the call was successful.
ntfy_timeout	is the amount of time the blocking application waits for the notification before returning.
ntfy_rsv	is reserved for future use.
proc_id	is for internal driver use when notifying via signals.

`sig2send`	specifies the signal number desired for a notification via a signal.
`ev_id`	specifies the event identifier for a notification via an event.
`ev_val`	specifies the desired event value for a notification via an event.
`ev_inc_val`	specifies the value to increment the event by when incrementing events.
`callback_func`	specifies the address of the call-back function for a notification via a call-back function.
`callbk_param`	specifies the desired parameter value for the call-back function for a notification via a call-back function.

Notification via Signals

When this application requests this type of notification, it only fills out the `sig2send` field of the notify union. The file manager automatically acquires the current process ID and initializes it in the `proc_id` parameter of the notify union.

Notification via Events

The application fills out the event ID as well as the value of the event in the notify union. If incrementing events are used, the `ev_inc_val` parameter should also be filled out.

Notification Extensions

Extensions allow for incrementing signals or events as well as sending Multimedia Application User Interface (MAUI) mailbox messages. At this point, MAUI messages are not used.

SoftStax Working Environment

Defs Files

The generic OS definitions are located in MWOS/SRC/DEFS or the DEFS subdirectory for the processor family (MWOS/OS9/DEFS or MWOS/OS9000/PPC/DEFS). All *mw*SoftStax definition files are found in MWOS/SRC/DEFS/SPF.

Keep in mind when you create an application and use the makefiles that make the examples, the previous discussion is handled automatically. These makefiles are defined to search the correct subdirectories for the correct target processor.

Driver Source Files

The following discussion elaborates on all the files found in the SPPROTO subdirectory.

defs.h

Contains standard include files for a driver. spf.h gets defined after SPF_DRSTAT and SPF_LUSTAT macros because these macros are referenced in spf.h.

SPF_DRSTAT, SPF_LUSTAT, SPF_PPSTAT definitions

These macros allow the driver to define its own variables in the logical unit driver static and per path static storage structures. For example, add variables x and y to the driver static, variable z to the logical unit, and a and b to the per path storage.

```
#define SPF_PPSTAT \
      u_int32 a; \
      u_int32 b;
#define SPF_DRSTAT \
```

```
      u_int32 x; \
      u_init32 y;
#define SPF_LUSTAT \
      u_init32 *z;
```

SPF_LUSTAT_INIT definitions

This is where the descriptor gets the initialized values for the logical unit static. Per the previous example, initialize `z` to 5.

```
#define SPF_LUSTAT_INIT \
      5
```

Note

The initialization for SPF_DRSTAT variable goes in `main.c` where the driver static is defined.

history.h

`history.h` includes edition/attributes for any editorials present on the driver and descriptors. The `history.h` file is principally a file to track the history of the driver and descriptors. There are two `_asm` instructions in this file. This is where the system attributes and revision status are updated. Type the following instruction to see the system attributes and status:

```
os9ident $ MWOS/OS9/68000/CMDS/BOOTOBJS/spproto
```

or on an OS-9 for 68K or OS-9 system enter:

```
ident MWOS/OS9/68000/CMDS/BOOTOBJS/spproto
```

This command lists all vital statistics for the module `spproto`. On your display, look through the `Edition` and `Ty/La At/Rev` lines. Note the same values appear as in the `_asm()` lines of the `history.h` file of `spproto`.

The `_sysattr` definition is especially important. For example, assume you have an old driver in ROM and the `_sysattr` value is 0xA001. You found a bug in this driver and fixed it. Before recompiling, you change the _sysattr value to 0xA002. You bring the target system up and it loads the old driver from ROM into memory. Instead of having to reprogram your ROM every time to check the fix, you can load the new driver into memory. OS-9 compares the _sysattr values of the driver and uses the one with the highest value. In this case, because you changed the value of your driver to 0xA002, it replaces the old driver. If you did not change it, the values would have been the same and the older module would remain in memory and be used.

The _sysedit value allows you to keep track of the driver and descriptor versions in the field, so you should always increment _sysedit values before releasing changes. You should also document those changes. This way, if problems occur, you can trace the driver via the _sysedit value and your documentation.

If you find two OS-9 modules with the same version number, they may still be different. A quick check to compare the module size and module CRC using the os9ident or ident commands show if they are different in any way.

proto.h

proto.h defines the function prototypes for the driver.

main.c

main.c contains the initialization variables OS-9 uses to initialize the driver static structure before calling the driver. Make sure to include initialized values for the variables declared in the SPF_DRSTAT macro.

entry.c

entry.c contains code for all the entry points in a driver. Typically, if implementing a state machine, you have separate source files to process incoming and outgoing packets used in dr_updata() and dr_downdata(). This provides source code modularity.

misc.c

The misc.c file can be found in /MWOS/SRC/DPIO/SPF/DRVR

This file contains two subtle but important standard functions hardware drivers used to get and free mbufs. The mbuf facility itself includes two standard calls to get and free packets: m_get() and m_free(). Do not use the m_get() call if this is a hardware driver using the DR_FMCALLUP_PKT() call.

For instance, on the receive side, SoftStax hardware drivers put the device entry handle of the next highest protocol on the stack in every receive mbuf to correctly pass the packet up the stack. The get_mbuf() function provided in misc.c automatically reserves room right after the mbuf header for the device entry pointer and puts this information into the mbuf. If the protocol layout is changed in the future, drivers using misc.c simply need to be recompiled. Whereas, drivers that do not use misc.c need to be modified.

There are also mbuf nofree support functions provided for drivers wanting to use the SPF_NOFREE functionality.

For More Information

Refer to Appendix B: The Mbuf Facility for detailed information concerning SPF_NOFREE.

Makefiles

Two driver makefiles (spfdrvr.mak and spfdbg.mak) are located in the SPPROTO subdirectory:

- The spfdrvr.mak creates a driver object without the debugging library functionality enabled. This object is stored in the CMDS/BOOTOBJS/SPF for the correct processor family.

- The spfdbg.mak uses the same source files, but includes the debugging functionality. This object is stored in the CMDS/BOOTOBJS/SPF/MON directory of the processor family.

When integrating your system, use the objects out of the CMDS/BOOTOBJS directory. If your communications software does not seem to be working correctly, load the objects out of the MON directory. These objects create debug modules in memory. When the software is not working, and the source code for the protocol is unavailable, viewing the log can sometimes help.

Sometimes, the debug module logs an error pointing to an incompatibility in the peer software with which it is trying to communicate. Re-create the problem. If it is a Microware protocol driver, save the debug module and send it to Microware.

When making a descriptor for a particular driver, the makefiles for those descriptors are in one of two places in the MWOS directory structure depending on whether it is a hardware driver or a protocol driver.

Hardware Driver makefile Descriptors

For hardware drivers where the object typically goes with a particular port to a board, the makefile for the descriptor is usually found in the processor family subdirectory under PORTS.

Example hardware driver makefile source code has been included for demonstration purposes. For example, the Z8530 source in SPF/DRVR/Z8530 makes a driver for use on the MVME147 CPU board. Therefore, if you look at the MWOS/OS9/68020/PORTS/MVME147 directory, you see the makefiles for this port-specific compile of the Z8530 driver with respect to the MVME147 board. Directly under the MVME147 subdirectory is the DEFS subdirectory. This contains the spf_desc.h file that can be added to or modified to change or create device descriptors for the sp8530 driver. The object is stored in the CMDS/BOOTOBJS directory within the PORTS/MVME147 subdirectory.

When making the objects for a specific port, the objects are stored in a local CMDS/BOOTOBJS directory and the makefiles are located in the corresponding PORTS directory such as

MWOS/OS9/68020/PORTS/MVME147. The makefiles found in this directory can be used as templates and are portable to other PORTS directories for making your own SoftStax driver and descriptor port.

Protocol Drivers makefile Descriptors

The descriptor makefiles are stored in the MWOS/SRC/DPIO/SPF/DRVR directory under the protocol driver name. For example, the SPPROTO directory contains defs.h, the file the descriptor needs to compile properly along with spfdesc.mak, the makefile for the descriptor. Directly under the SPPROTO directory is the DEFS subdirectory. It contains all the initialization information for creating the specific descriptors.

MON Directory

All objects for the file manager and drivers are in the CMDS/BOOTOBJS/SPF directory as shown in **Figure 1-4**. However, Microware has special debugging objects that create data modules in memory and log the processing of that driver. If problems occur, this debug module can be used to quickly find and fix the problem. Refer to the debugging appendix for extended details.

When making the objects for a specific port, the objects are stored in a local CMDS/BOOTOBJS directory as shown for this example. The makefiles found in this directory can be used as templates and are portable to other PORTS directories for making your own SoftStax driver and descriptor port.

Making a Driver using the SPPROTO Template

In this example, create your own driver using the `SPPROTO` directory as the template.

For More Information

Refer to Chapter 3: SPPROTO Driver in this manual for more information.

Complete the following steps to create a driver:

Step 1. Create a directory called `SPMYDRVR` under the `MWOS/SRC/DPIO/SPF/DRVR` directory. Perform this on your host machine.

Step 2. Copy the `SPPROTO` directory and its contents to the `SPMYDRVR` directory.

You should have a directory called `SPMYDRVR` with the same contents as `SPPROTO`.

Step 3. Create your driver by editing all the `.c` and `.h` files.

The `SPPROTO` template only provides one `.c` file, `entry.c`. You may want to create more `.c` files.

For example, say the `spmydriver` has `entry.c`, `foo1.c`, and `foo2.c`. Add `foo1.c` and `foo2.c` to the `makefile` so the driver compiles correctly. The best way to do this is to search through `spfdrvr.mak` for instances of “`entry`”, when you find one, add the same command lines for “`foo1`” and “`foo2`”.

Step 4. Edit the driver makefiles to create the driver, `spmydrvr`.

Step 5. Open the `spfdrvr.mak` file for editing.

Step 6. Change the `TRGTS` macro to:

```
TRGTS = spmydrvr
```

Step 7. Search the `spfdrvr.mak` file for all `spproto` occurrences and change them to `spmydrvr`.

Step 8. Customize each entry point to fit your hardware configuration.

Step 9. Customize device descriptors to fit your drivers.

Step 10. Create the debug version in the `spfdbg.mak` file.

You can also copy the `spfdrvr.mak` to `spfdbg.mak` and add the following debug macro to `spfdbg.mak`:

```
DEBUG = -dDEBUG=2000 -l=$(MWOS_DFTLIB)/dbg_mod.l
```

This is the only difference between the `spfdrvr` and `spfdbg` makefiles.

Step 11. Use the `make` utility to create object files.

Step 12. Test your driver.

Creating Device Descriptors

Go to the MWOS/SRC/DPIO/SPF/DRVR/SPPROTO directory and walk through the changes required by spfdesc.mak and DEFS/spf_desc.h to make a new descriptor.

The spproto driver comes with one descriptor: proto.

Complete the following steps to make a new descriptor:

Step 1. Open the spfdesc.mak file for editing.

Step 2. Add new descriptor names to the following line:

```
TRGTS = proto
```

The new line looks like:

```
TRGTS = proto proto1
```

This is the only change to the spfdesc.mak file.

Step 3. Go to the DEFS directory.

Step 4. Open the spf_desc.h file for editing. The file has a section with #ifdef proto and #endif wrapped around it.

Step 5. Copy this wrapper.

Step 6. Edit the values for the new section you created. Make sure to change the #ifdef line for the new section to read #ifdef proto1.

Step 7. Type the following after changing the makefile and adding the initialization for proto1.

```
os9make -f=spfdesc.mak
```

When completed, look in the MWOS/OS9/68000/CMDS/BOOTOBJS/SPF directory. You see your new device descriptor, proto1.

For More Information

Refer to Chapter 3: SPPROTO Driver in this manual for more information.

Makefile Summary

spfdrvr.mak

Makes the driver. When using this option, remember the following:

- define the `MWOS` macro correctly
- define the correct target(s)
- add `.c` files to `CFILES`, `RFILES/IFILES` macros
- use `OPTMZ` macro for optimization tuning
- any added files require the `.r` and `.i` dependency

spfdbg.mak

Makes the debugging version of the driver.

`spfdbg.mak` is exactly the same as `spfdrvr.mak`, but creates debugging version and stores it in the `MON` directory.

```
DEBUG=-dDEBUG=0x2000 -l=$(MWOS_DFTLIB)/dbg_mod.l
```

This defines the size of the debug data module as 2000H bytes and includes the debugging library appendix.

spfdesc.mak

Makes the device descriptors. When using this option:

- define `MWOS` correctly
- create all desired targets
- use `TRGTS` macro location where descriptor's names are defined

spf_desc.h

Contains macros defined to ensure the path options and ITEM structure initializes correctly for this descriptor.

Note

`spf.h` defines defaults for all macros in `spf_desc.h`. If macros are not defined, they are defined in a default manner by `spf.h`.

SoftStax Support Facilities for the Driver

Libraries

Libraries the driver uses can be found under the processor type `LIB` subdirectory as shown in the figure **Figure 1-4** (`OS9/68000/LIB` or `OS9000/PPC/LIB`).

Mbuf Library (mbuf.l)

The system mbuf facility is a pre-allocated pool of memory the OS reserves. This provides faster allocation than `malloc()` and buffers can be quickly allocated/deallocated by the system. This library uses:

- `Mbuf` structure
- `Mbuf` calls

For More Information

Refer to Appendix B: The Mbuf Facility for detailed information concerning the Mbuf facility.

Timer Service Library (timer.l)

The timer library enables drivers to set up one-shot and cyclic timers for doing protocol processing such as timeouts. These are not intended as accurate, high resolution timers and should not be used as such.

The timers are implemented using a single system alarm that schedules the receive thread to execute the requested funtion when the timer expires. This system alarm is run twice as fast as the shortest timer resulting in an

accuracy of approximately 1/2 the smallest timer interval.
The accuracy may further be reduced due to overhead of scheduling the receive thread to execute the timer function.

Include the <SPF/timer.h> header file in `defs.h` to use timer services.

The `timer_pb` structure is defined in `timer.h`. The first six parameters in the `timer_pb` structure are not to be written to by the driver. The timer service keeps these variables. The variables from `timer_type` on down are to be initialized by the driver for the desired timer.

Variables to be initialized by the driver for timer service:

`timer_type`	Defined as `TIMER_ONE_SHOT` or `TIMER_CYCLIC`.
`timer_call_type`	Drivers should always set this to `CALL_FUNCTION`.
`timer_interval`	Timer interval in milliseconds desired.
`t_func`	Function to call when timer expires.
`t_pb`	Parameter to pass to `t_func`.

Once these parameters are set, the driver can make the `timer_start()` call.

These structures are used for the lifetime of the timer by the timer service, so they are not reusable unless the timer for the `timer_pb` has been stopped.

WARNING
Timer functions can not be used in an interrupt service routine.

The following three API calls are available for the driver:

timer_start()

This starts a cyclic or one-shot timer depending on how you fill out the timer parameter block in `timer.h`. Calling `timer_start()` with an already started time restarts the timer.

timer_restart()

This restarts a cyclic timer and readjusts it to go off at the same interval, but at the current time the call was made. Calling `timer_restart()` with a timer that hasn't been started starts the timer.

timer_stop()

This stops a timer and takes it out of the timer queue.

Per Path Storage Library (ppstat.l)

The per path storage library provides standard calls drivers can make to create, delete, and search through the per path list as described at the beginning of this manual. The SPPROTO driver uses this library so you can see where these driver library calls are used.

Debugging Library (dbg_mod.l)

This facility allows writers to create debugging output data modules in memory to gather statistics in real time as well as log any errors that occur and might otherwise be hard to report.

Attributes of this library include the following:

- name is usually called `dbg_xxx` by convention
- one or more drivers can use the same `debug` data module
- `debug` driver interface calls

Use `rombug` to view the output.

For More Information

Refer to Appendix A: Debugging for detailed information.

Flow Control

Earlier in the manual, the `pd_readsz` and `pd_writesz` parameters were discussed. When data is enqueued on a path, SoftStax looks to see if the data now exceeds the `pd_readsz` value. If it does, SoftStax issues an `SPF_SS_FLOWON` setstat down the stack. At this point, if a driver implements flow control, it initiates messaging to stop the flow of data.

When the application reads data from the queue, SoftStax determines if the read queue size is now below the `pd_readsz` threshold. If it is, SoftStax issues an `SPF_SS_FLOWOFF` setstat down the stack. At this point, the driver implementing flow control sends messages to its peer to begin the flow of data again.

It is up to the hardware driver enqueuing transmit data to enforce the size of the transmit queue size using the `dr_writesz` parameter in the path.

Driver Considerations

Hardware Drivers

Attributes of hardware drivers include:

- volatile structures and variables/optimization
- interrupt service routines TX/RX Error reporting and `FMCALLUP_PKT`
- multiple protocols on top of the same driver

High-level Data Link Control (HDLC) Controllers

Standard HDLC does not allow for multiple endpoint addressing and multi-protocol support on a given single HDLC link. Because of this, the stacking and unstacking aspects of this kind of driver are slightly different.

Typically, HDLC hardware drivers keep the associated protocol driver above in another variable in the logical unit. (**Example:** `lu_prot_above`.) At `SPF_SS_PUSH`, the driver copies the `lu_updrvr` to `lu_prot_above`. If this logical unit is used to attempt another push, the protocol returns an `EOS_DEVBSY` error, meaning only one protocol can be pushed on this interface of the driver. When called at `SPF_SS_POP`, the `lu_prot_above` is still copied from `lu_updrvr`, but the driver once again allows the next `SPF_SS_PUSH` to occur on this logical unit successfully.

For More Information

Refer to the SP82525 driver source code for an implementation of this.

ATM Drivers

ATM communication protocol drivers allow multiple endpoint addressing via the VPI/VCI addresses. A given ATM interface (logical unit) can run multiple protocols above based on the VPI/VCI. This means ATM drivers end up allocating per-path storage for each VPI/VCI opened. Every path has a one-to-one correspondence with a VPI/VCI. The per-path storage stores the updriver for the path.

Data Link Layer Driver Considerations

Data link layers provide reliable data transmission (retransmission queues `SPF_NOFREE`). Data link layer protocols typically provide a reliable transport for protocols above. To provide reliability, all currently unacknowledged packets must be kept by the protocol in case they need to be retransmitted.

