USB Peripheral SDK for OS-9

Version 1.1.1

www.radisys.com

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

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

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

Revision C
August 2001

Copyright and publication information

Disclaimer

Reproduction notice

Table of Contents

Chapter 3: Running USB 23

Chapter 4: Device Drivers 29

Chapter 1: Introduction

This chapter contains the following sections:

- **Contents of this Package**
- **USB Overview**

Contents of this Package

This package is an add-on to Enhanced OS-9 that must be installed on top of an existing OS-9 installation.

Following is a list of directories and their contents. Each directory is included with the USB Peripheral SDK for OS-9 and located in the MWOS tree.

- `DOS/BIN`

  contains `hawkview.exe`

- `DOS/DRIVERS`

  contains a modified bulkusb driver and install file

- `OS9000/<PROCESSOR>/CMDS`

  contains hawkview daemon to run on the OS-9 machine

- `OS9000/<PROCESSOR>/PORTS/<BOARD>/CMDS/BOOTOBJS/SPF`

  contains SPF USB driver and descriptor

- `SRC/TOOLS/HAWKVIEW`

  contains headers for hawkview daemon and application

- `SRC/UTILS/OS/COM/HAWKVIEW`

  contains OS-9 source for hawkview daemon

USB Overview

USB is an external expansion bus that provides an easy and convenient method of adding peripherals to a PC. In addition, USB connectivity is becoming as prevalent in the embedded industry as it is in the PC industry. Many developers need a real-time operating system with well-integrated USB support to drive their hardware to the next level of connectivity; OS-9 provides this dynamic and flexible I/O architecture.

Features

Ease-of-use has been one of USB's most compelling features. An external universal serial bus port eliminates the need for multiple ports (modems, printers, disks, etc.). A single port may be used to add multiple devices without requiring the end user to add adapters to configure communication software. In addition, it is possible to attach as many as 127 peripherals, which may be connected through USB hubs.

Typical applications of the USB technology include the following:

- a hand-held data collection unit with a USB output to a PC
- a smart phone with USB "docking" capability
- a synchronized address book database with PC
- a PC-based Graphic User Interface for smart telephone control
- an in-car navigation data upload function to a laptop computer
- a smart device control and/or configuration from desktop PC or a laptop computer
- a high speed transport for audio/video devices such as digital cameras, DV cameras, or audio encoders

Performance

Electrically, there are three distinct types of devices that may be present in a USB network. These include the following:

- host (root hub)
- hub
- node (device)

There may be only one host for each USB network, and while loops are not allowed, multiple levels of hubs are permitted. These features make USB a tiered star topology. In addition, devices may only include “leaf” nodes or be integrated into hubs as functions.

Communication

USB provides communication between software on the host and its USB function (present on the device). Functions have different communication flow requirements for different client-to-function interactions.

- Software is an important part of USB’s communication flow.
- The USB is a polled bus; the Host Controller initiates all data transfers.
- All bus transactions involve the transmission of as many as three packets. Each transaction begins when the Host Controller sends a USB packet describing the type and direction of transaction, the USB device address, and the point number. This packet is referred to as the “token packet”. From here, the addressed USB device selects itself by decoding the appropriate address fields.

  In a given transaction, data is transferred from either host to device or device to host. The direction of data transfer is specified in the token packet. Following this, the source of the transaction sends a data packet or indicates that it contains no data transfer. In general, the destination responds with a handshake packet indicating whether or not the transfer was successful.

- The USB data transfer model between a source or destination on the host and an endpoint on a device is referred to as a “pipe”. There are two types of pipe data--message and stream. While message data contains a defined USB structure, stream data does not.

  Additionally, pipes have associations of data bandwidth, transfer service type, and endpoint characteristics such as directionality and buffer sizes. Most pipes come into existence when a USB device is configured.

Chapter 2: Hawkview

This chapter discusses Hawkview, including its components and capabilities. The following sections are included:

RadiSys.

MICROWARE SOFTWARE

Hawkview Overview

Hawkview is a demonstration application included with the USB SDK. With a layout similar to the Windows Explorer, Hawkview allows you to browse and modify the OS-9 system from your Windows desktop.

From Hawkview, you can browse to specific sets of information, such as active devices, open paths, and filesystem directory contents. For example, depending on your board, the PCF descriptor for the IDE PC card may either be `mhe1` or `mhc1`. By clicking on the appropriate descriptor, you can view its current directory.

