Home

Using SoftStax®

Version 4.7

RadiSys
THE POWER OF WE

www.radisys.com
Revision A • July 2006

Copyright and publication information

Disclaimer

Reproduction notice

Contents

Chapter 1: Overview

OS-9 Networking Overview ... 8
  OS-9 ... 8
  SoftStax ... 8
  RadiSys Hawk ... 8
System Requirements ... 9
  Protocol Developers ... 9
  Chipset Manufacturers ... 9
  Bridge, Router, Gateway, Internet Equipment Manufacturers ... 9
Architecture and Design Philosophy ... 9
  I/O Design ... 10
  The SoftStax Environment ... 12
Application Environment ... 12
  Application Environment Design Goals ... 12
    Application Development ... 12
    Understandable Applications ... 13
    Network independence ... 13
  Data Structures and Uses ... 13
  API and Services Provided ... 14
    Device-oriented calls ... 14
    Path-Oriented calls ... 14
    Call-Control Calls ... 15
    Data Manipulation Calls ... 15
    Asynchronous Notification Calls ... 15
  Using the Application Environment ... 15
Protocol Stack Framework ... 16
  Design Goals ... 16
    Software Baseline ... 16
    Open Architecture ... 17
    Stack and Layer Interoperability ... 17
    Protocol Stack Development ... 17
    Protocol Stack Add-ons ... 17
    Debugging Real-time Problems ... 17
  Driver Architecture ... 17
    Protocol Driver Data Structures ... 17
    Entry Points of a Protocol Driver ... 18
    Inter-driver Communication Primitives ... 18

Chapter 2: OS-9 Network I/O System Components

Components ... 20
  Protocol and Hardware Drivers ... 20
  Device Descriptors ... 21

Application Programming Interface (API) ... 21
The OS Library ... 21
The ITEM Library ... 22
Network Specific APIs ... 22
SoftStax Source File Directory Structure ... 23

Chapter 3: I/O APIs

ITEM Library Interface ... 26
Overview ... 26
ITEM Philosophy ... 26
Network Independence ... 26
Operating System Independence ... 26
Connection-oriented and Connectionless Networks ... 27
ITEM Definitions Files ... 27
Interactive Multimedia Channel Management ... 27
item.h Structures ... 27
device_type ... 28
addr_type ... 33
notify_type ... 36
Initializing notify_type Structure Fields ... 37
Example Using ntfy_ctl ... 39
ite_cctl_pb ... 42
Creating Your Own Library Call Extensions ... 43
spf_ss_pb ... 44
Using the Parameter Block in Setstat/Getstat Calls ... 45

Chapter 4: The SoftStax Device Descriptor

Descriptors ... 50
Internals ... 50
Example: Create the Proto2 Device Descriptor ... 50
Example: Using Logical Unit Number ... 51
The SoftStax Driver ... 55
Driver Conventions ... 55
Driver Names ... 55
Device Descriptor Names ... 55

Chapter 5: Advanced Programming Topics

SPF Protocol Stacking ... 58
Creating a Protocol Stack on a Path ... 58
Passing a Protocol Stack with an Open Call ... 58
Pushing and Popping ... 59
Push and Pop Details ... 59
Using the PROTSTAK Field ... 60

Chapter 6: Testing Applications and Protocols with SLOOP

About SPLOOP ... 62
Connection Oriented vs Connectionless Descriptors ... 63
Using SPLOOP For Application Testing ... 63
Using SPLOOP For Protocol Testing ... 64
... 66

Appendix A: Examples

Example Applications 68
How to Make an Application 68
Example 1: Standard Telecommunications Application 68
ex1_snd.c 68
Example 2: Using os_lib.l 84
spf_test.c 85
Example 3: Loopback Process Application 91
example3.c 91

Appendix B: Using SoftStax with Multimedia Devices

Networked Multimedia Device Basics 104
Multimedia Device Specifics 104
Multimedia Device Run-Time Model 106
DET Software Configuration 106
ITEM Functions 107
ITEM Interface 107
Control Channel Interface 107
Data Channel Interface 108
Resident Network Communications Protocol 108
BLOB Loader 108
Network Protocol 108
Server Protocol Stack 108
Network Management Entity (NME) 109

1 Overview

SoftStax® is the core component for communications and networking in the Microware OS-9® environment.

For information about... **Go to this page...**

OS-9 Networking Overview 8
System Requirements 9
Architecture and Design Philosophy 9
Application Environment 12
Protocol Stack Framework 16

OS-9 Networking Overview

Before delving into the specifics of the OS-9 networking environment, it is extremely important to understand the overall communications and networking product line.

The OS-9 networking environment includes many integrated components. Together these create an off-the-shelf software solution for communications equipment that is inter-operable with virtually any standard or proprietary network.

This package forms the basis of the OS-9 networking environment. For development, you need a minimum of SoftStax and Microware Hawk. For testing, you also need the OS-9 objects available in the OS-9 Development Kit for your specific target processor.

OS-9

This component, which is specific to your OS-9 target, contains all the binaries, definitions files, and header files needed for developing and testing the OS-9 target.

SoftStax

SoftStax includes the following components:

- ITEMIntegrated Telecommunications Environment for Multimedia—an application programming interface
- SPF Stacked Protocol File manager—the network infrastructure
- SPPROTO a template driver that can be used to create your own protocol drivers
- SPLOOP a loopback driver that emulates connection-oriented or connectionless networks for application and protocol testing, source and objects for a variety of HDLC controllers, and Serial Communications Controllers (SCCs) for integrated microprocessors
- examples sample applications for testing your system
- LAN Communications: TCP/UDP/IP protocols, PPP/SLIP, and hardware drivers for Ethernet chipsets

RadiSys Hawk

RadiSys Hawk is an open, integrated development environment that supports many of the embedded industry’s third-party development tools, such as compilers, in-circuit emulators, debuggers, and testing tools.

System Requirements

Protocol Developers

Protocol developers require OS-9 for Embedded Systems. You can test your protocol stack without requiring a real network by using the SPLOOP driver and creating network side emulation software.

Refer to *Chapter 6, Testing Applications and Protocols with SLOOP* for more detailed information on how to use the SPLOOP driver.

Optionally, the protocol stack or ethernet access to the OS-9 target can be performed using LAN Communications.

Chipset Manufacturers

Chipset manufacturers require OS-9 for Embedded Systems. Source code for SPF hardware drivers is available in SoftStax. This code can be used as a starting point to develop your driver. You can use the SPPROTO template to create your driver, but it is recommended that you use the sp82525 driver as a template. The sp82525 driver is specific to a hardware driver and is, in most cases, a better starting point for your development.

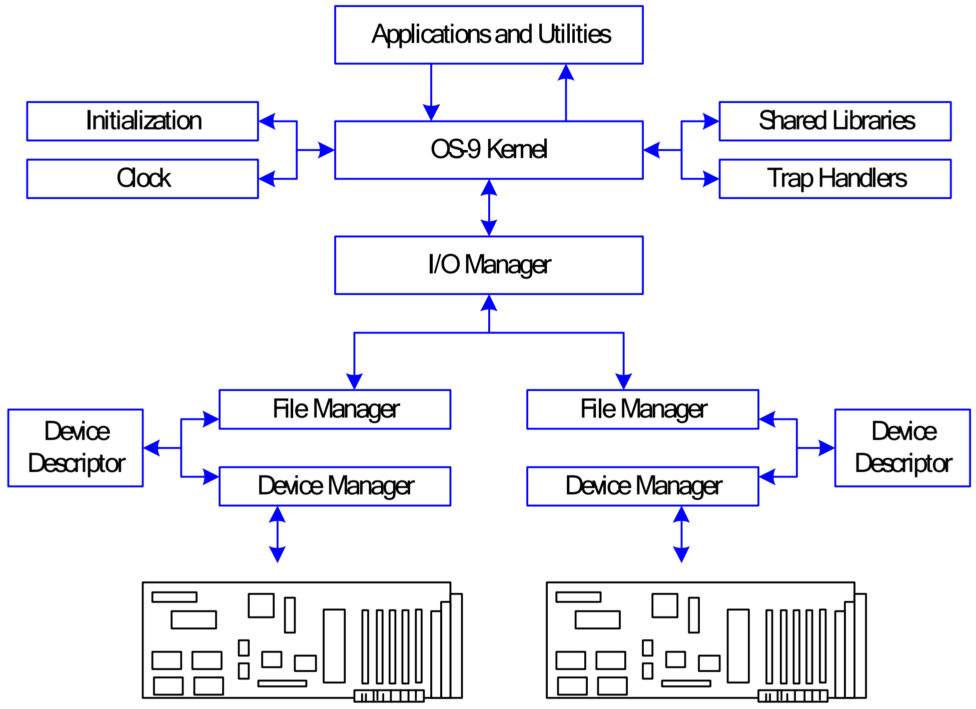
Bridge, Router, Gateway, Internet Equipment Manufacturers

LAN equipment manufacturers require OS-9 for Embedded Systems. Depending on the multi-network operation of the equipment, other communications paks can be integrated providing quick multi-protocol support for any environment.

Architecture and Design Philosophy

OS-9 implements a unified Input/Output (I/O) system. The programming interface used by the application is identical whether the application is using a hard drive, serial device, or network interface. This programming interface consists of calls to open, close, read, write, and set/get (called setstats and getstats) I/O configuration information .

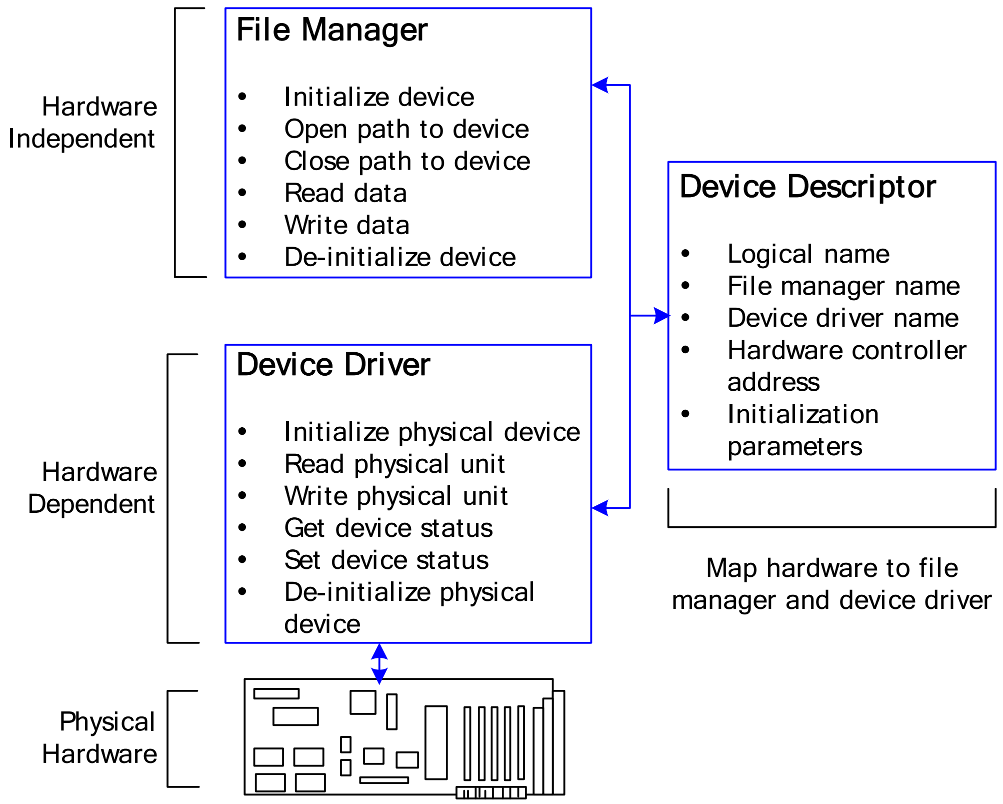
Figure 1-1. OS-9 I/O System



I/O Design

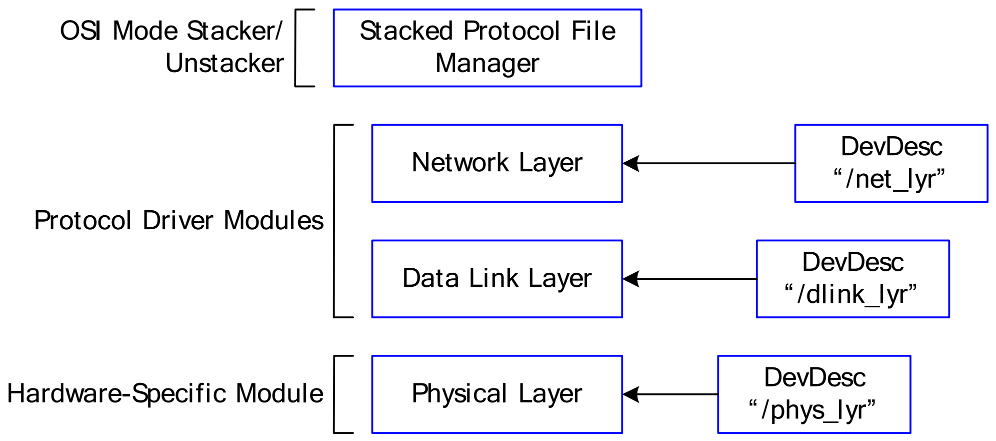
Every I/O system for OS-9 consists of a file manager, device driver, and device descriptor. The file manager performs all logical features of the specific I/O system—implementing the Hardware Abstraction Layer (HAL) for the system. The device driver controls the specific hardware, distilling driver creation down to hardware initialization, termination, and an interrupt service routine. The device descriptor is identifiable by the application that dynamically links all the modules. The application opens a path using a device descriptor module name. The OS-9 then uses the information contained in the device descriptor as a roadmap to create a link between the application, file manager, and device driver. The link created by OS-9 for the application is called a path. The application uses the resulting path to access the services provided by the I/O system. All modules in the system are fully re-entrant and position independent.

Figure 1-2. SoftStax I/O System



SoftStax™ extends the I/O system philosophy by enabling the mapping of more than one driver on a given path; it allows multiple drivers to be stacked onto one another. This extension represents the implementation of the OSI Model as defined by the International Standards Organization.

Figure 1-3. SoftStax and the OSI Seven-Layer Model

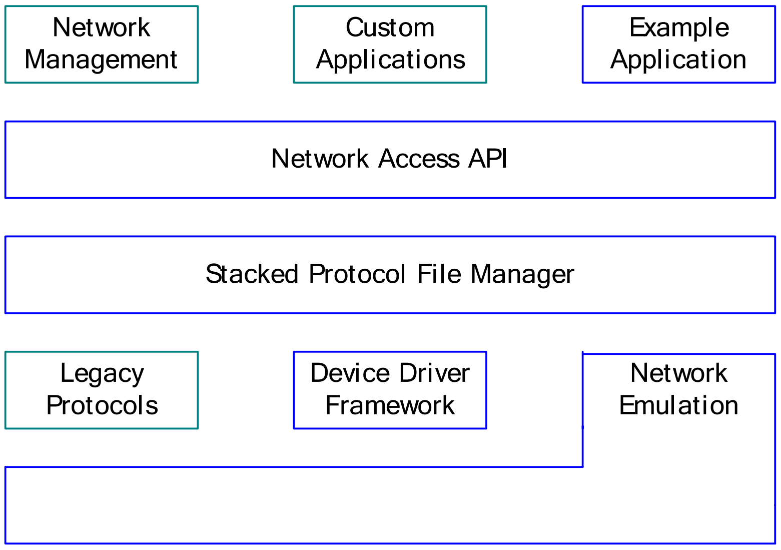


Protocol stack implementation begins with a specification that uses the OSI Model as the abstract framework. This enables a protocol layer implementation to be interoperable with other protocol layer (or protocol driver) implementations for OS-9.

The SoftStax Environment

The SoftStax environment consists of the following: an Application Programming Interface (API) called ITEM (Integrated Telephony Environment for Multimedia), the Stacked Protocol File Manager (SPF), a template protocol driver (SPROOTO), a network emulation driver (SPLOOP), and various HDLC driver implementations. Network-independent application examples are also provided as guidelines for application development.

Figure 1-4. SoftStax Architecture



Application Environment

To write an application for SoftStax you must understand the following concepts:

- application environment design goals
- data structures and their uses
- API and services provided

Application Environment Design Goals

ITEM defines the application environment of OS-9 for Communications and the SoftStax framework.

Application Development

One design goal of ITEM is to eliminate the complexities involved with an application using network services. Applications are not forced to build pieces of network-specific messages and pass them through the API to perform call control. For example, an ISDN application is not required to pass in the channel ID, bearer capability, or low-layer compatibility information elements like parameters in order make a connection. This simplifies the application, frees the application from being network specific, and does not require the programmer to be "ISDN-literate."

Understandable Applications

ITEM is intuitive to the general programmer and does not reflect a specific network protocol state machine.

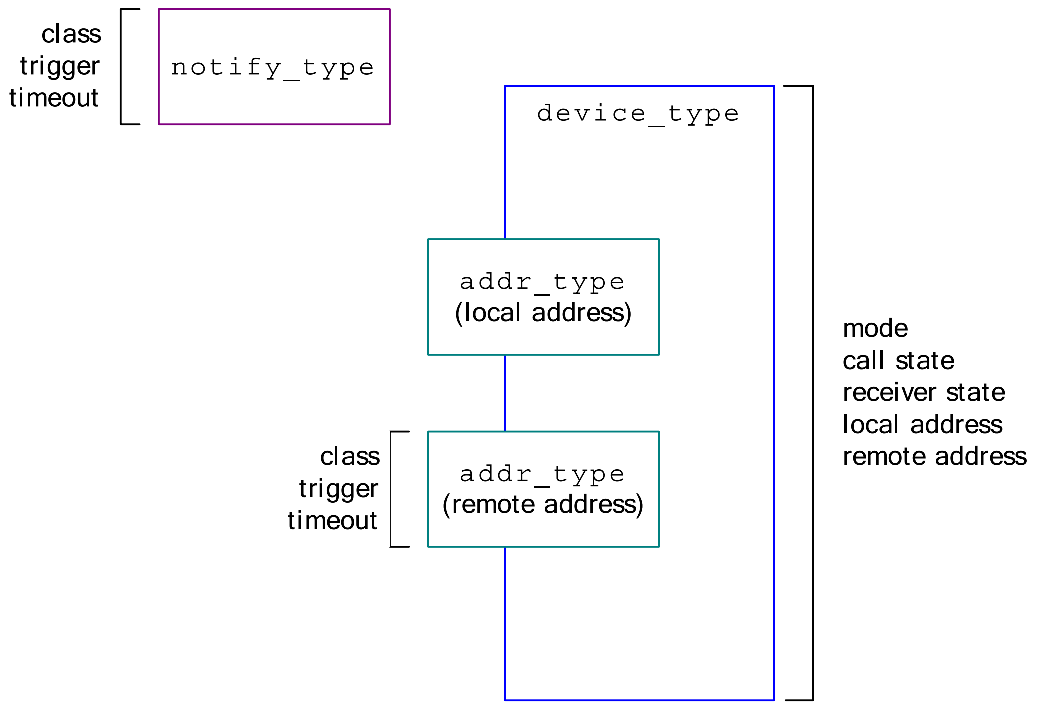
Network independence

ITEM enables application binaries to run across multiple network topologies without recompiling or relinking. Network independence is achieved by abstracting properties of the network.