The mbuf facility provides for an `SPF_NOFREE` flag in the `m_flags` field. When the hardware driver completes transmission of the mbuf, it calls `m_free_p()` to release the packet. This library call checks for the `SPF_NOFREE` bit. If it is set, the library simply sets the `SPF_DONE` bit. Otherwise, it returns the mbuf to the memory pool. Therefore, when the data link layer driver really wants to return the mbuf, the protocol must clear the `SPF_NOFREE` bit before calling an `m_free_xxx` library call.

Mbuf leaks occur if drivers using the `SPF_NOFREE` bit do not clear it before using the library to return an mbuf to the memory pool using `m_free_p`.

More Hold On Close

Hold on close allows you to hold the path open while the path closes for graceful closing.

For some protocols, there are multiple messages that must go back and forth in order to gracefully terminate a connection. These messages are sometimes initiated by the application calling the `ite_path_close()` or `_os_close()` calls. The path descriptor, `deventry`, `drstat`, `lustat`, and `pp_stg` must be present until the messaging completes. However, the OS deallocates the path descriptor immediately after `close()`. If this is the last path open to the protocol, the `deventry`, `drstat`, and `lustat` also returns immediately after close.

If you increment the `stk_hold_on_close` parameter during the `SPF_GS_UPDATE`, your structures are present until termination has completed allowing graceful protocol termination for the path.

When the application closes the path, SPF calls your driver at SPF_SS_APPCLOSE. In this setstat you should initiate the termination messaging, then set a timer and return. When incoming confirmation messages come in, send the appropriate reply message and on completion of the messaging (or timeout waiting), call the DR_FMCALLUP_CLOSE() macro to tell SPF the protocol has completed the close. SPF then immediately calls SPF_SS_CLOSE on the path.

Network Layer Drivers

ITE_DIAL

Uses ite_conn_pb in item_pvt.h. At this point, your protocol should send the connect request message, change states, and set the ITEM device type dev_callstate field to ITE_CS_CONNEST. Notice this call uses a notify parameter block. When the incoming connect confirmation comes in, change the ITEM dev_callstate field to ITE_CS_ACTIVE and send the notification in the ON_CONN element of the path's notification list.

ITE_HANGUP

Uses the spf_ss_pb in spf.h. The parameter points to the caller's ite_cctl_pb found in item.h. Some protocols allow response and reason fields as well as user data to be passed between endpoints when disconnecting. This parameter block is filled by the user. If the parameter field is NULL, the protocol should use default values if needed by the protocol. The protocol always assumes normal clearing procedures on hangup unless otherwise noted.

ITE_ANSWER

Uses the ite_conn_pb and ite_cctl_pb in the cctl_pb field of the ite_conn_pb. The protocol should send the connect confirmation and either set the dev_callstate to ITE_CS_ACTIVE, or if another confirmation from the network is required, wait until it arrives.

Hold-on-Close (HOC)

Hold-on-close (HOC) drivers must call up after they have attempted to gracefully close. If they do not, SPF never closes the path it has duped and extra paths are left around after every use of the protocol.

HOC drivers must set the PATH_HOLDONCLOSE variable in spf_desc.h to PATH_HOLD. If drivers do not have this macro set, it defaults to PATH_NOHOLD or historical operation.

HOC drivers must add the following to the SPF_GS_UPDATE getstat:

```
IF lustat->lu_dndrvr {
add: upb->stk_hold_on_close +=1;
               /* Make sure increments! */
} ELSE {
add: upb->stk_hold_on_close=1;
}
```

When HOC drivers get called at SPF_SS_CLOSE for a path before getting called at SPF_SS_APPCLOSE, the driver should simply terminate immediately without attempting graceful closing of the path. This is because the user has issued a pop of this protocol off of the stack.

HOC drivers must implement the SPF_SS_APPCLOSE setstat. This is the setstat code used when SPF is indicating to the HOC driver to begin graceful closing with its peer.

The final issue concerning SPF_SS_APPCLOSE is non-HOC drivers need not implement this setstat because it gets passed down transparently to the protocols that understand it. This implementation is backwards compatible with old protocol and device drivers.

Do not sleep during SPF_SS_CLOSE because SPF_SS_CLOSE is issued by the SPF receive thread.

HOC Scenarios

The following three scenarios are diagrammed to assist you in understanding the flow through the system:

1. Open a no-HOC path. Push HOC driver #1. Push HOC driver #2. Pop HOC driver #2. Pop HOC driver #1. Close the resulting no-HOC path.

2. Open a path with one HOC driver and close the path.
3. Open a path with two HOC drivers and close the path. All other scenario derivatives are proven by correct operation of these.

Scenario #1

Step 1. Open no HOC path

```
fm_open as usual.
```

Step 2. Push HOC driver #1.

```
SPF_SS_PUSH:
   push()
   update()
      -HOC 0->1 so _os_dup() called on path
      {pathcount=2, HOC=1}
```

Step 3. Push HOC driver #2.

```
SPF_SS_PUSH:
   push()
   update()
      -HOC 1->2, (only dup if HOC was 0, and now non-0)
```

Step 4. Pop the path (HOC driver #2)

```
SPF_SS_POP:
   pop()
      -HOC driver #2 gets called at
SPF_SS_CLOSE:
   /* HOC driver notes SS_CLOSE called before
   SS_APPCLOSE, so it must just close without any
   graceful messaging*/
   update()
      -HOC 2->1, (SPF does nothing)
```

Step 5. Pop the path again (HOC driver #1)

```
SPF_SS_POP:
   pop()
      -HOC driver #1 gets called at SPF_SS_CLOSE:
   /*HOC driver notes SS_CLOSE called before
   SS_APPCLOSE, so it must just close without any
```

```
   graceful messaging*/
   update()
      -HOC 1->0, Since HOC was non-zero and is now zero,
      SPF calls _os_close()
      fm_close()
         -pathcount=1, but HOC=0, do nothing and exit.
```

Step 6. Close the no HOC path

```
fm_close()
   -Normal close as before.
```

Scenario #2

Step 1. Open one HOC path

```
fm_open as usual.
```

Step 2. Close one HOC path.

```
fm_close()
   -pathcount=1, but HOC=0, do nothing and exit.
```

Step 3. Close the no HOC path

```
fm_close()
   {pathcount=1, HOC=1} so SPF calls SPF_SS_APPCLOSE
   SPF_SS_APPCLOSE:
   <HOC protocol initiates close messaging and returns>
   <After either messaging has completed successfully or
   protocol times out, the driver will use the
   DR_FMCALLUP_CLOSE() macro in spf.h>
   DR_FMCALLUP_CLOSE (in spf)
   hoc_func()
      fm_close()
      pathcount=0, so close normally.
```

Scenario #3

Step 1. Open two HOC paths

```
fm_open as usual.
   -update()
      HOC 0-> so SPF will _os_dup() path (SPF checks
      0 to non-0) {pathcount=2}
```

Step 2. Close two HOC paths.

```
fm_close()
   {pathcount=1, HOC=2} so SPF calls SPF_SS_APPCLOSE
      SPF_SS_APPCLOSE: <first HOC protocol initiates
      close messaging and returns>
<After either messaging has completed successfully or
protocol times out, the first driver uses
DR_FMCALLUP_CLOSE() macro in spf.h>
DR_FMCALLUP_CLOSE (in spf)
   HOC 2->1 so SPF issues the SPF_SS_APPCLOSE again
   <1st HOC protocol stored the fact for this path, it
   already completes the APPCLOSE, so it passes it
   transparently down>
   <2nd HOC protocol gets SPF_SS_APPCLOSE and initiate
   its messaging and returns>
<After either messaging has completed successfully or
protocol times out, the second driver uses the
DR_FMCALLUP_CLOSE() macro in spf.h>
DR_FMCALLUP_CLOSE (in spf)
   HOC 1->0 so SPF calls _os_close()
   _os_close (HOC path)
      fm_close()
pathcount=0, so close normally.
```

Out-of-band Protocol Considerations with ITEM

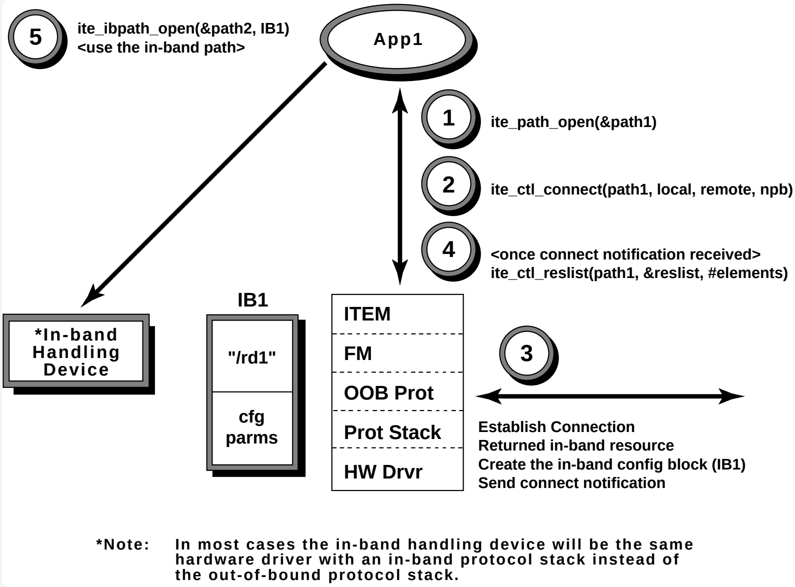
There are two key issues to discuss when talking about out-of-band protocol stacks. The first is the ability to configure the ib-band connection correctly once the out-of-band signalling is completed. The second is to provide quality of service labels, or profiles, which allow applications to make out-of-band calls using a simple profile which translates into more complex messaging for the protocol stack below.

In-Band Configuration of Out-of-Band Connections

Much like standard Plain Old Telephone Service (POTS) today, many protocols are *Out-of-band* protocols. That is you dial digits to the network administration entity to get an end-to-end connection to somewhere else.

For example:

Figure 2-10 Sequence of Events for Out-of-Band Protocol Connection



The figure Figure 2-10 Sequence of Events for Out-of-Band Protocol Connection shows the 5 steps that take place when an out-of-band (oob) connection is created between the entity labelled *App 1* and another endpoint. From the application's perspective, everything is the same except for the additions of steps 4 and 5.

The `spf_oob.h` definitions file contains structures and definitions for the in-band configuration block. The idea here is that the application makes a connection. Once notified that the connection is complete, the application requests the in-band configuration block from the protocol stack by making the `ite_ctl_reslist()` call. The second parameter in the call is a pointer to the in-band configuration block `ib_cfg_pb`. If the pointer contains `NULL`, no in-band device is needed and end-to-end communication can take place on this (the signalling) path. If a non-NULL pointer is returned, an in-band path must be opened using this block.

Declaration

The `ib_cfg_pb` structure is declared in the file `SPF/spf_oob.h` as follows:

```
typedef struct ib_cfg_pb {
   Ib_cfg_pb         ib_next;
   char              ib_name[IB_NAME_SIZE];
   u_int32           oob_syspath;
   u_int16           ib_flags;
      #define NEW_IB_RES   0x0001
      #define CLEAR_NEWIB  0xFFFE
   u_int16           ib_obj_type;
      #define IB_OBJ_ATM   1
      #define IB_OBJ_BRI   2

   u_int8ib_object[32];
   error_code  (*ib_callback) (
      Ib_cfg_pb      ibpb,
      void           *spb);
   error_code  (*oob_callback) (
      Ib_cfg_pb      ibpb,
      void           *spb);
      #define IB_STATE_CHANGE 0x01
      #define IB_CHECK_IN  0x02
      #define CB_CONN_TERM 0x03
   void              *ib_deventry;
   void              *oob_deventry;
   u_char            ib_state;
      #define IB_FREE      0x00
      #define IB_AWAIT_ADD 0x01
      #define IB_ACTIVE    0x02
      #define IB_AWAIT_DEL 0x03
      #define IB_RESERVED  0x04
      #define IB_WAIT_CHECKIN 0x05
   u_char            ib_rsv2[3];
};
```

Description

The application may read the variables down to the `ib_object` array. From the `ib_callback` field on down is only used for communication between the in-band and out-of-band drivers.

Fields

`ib_next`	points to next block in list (Driver use only).
`ib_name`	is the name of the descriptor to use for the end-to-end communication.
`oob_syspath`	is the system path number of the out-of-band path that created this block.
`ib_flags`	is the flags field. `NEW_IB_RES`: Set if this is a new block not yet received by the application.
`ib_obj_type`	is the in-band object type. Used to interpret contents of the `ib_object` array.
`ib_object`	includes in-band specific information for proper in-band driver configuration.
`ib_callback`	is used for communication from oob protocol to ib protocol
`oob_callback`	is used for communication from ib protocol to oob protocol.
`ib_deventry`	is the in-band driver's device entry.
`oob_deventry`	is the oob driver's device entry.
`ib_state`	is the current state of the ib connection.
`ib_rsv2`	is used for long word alignment/reserved for future use.

For ease of use, there is the `ite_ibpath_open()` ITEM call used to open an in-band path without needing to look inside the in-band configuration block. The application also has the option of performing a standard `open()` call using the `ib_name[]` string in the block.

There are 2 things that must be added to the `spf_oob.h` file for proper registration of your oob protocol. First, in order for all applications and protocol stacks to be interoperable with your protocol (IP over ATM and

MPEG over ISDN), we need to create a profile ID for your protocol along with an in-band object structure. This way, based on the profile ID, the application can map the `ib_object` structure correctly.

Creation of a protocol ID is simply defining a `PR_STRUCT_...` macro in `spf_oob.h` with the others. Contact Microware to register this value.

Creating an in-band object structure to cast the `ib_object` array with is protocol dependent. Notice for the ATM example, the in-band object maps to a VPI/VCI connection. ISDN maps to a call reference and channel ID. Depending on what kind of addressing identification your protocol has, this structure reflects that for the `ibobj_xxx` protocol.

The two setstat codes for getting the parameter block and setting the in-band resource configuration for an in-band driver are listed below:

ITE_RESOURCE_LIST

FROM: `ite_ctl_reslist`

This getstat is implemented by the out-of-band protocol. The `ITE_RESOURCE_LIST` setstat uses the `ite_rescfg_pb` in `item_pvt.h`. The out of band protocol should return the number of in-band connections (called resources) `pb.spb.size`. Usually, this number is only `1`. However, some interactive TV session control protocols create many in-band connections as a result of one service call. The pointer to the in-band resource list (`ib_cfg_pb` in `spf_oob.h`) is returned in `pb.spb.param`.

ITE_IBRES_CFG

FROM: `ite_ibpath_open`

This setstat is optionally implemented by the in-band protocol. If this setstat is not implemented, the device descriptor used to open the in-band path configures the device, not the in-band parameter block. After `SPF_SS_OPEN`, the device gets called at this setstat to configure itself per the parameters negotiated by the out-of-band protocol. A pointer to the `ib_cfg_pb` structure is in the `spb->param` field. Inter-driver communication is achieved through this in-band parameter block between the in-band device and out-of-band device using the `ib_callback()` and oob_callback entry points in the ib_cfg_pb. This way, the in-band resource can know things like when the out-of-band connection has gone away without depending on the application being correctly written.

Profiles for out-of-band connectivity

The second part of the spf_oob.h file deals with profiles for out-of-band protocols. A profile is a simple mapping of a requested service type from the application's perspective to various parameters and information elements that must be used in order to set up an in-band connection that can provide that kind of service. Applications remain portable because they simply ask to set a profile like *IP connection* or *MPEG connection*. These profile identifiers map to stored structures for particular protocols that in turn use the profile type to know how to create the connection request message.

The spf_oob.h file specifies the types of profiles using the ITE_SVC_ prefix. These profiles are passed between applications and protocols using the conn_type structure. The conn_type structure also contains other generic kinds of service options like data rate and subaddressing if applicable to the protocol stack.

Profile Implementation at the Driver Level

First of all, the protocols key off of the ITE_SVC_xxx macros in spf_oob.h. The specification of the protocol also determines whether the protocol can distinguish between profiles or not (the protocol requests the connection the same regardless of the kind of data being sent and received over the connection). If the protocol makes no distinction, then profiling for the protocol returns unknown service errors.

The next step is to create an xxx_pr.h file (where xxx is the name of your protocol). Examples of these files are isdn_pr.h and atm_pr.h found in the ISDN or ATM Communications Paks respectively. This file contains the profile structure as well as all the protocol specific structures and definitions needed to implement the profile options for the services.

If we are implementing protocol xxx, we would create an xxx_profile structure in xxx_pr.h. A sample xxx_profile structure is shown on the following page:

Declaration

```
typedef struct bri_profile {
   u_char                 pr_struct_type;
   u_char                 pr_svc_type;
      ITE_SVC_VOICE             1
      ITE_SVC_DATA_ANY          2
      ITE_SVC_DATA_MPEG         3
      ITE_SVC_DATA_IP           4
   u_int16                pr_size;
   char                   pr_desc[16];
   bri_profile_body       pr_body;
} bri_profile, *Bri_profile;
```

Description

The general concept is that the first variables are present in the structure regardless of the specific profile, then there is a structure that follows these variables which contains the protocol specific piece of the profile.

Fields

`pr_struct_type`	should be `PR_STRUCT_xxx` found in `spf_oob.h`. Contact Microware to register your profile ID.
`pr_service_type`	is the profile key macro in `spf_oob.h` (`ITE_SVC_`...).
`pr_size`	is the size of the `xxx_profile` structure.
`pr_desc[16]`	is the in-band descriptor string to open for this profile. This string is copied into the `ib_name[]` field when `ib_cfg_pb` is created.
`pr_body`	is a structure created with the protocol specific parameters to provide an in-band connection for this profile.

Now that we have created the profile structure, its time to put the profiles in the driver. Since we want the descriptor to hold all the profiles understood by the driver, an array of `xxx_profile` structures are kept in the logical unit (lu_profile_list[]). This way the spf_desc.h file can be used to initialize these array profiles as needed for voice, data, IP, and MPEG call operation. Also, the logical unit should have one of these profiles to use as the default profile if none is specified explicitly (lu_profile_default). This lu_profile_default value is the array element to use in the list as the default. Notice also that the ITE_SVC_xxx macros in spf_oob.h go from 1 to 4. These values can be used as array indexes for ease of profile listing (the [1] element in the profile list is ITE_SVC_VOICE profile).

After the profile array has been implemented, there should be an xxx_profile structure in the per path storage also. The profiles in the logical unit are not changeable without changing the descriptor. Once a path is opened, the default profile gets copied into the xxx_profile structure stored in the per path storage. It is this copy that can be read and modified by the application owning the path.

The following information shows additions to the defs.h file for the driver and a simplified example of an xxx_pr.h file.