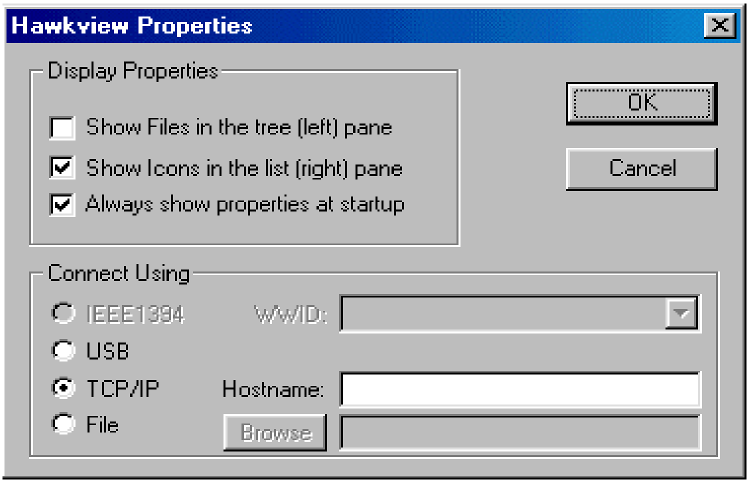
In addition, Hawkview allows you to browse current processes, IRQ vectors, events, and memory lists.

Running Hawkview

When the USB device driver has been `iniz`'d and the Hawkview daemon on the OS-9 target is running, you can start Hawkview. Complete the following steps to run Hawkview.

Step 1. Execute `hawkview.exe` on your Windows PC. The `Hawkview Properties` window should appear, as shown in **Figure 2-1** .

Figure 2-1



Step 2. Under `Connect Using`, select the USB radio button; click `OK`. Hawkiew attempts to connect to the USB device.

This connection attempt may fail, causing the following error message to appear:

```
USB device not present or not functioning
Couldn't connect via USB
```

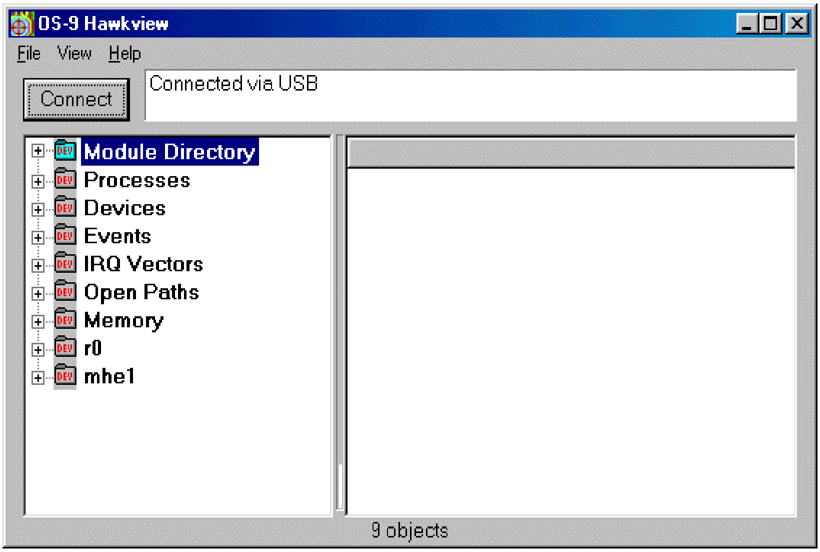
The list below shows some possible causes and solutions associated with the above error message:

- **The USB device is not connected**. To correct this error, simply plug in the USB device.
- **Device enumeration with the host failed.** There is an unknown device item under the USB items in the System Properties. To correct this error, cause a USB reset and try to connect with Hawkview again.
- **The Hawkview daemon is not running on the target.** Run `procs` on the target to make sure it is running. If not, run `hawkview &` to start the daemon.

Browsing the OS-9 Target

Once connected, Hawkview shows a list of browseable items in the left side of the OS-9 target window, as shown in **Figure 2-2** .

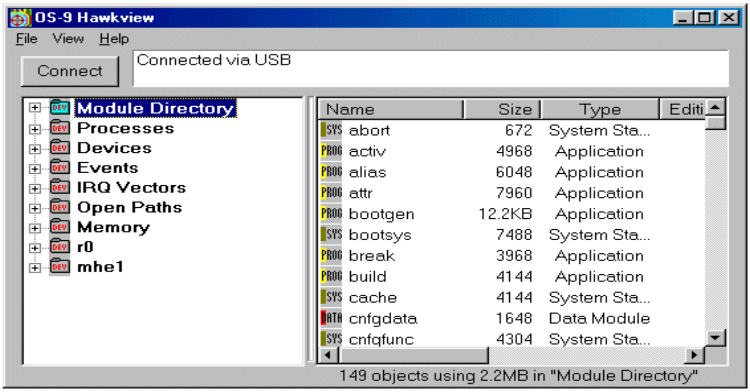
Figure 2-2



Module Directory

The module directory (shown in **Figure 2-3**) displays the current modules in memory on your board. From this directory you can view the current modules in memory on the OS-9 target board. The module directory also displays extended module information, such as who the owner of the module is, what type of module it is, what the module's edition number is, where it is in memory, and how much stack it requires.

Figure 2-3



In addition, you can retrieve modules from the target by right-clicking on the module, then selecting `Retrieve from Target`, and choosing a local directory on your PC.