Data Structures and Uses

Data structures were created to achieve application environment design goals and enable the application to remain network independent. Abstracting application-visible aspects of any network is the key to making network independence a reality. Abstractions for the network device and network addressing were created using structures called `device_type` and `address_type`. The third data structure in ITEM abstracts the asynchronous notification method called a `notify_type` structure. This provides a level of operating system independence.

Figure 1-5. SoftStax Data Structures



The descriptor automatically initializes all of the parameters in the `device_type` and `addr_type` structures when the path is created. Since automatic initialization occurs as an implicit kernel service, applications need not be aware of these two structures. This enables applications in their most simple form to continue operating with ITEM. If required, ITEM contains API calls to get and set all variables within the `device_type` and `addr_type` structures.

Applications use the notify_type structure for network event registration and removal. Notification requests can be set through the ITEM API for the following items:

- link down/link up
- incoming call
- connection active/far-end hangup
- data available to be read
- end of MPEG-II program
- flow control on/off
- custom protocol or device driver network events

API and Services Provided

The API is another important characteristic of the application environment. The ITEM API is modeled after the telephone, a paradigm with which everyone is familiar. ITEM provides scalable capabilities and is simple to use. However, in cases where applications require complex network-specific services, add-on communications paks come with APIs that expose detailed access to particular network topologies. For example, the Microware LAN Communications includes a BSD4.4 compatible socket library. The advantage of this approach is that developers know the level of network independence for the libraries used by the application.

The ITEM API contains five main categories of service:

- device oriented
- path oriented
- call control
- data manipulation
- asynchronous notification

Device-oriented calls

These calls manipulate individual protocol layers or device drivers. They include calls to initialize, terminate individual layers, get the layer name, get the type of service the layer provides, and get and set permissions for a layer.

Path-Oriented calls

These calls manipulate entire protocol stacks for a given path. Also available are calls to open and close incarnations of a protocol stack and to dynamically add and remove protocol layers. Profiles are used to simplify the correct quality of service for connections by the applications. These profiles are identified by the application as primitives (i.e. VOICE, DATA, MPEG, IP, etc.). Therefore, applications can request connections based on a service profile primitive. The protocol layer maps the primitive to the specific connection messages required to create the correct type of connection for the desired service.

Call-Control Calls

This group of calls provides call-control services required for connection-oriented networks. The SoftStax framework allows these calls to be made successfully even if the application is running over a connectionless network for true portability across all types of network topologies.

Data Manipulation Calls

The data manipulation calls enable synchronous or asynchronous reading and writing operation. Zero copy across the user interface is available not just with TCP/IP, but with all SoftStax protocols through the read and write mbuf calls. For convenience, data can also be read by packets or individual bytes.

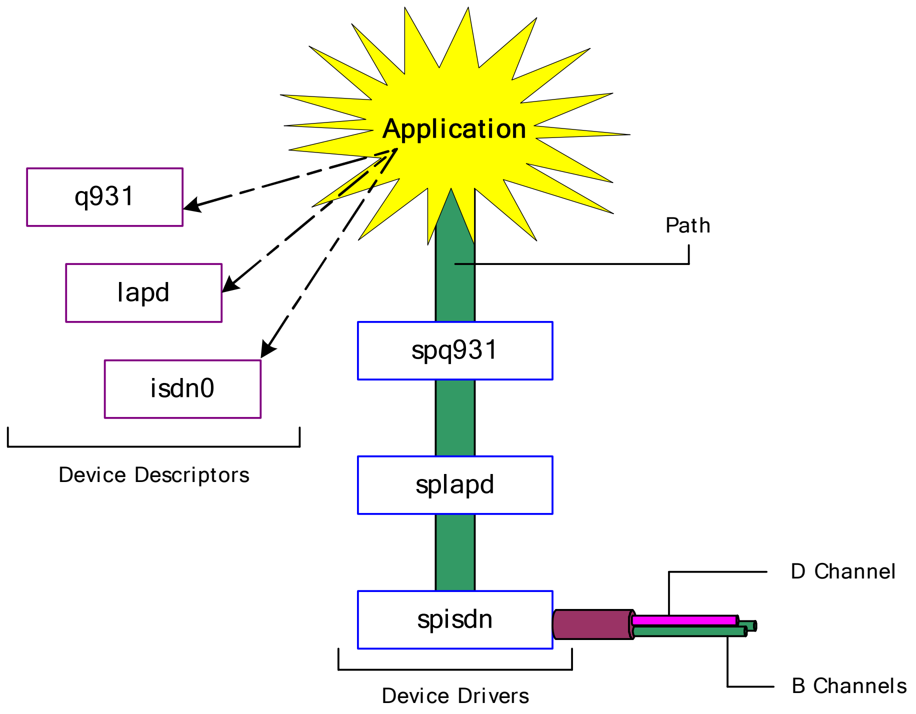
Asynchronous Notification Calls

In addition to the asynchronous calls defined by the previous sections, far-end hangup and protocol stack status change can also be registered by the application. The facility also allows layer-specific notifications if required.

Using the Application Environment

Below is an example of a stack consisting of an ISDN driver, LAP-D data link layer, and Q.931 network layer.

Figure 1-6. Example ISDN Stack



There are three ways the application can invoke this configuration:

- explicitly, 1 call

  ```
  ite_path_open("/isdn0/lapd/q931", READ | WRITE, &pathID, NULL);
  ```
- explicitly, 3 calls

  ```
  ite_path_open("/isdn0", READ | WRITE, &pathID, NULL);
  ite_path_push(pathID, "/lapd");
  ite_path_push(pathID, "/q931");
  ```
- implicitly

  ```
  ite_path_open("/network", READ | WRITE, &pathID, NULL);
  ```
  In this case, the isdn0 descriptor is configured to contain an implicit push of the `/lapd/q931` stack. This descriptor is then named /network. In this manner, the application simply opens /network. In addition, new descriptors containing different protocol stacks can be loaded into the OS-9 system. This method enables the application to run over different network topologies without disruption.

Addressing can be defined by using the '#' delimiter when opening each layer. Referring to the ISDN example above, spisdn uses D channel, splapd uses TEI/SAPI {00}, and spq931 uses 515-223-8000 for their respective addresses. The open call would look like the call below:

```
ite_path_open("/isdn0#D/lapd#00/q931#5152238000", READ|WRITE, &pathID,
NULL);
```

Protocol Stack Framework

To write an application for SoftStax, developers must define the following:

- design goals
- driver architecture
- optimized driver services
- data and control flow through the architecture

Design Goals

SoftStax defines the communications software framework within OS-9 for Communications.

Software Baseline

The SoftStax footprint is 20Kb RAM and 25Kb ROM for all processor architectures. SoftStax is a kernel extension that uses services unique to OS-9 to provide a run-time communications architecture that maximizes performance and minimizes footprint and CPU utilization.

Open Architecture

SoftStax is completely specified and documented to allow all third-party protocol stack companies, OS-9 for Communications users, and hardware driver providers to efficiently implement their technologies for OS-9.

Stack and Layer Interoperability

SoftStax provides one universal framework for every protocol layer. This enables protocols implemented by multiple parties to be interoperable.

Protocol Stack Development

SoftStax provides a protocol layer template driver, a network emulation driver, timer services, and buffer management services.

The template driver provides a "null layer" implementation to which a protocol state machine can be immediately added. The network emulation driver enables validation of protocol stacks without requiring access to the network. Timer services and buffer management services are also provided.

Protocol Stack Add-ons

Communications software development requires integration of an RTOS, application, one or more protocol stacks, and device drivers--all written to different frameworks. Through this, SoftStax enables developers to immediately understand a common baseline regardless of the OS-9 for Communications product add-on.

Debugging Real-time Problems

SoftStax provides a facility for tracing the events that lead up to real-time bugs. This facility is provided through a debugging library for real-time execution capture.

Driver Architecture

Protocol Driver Data Structures

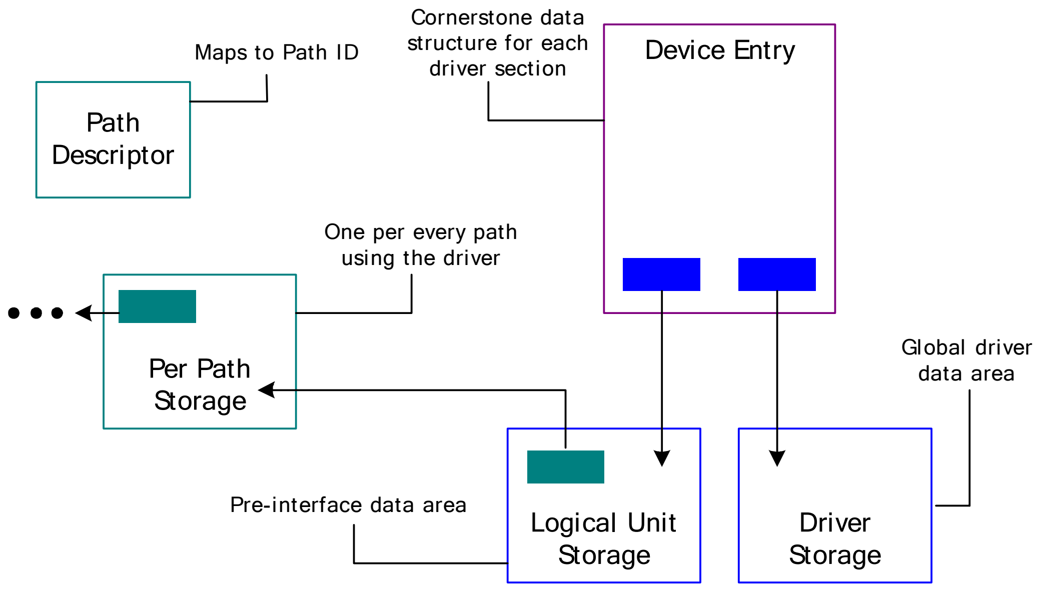
The OS-9 kernel provides automatic allocation and initialization of data structures for drivers. This service is used by OS-9 for Communications to allocate and initialize data areas for protocol drivers without requiring creation of code to allocate and initialize data areas.

OS-9 automatically creates four data structures for a driver, including the following:

- device entry
- driver storage
- logical unit storage
- path descriptor

A library is also provided to create a per path data structure for the driver, called the per path storage.

Figure 1-7. SoftStax Driver Architecture



Entry Points of a Protocol Driver

All ITEM API calls are realized at the driver layer as DrGetstat and DrSetstat calls. Parameter blocks are formatted with the ITEM service request and associated parameters. For device drivers, the DrUpdata entry point is not used, and an interrupt service routine, which can be considered as the incoming data entry point for a device driver, is implemented.

Inter-driver Communication Primitives

Inter-driver communications primitives are implemented not as inter-process communication, but as direct jumps to the entry point of the driver above and below. This aspect is the key to a high-performance system.

The `DR_FMCALLUP_PKT` macro minimizes the amount of time spent in an interrupt service routine by queuing the data on a receive queue for processing by the receive process.

2 OS-9 Network I/O System Components

This chapter provides an in depth look at the components that comprise OS-9 Network I/O.

For information about... **Go to this page...**

Components 20
SoftStax Source File Directory Structure 23

Components

SoftStax enables a wide variety of network devices to connect to local and wide-area networks. For low-end network devices, SoftStax may be all that is required to connect a device to a network. SoftStax components consist of a file manager—Stacked Protocol File manager (SPF), protocol drivers, hardware drivers, device descriptors, application interface libraries, and example applications. These components provide the flexibility needed to create network hardware and protocol independence.

SPF is the OSI seven-layer model stacker that application data uses to travel through OS-9 and the network protocol stack. It provides the ability to stack and unstack drivers used by the application.

SPF performs the following tasks:

- stacks and unstacks protocol drivers on a given OS-9 path
- queues receive data for each path
- blocks and unblocks applications attempting to read or write data
- provides the standard OS-9 I/O (input/output) interface

Protocol and Hardware Drivers

Protocol drivers typically implement a protocol state machine as defined by some national or international standards body. For example, LAP-B and UDP are typical protocols. Protocol drivers share the same characteristics as hardware drivers.

The term *driver* used without qualification is applicable to both hardware and protocol drivers. Protocol drivers are usually software protocol state machines and hardware drivers interface directly to network hardware.

SoftStax provides a template protocol driver called SPPROTO, which serves as the starting point for developing new protocol drivers.

The base package also provides source and objects for a variety of HDLC controllers and serial communications controllers found in integrated microprocessors.

SoftStax enables you to create and completely test client/server network applications without having access to the network. This is accomplished by opening paths to the SPLOOP driver provided in SoftStax. There following lists three types of descriptors the application can open:

- straight loopback
- descriptors emulating a connection oriented network
- descriptors emulating a connectionless network

Using a single OS-9/SoftStax target machine, the applications can execute and interact with each other while SPLOOP emulates the correct type of network. Applications can be completely validated in this manner, which drastically reduces development time for networked equipment and greatly reduces the cost of developing the application.

Refer to *Chapter 6, Testing Applications and Protocols with SLOOP* for more information on using the SPLOOP driver.

Device Descriptors

Device descriptors contain default information for a given driver. A device descriptor might contain time-out values, initial values having to do with a particular protocol state machine, or hardware initialization variables for hardware drivers. Optionally, a string can be embedded in the device descriptor telling the file manager which protocol drivers and hardware drivers to use when a path is opened using this descriptor.

Refer to *Chapter 4, The SoftStax Device Descriptor* for more information on creating and modifying device descriptors.

Application Programming Interface (API)

APIs provide an easy-to-use interface enabling applications to interact with other applications over various types of networks. Some APIs are general purpose and allow applications to communicate with one another, independent of the network architecture.

The Integrated Telecommunications Environment for Multimedia (ITEM) is a general purpose API provided with SoftStax. The goal of this API is to provide network independent multimedia access.

Other APIs are specific to a particular type of network. The `socket.l` API library is an example of this. An application using this type of library has the disadvantage of becoming less portable across networks. However, the application using a network-specific API has access to and can take advantage of special services the network provides.

The OS Library

The OS-9 operating system environment has an API named `os_lib.l`. This library is used for access to all OS-9 I/O systems, not just SoftStax.

The following table outlines the most commonly used calls from this library with a brief description.

Table 2-1. os_lib.l I/O Calls

Call	Description
`_os_attach()`	Initialize a device
`_os_detach()`	Deinitialize a device
`_os_open()`	Open a path to a device

Table 2-1. os_lib.l I/O Calls (Continued)

Call	Description
`_os_close()`	Close a path to a device
`_os_ss_sendsig()`	Send a signal when data is available to be read on this path
`_os_gs_ready()`	Return the number of bytes available to be read on this path
`_os_read()`	Read data
`_os_write()`	Write data

- Refer to the section Creating Your Own Library Call Extensions in *Chapter 3, I/O APIs* for more information about using this library to create your own specialized API access to a specific network protocol stack.
- Refer to the *Ultra C/C++ Library Reference Manual* for more information about these calls.

The ITEM Library

SoftStax has its own special API called ITEM. This API allows the same access as the OS library, but extends the API to include network independent call control, channel management, MPEG stream information gathering, and notification on different kinds of network events.

The ITEM library is covered in more detail in the section in *Chapter 3, I/O APIs*.

Network Specific APIs

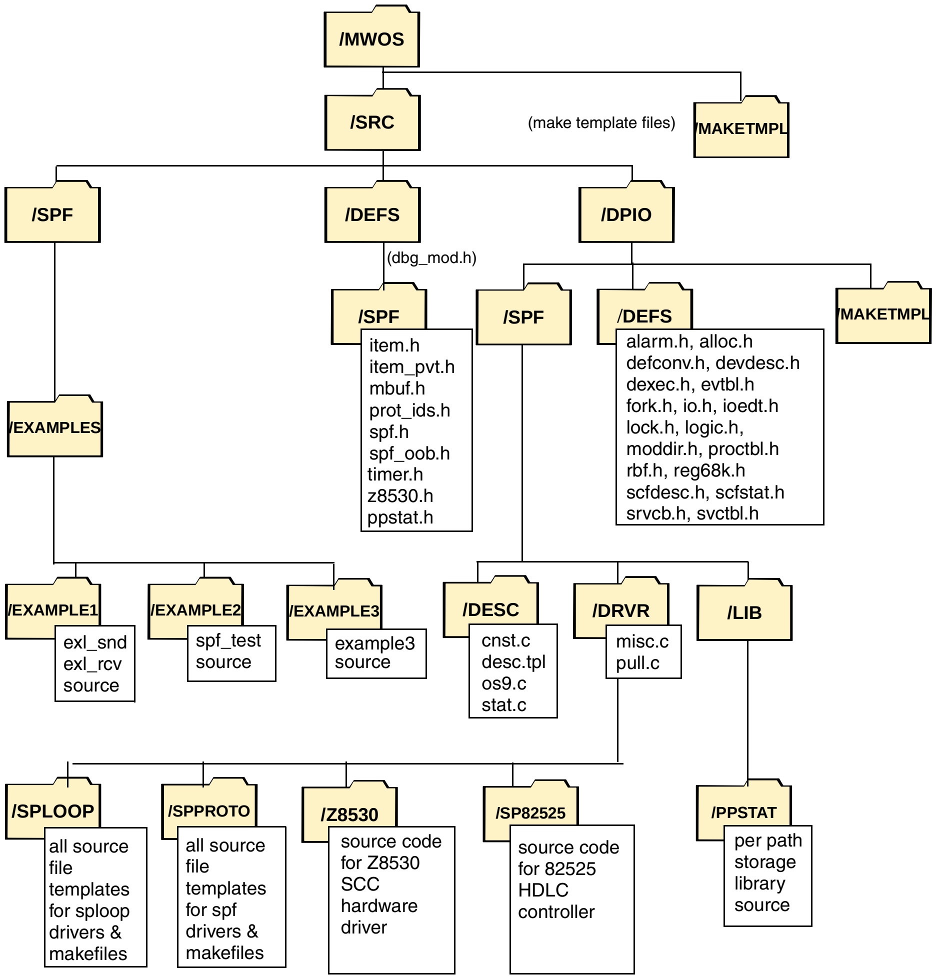
Most SoftStax Communications using a specific protocol stack provide a special API to access the unique properties of that specific protocol stack. However, there are trade-offs involved for using these APIs.

If an application only uses ITEM to communicate, it is guaranteed the ability to run over any network architecture and to be interoperable with Communication Paks available for OS-9/SoftStax. Once APIs specific to a particular network are used by the application, the application may be able to take more sophisticated control of the network. However, because the application is dedicated to a specific network, it is more difficult to port the application and have it work correctly on a different type of network.