Sample xxx_pr.h

```
typedef struct xxx_profile_body {
   u_char               qos1[8]; /* Example quality of*/
                                  /* svc param */
} xxx_profile_body, *Xxx_profile_body;

typedef struct xxx_profile {
   u_char               pr_struct_type;
   u_char               pr_svc_type;
   u_int16              pr_desc[16];
   xxx_profile_body     pr_body;
} xxx_profile, *Xxx_profile;
```

Additions to defs.h

Initialization defaults for SPF_LUSTAT_INIT:

```
#ifndef QOS_VOICE
#define QOS_VOICE {\
   PR_STRUCT_XXX,\
   ITE_SVC_VOICE,\
   {"/voice_desc"},\
```

```
#ifndef QOS_DATA
#define QOS_DATA {\
PR_STRUCT_XXX,\
ITE_SVC_DATA_ANY,
{"/data_desc"},\
```

```
   {0,0,0,0,0,0,0,0}       {1,1,1,1,1,1,1,1}\
}     }
#ifndef DEFAULT_PROFILE
#define DEFAULT_PROFILE ITE_SVC_VOICE
#endif
```

Added variables to logical unit:

```
xxx_profile          lu_profile_list[2];
                        /* Only distinguished voice/data */
u_int32              lu_profile_default;
                        /* Default profile index*/
```

Added variable to per path storage:

```
xxx_profile          pp_profile;
                        /* The profile used for this path*/
```

`SPF_LUSTAT_INIT` additions:

```
{\
QOS_VOICE,\
QOS_DATA,\
},\
{DEFAULT_PROFILE - 1},\
```

Profile API calls

The last thing to cover is the driver implementation of the profile API calls found in ITEM and prototyped in `spf_oob.h`.

Now that we know how the profile mechanism is implemented in the driver, we need to expose enough flexibility to the application to allow it to change or customize profiles for its own use. Two API calls have been created for this, `ite_path_profileget()` and `ite_path_profileset()`.

For More Information

Refer to the ***SoftStax Programming Reference*** for more information about the `ite_path_profileget()` and `ite_path_profileset()` calls.

Chapter 3: SPPROTO Driver

This chapter details the SPPROTO driver and provides a template for writing other drivers to be written.

SoftStax Driver Walk Through: spproto

The SPPROTO driver is the template driver provided with SoftStax. It is composed of the following basic source files:

- `defs.h`
- `history.h`
- `proto.h`
- `entry.c`
- `main.c`

defs.h

This file includes all definition files required for the driver to compile properly. Most of the include files are standard OS-9 for 68K include files. One of the exceptions is `defconv.h`. This file equates all OS-9 structure names and macros to their corresponding names in OS-9 for 68K. This is part of the Dual Ported I/O (DPIO) support. SoftStax driver source is source code portable across processors.

The `spf.h` file is included in `defs.h`, as well as the local files `proto.h` and `history.h` described on the following pages.

One important note about drivers using `spf.h` include files is the line `#define SPF_DRVR`, which must be included before including `spf.h`. This is because `spf.h` is conditionalized into the following three sections:

- application oriented
- driver and descriptor oriented
- file manager oriented

Applications including `<spf.h>` without defining `SPF_DRVR` or `SPF_FM`, get the first part of `spf.h`, which includes macro definitions and function prototypes. If you define `SPF_DRVR` before including `spf.h` (as spproto does), you bring in definitions for all structures the driver needs to know about. Finally, if you define `SPF_FM`, the entire `spf.h` file is brought in.

WARNING
Only the SPF manager should include the `SPF_FM` macro. Otherwise, your driver runs the risk of being incompatible with future versions of SoftStax.

The other part of `defs.h` is the declaration of the device-specific part of the logical unit. The definition begins with `#define SPF_LUSTAT`. You can find this macro used in the `spf_lustat` structure of `spf.h`.

The driver then defines `SPF_LUSTAT` as all the variables that should be part of the logical unit for the driver specific portion. `spproto` only defines two variables in its logical unit specific static storage area, `lu_dbg` and `lu_dbg_name`. These pointers are used by the debugging data module version of the driver to identify the pointer to the debug data module and the name of the debug data module to create or link.

Note

`spf.h` should be included *after* the `SPF_LUSTAT` macro is defined. If not, `SPF_LUSTAT` will not be defined when the compiler is resolving the `spf_lustat` data structure in `spf.h`, and the logical unit specific storage for these two structures will not be included.

For More Information

Refer to Appendix A: Debugging for more information on debugging data module support.

`SPF_DRSTAT` is also declared here. This is very much like the logical unit discussion previously, only for the driver static. Again, notice in the `spf.h` `spf_drstat` structure (at the bottom), there is a `#ifdef SPF_DRSTAT`.

For More Information

Refer to Chapter 2: Creating SoftStax Drivers for more information about `spf_drstat`.

When you write drivers, declare the variables you need in the device-specific portion of the driver static, the same as in the logical unit. Only the declared variables can be found in `defs.h`. The initialization of the entire driver static structure is discussed in `main.c`.

The SPPROTO driver does not have any device-specific definitions in the driver static structure. In this `defs.h` file, there is only the comment pertaining to where this declaration goes above the logical unit definitions.

history.h

The `history.h` file is principally a file to track the history of the driver and the descriptors. There are two `_asm` instructions in this file. This is where the system attributes and revision status are updated. Type the following instruction to see the system attributes and status:

```
os9ident $ MWOS/OS9/68000/CMDS/BOOTOBJS/SPF/spproto
```

or on an OS-9 for 68K or OS-9 system enter:

```
ident MWOS/OS9/68000/CMDS/BOOTOBJS/SPF/spproto
```

This command lists all vital statistics for the module `spproto`. On your display, look through the `Edition` and `Ty/La At/Rev` lines. Note the same values appear as in the `_asm()` lines of the `history.h` file of `spproto`.

The `_sysattr` definition is especially important. For example, assume you have an old driver in ROM and the `_sysattr` value is 0xA001. You found a bug in this driver and fixed it. Before recompiling, you change the `_sysattr` value to 0xA002. You bring the target system up and it loads the old driver from ROM into memory. Instead of having to reprogram your ROM every time to check the fix, you can load the new driver into memory. OS-9 compares the `_sysattr` values of the driver and uses the one with the highest value. In this case, because you changed the value of your driver to 0xA002, it replaces the old driver. If you did not change it, the values would have been the same and the older module would remain in memory and be used.

The `_sysedit` value allows you to keep track of the driver and descriptor versions in the field, so you should always increment `_sysedit` values before releasing changes. You should also document those changes. This way, if problems occur, you can trace the driver via the `_sysedit` value and your documentation.

If you find two OS-9 modules with the same version number, they may still be different. A quick check to compare the module size and module Cycle Redundancy Check (CRC) using the `os9ident` or `ident` commands shows if they are different versions.

proto.h

This file contains all function prototypes for the driver. Any time a new function is added to the driver, place the function prototype for that function in this header file. The `dr_xxx` prototypes defined in this file are the standard entry points for the driver. The prototypes defined after the `dr_xxx` prototypes are those of locally defined functions in the driver.

main.c

This file contains the initialized values for the driver static storage structure. Note the initialized values are declared for the entire `spf_drstat` structure. They are also declared this way because the driver static structure needs to be declared as a globally accessible structure for the driver.

An interesting part of the `main.c` file is the assembly language code at the bottom. This code defines the `_m_share` field in the module header for portability reasons. When OS-9 sets up the global static for the driver before calling an entry point, it looks at the `_m_share` field to find the driver static structure. This is because the optimizing compiler is allowed to put the static storage anywhere in the code space of the module for optimization reasons. OS-9 for 68K cannot assume the driver static is the first thing after the module header.

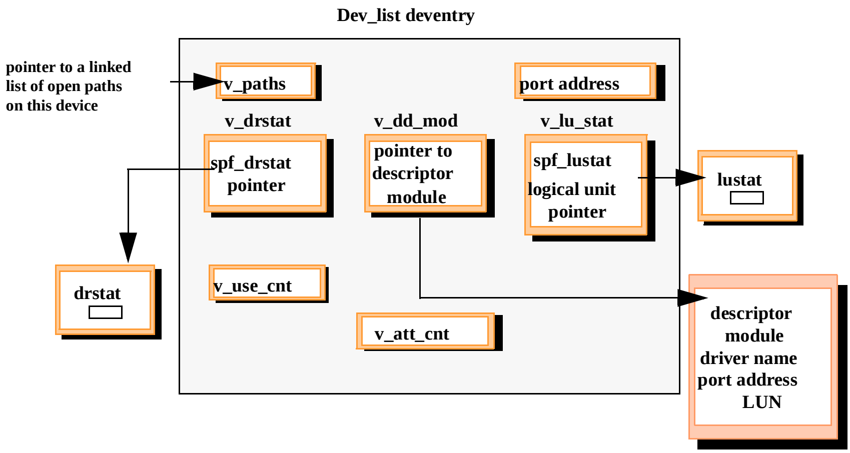
OS Allocated Memory Available for Driver Use

Before discussing the `spproto` template driver source code, let's review the memory structures the operating system allocates and makes available to the driver.

When an application makes an attach call using `/device_name`, the operating system must assign or map a device list entry with the `dev_list` structure to the device name being attached. If the device was previously attached, the attach count in the established `dev_list` structure would be incremented and the pointer returned to the application. If this is the first attach to this device name, a new `dev_list` structure would be allocated, initialized, and passed back to the application.

For every unique device descriptor being used in the system, there is a corresponding `device_list` entry structure. This `device_list` structure uniquely identifies a particular device within the OS-9 system and is therefore the cornerstone structure used by SoftStax I/O systems for stacking drivers. The `updrvr` and `dndrvr` pointers point to the `dev_list` structures of the protocols stacked above and below the current protocol. In every driver entry point, SoftStax passes a pointer (to the `dev_list` structure) to the driver. The following figure shows a graphical representation of the `dev_list` structure and some of the things that can be accessed using it.

Figure 3-1 The Dev_List Structure



Attaching the same device name multiple times causes the `v_att_cnt` in the `dev_list` structure to be incremented.

Opening the same device name multiple times causes the `v_use_cnt` in the `dev_list` structure to be incremented. The path descriptor created as a result of the `open()` call is also added to the `v_paths` list. (The `v_paths` list is generally not used because SoftStax puts the current path descriptor making the call in the `lu_pathdesc` parameter in the logical unit).

Pointers to the driver static and logical unit static structures are also found in the `dev_list` structure.

Two devices in the system have the same driver static if the driver names are the same *and* the port addresses are the same.

Two devices in the system have the same logical unit static if the driver names *and* port addresses *and* logical unit numbers are the same.

Allocation Example

Table 3-1 Devices

/a	/b	/c	/d
drvr_y	drvr_y	drvr_y	drvr_y
portaddr 10	portaddr 20	portaddr 20	portaddr 20
LUN 1	LUN 1	LUN 2	LUN 2
DESC 1	DESC 1	DESC 1	DESC 2

Table 3-2 Device Descriptors

Steps	Dev_list	driver static	logical unit static
attach /a:-->	DEVLIST1	DRVRSTAT1	LUSTAT1
attach /b:-->	DEVLIST2	DRVRSTAT2	LUSTAT2
attach /c:-->	DEVLIST3	DRVRSTAT2	LUSTAT3
attach /d:-->	DEVLIST4	DRVRSTAT2	LUSTAT3

Since `/a` is the first attach, unique storage is created for all three structures.

When `/b` is attached, the port address is different, so unique driver and logical unit static is created and a new `dev_list` structure is used.

Now /c gets attached. /c has the same port address but a different logical unit number than device /b, so a different logical unit static and dev_list is created, but the driver static is the same as device /b.

Then /d gets attached. /d has the same port address and the same logical unit number but a different descriptor number than device /c, so a different dev_list is created, but the driver static and the logical unit static are the same as device /c.

Allocating Per Path Storage for the Driver

The driver and logical unit static storage areas tend to be used for board and chip level storage areas. But what about unique storage for each path using the driver? Historically, this has been an easy answer: the path descriptor.

The OS allocates a unique path descriptor for each path opened in the system. However, SoftStax is the only OS-9 I/O system allowing multiple drivers to be stacked on the same path. Before, with a 1-to-1 correspondence between the path and driver, you just reserve storage for the driver in the path descriptor. But with the 1-to-n correspondence SoftStax has between a path and the driver stack being used, this is not possible. Drivers that de-multiplex up or down the stack need to keep per path static storage.

The initial structure for the per path storage is found in spf.h as the structure spf_ppstat. Notice this structure defines the SPF_PPSTAT, just like the logical unit and driver static structures, so each driver can customize the per path storage to their requirements.

When the driver is called at the SPF_SS_OPEN, SPF_SS_PUSH, and SPF_SS_POP setstats, the lu_updrvr and lu_dndrvr variables in the logical unit are correct, that is, they point to the drivers dev_list for this path's stack directly above and below this driver respectively. However, subsequent opens to the same device may cause the variables in the logical unit to change. It is for this reason, at SPF_SS_OPEN, SPF_SS_PUSH, and SPF_SS_POP time, lu_updrvr, lu_dndrvr, and lu_pathdesc should be stored in the per path storage area.

Allocation Example

There are three protocol drivers in a system, Internet Protocol (IP), User Data Packet (UDP) and Transmission Control Protocol (TCP). Path #1 opens TCP/IP. When IP gets called at `SPF_SS_OPEN`, IP allocates per path storage for path #1. IP stores the path descriptor for path #1, the pointer to the driver below `(lu_dndrvr = NULL)`, and the pointer to the driver above `(lu_updrvr = path #1)`. Then IP gets called at `SPF_SS_PUSH` time (pushing TCP). IP sets the new `updrvr` variable in the per path storage for path #1 to the new driver above (TCP). TCP also gets called at open and allocates per path storage for path #1.

At this point, IP has one per path storage element in the per path storage list as does TCP. The IP de-multiplexes incoming packets based on the protocol type in the incoming packet and the protocols above it on all the paths. TCP de-multiplexes incoming packets based on the destination port/IP address matching the local socket address in the path's `our_num addr_type` structure.

Now, path #2 opens /UDP/IP. The IP gets called at `SPF_SS_OPEN`. This time the IP's `lu_updrvr` points to the `dev_list` for path #2, overwriting the previous pointer which pointed to TCP's `dev_list`. This works since you have already stored the previous `lu_updrvr` in the per path storage for path #1. IP creates a new per path storage structure for path #2 and stores the path descriptor for path #2, the `lu_updrvr` and `lu_dndrvr` in the new per path storage. IP gets called at `PUSH` for path #2 (pushing UDP). At this point, IP stores the new `lu_updrvr` which points to UDPs `dev_list` in path #2's per path storage area. UDP also gets called at open and creates its own per path storage for path #2.

IP has two per path storage areas: one for path #1 (TCP) and one for path #2 (UDP). When a packet is received, IP goes through the per path storages and matches the protocol types in the packet with the same protocol type in the per path storage list. If the packet was type TCP, it would match path #1 storage and use the `updrvr` pointer in the per path storage to send the packet to TCP. TCP compares the destination port and IP address with the socket address of each path to pass the incoming packet up to the right path.

entry.c

The `entry.c` file contains all entry points called in the driver by the file manager. The `spproto` source code is well documented so as you write drivers, the comments tell you under what conditions this entry point gets called by the file manager or upper/lower layer drivers, as well as what to watch out for within particular entry points. This section details each entry point of `spproto`. Then, interrupt service routine conventions for the *mw*SoftStax hardware drivers are covered.

For More Information

Refer to Chapter 2: Creating SoftStax Drivers for more information about driver entry points.

dr_iniz()

The `dr_iniz` entry point allows the driver to set up and initialize anything that must be available when an `_os_attach()` or `ite_dev_attach()` is performed at the application level. Typically, the driver installs the interrupt service routine for the driver in this entry point. Because the driver static and logical unit static are already initialized by the operating system during the attach, there is no need for the driver to do it again in the `dr_iniz()` entry point unless there are relocatable pointers that must be stored dynamically at attach time.

One of these relocatable pointers might be for MPEG-2 devices that talk to the `duxman` module. These drivers would attach to the `duxman` module and get a pointer stored in an appropriate variable in the driver static. Note `DEBUG` is defined and a debug data module `dbg_proto` is being created.

SoftStax calls the `dr_iniz()` entry point only when it is the first to attach to a particular logical unit for the driver (`lu_att_cnt = 1`). SoftStax pre-increments the `lu_att_cnt` and the `dr_att_cnt` variables, so the respective attach counts include the current `attach` being processed.

dr_term()

This entry point should undo whatever was done in the `dr_iniz()` entry point. If an interrupt service routine was installed, it should be de-installed here. If linking and registration with the `duxman` module happened during `dr_iniz()`, unregister the device with `duxman` at this time. The only exception is you do not get rid of the debug data module at this point, as you might want to look at it after termination.

SPF calls the `dr_term()` entry point only when it is the last detach to a particular logical unit for the driver (the `lu_att_cnt = 0`). SPF pre-decrements the `lu_att_cnt` and `dr_att_cnt` variables, so the respective attach counts are minus the one being currently processed.

dr_getstat()

This entry point should support the following standard getstats listed, as well as any specific functionality.

For More Information

Implementing driver-specific getstats and generating the library binding is described in the `os_lib` API section ***Using the Parameter Block in Setstat/Getstat Calls*** in ***Using SoftStax***.

SPF_GS_DEVENTRY

For this getstat, the driver places the `deventry` pointer into the `param` field of the `spf_ss_pb` as shown in spproto's `entry.c`.

SPF_GS_PROTID

As with IP, some protocols demultiplex based on the protocol types above them. `spf.h` defines all the protocols Microware supports. `prot_ids.h` can be used to add user specific protocol types. This `getstat` should return your protocol type in the `param` field of the `spf_ss_pb` structure (all defined protocol identifiers are prefixed by `SPF_PR_xxx`).

SPF_SS_UPDATE

For increased efficiency, protocols are expected to pass certain parameters to SPF so it can act intelligently when performing I/O. These parameters are:

- Maximum Transmission Unit (MTU). This is saved as `lu_txsize` in each logical unit of the driver.
- Transmit Offset. States how much room at the front of each transmit packet the protocol needs for its header (saved as `lu_txoffset` in the logical unit of each driver).
- Transmit Trailer. Specifies how much room the protocol needs for its trailer at the end of each transmit packet (saved as `lu.txtrailer`).
- I/O enabled. Saved as `lu_ioenabled` in the logical unit of each driver.