Processes

The `Processes` item shows you the current state of every process in the system. This includes information such as the current process state, the CPU used, the number of OS and I/O calls, and the last system call made. Browse `Processes` if you would like extended information on processes currently running on the OS-9 target.

Devices

The `Devices` list shows all of the currently `iniz`'d devices in the system. For each of the descriptors, the list gives the name of the driver and location of the driver static storage. It is also useful for determining whether or not a specific driver is initialized.

Events

The `Events` list shows all of the current events that have been created on the system. If you are creating events in your application or driver, this list is useful for determining whether or not the event has been set up correctly.

IRQ Vectors

The `IRQ Vectors` list helps you find out which vectors are being used and which modules control a specific vector.

Open Paths

The `Open Paths` list allows you to see all of the open paths in the sytem and which processes have them opened.

Memory

The `Memory` item allows you to see what memory is allocated. For example, browse the `Memory` item if you are writing an application and the OS-9 system runs out of memory. For debugging purposes, you may want to learn where memory is being allocated and for what purpose.

You can also use the `Memory` item to view the OS-9 target's memory. Once you have chosen a specific set of memory, you can retrieve the contents of that memory space by right-clicking on the space and selecting `Retrieve from Target`.

Disk Devices

The disk devices are browseable file systems that navigate in much the same way as the Module Directory; they also give a similar type of information. Common disk descriptors are `r0`, `mhc1`and `mhe1`.

Modifying the OS-9 Target

The following sections describe ways in which Hawkview allows you to modify your OS-9 target.

Moving Modules

You can drag and drop modules from your Windows desktop to a disk device or the module directory of your OS-9 target. You can also drag modules from the OS-9 system disk devices to the Module Directory or another disk device.

Deleting Items

Hawkview allows you to delete items from the `Module Directory` list, `Events` list, and disk devices. To do this, right-click on the item you want to delete and choose `Delete` from the menu.

Creating Directories

Hawkview supports creating directories in the `Module Directory` or any of the disk devices. To do this, right-click on the directory and choose `New Folder`.

Chapter 3: Running USB

This chapter provides information on installing and running the Universal Serial Bus (USB) Software Developer's Kit (SDK). It includes the following sections:

- **Requirements and Compatibility**
- **Installing USB on the Target**

Requirements and Compatibility

The following describes the assumptions and requirements associated with using USB for OS-9.

Assumptions

This manual assumes you have installed OS-9 on your PC and created an OS-9 bootfile for your reference board.

Host Hardware Requirements (PC Compatible)

Your host PC should have the following hardware:

- a 266 MHz PC w/ Host USB (Universal Serial Bus) port
- an IDE PC card reader/writer
- an IDE PC flash card

Host Software Requirements (PC Compatible)

Your host PC should have the following software:

- Windows 98 w/ Service Pack 1 or Windows 2000
- Hawkview Application
- bulkusb.sys (located in `$(MWOS)/DOS/DRIVERS`)

Target Hardware Requirements

The device requires one of the following boards:

- SuperH 7709SE01 board
- Assabet board

Target Software Requirements

The device requires the following software:

- OS-9 or OS-9 for 68K port to target (with SoftStax)
- Hawkview Application for OS-9
- USB driver

Installing USB on the Target

Complete the steps below to install the USB SDK onto your target.

Note

For ARM users:

If you are installing USB on an ARM platform, you must set the system speed to 191 MHz in order for Hawkview to work properly.

Step 1. Copy the following files from your MWOS directory onto the PC card or in the boot:

- `$(MWOS)/OS9000/<PROCESSOR>/CMDS/hawkview`
- `$(MWOS)/OS9000/<PROCESSOR>/PORTS/<board>/CMDS/BOOTOBJS /SPF/<driver>`
- `$(MWOS)/OS9000/<PROCESSOR>/PORTS/<board>/CMDS/BOOTOBJS/SPF/usb0`

Step 2. Plug in all of the necessary cables, including those listed below:

- Power
- Serial
- USB

Boot your board, if necessary.

Step 3. At the OS-9 shell prompt, load the modules that have been placed on the PC card by entering the following command on the command line:

```
chd /
load -d hawkview <driver> usb0
```

Step 4. Initialize the OS-9 USB driver by typing `iniz /usb0` on the command line.

Step 5. Run the `hawkview` daemon in the background by entering `hawkview &` on the command line.

Step 6. Cause a USB reset by unplugging the USB cable from the board and plugging it back in.

Step 7. Using Windows Explorer, browse to `$(MWOS)/DOS/DRIVERS`; pick the `BULKUSB.INF` file to install the driver. This only must be done once on the Host PC.

If the driver has not been previously installed on your Windows 98 machine, Windows brings up the `Add New Hardware` wizard. After completing the fields in the wizard, proceed to step eight.