SoftStax Source File Directory Structure

If you are porting or creating an application, the templates for source code and makefiles can be found in the EXAMPLES subdirectory as shown in the following figure:

Figure 2-1. Source File Directory Structure for SoftStax



Typically, applications have two places in which their objects can be placed. The examples place their objects in the local EXAMPLES/CMDS/PPC directories. Your application may place the object under the appropriate CMDS directory under the processor family type (such as MWOS/OS9000/PPC/CMDS).

3 I/O APIs

This chapter discusses the functions and structures available in ITEM and `os_lib.l`.

For information about...	Go to this page...
ITEM Library Interface	26
Creating Your Own Library Call Extensions	43

ITEM Library Interface

This section provides an overview of ITEM (Integrated Telecommunications Environment for Multimedia), describes the ITEM library, and discusses each component of the `item.h` file in the order listed in that file.

Overview

The ITEM library provides the application with a network-independent application programming interface. Since an application can use generic call control library calls to communicate with the SoftStax I/O system, it does not need to know the type of network being used. This is important if application portability is a requirement. You can use ITEM to communicate with, and over, connection-oriented or connectionless networks.

ITEM also provides calls specific to the digital television environment. These calls include operations for channel management and MPEG program control and can be found in the Microware Digital Broadcast Environment (DBE) Communications Pak. Extensions to ITEM are also available in other SoftStax Communication Paks.

Most standard networking APIs expose too many network details and are far more complex than they need to be. Application programmers do not see programming to a network interface API as a fun exercise, but as a necessary evil. ITEM allows the programmer to write applications with just a few calls, but also has the flexibility to allow more detailed API access if needed.

Unlike most network APIs, ITEM uses a paradigm with which everyone is familiar: the telephone. As a result, the API is very intuitive. Call indicator applications connect, register for far-end hang-up, interact, and then disconnect. Call receiver applications register to be notified on incoming calls. Once notified, the call can be screened, answered, registered for far-end hang-up, interacted with, and then disconnected from.

ITEM Philosophy

Before going through the ITEM library, it is important to understand the philosophy behind this API.

The goal of ITEM is to provide a network independent and operating system independent API. If these two things are the goal of the API, then ITEM must abstract issues that could potentially be network or operating system specific.

Network Independence

In order to provide network independence, the API must abstract the connection management, addressing, and the network device and protocol stack.

Operating System Independence

In order to provide operating system independence, the asynchronous notifications (signals, semaphores) must be abstracted.

Connection-oriented and Connectionless Networks

Before data can be sent and received by the two network endpoints, the network must perform call control to create a connection between those endpoints. Once this connection is established, data follows the same route to and from those endpoints. As a result, once the end-to-end connection is set up, the round-trip delay is known. This is a connection-oriented network similar to the standard telephone system.

A connectionless network does not perform call control procedures for sending data between endpoints. An end-point wraps the data with the source and destination address of the packet and sends it to the network. In turn, the network keeps sending the packet along until it reaches its destination.

Round-trip delay is an important characteristic of a connectionless network. Caused by packets traveling between the same source and destination endpoints, but using completely different routes, connectionless networks have an unpredictable round-trip delay. The internet and Internet Protocol (IP) networks are examples of connectionless networks.

ITEM Definitions Files

The `item.h` file in the `MWOS/SRC/DEFS/SPF` directory that contains the core functionality applications must have to use the ITEM interface. This file contains all structures, macros, and function prototypes for device control, path control, and connection control over the network.

Interactive Multimedia Channel Management

The ITEM API also provides channel management and MPEG program control calls for the digital TV industry. These calls can be found in The Microware Digital Broadcast Environment (DBE) Pak available for use with SoftStax.

item.h Structures

The `item.h` file contains two core support structures and one substructure:

- `device_type` (contains the substructure `addr_type`)
  The `device_type` structure provides a network device abstraction for the application, enabling the application to deal with all network devices identically, regardless of the network device specifics.

- `notify_type`
  The `notify_type` structure enables the application to customize the method by which it should be notified when specific network events occur. This allows for operating system independence.

The following pages show the declarations for these structures.

device_type

The `device_type` structure contains general information about the network type, the current call state of the device, our-end and far-end address information, and a display array.

As described in *Chapter 4, The SoftStax Device Descriptor*, the device descriptor provides initial values for the path as it opens the device. An application that is unaware of the type of network it is running on is not required to know anything about the network device. It only needs to know the name of the device descriptor so it can open the path.

The application may assume that the device descriptor it used to open the path has all the right values for connecting to the far-end. Smarter applications enabling the user to set new addresses use the device type mechanism to allow unique addressing and call state information regardless of the network specifics.

The `open` call does not establish the connection. An `ite_ctl_connect()` must be executed before the connection is established. For connectionless networks, only an open call is needed to send and receive data through ITEM.

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_RSVD0x00
       #define ITE_ASGN_NONE0x01
       #define ITE_ASGN_THEIRNUM0x02
       #define ITE_ASGN_ANY0x03
       #define ITE_ASGN_PROFILE0x04
    u_char       dev_rsv1;
    u_int32      dev_rsv2
    addr_type    dev_ournum,dev_theirnum;
    char         dev_display[ITE_MAX_DISPLAYSIZE];
} device_type, *Device_type;
```

Fields

`dev_mode`

Characterize the mode of the device (for example, readable or writable). Legal values are: `FAM_READ`, `FAM_WRITE`, and `FAM_NONSHARE` found in `modes.h`.

`dev_netwk_in/dev_netwk_out`

Allow for independent characterization of the input and output sides of the network device. This enables an ITEM network device to behave differently, with respect to data transmission and reception characteristics.

Typically, both transmit and receive sides are of the same class (as defined in `item.h`). However, in asymmetrical networks (such as those used in some interactive TV trials), the transmit side of the network device may be one type, while the receive side is another. Legal values of these parameters do not describe specific networks, but identify a generic category for the network device.

Values for `dev_netwk_in/dev_netwk_out` are as follows:

Value	Description
`ITE_NET_CTL`	Control channel A description of the network device in the interactive TV environment. This device provides upstream data transmission from set top box to server.
`ITE_NET_DATA`	High-speed data device Similar to `ITE_NET_MPEG2`, but having the ability to receive high-speed data in formats other than MPEG-2.
`ITE_NET_MPEG2`	MPEG-2 Data channel network device A network device receiving the high-speed MPEG-2 data and delivering it to the transport demultiplexer chip or Stream Control Block (SCB) buffers for display of the audio and video data by the MPEG I/O system.

Value	Description (Continued)
`ITE_NET_CHMGR`	Channel management network device This device is used only to receive channel entry information from the network.
`ITE_NET_OOB`	Signalling device This network device is used primarily to perform signalling with the network and to establish the end-to-end connections requested. No end-to-end data is sent or received on this device.
`ITE_NET_VIPDIR`	Video Information Provider network device Device used to only receive VIP directory information.
`ITE_NET_SESCTL`	Session Control Device responsible for performing session setup/termination, and miscellaneous functions in the Interactive TV environment.
`ITE_NET_ANY`	All-purpose device General all-purpose device providing some multiple of the above network interfaces described by the above values.

`dev_callstate`

Shows the current call state of the device.

The call state values are defined as bit fields. This allows applications to be notified when one (of a group of call states) is reached.

For example, if a protocol driver allows notification when a specific call state is reached, you would write code similar to the following:

```
notify_on_callstate(ITE_CS_ACTIVE|ITE_CS_CONNTERM)
<attempt to dial>
<wait for notification>
```

In this case, you are notified when your dial was answered and you are connected. You are also notified if the call was unanswered and a timeout occurred.

When notification is received, the application performs an `ite_ctl_connstat()` to determine which state the connection is in since notification was set up for `ITE_CS_ACTIVE` and `ITE_CS_CONNTERM`.

Value	Description
`ITE_CS_IDLE`	No end-to-end connection There is no valid end-to-end connection at this time and none is attempting to be established.
`ITE_CS_INCALL`	Incoming call An incoming call has come in on this device.
`ITE_CS_CONNEST`	Connection setup in progress The device is in the process of setting up a connection.

Value	Description (Continued)
`ITE_CS_ACTIVE`	Active connection The device has an active connection.
`ITE_CS_CONNTERM`	Connection termination in process The device is terminating a connection.
`ITE_CS_CONNLESS`	Connectionless device The device is connectionless. Therefore, no call control is needed to send and receive messages over this network device.
`ITE_CS_SUSPEND`	Call Suspended Call has been suspended (put on hold).

`dev_rcvr_state`

Present unique challenges within the operating system. For instance, network devices can initiate I/O by receiving a far-end connection request. Because any endpoint on the network has the ability to initiate a connection, you must deal with this type of asynchronous behavior. The `dev_rcvr_state` indicates whether anyone is registered to receive notification for an incoming call on this network device.

When the `ite_ctl_rcvrasgn()` API call is executed, SoftStax logs the assignment type in this field.

Value	Description
`ITE_ASGN_NONE`	No assignment If an incoming call comes in on this device, it is ignored because nobody has registered to receive notification of the incoming call.
`ITE_ASGN_THEIRNUM`	Assign their number This device listens for an incoming call request, but only notifies the path registered if the incoming call address information matches the `dev_theirnum` address information in the `device_type` structure. This is similar to providing a call screening mechanism for an application.
`ITE_ASGN_ANY`	Any address The registered path receives notification of any incoming call regardless of the calling address.
`ITE_ASGN_PROFILE`	Assign on matching profile The registered path receives notification of all incoming calls that match the profile setting for the path. Refer to the section, in the *SoftStax Porting Guide*, *Out-Of-Band Considerations With ITEM* for more information.

`dev_rsv1`

Reserved for future use.

`dev_rsv2`

Reserved for future use.

`dev_ournum/dev_theirnum`

Contain address information for our-end and far-end. This address information is contained in the `addr_type` structure within `item.h`. The `addr_type` structure contains generic information about the address, as well as a character string containing the address.

`dev_display`

This is the location at which protocols typically store display information such as caller ID when making connections. The application can retrieve this via the `ite_ctl_connstat()` ITEM call, then display the information.

addr_type

Declaration

The `addr_type` substructure 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;
```

Fields

`addr_class`

Defined as shown in `item.h` (in `MWOS/SRC/DEFS/SPF`).

The address class does not imply the protocol used to make the connection. For example, an address class of E.164 does not imply the ISDN Q.931 protocol will be used for the signalling, even though Q.931 does use E.164 addressing. Standard telecommunication also uses E.164, as do many other protocols. The format for the E.164 addr_class is: (xxx)yyy-zzzz, where xxx is the area code, yyy is the local code, and zzzz is the number for the specific end point.

Value	Description
`ITE_ADCL_NONE`	No addressing.
`ITE_ADCL_UNKNOWN`	Unknown address class.
`ITE_ADCL_E164`	E.164 address specification.
`ITE_ADCL_INET`	Standard Internet (`sockaddr` structure) addressing.
`ITE_ADCL_RSV1`	Reserved for future use.
`ITE_ADCL_ATM_ENDSYSTEM`	Asynchronous Transfer Mode (ATM) addressing.
`ITE_ADCL_LPBK`	Loopback addressing using logical units.
`ITE_ADCL_NSAP`	Network Service Access Point Addressing. This is a unique 20 byte value used for session control addressing in the interactive TV environment.
`ITE_ADCL_DTE`	Data Terminal Equipment address (LAP-B control byte).
`ITE_ADCL_DCE`	Data Communications Equipment address (LAP-B control byte).
`ITE_ADCL_LAPD`	Terminal Endpoint (TEI), Service Access Point (SAPI) 2 octet pair.

`addr_subclass`

Indicate that a particular address class may contain sub-classes.

Value	Description
`ITE_ADSUB_NONE`	No subclass address.
`ITE_ADSUB_UNKNOWN`	Unknown subclass.
`ITE_ADSUB_VC`	Virtual circuit.
`ITE_ADSUB_PVC`	Permanent virtual circuit.
`ITE_ADSUB_LUN`	Sub-address is based on the logical unit number (based on the `sploop` driver).
`ITE_ADSUB_SLINK`	Single-link address (point-to-point).
`ITE_ADSUB_MLINK`	Multi-link address (point-to-multipoint).

`addr_rsv1`

Reserved for future use.

`addr_size`

Contain the number of valid bytes in the `addr[]` array.

`addr[32]`

Contains the specific address value. The application can set and read this array. The `addr[]` field should be interpreted based on the address class and sub-class as described in the `addr_class` and `addr_subclass` fields.

For example, the string 2267786 translates into the following bytes in the `addr[]` array: `addr[0x32/0x32/0x36/0x37/0x37/0x38/0x36]`

If the address class is labelled as `ITE_ADCL_E164`, this address is interpreted as an E.164 address or 226-7786. If the class is UDP/IP, it is interpreted as a `sockaddr` structure, address family 0x3232, port address 0x3637, IP address 0x37.0x38.0x36.0x00.

notify_type

The `notify_type` structure provides an abstraction for notification of all network asynchronous events. OS-9 provides various ways to notify applications of asynchronous events, including sending a signal, setting events, or callbacks.

The `notify_type` structure enables the application to customize how it is notified of asynchronous events. When the application uses asynchronous calls, a pointer to the `notify_type` structure is passed as one of the parameters so SoftStax knows to send the correct notification information to the application, as well as to use the correct method when sending the notification. Also, when applications allocate and send the `notify_type` structure as a parameter of the asynchronous function call, SoftStax copies the contents of the structure passed and keeps its own copy of the structure in the path storage area.

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;
```

Initializing notify_type Structure Fields

Macros are defined in `item.h` to easily fill out `notify_type` structures. For example, if you want to be notified of an incoming call, allocate a `notify_type` structure using the `ITEM_SIGNAL_NOTIFY()` macro to acquire signal number 1000 and a timeout value of 500 ticks, when SoftStax detects an incoming call.

```
    notify_type incall_npb;
    NPB_INIT_SIG (incall_npb, 500, 1000);
```

At this point use the `ite_ctl_rcvrasgn()` call, which passes a pointer to the `notify_type` structure you built.

SoftStax copies the information into a local `notify_type` structure in the path descriptor structure.

If you change any of the parameters in your `notify_type` structure after the receiver assignment call is made (such as putting a 2000 in the `sig2send` parameter), the change does not affect the copy stored by SoftStax. You still get signal 1000 sent to you when SoftStax detects an incoming call.

Fields

`ntfy_next`

Enable SoftStax to chain notifications. This should always be set to `NULL`, which is handled automatically by the `NPB_INIT_xxx` macros.

`ntfy_class`

Identify the type of notification you want to receive. This field is set to `ITE_NCL_SIGNAL` by the `NPB_INIT_SIG` macro, and to `ITE_NCL_EVENT` by the `NPB_INIT_EV` macro.

Value	Description
`ITE_NCL_BLOCK`	Block waiting for notification.
`ITE_NCL_SIGNAL`	Send a signal.
`ITE_NCL_EVENT`	Set an event.
The following field values are reserved and not implemented:	
`ITE_NCL_MMBOX`	Send a MAUI mailbox message.
`ITE_NCL_CALLBACK`	Call a callback function.
`ITE_NCL_SIGINC`	Send incrementing signals.
`ITE_NCL_EVENTINC`	Send incrementing events.

`ntfy_on`

Identify the trigger event telling SoftStax on which asynchronous event you want to be notified. You may be familiar with some of these events, while others are new because of unique network environments.

Value	Description
`ITE_ON_LINKDOWN`	Notifies the caller if the end-to-end link goes down. If communication fails at any layer of the protocol stack, a link-down notification is sent.
`ITE_ON_INCALL`	Notifies the caller of an incoming call. Only the network layer protocol driver implements this notification based on the `dev_rcvrstate` field description.
`ITE_ON_CONN`	Notifies the caller when an outgoing call connection is established and active.
`ITE_ON_DATAVAIL`	Notifies the caller when incoming data is received and available to be read (send a signal on data ready).
`ITE_ON_ENDPGM`	Notifies the application when a specific MPEG-2 program is being viewed. This is an MPEG-2 specific call.
`ITE_ON_FEHANGUP`	Notifies the application if the far-end initiates a hang-up.
`ITE_ON_DNLDONE`	Notifies the initiator after the client-side application is downloaded. An interactive TV-specific notification.
`ITE_ON_MSGCONF`	Notifies application that the confirmed message was successfully received by the far-end.

Value	Description (Continued)
`ITE_ON_RESADD`	Notifies application when requested resources have been added. This call is used in the interactive TV environment for session control.
`ITE_ON_LINKUP`	Notifies the caller if the link comes back up after being down.
`ITE_ON_FCTLON`	Not used.
`ITE_ON_FCTLOFF`	Not used.

`ntfy_rsv1`

Reserved for future use.

`ntfy_ctl_type`

Imply the structure of the `ntfy_ctl` pointer.

`void *ntfy_ctl`

Indicate some notifications may need to have special parameters passed to the driver to implement a specific or protocol-defined notification. In this case, a structure is created so the application and protocol driver can agree on the parameters. Next, a notify type value is assigned to indicate the parameter being passed in `ntfy_ctl()` is a pointer to a specific structure.

Example Using ntfy_ctl

Suppose James wants to be notified if Mark, Staci, Darrin, or Mike calls. James needs a protocol driver supporting this notification. Follow the procedure listed below to create such a driver:

Step 1. Create a structure that both James and the protocol driver acknowledge.

```
typedef struct ntfy_names
{
char *namelist[4];
u_int32 namelist_count;
char caller[8];
} ntfy_names, *Ntfy_names;
```

Step 2. Define a `ntfy_ctl_type` value (user-defined values are 0xA0 through 0xFE).