Whenever the protocol stack for a given path changes, SPF performs an `SPF_SS_UPDATE` getstat to gather the updated parameters. For example, when an `SPF_SS_PUSH` or `SPF_SS_POP` is successfully executed, SPF performs an `SPF_SS_UPDATE` getstat to gather the new I/O information for the stack.

When a driver is called at this entry point, this is the expected processing:

Step 1. Call the lower driver with the same getstat (if there is a lower driver).

Step 2. Fill out the `spf_update_pb` passed into the getstat. If the driver is at the bottom, it needs to fill the following fields:

- `stk_txsize` field in the `spf_update_pb` with the driver's MTU.
- `stk_txoffset` with the number of bytes the driver needs to add to the header.
- `stk_txtrailer` with the number of bytes the driver needs to add to the end.
- `stk_ioenabled`, based on whether the driver is able to send and receive data for this path.

If the driver is not the bottom driver on the stack, it must fill out the `spf_update_pb` per the following parameter requirements.

stk_txsize Parameter

This is the most complex of the parameters. If your driver does not have segmentation (fragmentation) and re-assembly capabilities, your `lu_txsize` is the MTU of your protocol. For example, LAP-B has no fragment/reassemble capabilities and its MTU is 4096 bytes. Most hardware drivers do not have fragment/re-assemble capabilities. Therefore, the algorithm for setting the `stk_txsize` field is to fill it with the smallest value between your `lu_txsize` field and the `stk_txsize` field passed up when you did the getstat. SPF automatically fragments packets for this stack so the MTU is never exceeded.

There is, however, one note of qualification. If the application using SPF is itself a protocol and it sends down a packet bigger than the stack's MTU, SoftStax fragments that packet. It then shows up on the other end as multiple packets and is not reassembled into the one original packet sent. For applications doing byte count reads, this is not a problem. But if the application needs to do packet-by-packet reads, this can cause a problem. In this case, the application must ensure the protocol stack can support its MTU or make sure there is a protocol on the stack supporting segmentation and reassembly.

If your driver does support segmentation and re-assembly (such as X.25), your MTU effectively becomes `0xFFFF`, which causes added responsibilities. You always send up a `stk_txsize` of `0xFFFF`, but when a packet gets passed to you, remember the MTU of the stack below you and fragment the packet to protect it from getting packets that are too large.

stk_txoffset Parameter

Add your header requirements to the `stk_txoffset` field passed back to you by the getstat (to the lower protocol). When it gets passed up to SPF, it is stored with the path, and when writes occur, SPF adds the appropriate header size to reserve room for the entire stack's header requirements.

stk_txtrailer Parameter

This parameter is used to add your trailer requirements to the `stk_txtrailer` field passed back by the getstat (to the lower protocol). When it gets passed up to SPF, it is stored with the path when writes occur. SPF adds the trailer size for packet allocation so there is room for the protocol's trailer bytes.

stk_ioenabled Parameter

This parameter indicates whether reads and writes can be performed on this stack. If your driver's I/O is not yet established, it makes no difference what the stack is below. `stk_ioenabled` is `DRVR_IODIS` (or I/O disabled). Otherwise, if your protocol is enabled, `stk_ioenabled` becomes the value of the lower driver's `lu_ioenabled` field.

Note

DRVR_IOEN means the protocol is successfully communicating with its peer. It does not imply a connection has been established.

SPF_SS_UPDATE is initiated by SPF on any change in the protocol stack for a given path. It bubbles all the way up to SoftStax for storage in that specific path static. If the lu_ioenabled status changes for any driver, that driver should use the SPF_SS_UPDATE setstat to call and notify SPF of the change in the driver's I/O status.

Unknown Codes

The convention for unknown codes is to find out whether the getstat is going up or down the stack. To do this, use the updir field in spf_ss_pb. If the getstat is going up the stack, pass the data up via the SMCALL_GS macro in spf.h. If the getstat is going down the stack, check if you have a lower driver. If you do not, return EOS_UNKSVC. If you do, pass the getstat down to the lower driver and return the results transparently.

dr_setstat()

The dr_setstat entry point is where all call control requests come through to be processed. Some setstat codes must be supported by all drivers. Other setstat codes are only supported by the drivers implementing the code's functionality.

For example, ITEM supports many call control setstats (ITE_DIAL, ITE_ANSWER, and ITE_HANGUP). All of these codes represent layer three, or network layer services in the OSI model. Therefore, the protocol driver taking care of the layer three functionality should incorporate these setstats.

Most hardware drivers are only concerned with sending and receiving packets. In this case, they would not support the previous call control setstats.

The following section discusses which codes must be supported by all drivers. Then, discussion continues concerning the rest according to which layer protocol would typically support the particular setstat.

Setstat Codes That Must be Supported by All Drivers

SPF_SS_OPEN

FROM: `_os_open()` or `ite_path_open()`

SPF calls the driver every time a new path opens using this driver. It is usually at this point, if not already established, the protocol driver attempts to communicate with its peer and initialize the link. If you are writing a protocol driver, be aware of the layer level at which your protocol is used. In general, a hardware driver does most of its own setup and initialization at `dr_iniz()` time.

Protocol drivers at the second layer level do most of the initialization during the `SPF_SS_OPEN` setstat. For every path it translates into initializing a new data link layer entity. Q.931 is like this, where for every open path in the system, a *Terminal Endpoint* number is assigned by the network. In this case, this assignment would happen at `SPF_SS_OPEN` time.

Data link layer protocols having a unique initialization for every port (such as LAP-B) initialize at `dr_iniz()` time for each new logical unit.

Layer three protocols typically keep a table of the path descriptor and a potential connection. For example, X.25 keeps a table of all the paths. When connections are made on those paths, the connection information is kept in the table with the path descriptor. In this way, when X.25 receives data over a given connection, it knows for which path the data is destined.

SPF_SS_CLOSE

FROM: `_os_close()` or `ite_path_close()`

Close allows the driver to undo, for the closing path, whatever was done in the `SPF_SS_OPEN` setstat. SPF calls this setstat only if this is the last process to close the path.

SPF_SS_PUSH

FROM: `ite_path_push()`

This setstat is called by the file manager to notify the driver currently on the top of the stack that a driver is getting pushed onto the top of the stack. The file manager takes care of setting up the links to the new protocol on the path. Some protocols might want to integrity check the protocol being pushed by calling up the stack for its protocol ID. In most cases, the protocol drivers do not need to do anything.

SPF_SS_POP

FROM: `ite_path_pop()`

This setstat is called by the file manager to notify the driver (the new top of the stack after the pop is completed) that the driver above it is being popped off. Again, in most cases the driver being called does nothing and returns successfully.

Codes Implemented Only by Drivers With Flow Control Ability

SPF_SS_FLOWOFF / SPF_SS_FLOWON

SPF implements a backflow mechanism when receiving data to prevent receive buffer overflow. This backflow method is governed by the `pd_readsz` field in the path options section of the path descriptor. This variable, when non-zero, tells SPF when to tell the lower drivers to stop sending received packets, thus creating data backflow. A protocol able to tell its peer to stop sending data should generate this packet when the SPF_SS_FLOWON setstat code is received. Conversely, when the SPF_SS_FLOWOFF setstat is received, the packet telling the peer to start sending data again should be generated.

For example, LAP-B has as one of its services, flow control via the receiver ready (RR) and "receiver not ready" (RNR) messages. When SPF_SS_FLOWON is received by splapb, it sends the RNR message to its peer and the peer's protocol driver stops sending data. Then, when the application has read less than the number of bytes specified by the pd_readsz field's worth of data, SPF sends the SPF_SS_FLOWOFF setstat. This causes splapb to send the RR message to its peer. The peer LAP-B protocol then starts sending data again. A pd_readsz of zero causes SPF to never send the flow control setstats.

Codes Implemented Only by Network Layer Protocol Drivers

ITE_DIAL

FROM: ite_ctl_connect()

CAUSE: An ite_ctl_connect() call was made by the application.

EXPECTED SERVICE: This setstat causes the network layer protocol driver to initiate connection setup between two midpoints. The protocol driver should use either the our_num/their_num addresses passed into the setstat (if non-NULL) to establish the connection, or use the our_num/their_num values in the conn_info structure of the path descriptor's path_type structure in item_pvt.h if either or both addresses are passed in as NULL.

ITE_ANSWER

FROM: ite_ctl_answer()

CAUSE: The application was notified of an incoming call and is expected to answer it.

EXPECTED SERVICE: The protocol driver communicates with the network to create an active connection. After the protocol driver initiates the call-answering procedure, it returns. The caller passes a `notify_type` pointer in the `param` field of the parameter block. When the network confirms the connection, the driver performs the proper notification. If the parameter field is `NULL`, the caller needs to call the `ite_ctl_connstat()` to see if the connection is active.

ITE_HANGUP

FROM: `ite_ctl_disconnect()`

CAUSE: The application needs to disconnect the active connection.

EXPECTED SERVICE: The protocol driver initiates call termination procedures and returns.

ITE_FEHANGUP_ASGN / ITE_FEHANGUP_RMV

FROM: `ite_fehangup_asgn()`/`ite_fehangup_rem()`

CAUSE: The application needs to be notified when the far-end hangs up. The caller assumes there is an active connection when this call is made.

EXPECTED SERVICE: SPF takes care of inserting all notification requests into the notification array. The driver is expected to send notification if the far-end initiates hang-up procedures. If this is a removal, SPF takes care of removing the request from the notification list. The driver is expected to clear the local storage. Therefore, if far-end hang-up occurs, no notification is sent.

ITE_RCVR_ASGN / ITE_RCVR_RMV

FROM: `ite_rcvrasgn()`/`ite_rcvrrem()`

CAUSE: The application needs to be notified when and if an incoming call occurs.

EXPECTED SERVICE: SPF takes care of inserting all notification requests into the notification array. The protocol driver is expected to send the notification if there is an incoming call. Also, the protocol driver needs to store the `device_type` structure for this notification. If the `dev_rcvr_state = ITE_ASGN_ANY`, notification is always performed.

If `dev_rcvr_state = ITE_ASGN_THEIRNUM`, notification is performed only if the calling address matches the `dev_theirnum` address for the path to be notified. If this is a removal, SPF takes care of removing the request from the notification list and the protocol driver is expected to clear local storage. If an incoming call occurs, no notification is sent.

With connectionless protocol drivers, the convention is call control setstats cause connectionless protocols to return an `EOS_UNKSVC` error to the caller.

Unknown Codes

The convention for unknown codes is to use the `updir` field in the `spf_ss_pb` to determine if this setstat is going up or down the stack.

- If the setstat is going up, you pass the data up via the `SMCALL_SS` macro in `spf.h`.
- If the setstat is going down, first check if you have a lower protocol driver in the stack. If you do not, return `EOS_UNKSVC`. If you do, pass the setstat down to the lower protocol driver and return the results transparently.

dr_updata()

This entry point is called by the lower-layer protocol driver when incoming data is received. Hardware drivers never get called by the `dr_updata()` routine because their interrupt service routine is what is used to receive the incoming data. Think of this entry point as a protocol driver’s interrupt service routine. During this entry point, deal with the protocol header of the incoming data packet and send responses to the `dr_downdata()` entry point of the lower layer protocol as needed.

SPF_FMCALLUP_PKT / SMCALL_UPDATA

The `spf.h` file defines two macros: `SPF_FMCALLUP_PKT` and `SMCALL_UPDATA`. The first macro calls the packet call-up function in SPF while the second calls the upper-layer protocol directly.

Typically, receive packet data flow begins when the interrupt service routine receives one complete packet. At this point, the driver is running in the interrupt service routine. While the driver is executing code in this routine, interrupts are masked to the level of the hardware driver. Therefore, the processing in the interrupt service routine context should be as short as possible.

If the receive interrupt service routine called `SMCALL_UPDATA()` directly, the higher layer protocol(s) process the incoming packet and send out any response messages in an interrupt service routine context. This makes the possibility of missing the next received packet quite high and results in poor system performance.

To handle this, SPF starts an independent system-state receive process when the first SPF path open occurs. Then, when the interrupt service routine receives a complete packet and the `FMCALLUP_PKT` macro is called, SPF queues the receive packet and sets an event for the receive thread process to wake up and send the packet to the next protocol above the hardware driver. The only data packet reception happens on interrupt service routine context. All subsequent processing of the packet occurs on SPF receive thread process context. At this point, after the next higher protocol receives the packet, it uses the `SMCALL_UPDATA` macro to pass the packet up the chain.

Protocol drivers also need to know the SPF receive process is a system-state process. Since SoftStax does not allow system state time slicing, this thread is not time sliced. Therefore, protocol drivers should never sleep in their `dr_updata` entry point routines. If the drivers are sleeping on the receive thread, all receive packets stop being processed.

dr_downdata()

This entry point is typically straight-forward for drivers and encompasses the following procedure:

Step 1. Receive an mbuf to be transmitted.

Step 2. Back off the `m_offset` field by the number of bytes the newly added header takes.

Step 3. Add your header and trailer as needed.

Step 4. Increment the `m_size` field of the mbuf by the number of bytes the driver added for the new header.

Step 5. Store the mbuf on a retransmission queue if the protocol driver implements retransmission.

Step 6. Set the `SPF_NOFREE` bit (if your protocol implements retransmission).

Step 7. Send to the next lower driver in the stack.

Note

If a driver sets the `SPF_NOFREE` bit on an mbuf, it is expected to clear it before freeing the mbuf. If you call `m_free()` with an mbuf that has the `SPF_NOFREE` bit set, `m_free()` simply sets the `SPF_DONE` bit in the mbuf and returns.

Driver Interrupt Service Routine Conventions

This section describes situations to watch for when writing an interrupt service routine.

Writing and Installing the Interrupt Service Routine (ISR)

Write your interrupt service routine and then install it in the OS-9 interrupt table. The following code segment shows an example function prototype for the interrupt service routine and a conditionalized definition:

```
error_code hw_isr(Dev_list dev_entry);

#if defined(_OS9000)
   #define HW_ISR hw_isr
               /* OS-9000 interrupt service routine */
#elif defined(_OSK)
   extern void hw_isr_os9();
   #define HW_ISR hw_isr_os9
               /* OS-9 interrupt service routine */
#endif
```

Defining a Macro as an Interrupt Service Routine

Defining a macro as an interrupt service routine allows your driver to be source code compatible across processors. Notice the real name of the interrupt service routine is `hw_isr()`. However, when running under OS-9 for 68K processor family, there is assembly language code converting OS-9 for 68K interrupt service routine conventions to the OS-9 interrupt service routine conventions. This code is labeled `hw_isr_os9`.

As shown in the above segment, when `_OS9000` is defined, the name of the ISR is `hw_isr()` only because the `hw_isr_os9` code segment is not needed. (The compiler automatically defines `_OS9000` when compiling for 80X86 and the OS-9 operating system. `_OSK` is defined by the compiler when compiling for the 68XXX family.)

Installing the ISR

This section describes the interrupt service routine installation found in the `dr_iniz()` entry point.

```
if ((err = _os_irq(lustat->lu_vector,
lustat->lu_priority, HW_ISR, dev_entry)) != SUCCESS)
{
   return(err);
}
```

The interrupt vector and priority can be found in the logical unit structure and are initialized by the device descriptor. The `HW_ISR` macro is in the `_os_irq()` call. The correct name gets resolved at compile time, so there is portability across processors. The last parameter in the `_os_irq()` call is `device_entry`. This parameter gets passed to the protocol driver's interrupt service routine when it runs. However, depending on the driver, you could pass pointers to other structures. Pass whatever is most useful for the interrupt service routine to have when executing.

Note

If there are two logical units of a driver with two different device entries sharing the same interrupt service routine, then it is better to install the interrupt service routine with the driver static and store both device entries in the driver static.

For More Information

For more information on the `_os_irq()` call, see the ***Ultra C Library Reference Manual***. This manual says `_os_irq()` is only an OS-9000 call. The `conv_lib.l` has created a binding to make this call valid for OS-9 also.

OS-9 Interrupt Service Routine Glue Code

This section describes the OS-9 interrupt service routine glue code. Glue code is code inserted into the driver to ensure compatibility between OS-9 for 68K and OS-9 systems so the driver can run in either or both operating systems without any further source code changes.

The following example is the interrupt service routine glue code included for OS-9. It ensures the `hw_isr()` routine is properly called. Do not modify this call unless you are using other labels for your functions.

```
#if defined(_OSK)
/* interrupt service routine glue-code for OS-9 */
_asm("hw_isr_os9:
   move.l      a2,d0;   /* put Dev_list in a2 into d0*/
   bsr         hw_is;   /* call interrupt service*/
         /*routine*/
   tst.l       d0;      /*see if SUCCESS returned*/
   beq.s       hw_isr_os9_exit;
         /*if so, return*/
   ori         #Carry,ccr;
         /* else E_NOTME returned */
         /*--set carry bit*/
hw_isr_os9_exit
   rts
");
#endif
```

Data Transmission Conventions

Every interrupt service routine has its own method by which to transmit data from its transmit queue. However, when packet transmission is complete, the interrupt service routine should use the `m_free()` function to free the transmitted packet. The following code segment comes out of `hw.c` for the `sp8530` driver. You can find the source in `/MWOS/SRC/DPIO/SPF/DRVR/Z8530`:

```
/* see if we just transmitted the last character in this packet */
if (lustat->lu_tx_left == 0) {
   Mbuf mb = lustat->lu_tx_hd;

#if defined(DEBUG)
   debug_data(lustat->dbg_ptr,”TxLastByte “,0);
```

```
#endif
   WR_8530(control->channel, 10,
      control->encoding|CRC_INIT_1|IDLE)
      control->new_packet = 2;

   /* change to next mbuf packet chain -return old one */
   lustat->lu_tx_hd = lustat->lu_tx_mb = mb->m_qnext;
   lustat->lu_tx_out = mtod(lustat->lu_tx_mb, u_char*);
   lustat->lu_tx_left = lustat->lu_tx_mb->m_size;
   mb->m_qnext = NULL;
   m_free( mb);
}
```

This code segment checked to see if the last character in the packet had been sent, and if so, manipulated the local pointers to point to the next packet if there was one, and called `m_free()` to free the transmitted mbuf.

If there is a protocol driver (driver one) above the protocol driver (driver two) implementing retransmission, driver one should keep a pointer to the just released mbuf until its peer acknowledges with a successful packet reception. Problems result if the peer does not acknowledge the mbuf. The higher layer protocol (driver one) attempts to use the pointer to the mbuf already freed by the code segment to re-send the packet.