Step 8. Execute `hawkview.exe` on your PC. It is located in `$(MWOS)/DOS/BIN`.

The `Hawkview Properties` box appears. Select the `USB` radio button.

Step 9. Click `OK`.

Hawkview should display a tree of browseable items in the left side of the OS-9 target.

Chapter 4: Device Drivers

This chapter defines the design standards for an OS-9 *Soft*Stax Universal Serial Bus (USB) device driver implementation. It includes the following sections:

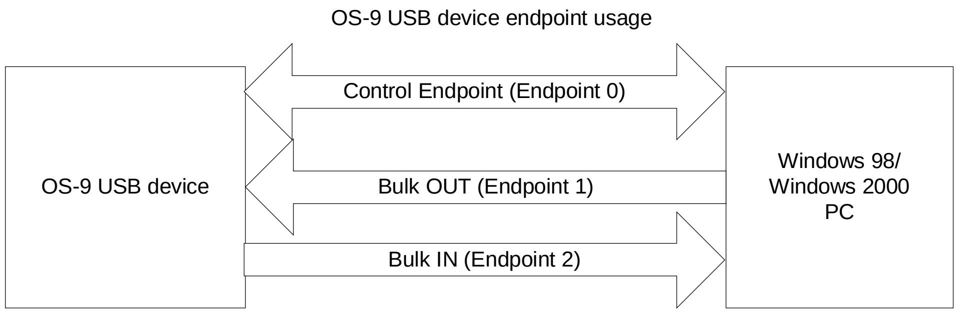
- **Overview of Device Driver**
- **Driver Files**
- **USB Device Drivers**
- **Common USB Driver Codes**
- **Common Code Structures**
- **Writing the USB Device Driver**
- **Writing Applications**

RadiSys.
MICROWARE SOFTWARE

Overview of Device Driver

The standard OS-9 Universal Serial Bus (USB) Device Driver is configured to use one control endpoint, endpoint 0. This endpoint negotiates with the USB host and two bulk endpoints, Bulk IN and Bulk OUT, for bi-directional communication with the host.

Figure 4-1 OS-9 USB Device Endpoint Use



Driver Requirements

OS-9 USB device drivers are innumerable through communication between endpoint 0 and the USB host. In order to run the Hawkview application, the base USB device driver must have two bulk endpoints (endpoint 1 is a Bulk OUT and endpoint 2 is a Bulk IN). All requirements of both the control and bulk endpoints come from the USB specification. However, certain hardware may have limitations, forcing a specific configuration.

Endpoint 0 should be capable of sending at least a 64-byte packet to accommodate descriptors needing returned. Endpoints 1 and 2 have no size requirements as long as the endpoint descriptor lists the maximum sizes.

Standard Driver Information

The following details conventions for driver and device descriptor names.

Driver Names

*Soft*Stax USB device side drivers usually start with the `spusbd` prefix. The `spusbd` denotes a *Soft*Stax USB device side driver. Examples include `spusbdsa`, `spusbdsl11`, and `spusbd823`.

Device Descriptor Names

Device descriptors for USB Drivers should be `usbX` (where `X` is a number). Most devices only have one device interface; therefore, they have `usb0` as their descriptor.

Driver Files

The following is a typical list of driver files found in the MWOS tree. These files implement a USB device side hardware driver and USB protocol.

```
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/cproto.h
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/entry.c
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/usb.c
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/pp_stg.c
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/<DRVR>/defs.h
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/<DRVR>/history.h
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/<DRVR>/proto.h
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/<DRVR>/main.c
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/<DRVR>/hardware.h
$(MWOS)/SRC/DPIO/SPF/DRVR/USBD/<DRVR>/hardware.c
$(MWOS)/<OS>/<CPU>/PORTS/<PORT>/SPF/<DRVR>/DEFS/spf_desc.h
$(MWOS)/<OS>/<CPU>/PORTS/<PORT>/SPF/<DRVR>/makefile
$(MWOS)/<OS>/<CPU>/PORTS/<PORT>/SPF/<DRVR>/spfdesc.mak
$(MWOS)/<OS>/<CPU>/PORTS/<PORT>/SPF/<DRVR>/spfdrvr.mak
$(MWOS)/<OS>/<CPU>/PORTS/<PORT>/SPF/<DRVR>/spfdbg.mak
```

Common Files

The following describes common files found in the above driver source:

`cproto.h`	contains prototypes for the common functions
`entry.c`	contains all of the entry points called by the SPF file manager These calls are detailed in the **Device Driver Entry Points** section of this document.