```
#define NTYPE_NAMES   0xA0
```

Step 3. Point the `namelist` pointer to the string names of Mark, Staci, Darrin, and Mike in the `namelist` string array.

Step 4. Place a `4` in the `namelist_count` field.

Step 5. Place the pointer to the `ntfy_names` structure in the `ntfy_ctl` field of the notify parameter block.

Step 6. Fill `ntfy_ctl_type` with `NTYPE_NAMES`.

Step 7. Execute the protocol-specific notification by the application.

If the protocol driver has the notification implemented properly, James receives notification and is able to look in the caller field of the `ntfy_names` structure and find out who just called him.

`ntfy_timeout`

Handle time-outs (in seconds) for notification requests. For example, if you make a phone call, you usually let the phone ring a set number of times before you decide no one is home and hang-up. The timeout parameter sets a limit on the amount of time to wait for the notification trigger to occur. If the timeout value is 0, you wait indefinately or until the trigger event occurs.

`ntfy_rsv`

Reserved for future use.

`Notify union`

Depending on the value in the `ntfy_class` field of the `notify_type` structure, the application selects one of the following unions (`sig` or `ev`) to respond to a notification.

`sig`

Of type union and depends on the notification class. The structure for this signal provides the process ID and signal to send. The proc_id field is automatically specified by SoftStax, so the application only needs to set the value in the sig2send field. The sig2send field is filled automatically by the `ITEM_SIGNAL_NOTIFY()` macro.

`proc_id`

The process identifier for the application. Typically, the application does not know the process IDs, so SoftStax sets this field for the application.

`sig2send`

Indicate the signal number to send. Microware reserves signal codes up to 255, so this number should be greater than 255.

`ev`

Of type union and allows applications to have events set for notification. The application is expected to create the event, and specify the event ID and event value fields. The `ev` structure is automatically filled out by the `ITEM_EVENT_NOTIFY()` macro.

`ev_id`

The event identifier. This value is returned as a result of an `_os_ev_creat()` call.

`ev_val`

The event value set when the asynchronous trigger event occurs.

`inc_ev`

Of type union and allows applications to set incrementing events for notification.

`ev_id`

The event identifier. This value is returned as a result of an `_os_ev_creat()` call.

`ev_inc_val`

Event value is incremented by this amount each time the notification is triggered.

`mmbox`

Not currently implemented.

`mmbox_handle`

Not currently implemented.

`callback_func`

Not currently implemented.

`callbk`

Of type union and allows applications and drivers to be notified via a callback function.

`callbk_param`

Point to a parameter to pass to the callback function.

`callback_func`

Points to the function to call.

The Ultra C manuals contain a thorough explanation of events and how to use them. Contact your Microware sales representative for information on how you can obtain these manuals.

ite_cctl_pb

This structure is the call control parameter block. It is used to support out-of-band signalling through ITEM. Within ITEM, the `notify_type` structure supports standard connectivity functions like `ite_ctl_connect()`, `ite_ctl_disconnect()`, and the `ite_ctl_answer()`. When creating the notify type structure, the `ntfy_ctl_type` must be set to `NTYPE_SESSCTL` and the `ite_cctl_pb` must be allocated with the `ntfy_ctl` pointer pointing at it. This enables end-to-end user data to be passed between the caller and receiver during the signalling procedures allowed by many protocols. If none of these special parameters are needed, the `ntfy_ctl` field can be set to `NULL` and not used.

Declaration

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

```
typedef struct ite_cctl_pb {
    void          *ib_reslist;
    u_int16       response,
                  reason,
                  rsv1,
                  usr_data_cnt;
    void          *usr_data;
} ite_cctl_pb, *Ite_cctl_pb;
```

Fields

`ib_reslist`

When using out-of-band signalling protocols, the resulting in-band resources allocated (as a result of the connection) is returned in this field when the `ite_ctl_answer()` call is executed.

`response/reason`

Contain network-specific information about the result of the request. For example, as defined by your network, you can send `ite_ctl_answer` and instruct SoftStax to send a `ConnectFailure` response by reason of `LocalNodeBusy`. Likewise with `ite_ctl_answer()`, codes can be returned to indicate success or failure of the operation. These codes are network-dependent and are defined in communication packs that support these fields.

`rsv1`

Reserved for future use.

`usr_data_cnt/usr_data`

Send data opaquely through the network to the endpoint you are attempting to signal. For instance, during the `ite_ctl_connect()` call, you can pass “Hello Mike” as user data to the endpoint at which you want to connect. User data sent from a remote site is placed on your path’s receive queue and could be read if you performed an `ite_data_read()` call on your path.

Creating Your Own Library Call Extensions

`os_lib.l` is the standard operating system library provided with OS-9 to access all file managers, including SPF. These file managers all use `_os_lib.l` calls to access SPF and the drivers below it.

You can also create your own extensions to the API libraries. For example, to create a protocol driver with special functions that applications need to access, do the following:

Step 1. Use the `_os_setstat()` and `_os_getstat()` calls provided in `_os_lib.l` and the `spf_ss_pb` structure in `spf.h`.

The `_os_setstat()` and `_os_getstat()` calls require the following parameters:

`Path ID`

contains the path ID returned when you open the device with the `_os_open()` or `ite_path_open()` call.

`Code`

must always contain the code `SS_SPF`. The application uses this code to communicate with SPF.

`Parameter block`

contains the address to the parameter block structure required by the driver. The following section explains how to create and use the parameter block. SPF assumes the parameter block has the structure of `spf_ss_pb` as defined in the `spf.h` file.

Step 2. Add the SoftStax driver base code or protocol ID to the `port_ids.h` definitions file.

spf_ss_pb

Declaration

The `spf_ss_pb` structure is declared in the file `SPF/spf.h` as follows:

```
typedef struct spf_ss_pb {
u_int32       code,
              size;
void          *param;
u_int8        updir;
   #define SPB_GOINGUP 1
   #define SPB_GOINGDWN 0
u_int8        rsv[3];
} spf_ss_pb, *Spf_ss_pb;
```

Fields

`code`

The value of your special getstat or setstat code as outlined below.

`size`

The size in bytes of what the param field points to (if needed).

`param`

A pointer to an application buffer or structure.

`updir`

Always set to `SPB_GOINGDWN`.

`rsvis`

Reserved for future use.

Using the Parameter Block in Setstat/Getstat Calls

The best way to illustrate how to use the parameter block is by example. Suppose you wrote a hardware driver controlling the interface chips for the network. This chip has a special feature that generates Dual Tone Multi-Frequency (DTMF) tones. You want to include this feature in your hardware driver, but the libraries you are using do not contain generic calls.

The steps to create a generic call are listed below:

Step 1. Create a library called `dtmflib.l`.
It must contain one call that you create called `dtmf_send()`. The parameters passed are the path identifier and the DTMF tone number
{0-9,#,*}. Your function prototype might look like this:

```
dtmf_send(path_id path, u_char tone_val);
```

The goal is to organize the information passed into the call into the SPF parameter block and use the `_os_setstat()` call to send it to your driver.

Step 2. Assign a protocol type value for your hardware driver. The `spf.h` file (see the `MWOS/SRC/DEFS/(SPF_PR_xxx)` directory) has a listing of the values supported by Microware, as well a list of values to be released with the next release of the SoftStax. There is also a file (`prot_ids.h` in the `MWOS/SRC/DEFS/SPF` directory) used specifically by users to register their protocol and hardware drivers.

The `spf.h` file specifies that user-defined protocol IDs range from 0x0900 through 0xFFFF. Locate a free value and place it in `prot_ids.h`. For example:

```
#define SPF_PR_MYDRVR 0x0900
```

Step 3. Create a file (call it `mydrvr.h`) applications can include (just as `item.h` is included in `MWOS/SRC/DEFS/SPF`) to access the special features of your driver.

Step 4. Define the base value for your driver-specific setstats and getstats:

```
#define SS_MYDRVR_BASE(SPF_PR_MYDRVR << 16)
```

The above statement means any time your protocol driver gets a code with `SPF_PR_MYDRVR` in the high-order word of the code, this setstat/getstat is intended to be serviced by your driver.

Step 5. Add your setstats:

```
#define SS_DTMF_SEND SS_MYDRVR_BASE + 0x01

/* You might want to later allow for turning*/

/* DTMF on/off */

#define SS_DTMF_ON SS_MYDRVR_BASE + 0x02

#define SS_DTMF_OFF SS_MYDRVR_BASE + 0x03

and so on...
```

Step 6. Establish the conventions for the parameter block getting passed. This tells your driver where to find the variables within the parameter block. For the `dtmf_send()` call, place `SS_DTMF_SEND` in the `spf_ss_pb.code` field.

Step 7. Place the tone number in the `spf_ss_pb.param` field. The code might look something like the following:

```
error_code dtmf_send(path_id path, char tone_val)
{
   spf_ss_pb spb;


   /*
```

Step 8. Integrity check the parameters passed in. In this example, the only legal values for `tone_val` are 0-9 and # and *. Check it here and return an illegal argument error if the parameter is not valid.

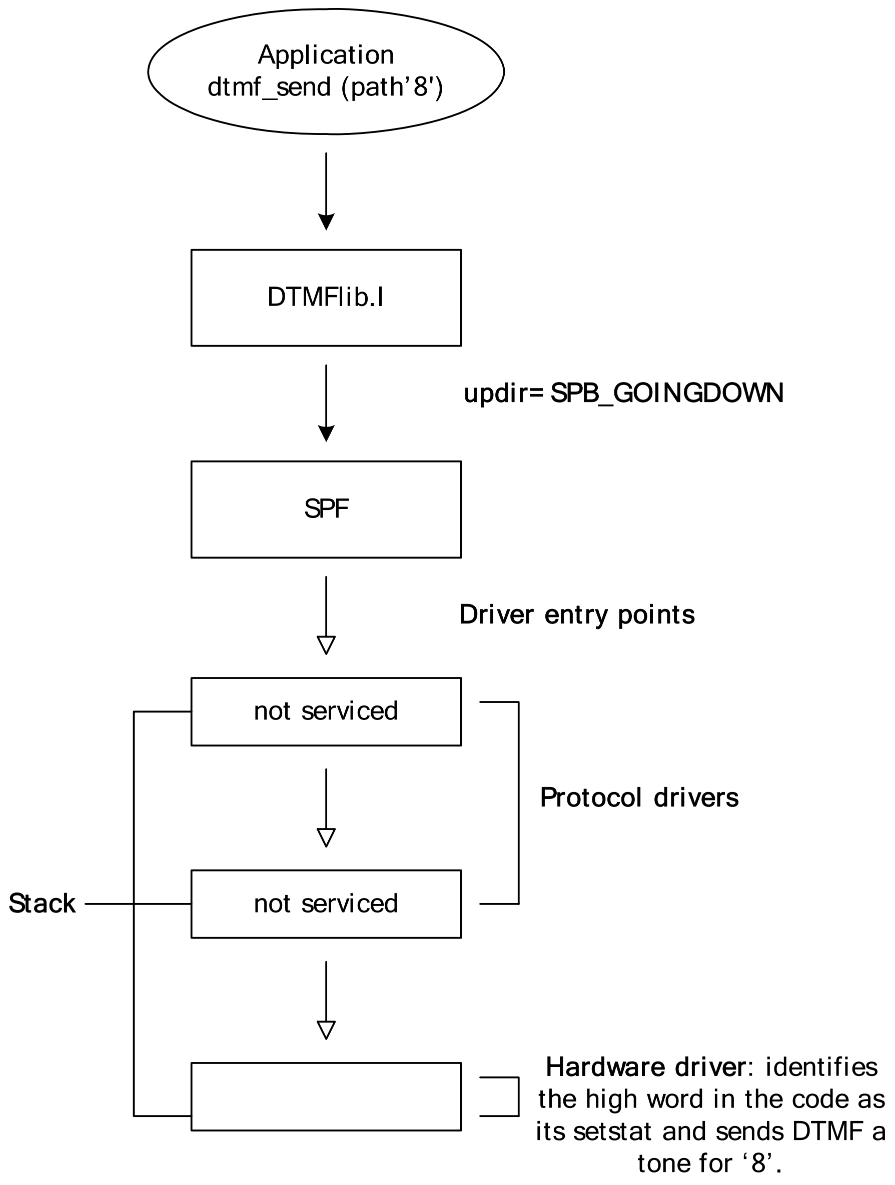
```
*/
 spb.code = SS_DTMF_SEND;
 spb.param = (void *)tone_val;
 spb.updir = SPB_GOINGDWN;


   return(_os_setstat(path, SS_SPF, &spb);
}
```

Step 9. Enter your `updir` value.

This example does not explain the `updir` field in the `spf_ss_pb`. `Updir` indicates the direction of the `SetStat` or `GetStat`. If `updir` = 1 (or `SPB_GOINGUP` as defined in `spf.h`), the driver below is passing up a `SetStat` request. If `updir` is `0` (or `SPB_GOINGDOWN` as defined in `spf.h`), the requester is above you in the stack. Therefore, the `updir` value tells the protocol driver processing the request to send the results of the operation either up or down the stack. Libraries always set this parameter to `SPB_GOINGDWN`.

Figure 3-1. Effect of updir Field Value



In this case, the fields provided in the `spf_ss_pb` are sufficient to hold all variables passed. However, if you have multiple parameters to pass, it is preferred that you define your own parameter block. The first structure in the parameter block must be the `spf_ss_pb` (as defined in `spf.h`). This is because SPF processes all subcodes (a subcode is the value you place in the `spb.code` field of the parameter block) expecting the `spf_ss_pb`.

4 The SoftStax Device Descriptor

This chapter looks at device descriptors and SoftStax drivers.

For information about...	Go to this page...
Descriptors	50
The SoftStax Driver	55

Descriptors

The device descriptor provides the following:

- default information that SoftStax requires in order to open a path on for an application
- all of the information required to initialize the network device
- default values for the path descriptor structure
- default initial values for the logical unit of the driver

To create or modify the device descriptor, change the parameters located in the `spf_desc.h` file.

The `spf_desc.h` file can be found in one of two places, depending on the driver to which it belongs. A hardware device descriptor is found in the `PORTS` directory containing the makefiles used to make the driver and descriptor. For example, you can find an example `spf_desc.h` file in:

```
MWOS/OS9000/ARMV4/PORTS/SIDEARM/SPF/SPE509/DEFS
```

If it is a protocol driver, the `spf_desc.h` file is found in the `DEFS` directory in the source directory of the protocol driver. For example, the `spf_desc.h` file for spproto is found in:

```
MWOS/SRC/DPIO/SPF/DRVR/SPPROTO/DEFS
```

Internals

The example `spf_desc.h` file in this section belongs to the SPPROTO driver. Display this `.h` file on your workstation.

The first thing to notice is the `spf_desc.h` file includes `item.h`. This is because the `item.h` file contains many of the macros needed to initialize the item structures in the path descriptor.

You also see that each logical section within this file begins with `#ifdef <descriptor name>`. This is because this single file creates all device descriptors for a given SoftStax driver. The makefile templates to make these descriptors are set up to include the section of the `spf_desc.h` file `ifdef`ed with the intended name of the descriptor.

Example: Create the Proto2 Device Descriptor

To create a device descriptor named `proto2`, cut and paste a section between the `#ifdef` and `#endif` of the `spf_desc.h` file and change the `#ifdef` to `#ifdef proto2`. When you make the `proto2` device descriptor, the only section of the `spf_desc.h` file used during this compile is contained in the `#ifdef proto2` section.

Look at the values within one of these sections. For this example, you can look at the `#ifdef proto` section in the `spf_desc.h` file for SPPROTO. It is located in:

```
/MWOS/SRC/DPIO/SPF/DRVR/SPPROTO/DEFS
```

The first macro to define is `PORTADDR`. Since protocol drivers usually do not have a hardware port address, set this to `0`. For hardware drivers, this is the port address of the hardware.

The next macro is `LUN` (logical unit number). A logical unit is storage allocated to a driver containing specific information for a given hardware port.

For example, assume you have a circuit board with four serial controllers on it, but only one common interrupt register. Make descriptors `/sp0` through `/sp3` for each of the ports. Each descriptor has the same base port address value because of the common interrupt service routine registers. The driver static would probably contain fields relating to this interrupt logic. However, you cannot store variables specific to a given port in the driver static or another port; this may overwrite the data stored for a different port. This area, then, is where you can use the logical unit number field.

Set the logical unit number as follows:

Table 4-1. Logical Unit Number

Description	LUN
`/sp0`	0
`/sp1`	1
`/sp2`	2
`/sp3`	3

Example: Using Logical Unit Number

if the `LUN` is different, OS-9 allocates unique logical unit storage for each descriptor. Therefore, variables common to all ports are stored in the driver static storage area, and variables specific to a port are stored in the logical unit static.

For hardware drivers, a given port is usually a uniform offset away from its previous and next ports. For example, if a base address of the circuit board described above is at 0xA0000, then each of the four serial ports is located at 0xA0010, 0xA0020, 0xA0030, and 0xA0040 respectively. The port address for the circuit board would be 0xA0000. For convenience, specify the `LUN`s for `/sp0` through `/sp3` as 0x10, 0x20, 0x30, and 0x40.

The logical unit might contain a pointer to the register map structure representation of one instance of the serial port. When the receive interrupt service routine occurs, the base port address interrogates the interrupt-reason registers to find out which of the four ports generated the interrupt. Once this is known, the `LUN` for the interrupting port is added to the base port address and this address is used to access the registers of the channel generating the interrupt.

In this example for `proto`, `LUN` is set to 0x7F.

For more information on logical units see the Logical Units section in the *SoftStax Porting Guide*.

Following along in the `spf_desc.h` file:

`MODE`

sets the device mode. Most modes are set for read and write capability as shown in the macros from `MWOS/<os>/SRC/DEFS/modes.h`.

`ASYNCFLAG`

initializes the pd_ioasync variable in the path options. SPF uses a variable to determine whether or not to block. If the `IO_READ_ASYNC` bit is set in the `pd_ioasync` field, the read side of path is in asynchronous mode. When a read occurs on the path and no data is available, SPF returns an `EWOULDBLOCK` error instead of doing an `_os_sleep()` in the file manager. The reading and writing data paths can be independently controlled by only setting either the `IO_WRITE_ASYNC` bit or `IO_READ_ASYNC` bits. If the `IO_WRITE_ASYNC` is not set, SPF blocks if the application does a write and no mbufs are available. If set, the application returns an `EOS_NOBUFS` error instead of blocking the write.

`PKTFLAG`

initiates the `pd_iopacket` variable in the path options. This variable is a bit field used by SPF to determine packet-oriented operation for the path. Setting this field to `0` (`IO_CHAR`) causes normal character-oriented operation (for example, the requested number of bytes to read is returned regardless of packet boundaries).

- If the `IO_DGRAM_TOSS` bit is set in this field and the reader only reads a portion of the current packet, the rest of the packet is tossed (UDP Datagram operation).
- If `IO_NEXPKT_ONLY` is set and the requested read size is larger, SPF returns only the contents of the next packet.
- If `IO_PACKET` is set and the requested read size is larger, SPF returns all available mbuf packet chains. If there is no data available at the time of the read call, SPF will block one time to wait for incoming data. For example, the read queue has four packets of ten bytes. The user is requesting 80.

For `IO_CHAR`, 40 bytes is read and the application blocks for a timeout period specified by BLOCKTIME. If ASYNCFLAG is set, SPF returns 40 bytes and an EWOULDBLOCK error.

- For `IO_PACKET`, SPF returns 40 bytes without blocking.
- For `IO_NEXPKT_ONLY`, SPF returns ten bytes.

The next example shows the same four packets of ten bytes; the application performs a read of five bytes and `IO_DGRAM_TOSS` is set:

SPF returns five bytes, and the next five bytes in the packet would be thrown away. The resulting queue would have three packets of ten bytes.

`BLOCKTIME`

initiates the pd_iotime variable in the path options. This indicates how long to block in the `read()` operation if no data is available. The `BLOCKTIME` should be set to the number of ticks to wait for incoming data. If the `read()` operation times out before being fulfilled, SPF returns the number of bytes read along with the buffer the bytes were read into. In this case, the error code `ETIMEDOUT` is returned.

`READSZ`

is used for flow control. If the `READSZ` macro is defined as `0`, SPF does not perform receive-buffer flow control. It will not attempt to prevent receive packets from coming in during an overflow condition.

If `READSZ` is not `0`, and the number of bytes waiting to be read equals the `READSZ` value, SPF issues an `SPF_SS_FLOWON` setstat to the drivers. In turn, a driver implementing a flow control mechanism tells its peer to stop sending data until the application reads the data below the `READSZ` threshold. When the application reads the data below the threshold, SPF issues an `SPF_SS_FLOWOFF` setstat to the drivers below. The driver implementing the flow control mechanism (as above) tells its peer to continue transmitting.

`WRITESZ`

is used for flow control on the transmit queue of the driver. The hardware driver must enforce this value.

This functionality only works if there is a device driver implementing flow control in the stack.

The next group of parameters initializes the `device_type` structure ITEM uses for the path.

Refer to the ITEM definitions in *Chapter 3, I/O APIs* for information on these parameters.

`PROTSTAK`

allows the application to open a device name such as `/proto`. Within the `/proto` descriptor, the `PROTSTAK` macro could be defined as `/lapb/x25a`. This way, the application does not need to be aware of the protocol stack. It can open a generic device name and have the stack configured within the device descriptor. If the stack changes, the application object is untouched. You change the `PROTSTAK` macro to the new stack and recompile the proto descriptor.

It is important to remember if you open `/proto` and `protstak =/a/b`, the stack being opened is actually `/proto/a/b`.

`DRV_NAME`

is the driver name string. This is defined as SPPROTO for the `/proto` descriptor.

`TXSIZE`

determines the Maximum Transmission Unit (MTU) for this driver. If a protocol says the maximum amount of data to send in one packet is 100 bytes, this field should be set to 100.

`TXOFFSET`

tells SPF how many bytes to leave free at the beginning of the transmit packet so the protocol has enough room for the header. For example, LAP-B uses two bytes for a header, followed by the payload. In this case, LAP-B sets the `TXOFFSET` macro to `2`.

`TXTRAILER`

tells SPF how much room (in bytes) is needed for the encapsulation at the end of a transmit packet. When SPF creates an mbuf for transmission, the size of the mbuf will be the payload + `TXOFFSET` + `TXTRAILER`.

`PATH_HOLDONCLOSE`

allows the path to stay open even after the application has called `close()` to allow the protocols to gracefully terminate.

Refer to the *SoftStax Porting Guide* for detailed driver information. When making descriptors, it is best to consult the driver documentation to see if the `PATH_HOLDONCLOSE` macro should or should not be set.

`PROTTYPE`

sets to the defined protocol ID value in either `spf.h` or `prot_ids.h`.

Look at `spf.h` and find the place in the listing that says:

```
Device descriptor Macro definitions
```

This part of `spf.h` defines default values to the macros in `spf_desc.h`. Because of this, you have the option to omit a macro from the `spf_desc.h` file if you want its value to be set to the default value provided in `spf.h`.

The last line in the section is `#include <SPPROTO/defs.h>`.

Typically, the `defs.h` file within the driver source file contains the logical unit specific structure and the initialized data for that structure. This is why it is included here. However, if each descriptor has its own default setup for each logical unit that depends on the descriptor (such as in the sp8530 driver), put the macros for those variables into the `spf_desc.h` sections individually.

For example, assume you want the first port on your serial card to run at 4800 baud and the second to run at 9600 baud. Use a variable called `lu_baudrate` in your logical unit specific structure. In the initialized data section of the logical unit for `defs.h`, use a macro named `BAUDRATE`. Next, in the section for `/sp0` in `spf_desc.h` you could place `#define BAUDRATE 4800`. In the `#ifdef sp1` section of the `spf_desc.h`, put `#define BAUDRATE 9600`.

Refer to the *SoftStax Porting Guide* for more information about the `defs.h` file.

Review the source files for making the descriptor in `SPF/DESC`. The `makefiles` use the `spf_desc.h` file to automatically make these source files. Reviewing the `makefiles`, the `DESC` source files, and the `spf_pdstat` and `spf_desc` structures in `spf.h` provides you with a more in-depth understanding of the structure of an SoftStax device descriptor.

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 cards. The hardware driver is always on the bottom of the protocol stack for a path.

The protocol driver does not usually interface directly to any hardware. Typically, it is a state-machine implementation that processes incoming and outgoing data according to a 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 developing 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. The Digital Broadcast Environment Pak includes a real-time driver named `rt_drvr` and uses descriptor `rt0`.

Device descriptors for protocol drivers are slightly different. They are typically labelled by just the suffix of the protocol driver they describe and a number or letter suffix. 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.

5 Advanced Programming Topics

This chapter explains how SoftStax stacks protocols on a path, what options are available to the application, and the purpose and internal details for the SoftStax device descriptors and SoftStax device drivers.

For information about...	Go to this page...
SPF Protocol Stacking	58

SPF Protocol Stacking

The SPF manager controls protocol stacks on paths by initializing the storage areas of each driver. When a protocol stack is opened, the driver looks in its respective storage area to find the upper and lower layer protocols and establishes communication.

Creating a Protocol Stack on a Path

There are three methods used to create a protocol stack on a path:

- passing a protocol stack explicitly with an open call
- pushing and popping
- using the `PROTSTAK` field of the device descriptor (using `spf_desc.h`)

Passing a Protocol Stack with an Open Call

Create a protocol stack by passing in a protocol stack string in the open call. For example, to open an X.25 connection over an sp8530 serial controller chip, your open call might look like the following:

```
_os_open("/sp0/lapb0/x25a", mode, &path);
```

This causes SPF to perform the following:

1. Parse the device string by first opening the `sp0` device associated with the 8530 hardware device driver.
2. Stack the LAP-B device driver on top of the `sp0` device.
3. Stack the X.25 driver on top of the LAP-B device driver.

When the caller writes a packet, it is encapsulated with an X.25 header, then a LAP-B header, and is passed to the 8530 chip. The 8530 chip appends the CRC and transmits the packet.

You can also pass addressing information within the string. The "#" character is used to delimit this. For instance, if you want the far end address X.25 dials to be 8888, you open the following name:

```
/sp0/lapb0/x25a#8888
```

This passes the addressing information to the X.25 driver.

Be careful with this method. Certain protocol drivers may not understand addresses and therefore can not use the data behind the # character. Consult the documentation for the driver you are using for details on whether it implements the # delimiter.

Pushing and Popping

Use the `ite_path_push()` and `ite_path_pop()` calls to dynamically link and unlink the SoftStax protocol stacks on the SPF path.

Instead of explicitly opening the stack in the previous example, we could create the stack step-by-step if we used the following pseudo-code:

```
ite_path_open ("/sp0", mode, &path, NULL);
    /* first driver on the stack */
ite_path_push (path, "lapb0");
    /* Now two drivers stack on the path */
ite_path_push (path, "/x25a#8888");
    /* The stack is complete */
```

You can also push and pop protocols dynamically on an existing protocol stack. Using the previous example, assume you have an open X.25 path to far-end 8888. You are communicating with the far-end and now you must send credit card information. You do not want to send this information in plain text format, so you need some means of encrypting the data.

Now, assume you have an RSA protocol driver and descriptor labelled `/rsa1`. Execute `ite_path_push()` to link the RSA protocol driver at the top of your path's protocol stack. At this point, you do an `ite_data_write()` of your credit card information. The data being written is encrypted by the RSA driver, then passed down the X.25 stack as before. When you are done sending the secure information, perform an `ite_path_pop(path)` and the RSA protocol driver is unlinked from your path. You are now back to the X.25 protocol stack on your path.

Push and Pop Details

This section describes some important items that are required when pushing and popping protocols.

SoftStax allows the application to pop protocols off the stack until it reaches the last (or bottom) driver on the stack. When the bottom of the stack has been reached, subsequent attempts to pop the last driver on the stack causes SoftStax to return an `EOS_BTMSTK` error. If popping causes an `EOS_NOSTACK` error, there are probably no drivers associated with the path. This is checked and verified by SoftStax during the `pop()` call. SoftStax also assumes the application knows what it is doing when the stacks are being created on a path. Therefore, if the application stacks the protocols incorrectly, SoftStax attempts to process the data as the protocols were stacked. For example, opening a protocol stack such as `/proto/proto/proto` is legal, although it is not recommended.

There are no checks in place to review a protocol stack for redundancies, or to verify the stack for incorrectly stacked protocols.

Using the PROTSTAK Field

A third alternative is to use the `PROTSTAK` field in the `spf_desc.h` descriptor to specify the protocol stack to use. This allows the application to open a device according to its functions, without knowing about the protocol stack.

For example, an application might open a device descriptor called `/channel_mgr`. Within this device descriptor, the `PROTSTAK` field in the `spf_desc.h` used to make `/channel_mgr` might specify `/sp0/lapb0/x25a`. For another network, the `/channel_mgr` device might have a stack of `/sp0/Q2110/Q2931`. This allows the application to be completely portable.

Refer to *Chapter 4, The SoftStax Device Descriptor* for details about the `PROTSTAK` field.

6 Testing Applications and Protocols with SLOOP

This chapter examines setting up and using the SPLOOP driver for testing applications.

For information about... **Go to this page...**

About SPLOOP 62
Using SPLOOP For Application Testing 63
Using SPLOOP For Protocol Testing 64

About SPLOOP

The sploop driver enables protocol drivers and applications to be tested without needing access to the network. The SPLOOP driver consists of one SoftStax driver and 5 descriptors. The loopc0 and loopc1 descriptors are used to open connection oriented paths to SPLOOP. Example 1 provided with SoftStax shows how these descriptors are used. The loopcl5 and loopcl6 descriptors are used to open connectionless paths to SPLOOP. The loop descriptor opens a direct loopback path through SPLOOP. The number at the end of the descriptor indicates the Logical Unit Number (LUN) the descriptor uses. Each SPLOOP descriptor can only be opened once. Subsequent opens to the same decscriptor returns an `EOS_DEVBSY` error. Since the ITEM API uses a telephone paradigm, you can think of the logical unit number as the phone number for the SPLOOP descriptor.

Refer to *Appendix A, Examples* for more information about the examples.

The sploop driver uses the LUN as its ITEM addressing. If you look at the `spf_desc.h` file in the `SPLOOP` directory, you see these five descriptors initialize their ITEM addressing using `ADCL_LPBK` and `ADSUB_LUN`. The `our_addr` is the LUN of this descriptor. For connectionless descriptors, the `their_addr` contains the LUN that receives the data when the application or protocol sends data down the stack. For connection oriented descriptors, the `their_addr` is the LUN that called if an `ite_ctl_connect()` call is made. The `their_addr` is not used for the loop descriptor since data goes down and comes up the same path.

The SPLOOP driver keeps an array of logical unit static pointers. Every time a path is opened using an sploop descriptor, a pointer to the logical unit static is stored in the array indexed by the LUN of the logical unit. For example, when loopc1 is opened, the logical unit static created by opening loopc1 is stored in the [1] element of the logical unit array in SPLOOP. Array elements zero through four contain connection oriented logical units. Array elements five through ten contain connectionless logical units. Thus you can only create five connection oriented descriptors without changing the SPLOOP source code. You can create up to six connectionless descriptors. Creating an SPLOOP descriptor with a LUN greater than ten defaults that descriptor to being straight loopback.

The connectionless logical units can only communicate with another connectionless logical unit specified by the `their_addr` of the logical unit. Connection oriented paths can only call other connection oriented paths.

SoftStax sets these descriptors up so loopcl5 pairs with loopcl6 and loopc0 pairs with loopc1.

Connection Oriented vs Connectionless Descriptors

Paths opening connection oriented descriptors must perform call control to connect to another connection oriented path using SPLOOP before they can read or write data. Paths opening connectionless descriptors can immediately send and receive data assuming there are open paths to both sides of a connectionless descriptor pair.

If an application is being developed for a connection oriented network, the application client and server should use the loopc0/loopc1 pair for testing. Applications being developed for connectionless networks should use the `loopcl5/loopcl6` pair. Applications being developed for both, should test with both pairs.

Protocol driver testing always uses the `loopcl5/loopcl6` pair.

Using SPLOOP For Application Testing

Figure 6-1 and Figure 6-2 show how client server applications use SPLOOP to test on connectionless and connection oriented paths.

Figure 6-1. Typical Application Test Setup--Connection Oriented Environment

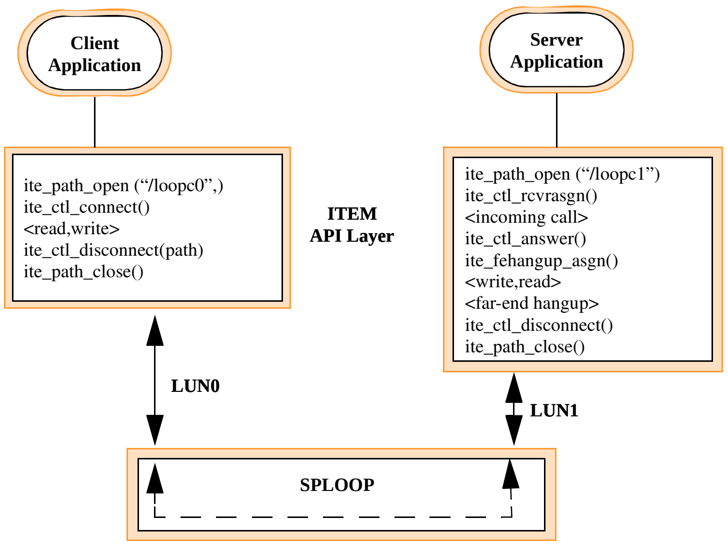
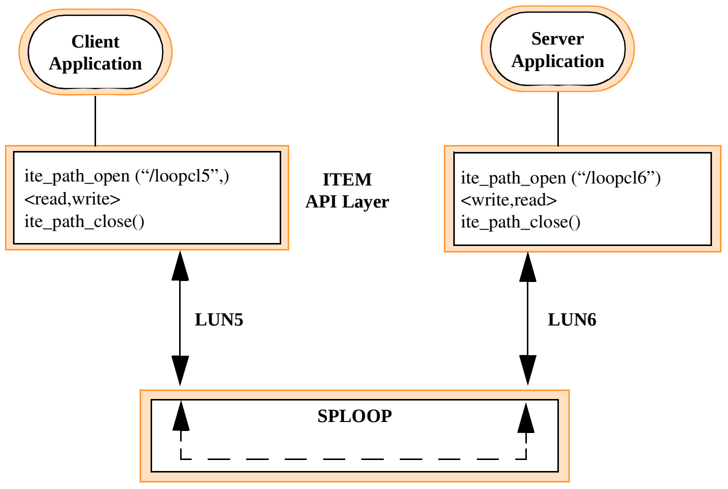


Figure 6-2. Typical Application Test Setup--Connectionless Environment



The first example provides an example program that using connection-oriented paths. You may wish to use this example as a starting point for easier development.

Using SPLOOP For Protocol Testing

Figure 6-3 shows how SPLOOP can be used to validate protocol drivers.

Step 1. Create the protocol driver. Depending on the services provided, you would also create an application that fully exercises all the services and functionality of the protocol driver.

Step 2. Create an emulator for the peer side of the protocol being tested.

Notice in the figure the test application opens the `/loopcl5/proto_to_test` stack. The emulator simply opens the loopcl6 descriptor. When the test application performs an `ite_ctl_connect()` for example, the protocol might generate some kind of connect packet and send it down the stack to SPLOOP. The SPLOOP driver would then send it up the loopcl6 path where the packet would be read, validated, and responded to by the peer protocol emulator.

Figure 6-3. Protocol Driver Test Setup

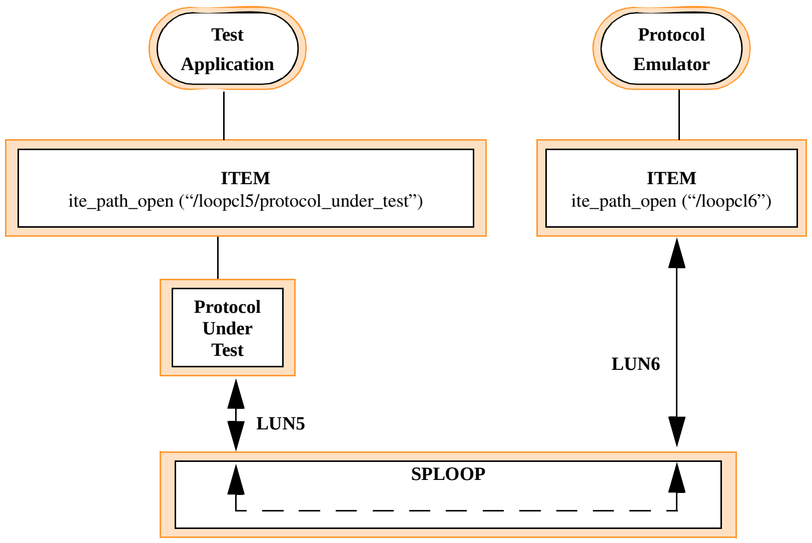
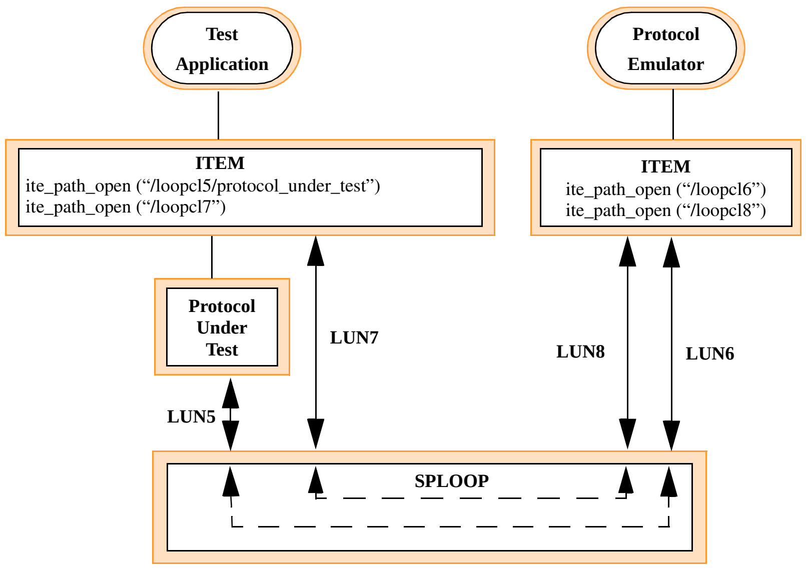


Figure 6-4 gives the test application a little more control over the emulator and test environment. The setup assumes that there is another sploop descriptor pair called loopcl7 and loopcl8. The test setup is identical to the previous one, but the pipe between the test application and the emulator using the loopcl7-loopcl8 pipe is used by the test application to control the emulator. This way, the test application can set the emulator to respond incorrectly or not at all to validate protocol timeouts, error conditions, and re-transmission.

Figure 6-4. Advanced Driver Test Setup


Test Application
Protocol Emulator
ITEM
ite_path_open ("/loopcl5/protocol_under_test")
ite_path_open ("/loopcl7")
ITEM
ite_path_open ("/loopcl6")
ite_path_open ("/loopcl8")
Protocol Under Test
LUN7
LUN8
LUN6
LUN5
SPLOOP

The SPLOOP driver then sends the loopcl6 path where the packet is read, validated, and responded by the peer protocol emulator environment for testing applications and protocol drivers. This is the fastest way to create high quality applications and protocols that work in the OS-9/SoftStax network environment and results in fewer errors.

A Examples

This appendix provides example applications using SoftStax.

For information about...	Go to this page...
Example Applications	68
Example 1: Standard Telecommunications Application	68
Example 2: Using os_lib.l	84
Example 3: Loopback Process Application	91

Example Applications

Now that you have a basic understanding of OS-9 modules and SoftStax architecture, you may want to look at some example applications and become familiar with how to use the SoftStax I/O system. Each of these three examples can be found in the `MWOS/SRC/SPF/DEMOS` directory.

- Example 1: Standard Telecommunications Application is a standard telecommunications application written in two processes.
- Example 2: Using os_lib.l is a send/receive application using `os_lib.l` to test protocol and hardware drivers.
- Example 3: Loopback Process Application is an application showing both connection-oriented and connectionless-oriented call control looping back on one path.

How to Make an Application

Source code for the example applications in this chapter is found in the `MWOS/SRC/SPF/DEMOS` directory.

For each of the example applications, there is a file named `makefile`. On an OS-9 system, typing the command `make` creates an example application using the `makefile`. On a UNIX system, the command to use is `os9make`.

The make process generates relocatable object (`.r`) files and an executable for each target processor. These are stored in individual target directories. For example, PowerPC relocatables are stored in `/RELS/ppc`. The final binary executables are stored in the appropriate CMDS directory such as:

`MWOS/OS9000/PPC/CMDS` for Power PC executables.

Example 1: Standard Telecommunications Application

The `ex1_snd.c` and `ex1_rvc.c` programs show a simple `hello world` application. Use these programs to develop a better understanding of the ITEM interface. Use the `SPLOOP` driver provided with the package to test this program.

You can also use this example to test drivers to make sure the standard telecommunications calls are correctly implemented.

ex1_snd.c

```
/* ex1_snd.c
*
* This source code is the connection initiator. It opens an ITEM path and
* makes a connection to the ex1_rcv.c program (which must be running on
* the system). After making a connection, this program sends a "hello
* world" message to the receiver program and awaits a response. The
* response message is displayed and a disconnect is performed before
* exiting.
*/
```

```
_asm("_sysedit: equ 1");/* set edition to #1 */

/* include files:

*          modes.h for file access modes (FAM_READ and
*             FAM_WRITE)

*          const.h for various constants (SUCCESS)

*          cglob.h for external _glob_data variable (needed
*             for _os_intercept call)

*          item.h for ITEM structures (device_type,
*             notify_type, *addr_type) and for ITEM function
*             prototypes

*/

#include <stdio.h>

#include <modes.h>

#include <const.h>

#include <cglob.h>

#include <SPF/item.h>

/* system-specific definitions:

* DEVICE is our ITEM device

* SND_MESSAGE is the message to send to the ex1_rcv.c
*      program upon connection.