The return `m_free()` checks for the `SPF_NOFREE` bit in the mbuf, which identifies whether a higher layer protocol is holding a pointer to this mbuf for possible retransmission. If this is the case, `m_free()` sets the `SPF_DONE` flag to indicate the driver has transmitted the packet. If the `SPF_NOFREE` bit is not set, `m_free()` returns the mbuf to the free pool.

Data Reception Conventions

`misc.c` provides a function to return mbufs to the pool. It also provides a function to get mbufs from the pool. Control information is embedded in the received mbuf to allow the SoftStax receive thread to pass the packet up the stack correctly. The `misc.c` file `get_mbuf()` call inserts the device entry pointer into the receive packet before it returns the packet to the caller.

The following code segment from the source of the sp8530 driver illustrates this use:

```
/* get head of receive packet chain */
if (get_mbuf(lustat->lu_updrvr, lustat->lu_rx_pktsz,
   &lustat->lu_rx_mb) != SUCCESS)
{
   /* return with lost-data error */
   lustat->lu_rx_err = RXERR_MBUF;
   return(-1);
}
```

The code segment shown is from the sp8530 source in `DRVR/Z8530`. This code segment checks to see if the head of the receive queue is `NULL`. If so, it gets an mbuf by executing the `get_mbuf()` call. The parameters to this call are as follows:

- the pointer to the device entry of the driver above (`lustat->lu_updrvr`)
- the size of the packet payload to allocate (here it is stored in the `lu_rx_pktsz` field)
- a pointer to where the returned mbuf pointer is stored.

ISR Summary

This concludes the discussion of the interrupt service routine conventions. The three main points to remember include the following:

- use the `hw_isr` macro and the `hw_isr_os9` glue code for portability across processors (if needed)
- use the `return_mbuf()` function provided in `misc.c` to free transmitted `mbufs`
- use the `get_mbuf()` function provided in `misc.c` to get mbufs from the `mbuf` pool

Chapter 4: SPLOOP Driver

This chapter details the SPLOOP driver which provides SoftStax with the following functionality:

- communicate standard Input/Output (I/O) data loopback over a single path
- connection-oriented network data loopback over a single path
- connectionless network emulation enabling applications to test their ability to run over connectionless networks
- connection-oriented network emulation enabling applications to test their ability to run over connection oriented networks

RadiSys.
MICROWARE SOFTWARE

Introduction to the sploop Driver

When writing applications, it is extremely important to understand the power and functionality of the SPLOOP driver. Because the ITEM (Integrated Telecommunications Environment for Multimedia) interface provides network independence, this driver allows you to write and test applications on a local OS-9 machine, just as if the applications were running over a real network. Examples 1 and 3 in Chapter 1 use SPLOOP to show the functionality of the ITEM API.

For More Information

Refer to ***Appendix A: Examples*** in ***Using SoftStax*** for more information about these examples.

The sploop driver can act as a connectionless network emulator, a connection-oriented emulator, or a straight loopback driver. The operation is determined by the logical unit number in the device descriptor.

The sploop driver keeps a 10-element array and divides this array into two parts. If you open a device with a logical unit from number 0 through 4, you have opened a connection-oriented path. If the device open has a logical unit number from 5 through 9, you have opened a connectionless device. If you open a device with a logical unit number greater than 9, you have opened a loopback device.

The descriptors provided with the software package indicate these types. The loop descriptor has a logical unit of 0x7F. Therefore, any paths opened using this device are loopback. `loopc0` and `loopc1` are connection-oriented device descriptors using logical units 0 and 1 respectively. `loopcl5` and `loopcl6` are connectionless devices using logical units 5 and 6.

Addressing

In the device descriptors for this driver, the address class and subclass parameters are `ITE_ADCL_LPBK` and `ITE_ADSUB_LUN` respectively. This means the SPLOOP driver has its own address class to emulate the addressing, and class uses logical unit numbers for individual addresses. The SPLOOP driver interprets the first character in the `addr[]` array of the `addr_type` structure as the logical unit.

For More Information

Refer to Chapter 2: Creating SoftStax Drivers for more information about the `addr_type` structure.

For example, if you put a 1 in the `dev_theirnum.addr[0]` field and executed an `ite_ctl_connect()`, the driver attempts to connect to the process with the open path to logical unit number 1.

Restrictions

Connectionless paths cannot send messages to connection-oriented paths. Likewise, connection-oriented paths cannot attempt to connect to connectionless paths. Loopback paths obviously only communicate after that single loopback path.

For More Information

For details on how applications use `sploop`, refer to Chapter 1: Getting Started and review Examples 1 and 3. These examples show how `sploop` emulates various environments.

For additional details on using SPLOOP for testing, refer to ***Chapter 8: Using SPLOOP to Test Applications and Protocols*** in ***Using SoftStax***.

For additonal details on the examples, refer to ***Appendix A: Examples*** in ***Using SoftStax***.

Chapter 5: sp8530 Device Driver

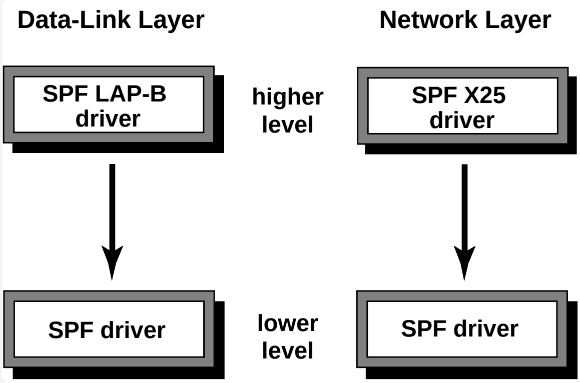
This chapter describes the sp8530 device driver provided for use with the Zilog Z85C30 SCC (Serial Communications Controller) on the Motorola MVME147 monoboard computers and on the MATRIX Corporation MS-SIO4A VME boards for controlling serial ports. The Z85C30 SCC handles the synchronous bit-oriented protocols SDLC and HDLC, synchronous byte oriented protocols, and asynchronous formats.

Introduction to sp8530 Device Driver

The sp8530 device driver is different from other Microware Z85C30 drivers as it uses the Z85C30 in SDLC mode, not in asynchronous mode.

It is the physical-layer SoftStax driver sitting below the SoftStax LAP-B driver in the data-link layer and the SoftStax X.25 driver in the network layer.

Figure 5-1 SPF Driver Layers



The sp8530 driver makes use of the following Z85C30 SDLC capabilities:

- **Automatic flag insertion between messages.**
  The flag character has the bit pattern 0x7E. This character is only transmitted between messages.
- **Abort sequence generation and checking.**
  The abort sequence is 7 to 13 consecutive ones (for example, 1111111). The transmitting Z85C30 generates an abort sequence after an underrun error condition is detected and then retransmits the message from the start.

Upon receiving an abort sequence, the receiving Z85C30 discards the partially received message before the abort sequence and expects the message to be retransmitted.

- **Automatic zero insertion and detection.**
  The flag character is guaranteed to be transmitted only between messages, that is, message payloads will not contain character 0x7E.

The transmitting Z85C30 guarantees this by watching the payload and inserting a 0 after all strings of five consecutive ones (11111). The receiving Z85C30 watches the receive stream for strings of five consecutive ones. If the next bit is a 0, it is deleted. If the next bit is a 1, the six consecutive ones are not recognized as data but as part of either a flag or abort sequence.

- **CRC generation, detection, and checking.**
  At the end of a message the transmitter appends a 16 bit CRC before sending the flag character marking the end of the current message and the beginning of the next message (if there is one).

After receiving the flag, the receiver uses the previous 16 bits as the CRC and checks the entire message against this CRC.

For More Information

See the *SCC Users Manual* for a complete description of the Zilog SCC hardware. It is available from this address:

Zilog, Inc.
Campbell, CA 95008-6600
Telephone 408-370-8000
Fax 408-370-8056

---

sp8530 Entry Points

The sp8530 device driver has the following SoftStax entry points.

dr_iniz()

This function is entered only if no other device descriptors are attached to the sp8530 driver. First, `dr_iniz()` installs the hardware interrupt service routine (ISR). Next, this function initializes the 15 Z85C30 write registers with data from the device descriptors logical unit specific static storage structure. Then, `dr_iniz()` sends commands to initialize the Z85C30 for operations.

dr_term()

This function removes the installed ISR (Interrupt Service Routines) and disables interrupts from the Z85C30.

dr_getstat()

All the SoftStax drivers have the following entry point:

SPF_SS_UPDATE

This is the lowest (device) level driver. This function only fills certain variables into the parameter block passed to it and returns.

dr_setstat()

This entry point handles the following three setstat subcodes:

SPF_SS_OPEN

Calls the adjacent upper-layer protocol at its `dr_setstat` with subcode SPF_SS_UPDATE to indicate the sp8530 driver is ready for I/O.

SPF_SS_CLOSE

Returns the device list entry of this driver's adjacent lower-layer protocol. Since this is a device driver at the lowest level, the `NULL` pointer is returned.

SPF_SS_NEWTOP

Changes this driver's adjacent upper-layer protocol driver and then calls the `dr_setstat` entry of this new upper-layer driver with subcode `SPF_SS_UPDATE` to indicate a successful change.

dr_downdata()

This function initiates transmission of an mbuf by calling `hw_tx_handler()` to place the first data byte of the mbuf on the Z85C30 output FIFO or by placing the mbuf in an existing queue of mbufs to be transmitted.

The transmission of this first byte causes Z85C30 to generate a transmit buffer empty interrupt. The Z85C30 ISR then calls `hw_tx_handler()` again to place the second byte on the output FIFO.

After the last byte of the mbuf has been placed on the output FIFO, the last call to `hw_tx_handler()` causes the CRC and flag to be transmitted.

dr_updata()

This function is included for compatibility with SoftStax. Since sp8530 is a device driver, no lower-layer driver uses this entry point to send data up the protocol stack.

The ISR for Z85C30 is included with the driver code.

Z85C30 ISR

The ISR handles the following interrupts from the Z85C30:

- Transmit Interrupts
- Receive Interrupts
- External/Status Interrupts

Transmit Interrupts

Transmit Buffer Empty

This interrupt occurs when the transmit buffer becomes empty. A byte must first have been placed in this buffer before it can become empty.

The ISR responds to the interrupt by:

Step 1. Writing the next byte to be transmitted to the output FIFO, if there is another data byte in the mbuf.

Step 2. Disabling this interrupt until the 16 bit CRC and flag have been transmitted, only if the last byte of the mbuf was written to the output FIFO.

Receive Interrupts

Receive Character Available

The ISR calls `hw_rx_handler()` to write the received byte into an mbuf.

End of Frame (SDLC)

The ISR checks for reception errors. If any are found, the mbuf is returned to the pool. If there are no errors, the mbuf is passed up to SoftStax `dr_fmcallup()` entry point which signals SoftStax receive thread.

The receive thread passes the mbuf to SoftStax `fm_rx()` entry point, which passes it to the next higher protocol driver by finding the device entry of this driver embedded in the mbuf (preceding the data).

External/Status interrupts

TxUnderrun/EOM

This interrupt occurs when a new data byte has not been written to the output FIFO as expected `(TxUnderrun)`. If the last byte has been transmitted, Z85C30 sends the CRC and flag. If bytes remain to be transmitted, Z85C30 sends the abort sequence to the receiving Z85C30.

The ISR resends the entire mbuf after sending the abort sequence.

Break/Abort

This interrupt is caused by receiving the abort sequence. The transmitter sends this when a transmit underrun occurs and more bytes remain to be transmitted.

The receiving ISR discards any bytes received for the aborted message by setting the current mbuf data pointer back to the start of the data area and reducing the count of bytes received by the number of bytes being discarded in the aborted message.

Chapter 6: SP82525 Driver

This chapter describes the sp82525 driver that controls the 82525 Dual Channel HDLC controller manufactured by Siemens.

Introduction to sp82525 Driver

The sp82525 driver is an OS-9 module that controls the 82525 Dual Channel HDLC controller produced by Siemens. This chip has the capability of controlling two independent HDLC or SDLC channels simultaneously. It also has built in capability to do some limited HDLC address detection, and CRC generation and checking as well as packet framing.

Data Reception and Transmission Characteristics

The sp82525 driver does not keep per path storage. The driver allows one protocol to be pushed on top per channel. If many paths open the same protocol stack on the sp82525, it is allowed. However, if there is already a driver stacked on the sp82525 and another path attempts to push a different driver on the same channel, the sp82525 driver returns an `EOS_DEVBSY` error.

Default Descriptor Values for sp82525

ITEM Addressing

The ITEM (Integrated Telecommunications Environment for Multimedia) addressing for this driver is specified as `ITE_NET_CTL` since this driver is typically used to send and receive control information. The driver's call state is defined as active.

Other Default Settings

The `/hscx_1A` and `/hscx_1B` descriptors contain default information for proper configuration when using Microware's ISDN prototype board. For proper operation with your hardware, make sure you modify the `PORTADDR`, `VECTOR`, `PRIORITY`, and `IRQLEVEL` macros in `bch_desc.h` to match the configuration of your hardware.

The `RX_TIMESLOT` and `TX_TIMESLOT` need to be modified to use the correct timeslot on the inter-chip TDM bus as required. This is configured for use with the AM79c30 ISDN Transceiver chip which used the SPI bus, timeslots `Bd` and `Be` to move data between the chips.

Considerations for Other Drivers

None. This is a standard SoftStax driver with no extra necessary interfaces.

Chapter 7: Using DPIO

Dual-ported input/output (DPIO) enables you to simultaneously develop file managers and device drivers in the C programming language for OS-9. This chapter describes how to use DPIO and includes the following sections:

- **Utilities**
- **DPIO Libraries**
- **Compiling**
- **The File Manager**
- **Device Driver**
- **The Device Descriptor**
- **Example DPIO Device Descriptors**

Utilities

Because the Ultra C compiler does not automatically create file managers and drivers for 68K systems, it is necessary to create them using the following two steps:

Step 1. Identify the DPIO module as a "Program" module when compiling. This allows the module to have initialized data, a key factor for OS-9 system modules.

Step 2. Use the `chtype` utility to change the module from type "Program" to type "File Manager" or "Driver".

Device descriptors are slightly different. The DPIO device descriptor is the concatenation of a 68K device descriptor and an OS-9 device descriptor. The process of compiling, merging, and linking these two modules requires the use of the `chtype` utility, along with a second utility, `rm_vsect`, which intercedes at an intermediate step to modify the compiler's assembly language output to allow the two modules to be merged before linking.

Makefiles distributed with OS-9 products, as well as the sample code found later in this manual, demonstrate all of these steps and give clear examples of how to perform them.

chtype

Modify Program Modules or Change Module Type

Syntax

```
chtype [<options>] <filename> [<options>]
```

Description

`chtype` is only available for OS-9 for 68K DPIO modules. It is a DPIO development tool that modifies program modules and changes the module type to file manager, device driver, or device descriptor.

The `chtype` utility accepts the following options:

`-?` Display the command's syntax, options, and description. For example, to display information about the `chtype` utility, type:
`chtype -?`

`-t[=]MT_xxx`

Change the module to the specified type. Valid module types are shown in the table below.

Table 7-1 -t Option Module Types

Type	Change Module To An OS-9:
`MT_DEVDESC`	Device descriptor module
`MT_DEVDRVR`	Device driver module
`MT_FILEMAN`	File manager module
`MT_PROGRAM`	Program module
`MT_SYSTEM`	System state module

For example, to change the module type of the file named `myfile` to a device descriptor, type:

```
chtype -t=MT_DEVDESC myfile
```

`-x` — The specified file is in the execution directory. For example, to indicate that myfile is in an execution directory and to change its type to a device descriptor, type:

```
chtype -xt=MT_DEVDESC myfile
```

To create an OS-9 for 68K DPIO file manager, perform the following:

Step 1. Compile the file manager as a program by linking it to `fmstart.r`.

Step 2. Run `chtype` on the module with `MT_FILEMAN` as the specified type:

```
chtype -t=MT_FILEMAN myfileman
```

Note

You **must** run `chtype` on all file managers, device drivers, and device descriptors before attempting to use them. If you do not, you will get a "module not found" error (`E$MNF`, `E_MNF`, or `EOS_MNF`).

rm_vsect

Remove Vsect Wrapper in Assembly File

Syntax

```
rm_vsect < input_file.a > output_file.a
```

Description

`rm_vsect` is a general development tool that takes an assembly language file (`.a` file) from the standard input, removes the vsect wrapper, and prints it out to the standard output. `rm_vsect`'s main use lies in making OS-9 device descriptors.

To remove the vsect declarations from a `.c` source file, perform the following:

Step 1. Compile the `.c` file to an assembly language (`.a`) file, using your compiler's command line options.

Step 2. Run the `rm_vsect` utility:

```
rm_vsect <file.a >file_new.a
```

Step 3. Assemble the new `.a` file into a `.r` file and continue making the descriptor.

DPIO Libraries

The libraries supporting DPIO file managers are located in the following directories:

Libraries and root psects for OS-9 for 68K target processors:

`MWOS/OS9/68000/LIB`

Libraries and root psects for OS-9 target processors are located in one of the following subdirectories.

For OS-9/PPC target processors:

`MWOS/OS9000/PPC/LIB`

Special root psects that support DPIO are located in the `MWOS/OS9/68000/LIB/DPIO` subdirectory and include the following:

Table 7-2 Root psects for DPIO

Name	Root Psect For
`drvstart.r`	OS-9 for 68K/DPIO device driver modules.
`fmstart.r`	OS-9 for 68K/DPIO file manager modules.

System-state Libraries

There are two system-state libraries supporting DPIO for OS-9 (for 68K): `conv_lib.l` and `lock.l`. A summary follows.

Note

All functions are only available in system-state.

conv_lib.l

The `conv_lib.l` library contains `_os_*` functions not provided in the standard 68K libraries. It contains the following functions for OS-9 (for 68K) system state only.

For More Information

Refer to the ***OS-9000 Technical Manual*** and the ***Ultra C Library Reference*** for more information about these functions.

```
_os_ev_wait()
_os_getstat()
_os_gs_luopt()
_os_initdata()
_os_irq()
_os_salarm_cycle()
_os_salarm_set()
_os_setstat()
_os_sleep()
_os_ss_luopt()
```

cpu.l

The `cpu.l` library is obsolete. It has been merged with `os_lib.l` for OS-9 for the 68K starting with DPIO 2.1, and OS-9 version 3.0.2.

This library formerly contained the following functions:

```
change_static()
get_static()
grab_static()
irq_static()
irq_disable()
irq_maskget()
irq_enable()
irq_restore()
```

```
irq_save()
swap_static()
```

lock.l

The `lock.l` library contains resource lock descriptor manipulation routines for OS-9 for 68K. The functions can only be used in system state.