`usb.c`	contains the code necessary to answer requests from the host to enumerate the device The **Common Code Structures** section of this document discusses these functions in detail.
`pp_stg.c`	contains the per-path storage functions for the driver

Driver Specific Files

`defs.h`	contains all of the SPF structure definitions needed to compile for a specific USB device driver This file includes `spf.h`, `proto.h`, `history.h`, and `hardware.h`. For USB, this file contains macros extending the SPF data structures. The **USB Device Drivers** section of this document outlines these macros.
`history.h`	contains the driver edition history and macros defining the edition of the driver
`proto.h`	contains all of the function prototypes for this driver Each time a new function is added to the driver, its prototype should be added to this file.
`main.c`	contains the initialized data for the driver static storage (`spf_drstat`)
`hardware.h`	contains all of the necessary hardware specific definitions
`hardware.c`	contains hardware specific functions, such as `hw_init()`, `hw_isr()`, `hw_term()`, etc.

USB Device Drivers

The following section describes device driver structures and endpoints.

Device Driver Structures

The following device driver structures can be found in `defs.h`.

SPF_DRSTAT

The SPF_DRSTAT macro in `defs.h` extends the `spf_drstat` SPF structure. The USB common code requires the following definitions to be in the driver static storage. Any further definitions are driver-specific.

```
#define SPF_DRSTAT \
  usb_desc_block  *descriptors; \
  error_code     (*cache_cctl)(u_int32 control, void *addr, u_int32 size); \
  void            *dr_cglobs;  \
  ... \
  ...
```

The initializer for the `spf_drstat` structure goes into `main.c`.

```
spf_drstat dr_stat = {
  SPF_VERSION,                      /* dr_version */
  NULL,                             /* dr_fmcallup */
  dr_iniz,                          /* dr_iniz */
  dr_term,                          /* dr_term */
  dr_getstat,                       /* dr_getstat */
  dr_setstat,                       /* dr_setstat */
  dr_downdata,                      /* dr_downdata */
  dr_updata,                        /* dr_updata */
  0,                                /* dr_att_cnt */
  NULL,                             /* dr_lulist */
  DR_ALLOC_LU_PERPORT,              /* dr_lumode */
  { 0 },                            /* dr_rsv1[] */
  0,                                /* dr_use_cnt */
  &descriptors,                     /* descriptor block */
  NULL,                             /* cache_cctl() */
  &_bdata,                          /* globals */
  ...,                              /* additional data */
  ...                               /* additional data */
};
```

SPF_LUSTAT

The SPF_LUSTAT macro in defs.h extends the spf_lustat structure. All the fields in this structure are driver-specific extensions for the logical unit stat. This macro looks similar to the following macro:

```
#define SPF_LUSTAT \
  Pp_udc_stat  lu_ppstat;       /* Per path static                   */\
  pp_udc_stat  lu_sdlc_const;   /* default initial values for ppstat */\
  void*        lu_dbg;          /* debugging pointer                 */\
  char         lu_dbg_name[16]; /* Name of debug module              */\
  u_int32      lu_irqlevel;     /* IRQ level                         */\
  u_int32      lu_vector;       /* IRQ vector number                 */\
  u_int32      lu_priority;     /* IRQ polling priority              */\
  u_int32      lu_irqmask;      /* IRQ mask level                    */\
  void*        lu_cache_static; /* static storage for cache_cctl()   */
```

The macro SPF_LUSTAT_INIT (shown below) lists the initial values for the extensions above to the spf_lustat structure.

```
#define SPF_LUSTAT_INIT \
  NULL,              /* lu_ppstat       */\
  DEFAULT_PPSTAT,    /* Default values  */\
  NULL,              /* lu_dbg          */\
  {DEBUG_NAME},      /* lu_dbg_name     */\
  IRQLEVEL,          /* lu_irqlevel     */\
  IRQVECTOR,         /* lu_vector       */\
  PRIORITY,          /* lu_priority     */\
  IRQMASK            /* lu_irqmask      */
```

spf_desc.h

There are no special settings for USB. Refer to the ***Using Soft*Stax** manual for more information about settings in spf_desc.h that are pertinent to building descriptors and device drivers.

Device Driver Entry Points

The USB device driver contains the following *Soft*Stax entry points:

entry.c

```
error_code dr_iniz(Dev_list deventry)
error_code dr_term(Dev_list deventry)
error_code dr_getstat(Dev_list deventry, Spf_ss_pb pb)
error_code dr_setstat(Dev_list deventry, Spf_ss_pb pb)
error_code dr_downdata(Dev_list deventry, Mbuf mb)
error_code dr_updata(Dev_list deventry, Mbuf mb)
```

`dr_iniz()`

entered only if no other device descriptors are currently attached (iniz'd) to the USB driver

First, `dr_iniz()` installs the hardware interrupt service routine (ISR). Next, it initializes the USB hardware on a specific platorm by calling `hw_init()`.

`dr_term()`

disables the USB hardware with a call to `hw_term()` and removes the installed ISR

The file manager calls `dr_term()` when the last path is closed on the device (`deiniz'd`).