* Modify these parameters for your particular test setup

*/

#define DEVICE   "loopc0"

#define SND_MESSAGE"hello world"

/* Define the signals used for notification */

#define CONNECT_SIG0x2001

#define FEHANGUP_SIG0x2002

#define DATAVAIL_SIG0x2003

/* global variables for the sender application:

*          connect_flag, datavail_flag, and fehangup_flag are
*          set to 1 by the sighand function upon receiving
*          CONNECT_SIG,DATAVAIL_SIG, or FEHANGUP_SIG,
*          respectively.*/

/* define signal receive flags to use with notification */

u_int8 connect_flag, fehangup_flag, datavail_flag;

/* The signal handler function intercepts any

* incoming signal and sets the appropriate global flag variable. Signal
* handlers are important due to the asynchronous nature of network
* communication. As a general rule,I/O should not be performed within the
* signal handler function.

*/
```

```
void sighand(int rcvd_signal)
{
    switch(rcvd_signal)
    {
        case CONNECT_SIG:
            connect_flag = 1;
            break;
        case DATAVAIL_SIG:
            datavail_flag = 1;
            break;
        case FEHANGUP_SIG:
            fehangup_flag = 1;
            break;
    }
    _os_rte();    /* return to program from signal handler */
} /* End signal handler */
void main(void)
{
/* main program variables:
* dev_name = pointer to the name of our DEVICE
* ite_path = path to our DEVICE
* device_info = structure used to obtain call statistics and information
* my_addr = address structure used to set our class and subclass
* connect_npb = connection notification parameter block
* fehangup_npb = far-end hang-up notification parameter block
* datavail_npb = data available notification parameter block
* rcv_size = used to remember the size of our data receive packets
* snd_size = used to remember the size of our data send packets
* rcv_buffer = data receive buffer
* snd_buffer = data send buffer
* err = used for error checking
*/
    char *dev_name = DEVICE;
    path_id ite_path;
    device_type device_info;
        /* The device_type structure is used to obtain call information and
         * statistics
         */
```

```
addr_type my_addr;

notify_type connect_npb, fehangup_npb, datavail_npb;

u_int32 rcv_size, snd_size;

u_char rcv_buffer[32], snd_buffer[32];

error_code err;

/* Most applications will need a signal handler due to the asynchronous

 * nature of using a network device. Be sure to reset any signal flags

 * used to zero!

 */

connect_flag = fehangup_flag = datavail_flag = 0;

if ((err = _os_intercept(sighand, _glob_data))
    != SUCCESS)

{

          printf("Error %03d:%03d installing signal
             handler\n", err/256, err%256);

          exit(0);

}

/* First, initialize the source address information

 * structure in ITEM. If the default source address

 * information in the descriptor is correct, you

 * do not have to do this part. You can use NULL where
 *     the src_info variable is in the ite_path_open()
 *     call. The address class is set to ITE_ADCL_LPBK
 *     because the loopback driver is used. The
 *     subclass is ITE_ADSUB_LUN to denote the
 *     Logical Unit Number is the address.

 */

my_addr.addr_class = ITE_ADCL_LPBK;

my_addr.addr_subclass = ITE_ADSUB_LUN;

/*     set up our notification blocks to receive
 *     signals for far-end hang-up (FEHANGUP_SIG),
 *     connection (CONNECT_SIG), and

 * data available (DATAVAIL_SIG). This sets up
 *     parameter blocks used

 * later to request notification.

 */

connect_npb.ntfy_class    = ITE_NCL_SIGNAL;

connect_npb.ntfy_timeout = 10;/* 10 second timeout */

connect_npb.ntfy_sig2send = CONNECT_SIG;

fehangup_npb.ntfy_class    = ITE_NCL_SIGNAL;

fehangup_npb.ntfy_timeout = 10;/* 10 second timeout */
fehangup_npb.ntfy_sig2send = FEHANGUP_SIG;

datavail_npb.ntfy_class    = ITE_NCL_SIGNAL;

datavail_npb.ntfy_timeout = 10;/* 10 second timeout */

datavail_npb.ntfy_sig2send = DATAVAIL_SIG;

/*     Open the ITEM path to our DEVICE for both READ and
       WRITE */

if ((err = ite_path_open(DEVICE, FAM_READ | FAM_WRITE,
          &ite_path, &my_addr)) != SUCCESS)

{

   printf("Error %03d:%03d on ite_path_open(%s)\n",
          err/256, err%256, dev_name);

   exit(0);

}

/* Now we get the device_type structure from our path.

 * The source address information was set correctly

 * from the ite_path_open call, but we'll check to

 * verify it was done when we get the structure back.

 */

if ((err = ite_ctl_connstat(ite_path, &device_info))
          != SUCCESS)

{

   printf("Error %03d:%03d getting connection status
          for path\n", err/256, err%256);

   exit(0);

}

if (device_info.dev_ournum.addr_class !=
          ITE_ADCL_LPBK)

{

   printf("Address class not set during open\n");

   exit(0);

}

if (device_info.dev_ournum.addr_subclass !=
          ITE_ADSUB_LUN)

{

   printf("Address subclass not set during open\n");

   exit(0);

}

printf("\nite_open(%s) successful\n", DEVICE);
```

```
/*  Now we make the call. Notice the source and destination address fields
 * are NULL because we use a loopback driver with descriptors containing
 * default source/destination address information. We also use our
 * connection notification parameter block to tell ITEM to notify us when a
 * connection is made.
*/

    if ((err = ite_ctl_connect(ite_path, NULL, NULL,
                &connect_npb)) !=SUCCESS)

    {

        printf("Error %03d:%03d during attempt to
                connect\n", err/256, err%256);

        printf("Are you sure the receiver program is
                running?\n");

        exit(0);

    }

    /* Go to sleep and await connection notification
     *      signal. The sleep time should be slightly longer
     *      than the connection timeout value. When a
     *      connection is made, the sleep() call will return
     *      immediately.

     */

    sleep(connect_npb.ntfy_timeout + 5);

    /*      Did we make a connection?

     *      If not, report a timeout error and exit.

     *      If yes, display a connection message along with the

     *      address we connected to. Note for subclass
     *      ITE_ADSUB_LUN, addresses are stored as a u_int8 in
     *      the first byte of the addr field.

     */

    if (!connect_flag)

    {               /* connection was not made */

        printf("Timeout error during connection attempt\n");

        exit(0);

    } else

    {

        printf("Connected to destination address %d\n",
                device_info.dev_theirnum.addr[0]);

    }

    /* Now, use the fehangup_npb notification parameter
     *      block to request ITEM notify us on far-end
     *      hang-up.

     */

    if ((err = ite_fehangup_asgn(ite_path, &fehangup_npb))
            != SUCCESS)
```

```
{
   printf("Error %03d:%03d during fehangup signal
          assignment\n", err/256, err%256);

   exit(0);

}

/* We want to be prepared to respond when the

 * receiver program sends a response to us.

 * To do this, we'll ask ITEM to notify us when data

 * becomes available (using our datavail_npb block).

 */

if ((err =ite_data_avail_asgn(ite_path,
     &datavail_npb)) != SUCCESS)
{
   printf("Error %03d:%03d during datavail signal
          assignment\n", err/256, err%256);

   exit(0);

}

/* Now that we have the end-to-end connection
 *     established, we'll send 'hello world' to the
 *     receiver program.

 */

strcpy(snd_buffer, SND_MESSAGE);

snd_size = strlen(snd_buffer) + 1;

if ((err = ite_data_write(ite_path, snd_buffer,
          &snd_size)) !=SUCCESS)

{

   printf("Error %03d:%03d during ite_data_write\n",
          err/256, err%256);

   exit(0);

}

/* Now, wait for the receiver to send a response.
 *     We know we've received a response when
 *     datavail_flag is set by our signal handler routine.
 */

if (!datavail_flag)

{

   sleep(datavail_npb.ntfy_timeout + 5);

}

/* If our data available flag has not been set, we
 *     timed  out while waiting for the receiver's response
 *     packet. Report the timeout error and exit.
 */

if (!datavail_flag)
```

```
{
   printf("Timeout error while awaiting response\n");
   exit(0);
}
/* We have been notified there is a response
 *     waiting for us, so lets find out how many bytes are
 *     in the response using ite_data_ready.
 */
if ((err = ite_data_ready(ite_path, &rcv_size))
          != SUCCESS)
{
   printf("Error %03d:%03d on ite_data_ready call\n",
          err/256, err%256);
   exit(0);
}
/* Knowing how many bytes are in the response message,
 * we read the incoming data into our rcv_buffer.
 */
if ((err = ite_data_read(ite_path, rcv_buffer,
          &rcv_size)) !=SUCCESS)
{
   printf("Error %03d:%03d during ite_data_read\n",
          err/256, err%256);
   exit(0);
}
/* Display the received response. */
printf("Response received = <%s>\n", rcv_buffer);
/* Time to disconnect from the receiver.
 * We are not using an in-band path (ib_path), so the
 * second parameter of the ite_ctl_disconnect call must
 * be set to NULL.
 */
if ((err = ite_ctl_disconnect(ite_path, NULL))
          != SUCCESS)
{
   printf("Error %03d:%03d on ite_ctl_disconnect\n",
          err/256, err%256);
   exit(0);
}
/* Close our ITEM path and exit. */
if ((err = ite_path_close(ite_path)) != SUCCESS)
```

```
    {
       printf("Error %03d:%03d on ite_path_close\n",
              err/256, err%256);
       exit(0);
    }
    exit(0);
} /* End ex1_snd.c */
/* In example 1, the receiver accepts incoming calls from
 * any caller and reads the incoming data. After reading  *the incoming
message, the receiver sends the
 * RESPONSE_MSG message back to the sender. The receiver
 * also demonstrates the caller identification
 * capabilities of ITEM if the network supports caller ID.
 */
/* ex1_rcv.c
 *
 * This source code is the connection receiver. It opens
 * an ITEM path, and uses the receiver assignment call to
 * wait for an incoming connection. Next, it reads the
 * incoming data and sends the RESPONSE_MSG response.
 * Note on connectionless networks, the
 * receiver assignment error EOS_CONN will occur. The
 * application should determine it is
 * attempting to receive data on a connectionless network,
 * and therefore the read should happen without waiting
 * for the notification by the system software.
 */
_asm("_sysedit: equ 1");/* set edition to #1 */
/* include files:
*      modes.h for file access modes (FAM_READ and
*          FAM_WRITE)
 *     const.h for various constants (SUCCESS)
 *     signal.h for signal value constants (SIGQUIT and
*          SIGINT)
 *     cglob.h for external _glob_data variable (needed for
 *         _os_intercept call)
 *     item.h for ITEM structures (device_type,
*             notify_type,
 *         addr_type) and for ITEM function prototypes
 */
```

```
#include <stdio.h>

#include <modes.h>

#include <const.h>

#include <signal.h>

#include <cglob.h>

#include <SPF/item.h>

/* system-specific definitions:

 * DEVICE        is the ITEM device

 * RESPONSE_MSG  is the message to send the
*                ex1_snd.c program upon receiving a
*                message.

 * Modify these parameters for your particular setup.

 */

#define DEVICE   "/loopc1"

#define RESPONSE_MSG"Message Received"

/* Define the signals used for notification */

#define INCALL_SIG0x2001

#define DATAVAIL_SIG0x2002

#define FEHANGUP_SIG0x2003

/*     Global variables for the receiver application:

 *     incall_flag and datavail_flag are set to 1 by the
 *     sighand function upon receiving INCALL_SIG or
 *     DATAVAIL_SIG, respectively. connected_flag is set to
 *     1 by the main program once a connection has been
 *     established. connected_flag is reset to 0 by the
 *     sighand function upon receiving a FEHANGUP_SIG
 *     signal. exit_flag lets the main program know when the
 *     user has pressed CTRL-E or CTRL-C to exit the
 *     program.

 */

u_int8 incall_flag, connected_flag, datavail_flag,
             exit_flag;

/* signal handler function -- its purpose is to intercept
 *  any incoming signal and set the appropriate global flag
 *  variable. Signal handlers are important due to the
 *  asynchronous nature of network communication. As a
 *  general rule, I/O should not be performed

 * within the signal handler function.

 */

void sighand(int rcvd_signal)

{

   switch(rcvd_signal)
```

```
    {
        case SIGQUIT:
        case SIGINT:
            exit_flag = 1;
            break;
        case INCALL_SIG:
            incall_flag = 1;
            break;
        case FEHANGUP_SIG:
            connected_flag = 0;
            break;
        case DATAVAIL_SIG:
            datavail_flag = 1;
            break;
    }
    _os_rte();/* return to program from signal handler */
} /* End signal handler */
void main(void)
{
   /* main program variables:
    * dev_name = pointer to the name of our DEVICE
    * ite_path = path to our DEVICE
    * device_info = structure used to obtain call
    * statistics and information, including the caller-id
    * string my_addr = address structure used to set our
    * class and subclass
    * incall_npb = incoming call notification parameter block
    * fehangup_npb = far-end hang-up notification
    * parameter block
    * datavail_npb = data available notification
    * parameter block
    * rcv_size = used to remember the size of data receive     * packets
    * snd_size = used to remember the size of data send
    * packets
    * rcv_buffer = data receive buffer
    * snd_buffer = data send buffer
    * err = used for error checking
    */
    char          *dev_name = DEVICE;
    path_id       ite_path;
    device_type   device_info;
    addr_type     my_addr;
    notify_type   incall_npb, fehangup_npb,
                  datavail_npb;
    u_int32       rcv_size, snd_size;
    u_char        rcv_buffer[32],
                  snd_buffer[32];

    error_code    err;

    /* Most applications will need a signal handler due
     * to the asynchronous nature of using a network
     * device. Be sure to reset any global notification
     * flags to zero!
     */
    if ((err = _os_intercept(sighand, _glob_data))
        != SUCCESS)

    {

             printf("Error %03d:%03d installing signal
                 handler\n", err/256, err%256);

             exit(0);

    }

    incall_flag = connected_flag = datavail_flag =
                 exit_flag = 0;

    /* Set up our address class, subclass, and address.
     * Our address class is set to ITE_ADCL_LPBK since we
     * are using a loopback driver. Our subclass is
     * ITE_ADSUB_LUN to denote our Logical Unit Number     * is our
address.

    my_addr.addr_class = ITE_ADCL_LPBK;

    my_addr.addr_subclass = ITE_ADSUB_LUN;


    /* Set up our notification blocks to let us receive
     * signals for far-end hang-up (FEHANGUP_SIG),
     * incoming call (INCALL_SIG), and data available
     * (DATAVAIL_SIG). Notice we are merely setting
     * up the parameter blocks.We will use these parameter
     * blocks later to request notification
     */
    incall_npb.ntfy_class    = ITE_NCL_SIGNAL;
    incall_npb.ntfy_timeout = 50; /* no timeout for
                 incoming calls */

    incall_npb.ntfy_sig2send = INCALL_SIG;

    fehangup_npb.ntfy_class = ITE_NCL_SIGNAL;

    fehangup_npb.ntfy_timeout = 10;
                 /* 10 second timeout */

    fehangup_npb.ntfy_sig2send = FEHANGUP_SIG;

    datavail_npb.ntfy_class = ITE_NCL_SIGNAL;

    datavail_npb.ntfy_timeout = 10;
                 /* 10 second timeout */

    datavail_npb.ntfy_sig2send = DATAVAIL_SIG;

    /* Open the ITEM path to our DEVICE for both READ and
                 WRITE */
```

```
printf("opening path...\n");

if ((err = ite_path_open(DEVICE, FAM_READ | FAM_WRITE,
        &ite_path, &my_addr)) != SUCCESS)

{

    printf("Error %03d:%03d on ite_path_open\n", err/
            256, err%256);

    exit(0);

}

printf("ite_path_open(%s) successful\n", DEVICE);


/* the big loop -- loop forever (waiting for calls and
 * answering them) until the user hits CTRL-E or CTRL-C
 * to exit.
s*/

while (exit_flag == 0)

{

/* initialize incall and connected flags */
    incall_flag = connected_flag = 0;

/* Ensure we do not have a data_available
 * assignment on our ITEM path left over from the
 * previous time through the loop.
 */
    if ((err = ite_data_avail_rmv(ite_path)) != SUCCESS)

    {

        printf("Error %03d:%03d Removing data available

            assignment\n", err/256, err%256);