For More Information

Refer to the ***Ultra C Library Reference*** for more information about these functions.

```
_os_acqlk()
_os_caqlk()
_os_crlk()
_os_dellk()
_os_rellk()
```

Compiling

When compiling your source code under DPIO for an OS-9 for 68K target processor, perform the following steps.

Step 1. Specify the `MWOS/SRC/DPIO/DEFS` directory as the first header file directory to search.

Step 2. The next directory searched must be the standard OS-9 for 68K header file directory located in `MWOS/OS9/SRC/DEFS`.

Step 3. Specify the `MWOS/SRC/DEFS` directory as the next header file directory to search.

The following is a makefile example command line for an OS-9 for 68K target:

```
xcc test.c -v=$(MWOS)/SRC/DPIO/DEFS
-v=$(MWOS)/OS9/SRC/DEFS   -v=$(MWOS)/SRC/DEFS
```

When compiling your source code under DPIO for an OS-9 target processor, perform the following steps.

Step 1. Specify the `MWOS/OS9000/SRC/DEFS` header file directory first.

Step 2. Next, specify the `MWOS/SRC/DEFS` header file directory.

The following is a makefile example command line for an OS-9 (non 68K) target:

```
xcc test.c -v=$(MWOS)/OS9000/SRC/DEFS
-v=$(MWOS)/SRC/DEFS
```

The File Manager

You must observe the following requirements for file managers that use DPIO to run on both OS-9 and OS-9 for 68K:

- Include the `defconv.h` Header File

  Each source file compiled for OS-9 (for 68K) DPIO file managers must include the `defconv.h` header file. In addition, `defconv.h` **must** be the first header file the source file encounters. It is located in the `MWOS/SRC/DPIO/DEFS` directory. This header file contains conversion macros that give OS-9 names to OS-9 (for 68K) types and macros. For example, the `open.c` source file might begin like this:

  ```
  #ifdef _OSK
  #include <defconv.h>
  #endif
  #include <srvcb.h>
  #include <io.h>
  #include <module.h>
  .
  .
  .
  ```

- Define Path Options

  Due to restrictions of path options under OS-9 (for 68K), dual-ported file managers **must** define a path options size of exactly 128 bytes. This preserves the compatibility between OS-9 and OS-9 (for 68K).

- Call File Manager Entry Points

  The DPIO glue code maintains a consistent interface between OS-9 (for 68K) IOMAN and dual-ported file managers. One of these characteristics involves calls to the `open()` and `close()` file manager entry points. When `open()` is called, the glue code calls the `attach()` entry point followed by the `open()` entry point. This is consistent with file managers written under OS-9, allowing the file manager to perform functions with each attach of a device. However, since OS-9 (for 68K) will call `attach()` for the first but not subsequent `open()` calls, the attach count is one more than the true device count for the system in an

OS-9 (for 68K) environment. To gain a true attach count, the system manager may wish to `iniz` the devices for DPIO before running applications. The glue code also ensures that when `close()` is called, the `close()` and `detach()` file manager entry points are called.

- Compile a DPIO File Manager

Many of the rules for compiling a DPIO file manager for OS-9 (for 68K) remain the same. For example, after the `fmstart.r` file, the first file found on the linker command line must be the file containing the addresses of the file manager's entry points. The following is a typical linker command line:

```
xcc $(MWOS)/OS9/68000/LIB/DPIO/fmstart.r main.r
status.r open.r  close.r $(LIBS)$(DEFS)-fd=myfm
```

`main.r` is the file containing the addresses of each entry point. OS-9 and OS-9 (for 68K) both require that the entry points be the first area of static storage encountered in the file manager.

Example: Test File Manager

The following is an example source for the Test File Manager (TFM). The example is not provided electronically.

Note that this source is intended only to show the concepts of DPIO file managers; it is not intended as an actual OS-9 or OS-9 (for 68K) file manager.

```
/**************************************************
* tfm.c
* This file contains the source for the Test File
* Manager
* (TFM) **************************************************/
/* Header Files ***************************************************/
#ifdef _OSK
#include <defconv.h>/* OS-9 DPIO conversion defs */
#endif
#include <types.h>/* Type defs */
#include <errno.h>/* Error code defs */
#include <module.h>/* Memory module defs */
#include <srvcb.h>/* Service control block defs */
#include <io.h>/* I/O defs */
#include <testfm.h>/* TFM defs */
/* Function Prototypes ***************************************************/
error_code open(), close();
/* File Manager Static Storage ******************************************/
testfm_fm_stat fm_stat = {
    open,  /* File manager entry points */
    close,
    0,0,0,0;/* File manager static storage */
}
/* Open Entry Point *****************************************************/
error_code open(ctrl_block, path_desc)
I_open_pb ctrl_block;/* parameter block pointer */
Testfm_path_desc path_desc;/* Test path descriptor type */
{
    /* Initialize path here */
    return SUCCESS;
}
/* Close Entry Point */
error_code close(ctrl_block, path_desc)
I_close_pb ctrl_block;/* parameter block pointer */
Testfm_path_desc path_desc;/* Test path descr type */
{
    /* Deinitialize path here */
    return SUCCESS;
}
/* End of tfm.c *****************************************************/
```

Example: Makefile for OS-9 (for 68K) Target Processor

The following example makefile produces a test file manager for an OS-9 for 68K target processor.

```
#
# makefile
#
# This makefile will make the Test File Manager (TFM).
# The target processor is OS-9/68000.
#
#
MWOS    =   /h0/MWOS
RDIR    =   RELS
ODIR    =   ../CMDS
DEFINES =
DEFS    =   -v=. \
            -v=$(MWOS)/SRC/DPIO/DEFS \
            -v=$(MWOS)/OS9/SRC/DEFS \
            -v=$(MWOS)/SRC/DEFS
LIBS    =   -l=$(MWOS)/OS9/68000/LIB/os_lib.l
COMP    =   cc $(DEFS) $(DEFINES) -eas=$(RDIR) -r -o=7\
            -to=osk -tp=68k
LINK    =   cc $(LIBS) -k -to=osk -tp=68k
START   =   $(MWOS)/OS9/68000/LIB/DPIO/fmstart.r
RFILES  =   $(RDIR)/tfm.r
$(ODIR)/tfm:            $(RFILES)
    $(LINK) $(START) $(RFILES) -fd=$(ODIR)/tfm
    chtype -t=MT_FILEMAN $(ODIR)/tfm
$(RDIR)/tfm.r:          tfm.c
    $(COMP) tfm.c
#
```

Example: Makefile for OS-9 Target Processor

The following example makefile produces a test file manager for an OS-9 target processor.

```
#
# makefile
#
# This makefile will make the Test File Manager (TFM).
# The target processor is OS-9000/80386.
#
#
MWOS    =   /h0/MWOS
RDIR    =   RELS
ODIR    =   ../CMDS
DEFINES =
DEFS    =   -v=. \
            -v=$(MWOS)/OS9000/SRC/DEFS \
            -v=$(MWOS)/SRC/DEFS
LIBS    =   -l=$(MWOS)/OS9000/80386/LIB/os_lib.l
COMP    =   cc $(DEFS) $(DEFINES) -r=$(RDIR) -r -o=7\
            -to=os9000 -tp=386
LINK    =   cc $(LIBS) -k -to=os9000 -tp=386

START   =   $(MWOS)/OS9000/80386/LIB/fmstart.r
RFILES  =   $(RDIR)/tfm.r
$(ODIR)/tfm:            $(RFILES)
    $(LINK) $(START) $(RFILES) -fd=$(ODIR)/tfm
$(RDIR)/tfm.r:          tfm.c
    $(COMP) tfm.c
#
#
```

Device Driver

You must observe the following requirements for device drivers that use DPIO to run on both OS-9 and OS-9 (for 68K).

- Include the `defconv.h` Header File

  The rules governing the `defconv.h` header file in the file manager also apply in the device driver. `defconv.h` **must** be the first header file found by the driver source when compiling for OS-9 (for 68K).

  Each source file compiled OS-9 (for 68K) DPIO device drivers must include the `defconv.h` header file. In addition, `defconv.h` **must** be the first header file the source file encounters. It is located in the `MWOS/SRC/DPIO/DEFS` directory. This header file contains conversion macros that give OS-9 names to OS-9 (for 68K) types and macros. For example, a source file might begin like this:

  ```
  #ifdef _OSK
  #include <defconv.h>
  #endif
  #include <srvcb.h>
  #include <io.h>
  #include <module.h>
  .
  .
  .
  ```

- Set Carry Bit: DPIO Device Driver Characteristics

  To maintain compatibility between operating systems, device driver interrupt service routines should have wrappers for OS-9 (for 68K) to set the carry bit when returning non-`SUCCESS` error codes.

- Compile a DPIO File Manager

  Many of the rules for compiling a DPIO device driver for OS-9 (for 68K) remain the same. For example, after the `drvstart.r` file, the first file found on the linker command line must be the file containing the addresses of the driver’s entry points.

Example: Test File Manager Device Driver

The following is an example source for the test file manager device driver (`tfmdrvr`). This file is not provided electronically.

This source is intended only to show the concepts of DPIO device drivers and is not intended as an actual OS-9 or OS-9 (for 68K) device driver.

```
/*******************************************************

  tfmdrvr.c

 This file contains the source for the TFM device driver.
******************************************************/
/* Header Files ******************************************************/
#ifdef _OSK
#include <defconv.h>/* OS-9 DPIO conversion defs */
#endif
#include <types.h>/* Type defs */
#include <errno.h>/* Error code defs */
#include <module.h>/* Memory module defs */
#include <srvcb.h>/* Service control block defs */
#include <io.h>/* I/O defs */
#include <testfm.h>/* TFM defs */
/* Function Prototypes **************************************************/
error_code  activate(), terminate();
/* Driver Static Storage ************************************************/
testfm_drvr_stat drvr_stat = {
    activate,/* Driver entry points */
    terminate,
    0,0,0,0;/* Driver static storage */
}
/* Activate Entry Point *************************************************/
error_code activate(dev_entry)
Dev_list dev_entry;/* OS-9000-like device entry */
{
    /* Turn on the hardware here */
    return SUCCESS;
}
/* Terminate Entry Point ************************************************/
error_code terminate(dev_entry)
Dev_list dev_entry;/* OS-9000-like device entry*/
{
    /* Turn off the hardware here */
    return SUCCESS;
}
/* End of tfmdrvr.c *****************************************************/
```

Example: Makefile for OS-9 (for 68K) Target Processor

The following example makefile produces `tfmdrvr` for an OS-9 for 68K target processor:

```
#
# makefile
#
# This makefile will make the tfmdrvr Device Driver.
# The target processor is OS-9/68000.
#
# NOTE: The compilation options in this file correspond to Ultra C
#
#
MWOS    =   /h0/MWOS
RDIR    =   RELS
ODIR    =   ../CMDS
TMPDIR  =   /dd
DEFINES =
DEFS    =   -v=. \
            -v=$(MWOS)/SRC/DPIO/DEFS \
            -v=$(MWOS)/OS9/SRC/DEFS \
            -v=$(MWOS)/SRC/DEFS
LIBS    =   -l=$(MWOS)/OS9/68000/LIB/os_lib.l
COMP    =   cc $(DEFS) -eas=$(RDIR) -r -o=7 -to=osk\
            -tp=68kc -td=$(TMPDIR)
LINK    =   cc -td=$(TMPDIR) $(LIBS) -k -to=osk -tp=68kc
START   =   $(MWOS)/OS9/68000/LIB/DPIO/drvstart.r
RFILES  =   $(RDIR)/tfmdrvr.r
$(ODIR)/tfmdrvr:        $(RFILES)
    $(LINK) $(START) $(RFILES) -fd=$(ODIR)/tfmdrvr
    chtype -t=MT_DEVDRVR $(ODIR)/tfmdrvr
$(RDIR)/tfmdrvr.r:      tfmdrvr.c
    $(COMP) tfmdrvr.c
```

Example: Makefile for OS-9 Target Processor

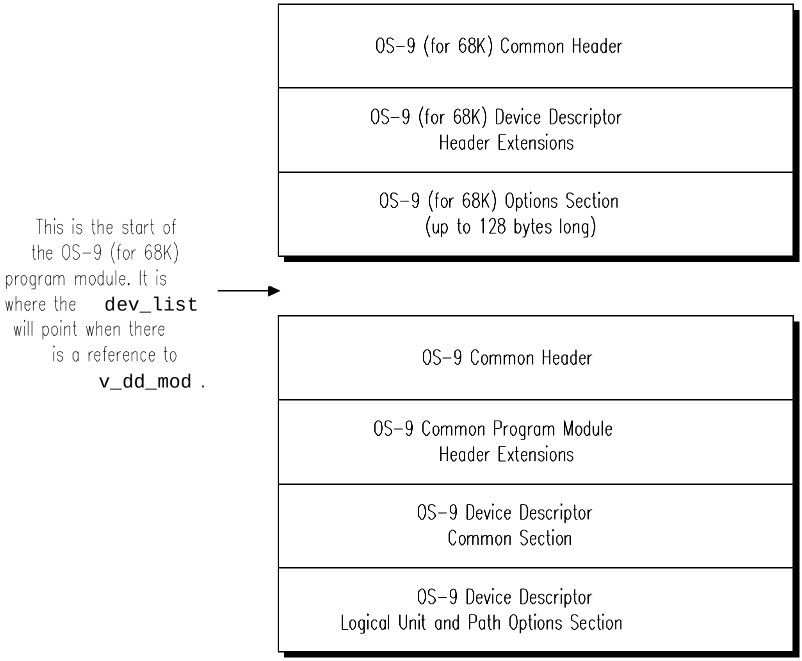
The following example makefile produces `tfmdrvr` for an OS-9 target processor:

```
#
# makefile
#
# This makefile will make the tfmdrvr Device Driver.
# The target processor is OS-9000/80386.
#
#   NOTE:   The compilation options in this file correspond to Ultra C  #
#
MWOS    =   /h0/MWOS
RDIR    =   RELS
ODIR    =   ../CMDS
TMPDIR  =   /dd
DEFS    =   -v=. \
            -v=$(MWOS)/OS9000/SRC/DEFS \
            -v=$(MWOS)/SRC/DEFS \
LIBS    =   -l=$(MWOS)/OS9000/80386/LIB/os_lib.l
COMP    =   cc $(DEFS) -eas=$(RDIR) -r -o=7 -to=os9000\
            -tp=386 -td=$(TMPDIR)
LINK    =   cc -td=$(TMPDIR) $(LIBS) -k -to=os9000 -tp=386
START   =   $(MWOS)/os9000/80386/LIB/drvstart.r
RFILES  =   $(RDIR)/tfmdrvr.r
$(ODIR)/tfmdrvr:        $(RFILES)
    $(LINK) $(START) $(RFILES) -fd=$(ODIR)/tfmdrvr
$(RDIR)/tfmdrvr.r:      tfmdrvr.c
    $(COMP) tfmdrvr.c
#
```

The Device Descriptor

The figure below illustrates an OS-9 (for 68K) DPIO device descriptor.

Figure 7-1 OS-9 (for 68K) DPIO Device Descriptor



Two module types form this descriptor. Each descriptor type is a complete module. `chtype` merges these modules and alters them to appear as a single OS-9 (for 68K) device descriptor module.

The upper half of the OS-9 (for 68K)/DPIO device descriptor structure is a complete OS-9 (for 68K) device descriptor containing the following:

- Module name
- File manager
- Driver name
- OS-9 (for 68K) path options section. DPIO file managers do not use this section

The lower half of OS-9 (for 68K)/DPIO Device Descriptor is an OS-9 device descriptor compiled in OS-9 (for 68K) as a program module. It contains the following:

- OS-9 device descriptor common section (dd_com)
- OS-9 logical unit initialized static storage
- OS-9 path option initialized static storage

A common technique for finding the dd_com section of a device descriptor is to use the following code:

```
dd_com *dd;
dd=(dd_com*)(((char*)(Dev_list->v_dd_mod))+(Dev_list
->v_dd_mod->m_exec));
```

Example DPIO Device Descriptors

Following is an example of the DPIO device descriptor `test1`. The `test1` device descriptor is made from the following files:

Table 7-3 test1 Files

File	Description
testdesc_const.c	OS-9 device descriptor common section.
testdesc_stat.c	Logical unit and path options section.
testdesc_os9.c	OS-9 (for 68K) device descriptor common section (for OS-9 (for 68K) descriptors only).
systype.h	Device descriptor definitions.
makek68k	OS-9 (for 68K) device descriptor makefile.
makep386	OS-9/80386 device descriptor makefile.

An OS-9 device descriptor is made up of the OS-9 common section and the logical unit/path options section. An OS-9 for 68K device descriptor is made up of three parts—the OS-9 for 68K common section, the OS-9 common section, and the logical unit/path options section.

To add a new descriptor for either OS-9 or OS-9 for 68K, simply add the new definitions to `systype.h` and the new targets to the makefiles.