`dr_getstat()`

All of the *Soft*Stax drivers have the `SPF_SS_UPDATE` entry point (explained below):

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

`dr_setstat()`

This entry point handles the `SPF_SS_OPEN` and `SPF_SS_CLOSE` setstat subcodes (explained below):

- `SPF_SS_OPEN` calls the adjacent upper-layer protocol at its `dr_setstat` with subcode `SPF_SS_UPDATE` to indicate the driver is ready for I/O.
- `SPF_SS_CLOSE` returns the device list entry of this driver's adjacent lower-layer protocol. Since this is a device driver at the lowest level, the `NULL` pointer is returned.

`dr_downdata()`	initiates the transfer of an mbuf by calling `hw_xmit()`
`dr_updata()`	called by the driver to send a received mbuf up the *Soft*Stax stack to the application

USB Driver Hardware Routines

The following functions are called from common code (`entry.c` or `usb.c`). They are hardware specific functions; thus, they need to be written for each USB device driver.

hardware.c

```
error_code hw_init(Dev_list dev)
error_code hw_term(Dev_list dev)
error_code hw_isr(Dev_list dev)
error_code hw_xmit(Dev_list dev, Mbuf mb)
error_code hw_setaddr(Dev_list dev, u_int8 address)
void hw_ep0_sendByte(Dev_list dev, u_int32 index, u_int8 byte)
void hw_ep0_sendBlock(Dev_list dev, u_int8 *src, u_int32 len)
void hw_ep0_sendDone(Dev_list dev, u_int32 len)
```

`hw_init()` — `dr_iniz()` calls this function when a path to the device is opened.

This function is responsible for initializing the hardware and allocating any memory required so the device can communicate

	with the USB host. This function is the appropriate place to initialize the DMA hardware, if applicable.
`hw_term()`	`dr_term()` calls this function when the last path to this device has been closed. `dr_term()`undoes the work that `hw_init()`performed. Its responsibilities include turning off the USB device, de-allocating the memory allocated by `hw_init()`, and, if necessary, turning off DMA.
`hw_isr()`	This function is the interrupt handler. The OS-9 interrupt handler calls `hw_isr()`each time the USB device gets an interrupt. Usually, this means that the state of the line or the state of the device has changed. Possible state changes include a packet received by the USB device, a packet requested by the host, a USB device suspended or resumed by the host, or an error condition. For each of these changes, a bit in the status register (SR) changes.
`hw_setaddr()`	The common code `handle_device_request()`calls this function to change the USB address of the USB device. Setting the USB device address is hardware-specific; thus, this function triggers a change after the next interrupt to the USB device.
`hw_xmit()`	This function, called by `dr_downdata()`, is responsible for passing the data in the mbuf from the application to the correct endpoint.

	It is also responsible for initializing a DMA transfer if DMA is used or copying the data directly to an outgoing FIFO. `hw_xmit()` must also keep track of the size of the data in case a transmit error occurs or the mbuf data is found to be larger than the outgoing FIFO can handle.
`hw_ep0_sendByte()`	This function is called by the USB common code to add a byte to the send buffer for endpoint 0.
`hw_ep0_sendBlock()`	This function loads a block of data into the send buffer for the control endpoint. The USB common code in `usb.c` calls this function.
`hw_ep0_sendDone()`	This function sets the length of the control endpoint's send buffer and to start sending the data. This function sets the length of the control endpoint's message and notifies the driver that the message is ready to be sent. Primarily, the USB common code in `usb.c` calls `hw_ep0_sendDone()`. In addition, it sends an empty packet or ends a data transaction in the driver code.

Common USB Driver Codes

The following is a list of common USB driver codes.

usb.c

```
error_code handle_device_request(Dev_list dev)
error_code handle_class_request(Dev_list dev)
error_code handle_vendor_request(Dev_list dev)
```

`handle_device_request()`

responsible for answering all of the standard device requests as stated in the USB Specification

This function handles the return of all of the configuration descriptors and makes the call to `hw_setaddr()`to set the USB hardware address.

`handle_class_request()` *Soft*Stax USB device drivers are currently not required to implement any class specific requests.

This function is present for completeness.

`handle_vendor_request()`

responsible for answering vendor-specific requests from the device

On some platforms, the function is used to work around specific USB hardware bugs.

Common Code Structures

The following provides a list of common codes structures.

usb_desc_block

This structure passes buffers and the USB device descriptor information from the driver proper to the USB common code. The common code uses `indesc` to read incoming descriptor data. `indesc` points to the input buffer where the device driver reads a request descriptor from the host.