    }

/* Request notification of an incoming call. This is
 * set up by the ite_ctl_rcvrasgn (receiver assignment)
 * call. Notice we are passing the address to our
 * incall notification block to tell ITEM to send a
 * INCALL_SIG upon noticing an incoming call.
 */
    if ((err = ite_ctl_rcvrasgn(ite_path, NULL,
            &incall_npb)) != SUCCESS)
    {
        printf("Error %03d:%03d performing receiver
            assignment\n", err/256, err%256);

    }

    printf("Waiting for incoming call...\n");


/* Sleep until an incoming call. Remember, although
 * sleep(0) will sleep forever, our process will be
 * awakened whenever a signal is received.
 */
    sleep(0);
```

```
/* Do we have an incoming call? If so, use
* ite_ctl_connstat to get the caller-id
* string and display it. It is possible to perform an
* ite_ctl_disconnect(ite_path, NULL) to refuse a
* connection if we are screening calls based on their
* caller-id strings. After displaying the caller-id
* string, answer the incoming call using
* ite_ctl_answer and set our connected_flag.
* We also need to request notification when the
* ex1_snd.c program disconnects from us (far-end
* hangup).
*/
   if (incall_flag)

   {

       if ((err = ite_ctl_connstat(ite_path,
          &device_info)) != SUCCESS)

       {

          printf("Error %03d:%03d performing
              ite_ctl_connstat\n", err/256, err%256);

          exit(0);

       }

       printf("Incoming caller-id: <%s>\n",
              device_info.dev_display);

/* Note we are not using an in-band path
* (ib_path), so the second parameter in the
* ite_ctl_answer call must be NULL.
*/
       if ((err = ite_ctl_answer(ite_path, NULL, NULL)) !=
              SUCCESS)

       {

          printf("Error %03d:%03d from ite_ctl_answer\n",
              err/256, err%256);

          exit(0);

       }

       printf("Connected\n");

       incall_flag = 0;

       connected_flag = 1;


/* request notification upon far-end hang-up */

       if ((err = ite_fehangup_asgn(ite_path,
              &fehangup_npb)) != SUCCESS)
       {

          printf("Error %03d:%03d performing fehangup
              assignment\n", err/256, err%256);

          exit(0);

       }
```

```
            } else
            {
        /* We awoke from a signal, but it was not due to an
         * incoming call. It is probably the user wanting to
         * exit the program. Let's remove our receiver
         * assignment (INCALL_SIG).
         */
                if ((err = ite_ctl_rcvrrmv(ite_path)) != SUCCESS)
                {
                    printf("Error %03d:%03d from receiver remove\n",
                        err/256, err%256);
                }
            }
        /* As long as we are connected to the sender program,
         * stay in this loop.
         */
            while (connected_flag)
            {
         * Request ITEM to notify us when data is available
         * from the sender program. Notice we are using
         * datavail_npb to have the DATAVAIL_SIG sent when
         * data is available.
         */
                if ((err = ite_data_avail_asgn(ite_path,
                        &datavail_npb)) != SUCCESS)
                {
                    printf("Error %03d:%03d performing data
                        available assignment\n", err/256, err%256);
                    exit(0);
                }
         /* Now, wait for the incoming data packet. Let's sleep for five
seconds longer than our requested timeout.
         * After sleeping, if we have not been notified of
         * available data but we are still connected, report a
         * timeout condition. */
                if (!datavail_flag && connected_flag)
                {
                    sleep(datavail_npb.ntfy_timeout + 5);
                }


                if (!datavail_flag && connected_flag)
                {
                    printf("Timeout waiting for data from sender\n");
                    exit(0);
                }
```

```
/* We have been notified data is either available
 * or far-end hang-up has occurred. If hang-up has
 * occurred, we drop out of our while loop and
 * wait for another incoming call. If data is
 * available, we need to find out how many bytes the
 * incoming data packet contains.
 */
       if (datavail_flag)
       {
          if ((err = ite_data_ready(ite_path, &rcv_size))
             != SUCCESS)
          {
             printf("Error %03d:%03d on ite_data_ready\n",
             err/256, err%256);
             exit(0);
          }
/* Knowing how many bytes are to be read, we read
 * the incoming data into our rcv_buffer.
 */
          if ((err = ite_data_read(ite_path, rcv_buffer,
             &rcv_size)) != SUCCESS)
          {
             printf("Error %03d:%03d during hello world
             read\n", err/256, err%256);
             exit(0);
          } else
          {
             printf("ite_data_read() result: [%s]\n",
             rcv_buffer);
          }
/* reset our datavail_flag */
          datavail_flag = 0;
/* Let's send our RESPONSE_MSG back to the sender
 * program to acknowledge we received the data.  */
          strcpy(snd_buffer, RESPONSE_MSG);
          snd_size = strlen(RESPONSE_MSG) + 1;
          if ((err = ite_data_write(ite_path, snd_buffer,
             &snd_size)) != SUCCESS)
          {
             printf("Error %03d:%03d during
             ite_data_write\n", err/256, err%256);
             continue;
             }
       }
```

```
        }
    /* We reach this point only if we have been
     * disconnected by the sending program. Report this
     * fact to the user.
     */
        printf("Disconnected\n\n");

        connected_flag = 0;

    }

     /* We reach this point only if the user has asked to
     * exit using CTRL-E or CTRL-C. We close our ITEM path
     * and exit.
     */
    if ((err = ite_path_close(ite_path)) != SUCCESS)

    {

        printf("Error %03d:%03d during ite_path_close\n",
               err/256, err%256);

    }

    exit(0);

} /* End ex1_rcv.c */
```

Example 2: Using os_lib.l

The `spf_test.c` program is an application that does not perform call control. It uses the I/O calls in `os_lib.l` to open paths and transmit and receive data. This program uses `_os_xxx` calls in `os_lib.l` to perform tests on the packets flowing through SoftStax. The syntax for this test is:

```
spf_test <descriptor name> <ITE or DCE> <packets to send> <number of
bytes in each packet>
```

Run this example program with the `loopback` driver by entering the following commands:

```
spf_test /loopc16 DCE 10 100

spf_test /loopc15 DTE 10 10
```

The first command causes `spf_test` to use `loopc16` to open a path. The DCE entry means the process waits for receive data first. The remainder of the command line indicates it receives, then transmits, ten packets of 100 bytes each.

The next command causes this incarnation of the `spf_test` process to use `loopc15` to open a path. The DTE entry means the process transmits ten packets of 100 bytes, then waits to receive ten packets of 100 bytes.

spf_test.c

```
/*****************************************************
 * SPF example/test program*
 *****************************************************

   /* Copyright 1995 by Microware Systems Corporation
    * Copyright 2001 by RadiSys Corporation
      Reproduced Under License
    *


    * This source code is the proprietary confidential
    * property of Microware Systems Corporation, and is
    * provided to licensee solely for documentation and
    * educational purposes. Reproduction, publication, or
    * distribution in any form to any party other than the
    * licensee is strictly prohibited.
   /*

_asm("_sysedit: equ 1");

_asm("_sysattr: equ 0xC001");



   /*

    * Header Files

   /*


#include <stdio.h>

#include <types.h>

#include <ctype.h>

#include <const.h>

#include <errno.h>

#include <modes.h>

#include <signal.h>


#include <module.h>


#include <SPF/spf.h>


   /*

    * Macro Definitions

   /*
   /*
    *Global Variables
    */
u_char buf[10000]= {0};
u_int32 COUNT= 1;
u_int32 BUFSZ= 100;
path_id path= 0;

   /*
    * Signal Handler
    */
void sighand(int sig)
{
   switch (sig)
   {
      case SIGINT :
      case SIGQUIT :
      case SIGHUP:
         fprintf(stderr,"Termination signal received\n");
         _os_close(path);
         _os_exit(SUCCESS);
         break;
      default :
         fprintf(stderr,"Unknown signal received
                %d\n",sig);
         break;
   }
}
   /*
    * Send Data
    */

error_code send_data()
{
   u_int32 loop;
   u_int32 count;
   u_int32 byte;
```

```
    printf("Sending data: \n"); fflush(stdout);

    for (loop=1;loop<=COUNT;loop++)
    {
        printf(">%d>",loop);
        for(byte=0;byte<BUFSZ;byte++)
        {
            buf[byte] = loop;
        }
        count = BUFSZ;
        if ((errno = _os_write(path,buf,&count)) != SUCCESS)
        {
            printf("ERROR: %s\n\n",strerror(errno));
            return(errno);
        }
    }
    printf("SUCCESSFUL\n\n");fflush(stdout);
    return(SUCCESS);
}

    /*
     * Receive Data
     */
error_code recv_data()
{
    u_int32 loop;
    u_int32 count;
    u_int32 byte;

    printf("Receiving data: \n"); fflush(stdout);

    for (loop=1;loop<=COUNT;loop++)
    {
        printf("<%d<",loop);
        buf[0] = 0;
        count = BUFSZ;
        if ((errno = _os_read(path,buf,&count)) != SUCCESS)
        {
            printf("ERROR: %s\n\n",strerror(errno));
            return(errno);
        }
        if (buf[0] != (loop%256))
        {
            printf(" ERROR: Out of Order Number Received\n");
            return(EOS_READ);
        }
    }
    printf(" ... SUCCESSFUL\n\n");fflush(stdout);
    return(SUCCESS);
}


    /*
     * Main Program
     */
void main(int argc, char *argv[])
{
    int32           val = 0,
                    ticks_left;
    error_code      err = SUCCESS;

    /* set up signal handler */
    signal(SIGINT,sighand);
    signal(SIGQUIT,sighand);
    signal(SIGHUP,sighand);

    /* print header */
    printf("\n***** %s *****\n\n",argv[0]);

    /* check command line arguments */
```

```
if ((argc < 3) || (argv[1][0] == '-'))
{
    printf("HELP:\n");
    printf("   Syntax: %s </device> <DTE/DCE>
            [<count:default=%d>]
            [<bufsize:default=%d>]\n\n",
            argv[0],COUNT,BUFSZ);
    _os_close(path);
    _os_exit(SUCCESS);
}

/* get new count */
if (argc >= 4)
{
    COUNT = atoi(argv[3]);
}
printf("COUNT = [%d]\n\n",COUNT);

/* get new buffer size */
if (argc >= 5)
{
    BUFSZ = atoi(argv[4]);
    if (BUFSZ > sizeof(buf))
    {
        printf("ERROR: maximum buffer size = '%d'\n\n");
        _os_close(path);
        _os_exit(EOS_PARAM);
    }
}
printf("BUFSZ = [%d]\n\n",BUFSZ);

/* open indicated device */
printf("Opening device [%s] ... ",argv[1]);
if ((errno =
    _os_open(argv[1],S_IREAD|S_IWRITE,&path))
    != SUCCESS)
```

```
{
    printf("ERROR: %s\n\n",strerror(errno));
    _os_close(path);
    _os_exit(errno);
}
printf("SUCCESSFUL\n\n");

if (strcmp(argv[2],"DCE") == 0)
{
    printf("DCE\n\n");



    /* receive data */
    if ((errno = recv_data()) != SUCCESS)
    {
        _os_close(path);
        _os_exit(err);
    }
    /* send data */
    if ((errno = send_data()) != SUCCESS)
    {
        _os_close(path);
        _os_exit(err);
    }
} else if (strcmp(argv[2],"DTE") == 0)
{
    printf("DTE\n\n");

    /* send data */
    if ((errno = send_data()) != SUCCESS)
    {
        _os_close(path);
        _os_exit(err);
    }

    /* receive data */
    if ((errno = recv_data()) != SUCCESS)
```

```
        {
            _os_close(path);
            _os_exit(err);
        }

    } else
    {

        printf("ERROR: Unknown Command [%s]\n\n",argv[2]);
        _os_close(path);
        _os_exit(EOS_ILLARG);
    }

    if ((ticks_left = sleep(1)) != 0)
    {
        printf("spf_test: signal received before the 1 sec sleep
complete\n");
    }
    _os_close(path);
    _os_exit(SUCCESS);
}
```

Example 3: Loopback Process Application

The `example3.c` program uses the `/loop` descriptor to perform call control and send and receive data over the same path. Note how the program registers to receive incoming calls, then connects. The incoming call signal comes over the same path. After the program answers, the connect signal is received and it sends and receives a test message.

example3.c

```
/*********************************************************
 * ID:     @(#) example3.c 1.2@(#)
 * Date:   6/26/96
 *********************************************************
 * Example to show connectionless and connection-oriented  *ITEM
communication

*********************************************************
 * Copyright 1996 by Microware Systems Corporation
 * Copyright 2001 by RadiSys Corporation

 * Reproduced Under License

 * This source code is the proprietary confidential
 * property of Microware Systems Corporation, and is
 * provided to licensee solely for documentation and
 * educational purposes. Reproduction, publication, or
 * distribution in any form to any party
 * other than the licensee is strictly prohibited.


**********************************************************

_asm("_sysedit: equ 1");/* set edition to #1 */


/* include files:

 *     modes.h for various file access modes (FAM_READ and
 *              FAM_WRITE)

 *    const.h for various constants (SUCCESS)
 *    cglob.h for external _glob_data variable (needed
 *              for _os_intercept call)
 *    item.h   for ITEM structures (device_type,
 *              notify_type) and for ITEM function prototypes

 */


#include <stdio.h>

#include <modes.h>

#include <const.h>

#include <cglob.h>

#include <SPF/item.h>

/* System-specific definitions:

 *        DEVICE      = our ITEM loopback device

 *        DATA_STRING = the message to send and receive on
 *            the ITEM path

 */

#define DEVICE   "/loop"

#define DATA_STRING"This is example #3 data."


/* Define the signals used for notification */

#define SIG_CONNECT0x2001

#define SIG_INCALL0x2002

#define SIG_DATAVAIL0x2003

#define SIG_FEHANGUP0x2004
```

```
   /* Global variables:
    * connect_flag = set to 1 by sighandler when a
    * SIG_CONNECT is received. incall_flag = set to 1 by
    * sighandler when a SIG_INCALL is received.
    * datavail_flag = set to 1 by sighandler when a
    * SIG_DATAVAIL is received.
    * fehangup_flag = set to 1 by sighandler when a
    * SIG_FEHANGUP is received.
    */
u_int8 connect_flag, incall_flag, datavail_flag,
          fehangup_flag;


   /* signal handler function -- intercepts any incoming
    * signal and set the appropriate global flag variable.
    */

void sighandler(int signal)
{
   switch (signal)
   {
      case SIG_CONNECT:
         connect_flag = 1;
         break;
      case SIG_INCALL:
         incall_flag = 1;
         break;
      case SIG_DATAVAIL:
         datavail_flag = 1;
         break;
      case SIG_FEHANGUP:
         fehangup_flag = 1;
         break;
      default:
   /* spurrious signal received */
         break;
   }
   _os_rte();/* return to program from signal handler */
} /* End signal handler */


void main(void)
{
   /* main program variables:
    * ite_path  = path to our DEVICE
```

```
 * data_length= used to store length of
   *          DATA_STRING message
 * count     = used in read/write functions
   *         to give # of bytes
 * buffer    = receive storage buffer

 * device_info= structure used to obtain
   *         caller-id string
 * connect_npb= connection notification
   *         parameter block
 * incall_npb= incoming call notification
   *         parameter block

 * datavail_npb= data available notification
   *         parameter block
 * fehangup_npb= far-end hang-up notification
   *         parameter block
 * err       = used for error checking

 */
path_id      ite_path;

u_int32      data_length, count;

u_char       buffer[256];

device_type  device_info;

notify_type  connect_npb, incall_npb,
             datavail_npb, fehangup_npb;

error_code   err;


/* set up our notification parameter blocks for
 * connection, incoming call, and data available.
 * Notice we're merely setting up the parameter
 * blocks... we'll use them later to request
 * notification.
 */
connect_npb.ntfy_class = ITE_NCL_SIGNAL;

connect_npb.ntfy_timeout = 10;/* 10 second timeout */

connect_npb.ntfy_sig2send = SIG_CONNECT;


incall_npb.ntfy_class = ITE_NCL_SIGNAL;

incall_npb.ntfy_timeout = 10;/* 10 second timeout */

incall_npb.ntfy_sig2send = SIG_INCALL;


datavail_npb.ntfy_class = ITE_NCL_SIGNAL;

datavail_npb.ntfy_timeout = 10;/* 10 second timeout */

datavail_npb.ntfy_sig2send = SIG_DATAVAIL;


fehangup_npb.ntfy_class = ITE_NCL_SIGNAL;

fehangup_npb.ntfy_timeout = 10;/* 10 second timeout */
fehangup_npb.ntfy_sig2send = SIG_FEHANGUP;


/* Initialize data_length, our signal flags and signal
          handler. */

data_length = strlen(DATA_STRING);

connect_flag = incall_flag = datavail_flag =
          fehangup_flag = 0;

if ((err = _os_intercept(sighandler, _glob_data))
          != SUCCESS)

{

   printf("Error %03d:%03d from _os_intercept\n", err/
          256, err%256);

   exit(0);

}


printf("\n** START OF CONNECTIONLESS COMMUNICATION
          **\n");

 /* For connectionless communication, the loopback
 * descriptors contain the default addressing
 * information.

 */
 /* Open an ITEM path to our DEVICE for both READ &
 * WRITE access. */

if ((err = ite_path_open(DEVICE, FAM_READ | FAM_WRITE,
          &ite_path, NULL)) != SUCCESS)

{

   printf("Error %03d:%03d from ite_path_open (%s)\n",
          err/256, err%256, DEVICE);

   exit(0);

}

printf("ite_path_open call successful.\n");


 /* Using our data available notification parameter
 * block, we ask ITEM to send us a SIG_DATAVAIL signal
 * when data is ready to be read from our ITEM path.