The following are listings of the six files used to make OS-9 and OS-9 (for 68K) DPIO device descriptors.

testdesc_const.c

The following is the `testdesc_const.c` file:

```
/*
*  $Workfile:   testdesc_const.c  $
*
*  This is the OS-9000 device descriptor constant
*  section. */
/* Header Files */

#include <systype.h>

/* Device descriptor constant data structure */

struct myconst {

    test_desc       desc;
    char            fm_name[32];
    char            drv_name[32];

} myconst = {

    /* desc */
    {
        /* Device Descriptor Common Fields */
        PORTADDR,/* dd_port: hardware base address */
        LUN,/* dd_lu_num: logical unit number */
        sizeof(test_path_desc),
       /* dd_pd_size: path descriptor size */
        DT_TEST,/* dd_type: device type */
        MODE,/* dd_mode: device mode capabilities */
        myconst.fm_name,/* dd_fmgr: file mngr name offset */
        myconst.drv_name,/* dd_drvr: driver name offset */
        DC_SEQ,/* dd_class: device class */
        0,/* dd_dscres: reserved for future IOMAN */
            /* Other descriptor specific fields here */
    },

    /* fm_name[32]: file manager name */
    {
        FMNAME
    },

    /* drv_name[32]: device driver name */
    {
        DRIVERNAME
    }
};
```

testdesc_stat.c

The following is the `testdesc_stat.c` file:

```
/**********************************************************
*  $Workfile:   testdesc_stat.c  $
 **********************************************************
*  These are the device descriptor logical unit static,
*  logical unit options, and path descriptor options
* sections.
 *********************************************************/
/* Header Files ********************************************************/

#include <systype.h>

/* Logical Unit Static Storage Declarations *********************************/

test_lu_stat my_lu = {

    /* Logical unit specific fields here */

};
```

testdesc_os9.c

The following is the `testdesc_os9.c` file:

```
/**********************************************************
 *  $Workfile:   testdesc_os9.c  $
 **********************************************************
 *  This is the OS-9 device descriptor section.
 *********************************************************/
/* Header Files ******************************************/
#include <systype.h>

/* Macro Definitions ************************************/

#define OPTS_SZ         128
#define MCOMMON_SZ      0x30
#define MDESC_SZ        0x18
#define FMNAME_SZ       32
#define DRVNAME_SZ      32
#define FM_OFFSET       (MCOMMON_SZ + MDESC_SZ  + OPTS_SZ)
#define DRVR_OFFSET     (FM_OFFSET + FMNAME_SZ)

/* Device descriptor OS-9 section data structure **********/

struct myos9 {
```

```
    /* from mod_dev in <module.h> */
    char *_mport;/* device port address */
    unsigned char_mvector;/* trap vector number */
    unsigned char   _mirqlvl;       /* irq interrupt level */
    unsigned char   _mpriority;     /* irq polling priority */
    unsigned char   _mmode;/* device mode capabilities */
    short_mfmgr;/* file manager name offset */
    short_mpdev;/* device driver name offset */
    short_mdevcon;/* device configuration offset */
    unsigned short  _mdscres[1];    /* (reserved) */
    unsigned long   _mdevflags;     /* reserved */
    unsigned short  _mdscres2[1];   /* reserved */
    unsigned short  _mopt;          /* option table size */
    unsigned char   _mdtype;        /* device type code */

    /* other needed fields */
    charopts[OPTS_SZ-1];/* for long-word alignment */
    charfm_name[FMNAME_SZ];/* file manager name */
    chardrv_name[DRVNAME_SZ];/* device driver name */

} myos9 = {

    /* OS-9 device descriptor section */
    PORTADDR+LUN,/* _mport: device port address */
    VECTOR,/* _mvector: trap vector number */
    IRQLEVEL,/* _mirqlvl: irq interrupt level */
    PRIORITY,/* _mpriority: irq polling priority */
    MODE,  /* _mmode: device mode capabilities */
    FM_OFFSET,/* _mfmgr: file manager name offset */
    DRVR_OFFSET,/* _mpdev: device driver name offset */
    0,     /* _mdevcon: device configuration offset */
    { 0 }, /* _mdscres[1]: reserved */
    0,     /* _mdevflags: reserved */
    { 0 },  /* _mdscres2[1]: reserved */
    OPTS_SZ,/* _mopt: option table size */
    DT_TEST,/* _mdtype: device type code */

    /* opts[OPTS_SZ-1]: long-word alignment *
     * NOTE: _mdtype is the first byte of the options
        field */
    {
        0
    },

    /* fm_name[FMNAME_SZ]: file manager name */
    {
        FMNAME
    },

    /* drv_name[DRVNAME_SZ]: driver name */
    {
        DRIVERNAME
    }
};
```

systype.h

The following is the `systype.h` file:

```
/*********************************************************
 *  $Workfile:   systype.h  $
 *********************************************************
 *  This is the device descriptor definitions file.
 ********************************************************/


/* Header Files ******************************************************/

#include <types.h>
#include <const.h>
#include <module.h>
#include <io.h>
#include <sg_codes.h>
#include <modes.h>
#include <reg68k.h>
#include <testdesc_defs.h>
/* Other needed header files included here */


/*********************************************************
* Makes device descriptor: test1 */
#ifdef test1

#define PORTADDR    0xFFFF0A01  /* Base address of hardware                 */
#define VECTOR      0x96        /* Port vector */
#define IRQLEVEL    5           /* Port IRQ Level */
#define IRQ_MASK    ((IRQLEVEL << 8) | SUPERVIS)
       /* CPU interrupt mask */
#define PRIORITY 16/* IRQ polling priority */
#define LUN       1/* Logical unit number */
#define MAXCREF   3/* Max # call refs in virtual unit */
#define FMNAME “test_fm”   /* Name of file manager */
#define DRIVERNAME  “test_drvr” /* Name of device driver */
#define MODE    S_ISIZE | S_IREAD | S_IWRITE
       /* Descriptor mode */
/* Other device specific definitions here */

#endif /* test1 ********************************************************/
```

Appendix A: Debugging

This chapter details debugging processes built directly into your driver for troubleshooting problematic drivers.

Debugging: dbg_mod.l Overview

The `dbg_mod.l` library allows programmers to incorporate debugging information into drivers they are developing. The library creates a data module in memory into which the driver can write information. The data module can be reviewed for debugging information. This library allows you to debug in real time; the system never stops as it does with `rombug`.

To use the library, define the following in the driver's or logical unit's static storage:

```
void* dbg_ptr;
```

This keeps the address of the debug information and passes it to all library calls. Include the `dbg_mod.h` header so you can acquire prototype information for the library calls.

Generally, the code to use the library is conditionalized so you can make a non-debug version of the driver. This conditioned code is in the form of:

```
#ifdef DEBUG
    <The library call>
#endif
```

Library Calls

The following library calls are used for debugging:

- `debug_data()`
- `debug_init()`
- `debug_string()`
- `debug_timestamp()`
- `debug_4data()`

debug_data()

Syntax

```
void debug_data (
  Dbg_stat     debug_stat,
  u_int32      *sptr,
  u_int32      data);
```

Description

This call writes a character string (no larger than 10 characters, not including the `NULL` byte at the end of the string) and one `u_int32` into the debug module. It is created to use one full line if you dump the debug module in `rombug` or if you are using the `dump` utility.

Parameters

`debug_stat`	points to modules `debug_stat` structure.
`*sptr`	points to an ASCII string with a maximum length of 10 bytes. To ensure readability of the debug module, always make the string exactly 10 bytes.
`data`	is a 32-bit data value.

For More Information

Refer to the *Debug Data String Conventions* in the following section for more information.

debug_init()

Syntax

```
error_code debug_init (
  u_int32      size,
  Dbg_stat     *dbg_ptr,
  char         *mod_name);
```

Description

This is the entry point where the caller tells the debug library which data module name to use and the data module size requirements.

It is recommended the module name and data module size parameters be kept in a device descriptor so the calling code can get them out and they do not have to be hard-coded anywhere within the module.

Parameters

`size`	contains the size of the data module. It must be greater than 64 bytes.
`*dbg_ptr`	points to the pointer defined in static storage.
`*mod_name`	contains the name of the data module. It must not be `NULL`.

debug_string()

Syntax

```
void debug_string (
  Dbg_stat  debug_stat,
  u_char    *dptr,
  u_int32   size);
```

Description

This call writes a character string of `size` bytes into the data module. The function rounds up to the nearest dump line, padding with zeros, so the debug module remains readable when dumped in `rombug` or when using the `dump` utility.

Parameters

`debug_stat`	points to modules debug_stat structure.
`*dptr`	points to an ASCII string.
`size`	contains the size of the string to place into the debug module.

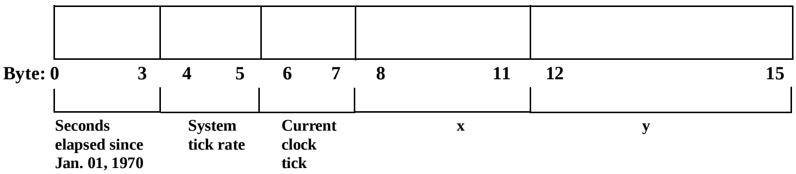
debug_timestamp()

Syntax

```
void debug_timestamp (
    Dbg_stat     debug_stat,
    u_int32      x,
    u_int32      y);
```

Description

This function writes a time-stamp into the debug data module. The time-stamp has the following layout:



Parameters

`debug_stat`	points to modules `debug_stat` structure.
`x`	is the first 32-bit data value. Unsigned long you can pass in for debugging.
`y`	is the second 32-bit data value. Unsigned long you can pass in for debugging.

debug_4data()

Syntax

```
void debug_4data (
  Dbg_stat  debug_stat,
  u_int32   a,
  u_int32   b,
  u_int32   c,
  u_int32   d);
```

Description

This function writes four longs of debug data.

Parameters

`debug_stat`	points to modules debug_stat structure.
a	is the first 32-bit data value.
b	is the second 32-bit data value.
c	is the third 32-bit data value.
d	is the fourth 32-bit data value.

Using Debug

A variety of debug information can be placed into the data module. There two methods used to look at the information include the following:

- rombug
- dump utility

When dumping the data, both rombug and the dump utility display 16 bytes of data in one line, both in hex and in ASCII. The `dbg_mod.l` library uses this, either truncates or pads strings to fit, and places all data into the module as four-byte `u_int32` values.

Rombug

When in rombug, view the debug data module by linking to it and dumping from an offset in the `.r7` register. The steps to view the debug data module are:

Step 1. Start at the `rombug:` prompt.

If you are not at this prompt already, break into rombug by typing `break`.

Step 2. Type `link X` at the `rombug` prompt to link to debug data module. `X` is the name of the data module to view. rombug can link to modules loaded into memory.

Once linked, the `.r7` relocation register points to the beginning of the module. From this pointer, it is easy to access the data area of a data module.

Step 3. View the information from the beginning of the module.

After the module is linked by rombug, obtain a pointer to the data section. The offset to the data differs between OS-9 for 68K and OS-9.

- for OS-9 for 68K, the offset is 0x34
- for OS-9, the offset is 0x64

Step 4. Obtain a pointer.

Step 5. Point the register to the data section.

Typically, the `.r7` register is modified to point to the data section.

To modify `.r7` for OS-9 for 68K, type:

```
.r7 .r7+34
```

To modify `.r7` for OS-9, type:

```
.r7 .r7+64
```

This sets the `.r7` register to the beginning of the debug static area of the module, and past the module header so the data module looks correct when dumping.

Step 6. Look at information from the end of the module.

The `.r6` register can be set to point to the last entry in the debug information.

Step 7. Type `.r6[.r7+4]` to set `.r6`.

This sets the register `.r6` line after the last debug statement put into the debug module.

The writing in the debug module rolls over to the beginning of the module after writing to the end of the module. You will never know where the last statement of the module is without setting the `.r6` register.

Step 8. Analyze the contents of the messages in the debug module by looking at the statements behind the statement pointed to by `.r6.`

dump Utility

Use the dump utility to look at the debug data module without entering rombug. However, use the dump utility mainly to save the debug data module to a file. Then, use a text editor to view or print the information.

For OS-9 for 68K, the dump command syntax is:

```
$ dump <module name> -m 34 >-<output file>
```

For OS-9, the dump command syntax is shown below:

```
$ dump <module name> -m 64 >-<output file>
```

Debug Data String Conventions

As described before in the `debug_data()` usage call, this call writes a string into the debug data module. In future releases of SoftStax, we may provide a windowing graphical user interface (GUI) application that will automatically monitor the debug data module and print out helpful debugging messages in real time while the test applications are running. This eliminates the need for using rombug after the fact when a bug or crash occurs. In order to create this utility, drivers drivers must adhere to a convention when creating the strings in the module. This convention allows the debugging application to identify the protocol and print standard helpful test messages to the screen when it identifies the standard debug strings in the data module.

Rule

Your protocol should have a unique two-character prefix that begins every string. For example, spproto uses `PR`. UDP/TCP/IP use `UP`, `TP`, and `IP` respectively. The `ER` prefix is reserved for reporting error strings in the debug module. Since all drivers have standard entry points, the same debug data string should be put into the data module when this entry point is hit. This is the format for those standard strings where XX stands for the protocol prefix:

```
-XXIniz, XXTerm
   3rd param = deventry
-XXUpdate, XXOpen, XXClose, XXPush, XXPop
   3rd param = path descriptor
-XXGetProtId
   3rd param = protocol ID value
-XXSsUnknown, XXGsUnknown
   3rd param = pb->code
-XXDN,XXUP for the debug4data() call
-XXDnMbEmpty, XXUpMbEmpty
   3rd param = mbuf pointer
```

The spproto driver already adheres to these standard debugging strings, so if the SPPROTO driver is used as the template, everything will be acceptable.

Some standard error strings that may occur in your protocol specific implementation not already in SPPROTO include the following:

Table A-1 SPPROTO Error Strings

Cause	String
`m_getn()` fails	`ERNoMbAvail`, status in 3rd parameter.
Can’t find per path entry	`ERNoPPEntry` 3rd param is `ptr` to per path list.
Anytime `srqmem()` fails	`ERSrqmem`, 3rd param = error returned from call.

Appendix B: The Mbuf Facility

An mbuf is a common data structure used to efficiently store variable-length data blocks. Mbufs can be queued, allocated, and deallocated, and are used for compatibility between local area networking packets and wide area networking packets. This appendix covers the following topics: installing the mbuf facility, mbuf structure, and its functions.

Installing the Mbuf Facility

Both the local and wide area network file managers use mbufs as the basis for data transfer. Thus, before starting any of these file managers, install the mbuf facility by performing the following steps:

Step 1. Load the `sysmbuf` module. `sysmbuf` controls the allocation and deallocation of mbufs from the system mbuf free pool. The `sysmbuf` module may be added to the system boot.

Step 2. Run the `mbinstall` utility, which installs the user-installed system call and allocates memory for use as the system mbuf free pool.

Optionally, on OS-9 for 68k family machines, the `sysmbuf` module may be named in the P2 Extension Module list of the init module and initialized as the system first boots up, eliminating the need to run `mbinstall`.

You should only call `mbinstall` once after a system reset to set up a system-wide mbuf pool.

Note

The default system mbuf pool size is 128K. This should be sufficient for all but the most heavily loaded systems. Systems with multiple network adapters may need to increase the available mbuf pool size.

OS-9 for 68K Systems

For OS-9 for 68K systems, use one of the following:

- `sysMbuf_010` for processors less than the 68020.
- `sysMbuf_020` for processors 68020 or greater. `sysMbuf_020` uses the more efficient bit instructions available with these processors.

By default, these modules both allocate 128K of memory for the system mbuf pool. You can patch the module to increase (or decrease) the amount of memory allocated for the system mbuf pool.

Because multiple IRQ service routines call and share the mbuf code, by default, the mbuf code masks IRQs to level 7 to protect allocation and deallocation requests. With the fast algorithm `sysmbuf` uses, this is usually not a problem. You can patch the module to limit the raising of the mask to the level of the highest IRQ service routine using mbufs. Use this feature with extreme care. If not done properly, this destroys the integrity of the mbuf free space resulting in a non-functioning system. Normally, you should not raise the mask to the level of the highest IRQ service routine using mbufs unless the operation of the network interferes with a higher-level IRQ routine.

By default, the minimum block size you can allocate is 64 bytes. A smaller block size uses more bitmap memory and requires more iterations through the code, but wastes less memory for small allocations. A larger block size uses less bitmap memory and requires less iterations through the search code, but wastes more memory for small allocations.

The 64 byte allocation size allows up to 2048 bytes to be bit mapped in one 32-bit search/load/store. Maximum allocation is dependent upon the memory available and memory allocated to other systems. SoftStax rarely requests blocks less than 64 bytes. 64 bytes has been demonstrated to be nearly the optimal size for SoftStax routines. Ethernet mbuf requests are never larger than 1536. However, in some SoftStax protocols, SoftStax may often request mbufs of less than 10 bytes for many types of packets.

The following locations in the module may interest you:

```
Addr   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 2 4 6 8 A C E
----   ---  ---  ---  ---  ---  ---  ---  ---  ---------------
0000  4afc 0001 0000 08de 0000 0000 0000 08d2 J|.....^.......R
0010  0555 0c01 a000 000a 0000 0000 0000 0000 .U.. ...........
0020  0000 0000 0000 0000 0000 0000 0000 1c50 ...............P
0030  0000 0050 0000 0000 0000 0826 0006 8020 ...P.......&...
0040  0007 8008 0000 0000 0000 0040 0002 0000 ...H.......@....
0050  48e7 0040 43fa 0012 4e40 0032 2c4b 6100 Hg.@Cz..N@.2,Ka.
```

Table B-1 Locations of Interest

Offset	Length	Meaning
`003c`	long	Processor identifier: 68010 or 68020 (not changeable).
`0040`	short	Maximum IRQ mask level: 7 (patchable).
`0042`	short	Reserved.
`0044`	long	Colored memory typecode: 0 (patchable).
`0048`	long	Minimum allocation blocksize: 64 (patchable).
`004c`	long	Memory to use for Mbufs: 128kb (patchable).

Note

OS-9 `SysMbuf` can only be installed using the `mbinstall` utility, while the OS-9 for 68K version can be installed using the `mbinstall` utility or by specifying it in the P2 Extension Module list in the Init module and including it in the boot.

OS-9 Systems

For OS-9 systems, `sysmbuf`, by default, allocates 128K of memory for the system mbuf pool.

To modify the default attributes of `sysmbuf`, use the `mbinstall` utility. `mbinstall` has three options that enable you to specify the memory pool size, the block size, and the memory color:

Table B-2 mbinstall Options

Option	Description
`-m=<size>`	Memory pool size in kilobytes. The default is 128K.
`-b=<size>`	Block size in bytes. The default is 64.
`-t=<num>`	Memory color. The default is 0.

For example, to set the system mbuf pool size to 256K, use the command line:

```
mbinstall -m256
```

The mbuf structure is declared on the following page.

Declaration

The mbuf structure is declared in the `mbuf.h` header file as follows:

```
typedef struct mbuf {
   struct mbuf     *m_pnext;
   struct mbuf     *m_qnext;
   u_int16         m_alloc;
   u_int16         m_size;
   u_int8          m_flags;
   u_int8          m_type;
   u_int16         m_offset;
} *Mbuf;
```

Fields

`m_pnext` chains multiple mbufs into a packet chain called a message unit.

`m_qnext` chains multiple packet chains into a message queue.

`m_alloc` indicates the total size of the memory allocated for this mbuf. This is set by sysmbuf and must not be altered. If you do alter this field, system crashes result.

`m_size` indicates the number of valid data bytes in this mbuf.

`m_flags` contains bit flags which may be set to give special meaning to various mbufs. Examples include:

- enabling protocols to implement reliable data transfer through acknowledgment timeout and retransmission by using the `SPF_NOFREE` and `SPF_DONE` bits.