The lengths and data for device and configuration descriptors include `device_len`, `device`, `config_len` and `config`. The common code uses these to pass descriptor information back to the USB host. A pointer to this structure is required in the `spf_dr_stat` structure.

```
typedef struct {
  usb_device_request *indesc;    /* pointer to the request descriptor */
  u_int32             device_len;/* length of the device descriptor */
  u_int8             *device;    /* pointer to the device descriptor */
  u_int32               config_len;/* length of the configuration descriptor */
  u_int8             *config;    /* pointer to the config. descriptor */
} usb_desc_block;
```

The following is an example of initialization for this structure:

```
usb_device_request inDesc[2];

#define DEVLEN 0x12
unsigned char udc_device[] = {
 0x12,USB_WVAL_DEVICE,0x10,0x01,0xff,3,0,0x40,0x5e,4,0x0a,0x93,0,0,0,0,0,1
};

#define CONFLEN 32
unsigned char udc_conf[] = {
  9,USB_WVAL_CONFIGURATION,CONFLEN,0,1,1,0,0x80,0x00,
  9,USB_WVAL_INTERFACE,0,0,2,0,0,0,0,
  7,USB_WVAL_ENDPOINT,0x82,2,0x40,0,0,
  7,USB_WVAL_ENDPOINT,0x01,2,0x40,0,0,
};

usb_desc_block descriptors = {
  &inDesc[0],
  DEVLEN,
  udc_device,
  CONFLEN,
  udc_conf
};
```

Writing the USB Device Driver

The following are the requirements for writing a device driver:

- OS-9/OS-9000 port to the platform
- *Soft*Stax port to the platform
- Win98 or Win2000 PC w/ USB host controller and Microsoft Windows 2000 DDK
- USB hardware knowledge
- USB traffic analyzer (recommended)
- VID and PID for USB device driver

The steps to write a USB device driver are described in the **Create New Driver Source Directory and Makefiles** section.

Choose a Hardware Solution

Choose hardware for the USB device that meets the needs of the project. For this example, Assabet is the board running OS-9 and the SA-1100 is the CPU. The SA-1100 has an on-chip ASIC for USB. The hypothetical example driver name is `spusbdsa1100`. The setup for this driver is one control endpoint, one Bulk IN endpoint, and one Bulk OUT endpoint.

Create New Driver Source Directory and Makefiles

The following steps lead you through creating a new (sample) driver source directory and makefile:

Step 1. Copy the files from $(MWOS)/SRC/DPIO/SPF/DRVR/USBD/SAMPLE to $(MWOS)/SRC/DPIO/SPF/DRVR/USBD/SPUSBDSA1100.

Step 2. Copy the files from $(MWOS)/OS9000/SAMPLES/USB/SPF/EXAMPLE to $(MWOS)/OS9000/ARM4/PORTS/<ASSABET>/SPF/SPUSBDSA1100.

Step 3. Make the following changes to spfdrvr.mak and spfdbg.

change the following:

```
TRGTS       =    spsample
TRGT_FNAME  =    spsample
PICSUB      =    # PICLIB
```

to the following:

```
TRGTS       =    spusbdsa1100
TRGT_FNAME  =    spusbdsa1100
PICSUB      =    -l=$(PORT)/LIB/pic1100.l
```

Step 4. Make the following changes in spf_desc.h.

change the following:

```
#define PRIORITY    0
#define IRQVECTOR   0x00
#define PORTADDR    0x00000000
#define IRQLEVEL    0
#define DRV_NAME    "spsample"
```

to the following:

```
#define PRIORITY    1
#define IRQVECTOR   0x4d
#define PORTADDR    0x80000000
#define IRQLEVEL    4
#define DRV_NAME    "spusbdsa1100"
```

Step 5. Modify the descriptors pointed to in usb_desc_block structure in hardware.c.

device_desc[] and config_desc[] are device-dependent structures that describe the device to the host. They must be accurate concerning the specification of the device; they tell the host which driver is to be used for the device, describe the endpoints, and give the host input on power consumption.

More information about these descriptors can be found in the ***The Universal Serial Bus Specification*** in the web site address, http://www.usb.org.

Step 6. Add code to hw_init() and hw_term() in hardware.c.

You should be able to iniz’d and deiniz’d the device after completing the code in these two functions.

If the activated device causes interrupts, you can set a breakpoint on hw_isr() before the device is iniz’d; the debugger should then stop on hw_isr(). Setting a breakpoint on hw_isr() verifies that the device is receiving interrupts and that you have initialized it properly.

Step 7. Add code to hw_isr() in hardware.c to handle interrupts.

hw_isr() is responsible for handling all of the interrupts that the USB device can generate. Usually, a change in the state of bits in the status register identifies these interrupts. However, only the hw_isr() routine handles all of the possible interrupts. In addition, the most important action is to implement code to handle an endpoint 0 interrupt in order to get the device enumerated by the host.

Place the data received by an endpoint 0 interrupt into the inDesc structure. Then, make a check to determine what type of request this is (device, class, or vendor) and call the correct handling routine (handle_devcie_request(), handle_class_request(), or handle_vendor_request()).