 */

if ((err = ite_data_avail_asgn(ite_path,
          &datavail_npb)) != SUCCESS)

{

   printf("Error %03d:%03d from ite_data_avail_asgn\n",
          err/256, err%256);

   exit(0);

}
```

```
 /* write data to ite_path... since we are using a
 * loopback driver, our data will come right back to
 * us, triggering a SIG_DATAVAIL signal to be sent.
 */
count = data_length;
if ((err = ite_data_write(ite_path, DATA_STRING,
          &count)) != SUCCESS)
{
   printf("Error %03d:%03d from ite_data_write\n", err/
          256, err%256);
   exit(0);
}
/* Sleep until data available signal is received. */
if (datavail_flag == 0) sleep(0);
if (datavail_flag == 0)
{
   printf("SIG_DATAVAIL not received on ite_path!\n");
   exit(0);
}
printf("SIG_DATAVAIL signal received correctly... %d
          bytes of data\n", count);


 /* Determine how many bytes of data need to be read.
 * It should be the same number of bytes we wrote out!
 */
if ((err = ite_data_ready(ite_path, &count))
          != SUCCESS)
{
   printf("Error %03d:%03d back from
          ite_data_ready\n",err/256, err%256);
   exit(0);
}


if (count != data_length)
{
   printf("Received length is not correct!\n");
   printf("%d bytes received... should be %d bytes.\n",
              count, data_length);
}
 /* Knowing how many bytes there are, read the data
 * into our buffer. Be sure to null-terminate the
 * string.
 */
if ((err = ite_data_read(ite_path, buffer, &count))
          != SUCCESS)
{
   printf("Error %03d:%03d from ite_data_read\n", err/
          256, err%256);
   exit(0);
}
buffer[count] = '\0';
/* Display the send/receive results. */
printf("Data sent: <%s>\nData received: <%s>\n",
          DATA_STRING, buffer);
/* Close our ITEM path. */
if ((err = ite_path_close(ite_path)) != SUCCESS)
{
   printf("Error %03d:%03d from ite_path_close on
          ite_path\n", err/256, err%256);
}
printf("ite_path_close call successful.\n");
printf("\n** START OF CONNECTION-ORIENTED
          COMMUNICATION **\n");
 /* For connection-oriented communication, the
 * loopback descriptors may or may-not contain the
 * default address information. In this example, we
 * assume the addressing information is held in the
 * descriptor. Below, we reinitialize our global flags.
 */
connect_flag = incall_flag = datavail_flag =
          fehangup_flag = 0;


 /* Open an ITEM path to our DEVICE for READ and WRITE
 * access. */
if ((err = ite_path_open(DEVICE, FAM_READ | FAM_WRITE,
          &ite_path, NULL)) != SUCCESS)
{
   printf("Error %03d:%03d from ite_path_open (%s)\n",
          err/256, err%256, DEVICE);
   exit(0);
}
printf("ite_path_open call successful.\n");
```

```
 /* The first thing a receiver program should do is a
 * receiver assignment to be notified of any incoming
 * calls. We use our incall notification parameter
 * block to ask ITEM to send us a SIG_INCALL signal
 * when we have an incoming call.
 */
if ((err = ite_ctl_rcvrasgn(ite_path, NULL,
          &incall_npb)) != SUCCESS)
{
   printf("Error %03d:%03d from ite_ctl_rcvrasgn\n",
          err/256, err%256);
   exit(0);
}


/* For a caller program, a call is placed using the
 * ite_ctl_connect function. We need to pass in our
 * connect notification parameter
  * block so ITEM sends a SIG_CONNECT when a connection
 *is established.
 */
if ((err = ite_ctl_connect(ite_path, NULL, NULL,
          &connect_npb)) != SUCCESS)
{
   printf("Error %03d:%03d from ite_ctl_connect\n",
          err/256, err%256);
   exit(0);
}
printf("ite_ctl_connect call successful.\n");


/* The receiver program will be awaiting an incoming
 * call signal (SIG_INCALL).
 */
if (incall_flag == 0) sleep(0);
if (incall_flag == 0)
{
   printf("Error -- SIG_INCALL not received!\n");
   exit(0);
}
printf("SIG_INCALL signal received correctly.\n");


/* After receiving a SIG_INCALL signal, the receiver
 * program can look at the caller-id information. If
 * this is not important, this can be omitted. Using
 * this technique, incoming calls can be screened. An
 * ite_ctl_disconnect can be used to refuse an incoming
 * call.  */
```

```
    if ((err = ite_ctl_connstat(ite_path, &device_info))
            != SUCCESS)
    {
        printf("Error %03d:%03d performing
                ite_ctl_connstat\n", err/256, err%256);
        exit(0);
    }
    printf("Incoming caller-id: <%s>\n",
            device_info.dev_display);


    /* After noticing an incoming call, the receiver
     * program answers the call using the ite_ctl_answer
     * function. This will establish a connection and send
     * a SIG_CONNECT to the caller program.
     */
    if ((err = ite_ctl_answer(ite_path, NULL, NULL)) !=
SUCCESS)
    {
        printf("Error %03d:%03d from ite_ctl_answer\n",
                err/256, err%256);
        exit(0);
    }
    printf("ite_ctl_answer call successful.\n");


    /* After issuing an ite_ctl_connect call, the caller
     * program will wait for a SIG_CONNECT to be sent,
     * meaning a connection has been established.
     */
    if (connect_flag == 0) sleep(0);


    if (connect_flag == 0)
    {
        printf("Error -- SIG_CONNECT never received after
                ite_ctl_answer!\n");
        exit(0);
    }
    printf("SIG_CONNECT signal received correctly.\n");


    /* Both the caller and receiver programs need to be
     * notified on far-end hang-up (the other party
     * disconnects). This is done using the
     * ite_fehangup_asgn call with our far-end hang-up
     * notification parameter block (fehangup_npb).
     */
```

```
if ((err = ite_fehangup_asgn(ite_path, &fehangup_npb))
          != SUCCESS)
{
   printf("Error %03d:%03d from ite_fehangup_asgn\n",
          err/256, err%256);
   exit(0);
}


/* Just as we sent our DATA_STRING in a connectionless
 * environment (above), we'll now send our DATA_STRING
 * the exact same way since our connection has now been
 * established. We start by having the receiver program
 * request ITEM send a SIG_DATAVAIL when data is
 * available for reading.
 */
if ((err = ite_data_avail_asgn(ite_path,
          &datavail_npb)) != SUCCESS)
{
   printf("Error %03d:%03d from ite_data_avail_asgn\n",
          err/256, err%256);
   exit(0);
}


/* Now, the sender will write data to the connected
 * ITEM path. This will cause the SIG_DATAVAIL signal
 * to be sent to the receiving program.
 */
count = data_length;
if ((err = ite_data_write(ite_path, DATA_STRING,
          &count)) != SUCCESS)
{
   printf("Error %03d:%03d from ite_data_write\n",
          err/256, err%256);
   exit(0);
}


 * The receiver program will await the SIG_DATAVAIL
 * signal.
 */
if (datavail_flag == 0) sleep(0);
if (datavail_flag == 0)
{
   printf("SIG_DATAVAIL not received on ite_path!\n");
   exit(0);
}
printf("SIG_DATAVAIL signal received correctly... %d
          bytes of data\n", count);


/* Determine how many bytes of data need to be read.
 * It should be the same number of bytes we wrote out!
 */

if ((err = ite_data_ready(ite_path, &count))
          != SUCCESS)
{
   printf("Error %03d:%03d back from ite_data_ready\n",
          err/256, err%256);
   exit(0);
}


if (count != data_length)
{
   printf("Received length is not correct!\n");
   printf("%d bytes received... should be %d bytes.\n",
          count, data_length);
}


/* Knowing how many bytes there are, read the data into
 * our buffer. Be sure to null-terminate the string.
 */
if ((err = ite_data_read(ite_path, buffer, &count))
          != SUCCESS)
{
   printf("Error %03d:%03d from ite_data_read\n",
          err/256, err%256);
   exit(0);
}
buffer[count] = '\0';


 /* Display the send/receive results. */
printf("Data sent: <%s>\nData received: <%s>\n",
          DATA_STRING, buffer);
```

```
     /* Our caller program now must disconnect from the
      * receiver program using the ite_ctl_disconnect
      * function. This causes a SIG_FEHANGUP to be sent to
      * the other end of the connection if fehangup
      * notification is active. Because we are not using an
      * in-band path (ib_path), the second parameter of
      * the ite_ctl_disconnect call must be NULL.
      */
    if ((err = ite_ctl_disconnect(ite_path, NULL))
              != SUCCESS)
    {
       printf("Error %03d:%03d from ite_ctl_disconnect\n",
              err/256, err%256);
    }


     /* The other party would now receive a SIG_FEHANGUP signal */
    if (fehangup_flag == 0) sleep(0);


    if (fehangup_flag == 0)
    {
       printf("Error -- SIG_FEHANGUP never received after
              ite_ctl_disconnect!\n");
    }
    else
    {
       printf("SIG_FEHANGUP signal received correctly.\n");
    }


     /* Close our ITEM path. */
    if ((err = ite_path_close(ite_path)) != SUCCESS)
    {
       printf("Error %03d:%03d from ite_path_close on
              ite_path\n", err/256, err%256);
    }
    else
    {
       printf("ite_path_close call successful.\n");
    }
    /* Exit program */
    printf("** END OF EXAMPLE #3 **\n\n");
    exit(0);
}
```

B Using SoftStax with Multimedia Devices

This appendix shows how to configure SoftStax to be used with Networked Multimedia Devices.

For information about...	Go to this page...
Networked Multimedia Device Basics	104

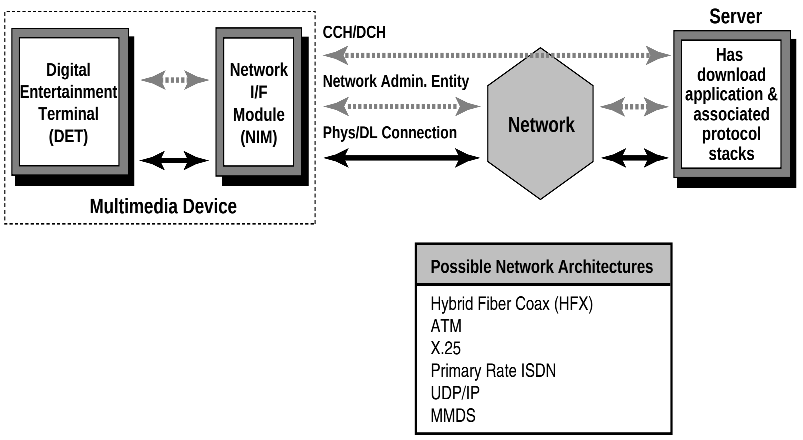
Networked Multimedia Device Basics

Figure B-1 shows the general end-to-end architecture of the multimedia delivery system.

The solid lines represent the physical and data link connections between each entity and its adjacent entity. The dashed lines represent the communication path through one or more of the entities.

The multimedia device in this figure has a default communication path, (not shown), between it and the network administrative entity. This path is used to make and break connections between the multimedia device and a given server. The result of the connection establishment is the control and data channel communications paths (CCH and DCH, respectively). Depending on the session being established, this may take the form of many messaging paths, one messaging path carrying MPEG, and one messaging path with a control protocol.

Figure B-1. General Network Topology for Multimedia Network Delivery



Possible Network Architectures

Hybrid Fiber Coax (HFX)
ATM
X.25
Primary Rate ISDN
UDP/IP
MMDS

The following information is specific to a deployed Digital Entertainment Terminal (DET) in an interactive multimedia network. SoftStax can also be used in a variety of network intelligent consumer devices such as PDAs and pagers.

Multimedia Device Specifics

This section examines the multimedia device architecture.

Every device has two logical components:

- Digital Entertainment Terminal (DET)

  This component has the MPEG decoders, graphics chips, and processing power to run a session between the multimedia device and the server. For example, the session may be Video-on-Demand (VOD), database applications, or interactive gaming. The default resident application also runs on the DET. The resident application in Figure B-2 is identified as the player shell.

- Network Interface Module (NIM)

  This component deals with the network-specific protocol. It establishes and terminates connections as well as transmits user data between the multimedia device and multimedia device server.

Figure B-2. Type 1 Multimedia Device Architecture

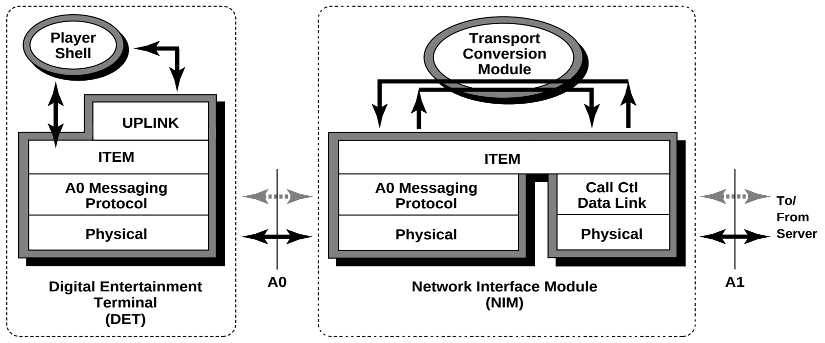
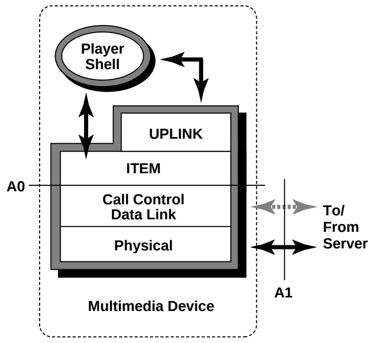


Figure B-3. Type 2 Multimedia Device Architecture



The important distinction between the architectures in Figure B-2 and Figure B-3 is where the application and network protocol processing is performed.

Type 1 multimedia devices have a motherboard containing the DET components and a plug-in module with a separate processor that comprises the NIM. Having the DET processor dedicated to application processing while the network processing is performed on a different processor is an advantage.

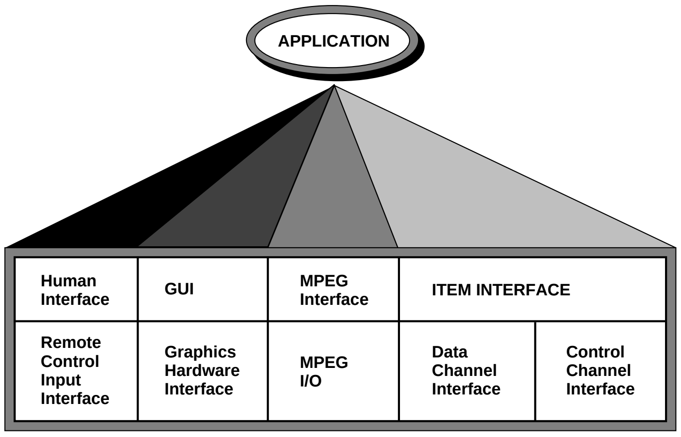
Type 2 multimedia devices have only one processor performing all application and network protocol processing. This limits the available computing power for the application.

Notice that there are two reference points on the model, the A0 and the A1. The A0 reference point is network-independent. The A1 reference point is the interface between the multimedia device and the network, which is network-dependent. The idea is that NIMs belong to the network provider, while DETs are customer equipment. This allows the DET to be portable across all networks.

Multimedia Device Run-Time Model

Figure B-4 shows the run-time model for a multimedia device. This section concentrates on the ITEM interface and the components below it. Higher layer software uses ITEM to receive private data not intended for video/graphics display hardware and to communicate with a network administrative entity or server using a protocol layer through the control channel interface.

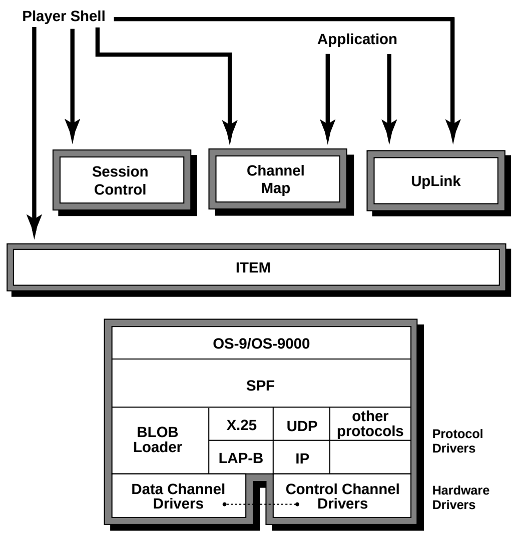
Figure B-4. Multimedia Device Run-time Model



DET Software Configuration

Figure B-5 shows the placement of the DET software modules and where ITEM fits in relationship to these modules.

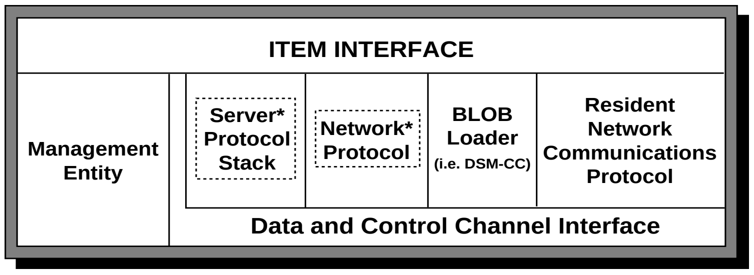
Figure B-5. ITEM Software Environment



ITEM Functions

In Figure B-6, a dashed box indicates a downloadable module.

Figure B-6. A Closer Look at the ITEM Interface


ITEM INTERFACE
Management Entity
Server* Protocol Stack
Network* Protocol
BLOB Loader (i.e. DSM-CC)
Resident Network Communications Protocol
Data and Control Channel Interface

ITEM Interface

The ITEM interface block provides an API for higher layer software to access the Microware network I/O system.

Control Channel Interface

The control channel interface communicates between the control channel hardware and the protocols above it as it sends and receives control channel information. This is typically implemented as a hardware driver sending and receiving packets over the physical interface.

Data Channel Interface

The data channel interface receives the high speed data input path. It provides the MPEG I/O system with MPEG data for playing audio and video assets and receives any private data intended for higher layer software over the data channel. Current implementations are receive data only. However, ITEM and the data channel interface do not preclude this from being a bi-directional path. The data channel interface is typically implemented as a hardware driver receiving (and in some cases, sending) packets (typically MPEG-2 streams) on the data channel.

Resident Network Communications Protocol

For a Type 1 multimedia device, this block implements the communication protocol between the DET and NIM, where the network-specific protocol is running between the NIM and the network.

For a Type 2 multimedia device, this block is either a resident network-specific protocol stack or a protocol requesting the network-specific protocol stack to be downloaded from the network when the multimedia device runs through its first initialization. The implementation can use the BLOB loader software block to accomplish the download of the network protocol.

BLOB Loader

This block implements the protocol used by the network and/or servers in the network to download modules to the multimedia device.

Network Protocol

This module can be resident on the multimedia device or downloaded to the multimedia device using the resident network communications module previously described. The network protocol implements the functions required to communicate to a specific network architecture, such as HFC, ATM UNI 3.1([2]), and ISDN D-channel layers 2 and 3.

For a Type 1 multimedia device, this module runs on the NIM processor. The A0 messaging protocol runs on the DET to send and receive commands between the DET and NIM. For a Type 2 multimedia device, this module runs on the DET processor.

Server Protocol Stack

The OS-9 environment provides a server protocol stack that can be dynamically downloaded and installed as part of the system software. There are two advantages to this environment:

- First, multiple applications can be downloaded and use the same server protocol stack without the overhead of sending the protocol stack with each application. The download procedure can inform the server that the desired protocol stack is already available, saving the time and memory required for a subsequent download of the same stack.

- Second, in environments where multiple applications need to use the same protocol stack to communicate with a server, the protocol stack saves memory. Protocol stacks embedded within applications are not re-entrant. However, if a protocol stack is loaded independently, multiple applications can access the same protocol stack simultaneously, saving the extra memory required for the same stack binary embedded in every application on the multimedia device. There are no software constraints on the number of server protocol stacks on the multimedia device at any given time. The available memory on a given DET determines the number of protocol stacks.

Network Management Entity (NME)

NME maintains a log of any exception condition, anomalies, or status reports generated by any module under ITEM. This information is accessed through the ITEM interface and returned to the application by the NME.