	•enabling drivers that indicate errored receive packets to protocols above without actually tossing the entire packet.
`m_type`	indicates the type of mbuf. The main use of the `m_type` field is to allow the network I/O system to distinguish certain types of mbufs that may be on a data queue. The main types used are `MB_DATA` and `MB_ADDR`. Others are defined but not widely used. The `MB_DATA` type indicates this is real payload the end application is wishing to read. The `MB_ADDR` indicates this mbuf is not payload, but contains addressing information the application may be returned when doing a `recvfrom()` socket call for example.
`m_offset`	indicates the offset, in bytes, from the beginning of the mbuf to the first valid bytes.

SPF_NOFREE/SPF_DONE

You are writing a protocol driver that enables reliable data transfer. Typically, you create a modulo array to store the pointers to data sent down for transmission, but not yet acknowledged by the far end. However, when the driver is through transmitting the packet, it typically performs an `m_free_p()` on the transmitted mbuf. If this were allowed to happen, the mbuf in the unacknowledged array would be lost. The mbuf library implements the `SPF_NOFREE` to indicate the mbuf must not be returned to the free pool. When any `m_free_x()` call is done, the library checks for the `SPF_NOFREE` bit. If it is set, the library does not return the mbuf to the free pool. Instead, the library only sets the `SPF_DONE` bit to indicate that the packet has been transmitted.

With this approach, hardware drivers can still call the regular `m_free()` functions and the library takes care of the `SPF_NOFREE` details.

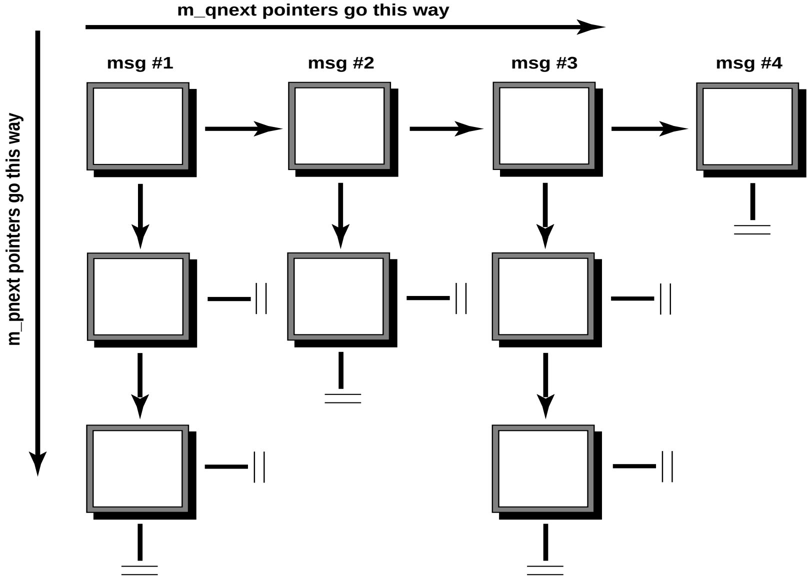
There are special calls provided in `misc.c` in the *mw*SoftStax driver section that enable you to get and free *nofree* mbufs. These calls internally set and clear the `SPF_NOFREE` bit for proper operation.

SPF_RXERR

If a driver receives an mbuf and detects an error, but the incoming packet may still be useful to the protocol driver above, the SPF_RXERR bit may be set in the mbuf's m_flags field. Not all drivers check this bit on incoming mbufs, so it should only be used in situations where it is known that a protocol driver above you understands the flag.

Example of mbuf Queue Structure

Figure 7-2 Example of mbuf Queue Structure



Each square in Figure 7-2 Example of mbuf Queue Structure represents one mbuf structure.

The `mbuf packet chain` consists of the series of mbufs pointed to by the `Mbuf->m_pnext` fields. In this figure, the mbuf packet chain for message #1 consists of three mbufs.

The `mbuf queue` consists of the series of mbuf packet chains pointed to by the `Mbuf->m_qnext` fields. In this figure, the mbuf queue consists of messages one through four.

Mbuf Functions

The following functions are provided in the `mbuflib.l` library.

Table B-3 mbuf functions

Function	Description
`m_adj()`	Adjust Pointers and Sizes
`m_cat()`	Append Mbuf
`m_copy()`	Allocate New Mbuf and Copy Data to New Mbuf
`m_deq()`	Remove First Mbuf Packet
`m_enq()`	Append Mbuf to Mbuf Queue
`m_flush()`	Free Empty Mbufs in Packet Chain
`m_free()`	Free Mbuf
`m_free_m()`	Return Mbuf to Free Pool
`m_free_p()`	Free Packet Chain
`m_free_q()`	Free Mbuf Queue
`m_get()`	Allocate Mbuf (Blocking)
`m_getn()`	Allocate Mbuf (Non-Blocking)
`m_len_p()`	Return Number of Data Bytes
`m_move()`	Copy Specified Number of Bytes

Table B-3 mbuf functions (continued)

Function	Description
m_msize()	Return Number of Bytes in Packet
m_pad(), M_PAD()	Advance Data Pointer
m_ptod()	Return Pointer to First Valid Byte
m_pullup()	Ensure First Mbuf Has Data
mtod()	Get Pointer to Mbuf Data
m_unpad(), M_UNPAD()	Move Data Pointer

The m_get(), m_getn(), and m_free() mbuf library functions request and free mbuf structures through a user-installed system call. The remaining functions operate only on the mbuf header fields and are entirely contained within the mbuflib.l library.

m_adj()

Adjust Pointers and Sizes

Syntax

```
#include <mbuf.h>
u_int32 m_adj(
  Mbuf      mb,
  u_int32   count);
```

Description

`m_adj()` adjusts pointers and sizes to remove data from the packet. `m_adj()` does not free the mbufs; only the pointers and sizes are adjusted as indicated. `m_adj()` returns the actual count of data removed.

Parameters

`mb`	is a pointer to the packet chain.
`count`	specifies the amount and location of data to remove. •If `count` is positive, the data is removed from the beginning of the packet chain. •If `count` is negative, the data is removed from the end of the packet chain.

See Also

`m_pad()`, `M_PAD()`
`m_unpad()`, `M_UNPAD()`

Syntax

```
#include <mbuf.h>
Mbuf m_cat(
   Mbuf    mb,
   Mbuf    nb);
```

Description

`m_cat()` appends mbuf packet `nb` to the end of packet `mb`. The result combines the two mbufs into a single packet. `m_cat()` returns `mb`, if `mb` is non-NULL. Otherwise, `m_cat()` returns `nb`.

Parameters

`mb`	is a pointer to the packet chain on to which to append `nb`.
`nb`	is a pointer to the packet chain to append to mb.

m_copy()

Allocate New Mbuf and Copy Data to New Mbuf

Syntax

```
#include <mbuf.h>
Mbuf m_copy(
   Mbuf        mb,
   u_int32     offset,
   u_int16     count,
   int32       *status);
```

Description

`m_copy()` allocates a new mbuf and copies data beginning from the original mbuf to the new mbuf `m_copy()` returns `NULL` if `offset` points past the end of the packet. If less than `count` bytes are in the packet following the `offset`, the returned mbuf is at least of size `count`, but it is not full. `m_copy()` calls `m_getn()` to allocate the new mbuf. If `m_getn()` fails, `m_copy()` returns `NULL` and places the error code status from `m_getn()` in `status`.

Parameters

`mb`	is a pointer to the original mbuf.
`offset`	specifies the location to begin the copy.
`count`	specifies the number of bytes to copy.
`status`	is a pointer to the returned error status code.

Fatal Errors

`EOS_ILLARG`	returned when the request is for a size greater than 32767 bytes.
`EOS_MEMFUL`	returned when no more system memory is available.
`ENOBUFS`	returned when an mbuf of the requested size is not currently available.

See Also

`m_getn()`

m_deq()

Remove First Mbuf Packet

Syntax

```
#include <mbuf.h>
Mbuf m_deq(
  Mbuf      *q);
```

Description

m_deq() removes the first mbuf packet chain from an mbuf queue. m_deq() returns a pointer to the removed mbuf packet chain.

Parameters

q	is a pointer to a pointer to the queue.

See Also

m_enq()

Syntax

```
#include <mbuf.h>
void m_enq(
   Mbuf       *q,
   Mbuf       mb);
```

Description

`m_enq()` appends an mbuf packet chain to the end of an mbuf queue.

Parameters

`q`	is a pointer to a pointer to the original mbuf queue.
`mb`	is a pointer to the mbuf packet.

See Also

`m_deq()`

m_flush()

Free Empty Mbufs in Packet Chain

Syntax

```
#include <mbuf.h>
Mbuf m_flush(
     Mbuf       mb);
```

Description

`m_flush()` frees all empty mbufs in the mbuf packet chain `mb`.
`m_flush()` returns a pointer to the new head of the mbuf packet chain.

Parameters

`mb`	is a pointer to the mbuf packet chain.

See Also

`m_free()`

m_free()

Free Mbuf

Syntax

```
#include <mbuf.h>
Mbuf m_free(
  Mbuf      mb);
```

Description

`m_free()` returns the specified mbuf to the free pool. `m_free()` returns a pointer to the next mbuf in the mbuf packet chain or `NULL` if this was the last mbuf in the packet chain.

Note

If the `SPF_NOFREE` bit is set in the `M_FLAGS` field of the mbuf header then the mbuf is not freed, instead the `SPF_DONE` bit is set in the `m_flag` field.

Parameters

`mb`	is a pointer to the mbuf to free.

See Also

`m_free_m()`
`m_free_p()`
`m_free_q()`

m_free_m()

Return Mbuf to Free Pool

Syntax

```
#include &lt;mbuf.h&gt;
int32 m_free_m(
   Mbuf      mb);
```

Description

`m_free_m()` returns the specified mbuf to the free pool. `m_free_m()` returns 0 if successful. Non-zero return values indicate the system memory data structure is corrupted.

Note

If the `SPF_NOFREE` bit is set in the `M_FLAGS` field of the mbuf header then the mbuf is not freed, instead the `SPF_DONE` bit is set in the `m_flag` field.

Parameters

`mb`	is a pointer to the mbuf to free. You must ensure `mb` is a pointer to an address returned by `m_get()` or `m_getn()`.

See Also

`m_free()`
`m_free_p()`
`m_free_q()`

Syntax

```
#include <mbuf.h>
Mbuf m_free_p(
  Mbuf      mb);
```

Description

m_free_p() returns an entire mbuf packet chain to the free pool. All mbufs in the packet are freed. m_free_p() returns a pointer to the next mbuf packet chain in the mbuf queue or NULL if this was the last packet chain in the queue.

Note

If the SPF_NOFREE bit is set in the M_FLAGS field of the mbuf header then the mbuf is not freed, instead the SPF_DONE bit is set in the m_flag field.

Parameters

mb	is a pointer to the mbuf packet chain to free.

See Also

m_free()
m_free_m()
m_free_q()

m_free_q()

Free Mbuf Queue

Syntax

```
#include <mbuf.h>
void m_free_q(
  Mbuf       *queue);
```

Description

`m_free_q()` returns the entire mbuf queue to the free pool. All mbufs for all packet chains in the queue are freed.

Note

If the `SPF_NOFREE` bit is set in the `M_FLAGS` field of the mbuf header then the mbuf is not freed, instead the `SPF_DONE` bit is set in the `m_flag` field.

Parameters

`queue`	is a pointer to a pointer to the mbuf queue to free.

See Also

`m_free()`
`m_free_m()`
`m_free_p()`

m_get()

Allocate Mbuf (Blocking)

Syntax

```
#include <mbuf.h>
Mbuf m_get(
   u_int16   size,
   int32     *status);
```

Description

Given a size, `m_get()` allocates an mbuf with a data area of `size` bytes. `m_get()` blocks until an mbuf of the requested size is available. If an error occurs, `m_get()` returns `NULL` and sets `status` to the error code. Otherwise, `m_get()` returns a pointer to the allocated mbuf.

Parameters

`size`	specifies the number of bytes to allocate for the data area. `m_get()` can allocate a maximum of 0XFFFF minus the size of the mbuf header, which is 0X010.
`status`	is a pointer to the returned error status code.

Non-fatal Errors

`EOS_ILLARG`	returned when the request is for a size greater than 32767 bytes.
`EOS_MEMFUL`	returned when no more system memory is available.
`ENOBUFS`	returned when an mbuf of the requested size is unavailable.

See Also

`m_getn()`

m_getn()

Allocate Mbuf (Non-Blocking)

Syntax

```
#include <mbuf.h>
Mbuf m_getn(
   u_int16   size,
   int32     *status);
```

Description

`m_getn()` allocates an mbuf with a data area of `size` bytes and returns a pointer to it. If the memory request cannot be granted immediately, `m_getn()` returns `NULL` and `ENOBUFS` is returned in `status`. If an error occurs, `m_getn()` returns `NULL` and sets `status` to the error code. Otherwise, `m_getn()` returns a pointer to the allocated mbuf.

Parameters

`size`	specifies the number of bytes to allocate for the data area. `m_getn()` can allocate a maximum of 0XFFFF minus the size of the mbuf header, which is 0X010.
`status`	is a pointer to the returned error status code.

Non-fatal Errors

`EOS_ILLARG`	returned when the request is for a size greater than 32767 bytes.
`EOS_MEMFUL`	returned when no more system memory is available.
`ENOBUFS`	returned when an mbuf of the requested size is unavailable.

See Also

`m_get()`

m_len_p()

Return Number of Data Bytes

Syntax

```
#include <mbuf.h>
u_int32 m_len_p(
  Mbuf        mb);
```

Description

`m_len_p()` adds up the total number of bytes used by each mbuf data allocation in the packet list. `m_len_p()` returns the total number of data bytes in the mbuf packet chain.

Parameters

`mb`	is a pointer to the packet.

m_move()

Copy Specified Number of Bytes

Syntax

```
#include <mbuf.h>
u_int32 m_move(
   Mbuf          mb,
   u_int32       offset,
   u_int32       count,
   char          *buffer);
```

Description

`m_move()` copies `count` bytes in the mbuf packet chain starting at `offset` into `buffer`. The mbuf remains unchanged. `m_move()` returns the count of bytes actually copied.

Parameters

`mb`	is a pointer to the mbuf packet chain.
`offset`	specifies the number of bytes from the front of the buffer into which to begin copying.
`count`	specifies the number of bytes to copy.
`buffer`	is a pointer to the buffer in which to copy the data.

m_msize()

Return Number of Bytes in Packet

Syntax

```
#include <mbuf.h>
u_int32 m_msize(
  Mbuf      mb);
```

Description

`m_msize()` adds up the total number of bytes used by the entire mbuf packet chain including headers, data, and non-used memory. `m_msize()` returns the number of bytes in the specified mbuf packet chain.

Parameters

mb	is a pointer to the mbuf packet chain.

See Also

`m_len_p()`

m_pad(), M_PAD()

Advance Data Pointer

Syntax

```
#include <mbuf.h>
u_int32 m_pad(
   Mbuf        mb,
   u_int16     count);
void M_PAD(
   Mbuf        mb,
   u_int16     count);
```

Description

`m_pad()` advances the data pointer in the mbuf. `m_pad()` returns `0` if successful and adjusts the data pointer. If the request exceeds the available space, the amount of space available is returned and the data pointer remains unchanged.

A macro, `M_PAD()`, is also available and has the same functionality as `m_pad()`. However, `M_PAD()` does not check to see if the request exceeds the available space.

Parameters

`mb`	is a pointer to the mbuf.
`count`	specifies the number of bytes to advance the data pointer.

See Also

`m_adj()`
`m_unpad()`, `M_UNPAD()`

m_ptod()

Return Pointer to First Valid Byte

Syntax

```
#include <mbuf.h>
char *m_ptod(
   Mbuf      mb);
```

Description

`m_ptod()` returns a pointer to the first valid data byte in an mbuf packet chain.

Parameters

`mb`	is a pointer to the mbuf packet chain.

See Also

`mtod()`

m_pullup()

Ensure First Mbuf Has Data

Syntax

```
#include <mbuf.h>
int32 m_pullup(
  Mbuf      *mb,
  u_int16   count);
```

Description

m_pullup() ensures the first mbuf on the packet chain contains at least count bytes of data. An implicit m_getn() is performed if necessary. m_pullup() returns 0 if successful. The replaced mbufs are automatically freed. It returns a non-zero error code if the m_getn() failed or the mbuf chain did not contain the requested amount of data. In this case, the original mbuf packet chain remains untouched.

Parameters

mb	is a pointer to a pointer to the first mbuf.
count	specifies the minimum amount of data for the first mbuf to contain.

Non-fatal Errors

EOS_ILLARG	returned when the request is for a size greater than 32767 bytes.
EBUFTOOSMALL	returned when count is greater than the memory within the mbuf.
ENOBUFS	An mbuf of the requested size is unavailable.

See Also

m_getn()

mtod()

Get Pointer to Mbuf Data

Syntax

```
#include <mbuf.h>
(t) *mtod(
     Mbuf      m,
     typedef   t);
```

Description

`mtod()` is a macro returning a pointer to the data area in an mbuf packet. `mtod()` returns a pointer with type `t` which points to the first byte of data in the mbuf packet. This pointer is obtained by adding the `m_offset` field of the mbuf packet to the beginning address of the mbuf.

Parameters

`m`	is a pointer to the mbuf.
`t`	is the type with which to cast the returned pointer.

See Also

`m_ptod()`

m_unpad(), M_UNPAD()

Move Data Pointer

Syntax

```
#include <mbuf.h>
u_int32 m_unpad(
  Mbuf      mb,
  u_int16   count);

void M_UNPAD(
  Mbuf      mb,
  u_int16   count);
```

Description

`m_unpad()` tries to move the data pointer so there are `count` bytes at the front of the mbuf. If successful, `m_unpad()` returns `0` and adjusts the data pointer. If the request exceeds the available space, the amount of space available is returned and the data pointer remains unchanged.

A macro, `M_UNPAD()`, is also available and has the same functionality as `m_unpad()`. However, `M_UNPAD()` does not check to see if the request exceeds the available space.

Parameters

`mb`	is a pointer to the mbuf.
`count`	specifies the number of bytes to leave before the data pointer.

See Also

`m_adj()`
`m_pad(), M_PAD()`

Index

Symbols

A

B

C

D

F

G

H

I

L

M

N

O

P

R

S

T

U

V

Z

Product Discrepancy Report

To: Customer Support

FAX: 515-224-1352

From:____

Company:_______________________________________

Phone:____

Fax:_____________________Email:__________________

Product Name:

Description of Problem:

____

____

____

____

____

____

____

Host Platform_____________________________________

Target Platform____________________________________

RadiSys.

MICROWARE SOFTWARE