The code should look similar to the code below:

```
{
  u_int8 type;
  type = (indesc->bmRequestType & USB_BMREQ_TYPE_MASK);
  if (type == USB_BMREQ_TYPE_STANDARD) {
    handle_device_request(dev);
  } else if (type == USB_BMREQ_TYPE_CLASS) {
    handle_class_request(dev);
  } else if (type == USB_BMREQ_TYPE_VENDOR) {
    handle_vendor_request(dev);
  }
}
```

Step 8. Add code to `hw_ep0_xxx()` and `hw_setaddr()` routines in `hardware.c`.

- `hw_ep0_sendByte()` adds a byte to the send buffer for transmitting data back to the host.
- `hw_ep0_sendBlock()` adds a block of bytes to the transmit buffer for transmitting data back to the host.
- `hw_ep0_sendDone()` signifies the end of processing for the input request and tells the driver that the buffer is ready to be sent.

The following code shows a possible way to implement the `hw_ep0_xxx()` routines (`obuffer[]` is the buffer the interrupt handler uses to transmit data, while `obuffer_ready` is a boolean indicating the buffer is ready to send and `obuffer_length` is the length of the descriptor to pass to the host):

```
u_int8 obuffer[256];  /* the output buffer for transmission on endpoint 0 */
u_int32 obuffer_ready=0;/* boolean, is the buffer ready for transmission */
u_int32 obuffer_length; /* output buffer length */

void hw_ep0_sendByte(Dev_list dev, u_int32 index, u_int8 byte)
{
  u_int8 *dest = ((u_int8*)&obuffer[0]) + index;

  *dest = byte;
}

void hw_ep0_sendBlock(Dev_list dev, u_int8 *src, u_int32 len)
{
  u_int8 *dest = &obuffer[0];

  while (len--) {
    *(dest++) = *(src++);
  }
}
void hw_ep0_sendDone(Dev_list dev, u_int32 len)
{
  obuffer_length = len;
  obuffer_ready = 1; /* this is cleared by the transmitter when finished */
}
```

When it receives the descriptor, `handle_device_request()` calls `hw_setaddr()` to set its device address. On some devices, you can set the address right away, but in others the address waits for the next message from the host controller. The behavior is device-dependent.

Upon completion of these steps and when the USB cable is plugged in, the host should enumerate the USB device. The errata for the SA-1100 tells the driver writer that the USB cable should be connected before the code touches USB registers. This may not be true with other hardware.

Step 9. Add code to `hw_isr()`.

Since the host has enumerated this device, add code to `hw_isr()` to receive information on the Bulk OUT endpoint. *Soft*Stax USB device drivers use mbufs to pass information up the SPF stack to the application. If DMA is used, data can be copied directly into an mbuf for transfer to the application. If there is no DMA, the FIFO must be copied by the CPU into an mbuf for transfer to the application.

Since the SA-1100 accesses its FIFOs using DMA, DMA directly into an mbuf. This means that you also need to flush the cache before Bulk data is received into the mbuf. Once there, it can be sent up the SPF stack using `DR_FMCALLUP_PKT (dev, dev, mbuf)`.

You can now test using the dump command on the target and `rwbulk.exe` from the Windows host. You can use `rwbulk` to send data to the device; `dump` displays this data on the OS-9 console. When everything is correct, proceed to the next step.

Step 10. Add code to `hw_xmit()`.

`hw_xmit()` receives an mbuf and queues it to a list to send to the host or start the send process if the queue is empty. If beginning the process, `hw_xmit()`is starts the DMA on the mbuf or copies the data from the mbuf to the outgoing FIFO. However, if the mbuf is added to a queue, `hw_isr()` sends each mbuf in the queue, but only when data is requested from the USB host (via an IN packet for that endpoint).

`hw_xmit()`should also use DMA to transmit the mbuf on the SA-1100.

Step 11. Test with `rwbulk.exe` and the loopback program in the **Writing Applications** section of this document; then, test using hawkview.exe for Windows and hawkview daemon for OS-9.

Writing Applications

The following is an example of a simple loopback program. When run, it opens the USB descriptor, giving it access to a bulk read and write endpoint. Following this, it loops, blocking on a read of a 64-byte packet. When it receives a packet it writes the same data back to the host.

```
#include <stdio.h>
#include <errno.h>
#include <modes.h>
#include <types.h>

#define DRVR_NAME "/usb0"
char * name = DRVR_NAME;
u_int32 mode = S_IREAD | S_IWRITE;
path_id ppid;

#define BUF_LEN 0x40
unsigned char buffer[BUF_LEN];

void main(int argc, char *argv[])
{
  error_code err;
  u_int32 count = BUF_LEN;

  err = _os_open(name, mode, &ppid);
  if (err) {
    exit(_errmsg(err, "Can't open: %s", name));
  }

  while (1) {
    if ((err = _os_read(ppid, buffer, &count)) == EOS_EOF) {
      _os_close(ppid);
      exit(_errmsg(err, "Reached EOF on %s", name));
    }
    _os_write(ppid, buffer, &count);
  }
}
```