Using JetBeam for OS-9

Version 2.1.1

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 D
August 2001

Copyright and publication information

This manual reflects version 2.1 of JetBeam for OS-9. Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

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

Reproduction notice

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

August 2001

Table of Contents

Chapter 1: Infrared Communication An Overview 7

Chapter 2: Using JetBeam™ for OS-9 21

Chapter 1: Infrared Communication An Overview

This chapter briefly describes infrared communication and gives an overview of the following topics:

- An introduction to infrared communication
- Using infrared technology
- IrDA—Setting industry standards for infrared communication
- IrDA-compliant system design basics
- Optional IrDA-compliant features
- IrDA Lite—minimum implementation standards

Note

This chapter provides a description of basic infrared communication principles. It is not comprehensive and does not include information specific to JetBeam™ for OS-9®. If you are already familiar with infrared communication, please skip to **Chapter 2: Using JetBeam™ for OS-9**.

An Introduction to Infrared Communication

Infrared (IR) communication technology connects computers, peripherals, hand-helds, and various other electronic devices without using cables or wires. Depending on physical proximity, infrared communication technology eliminates the need for cables to send a print job from a personal computer to a printer, or a data file from a PC to a notebook computer. Data is passed from one device to another via an infrared data link—using infrared lightwave frequencies. The technology that makes this possible is based on that used in television and VCR remote controllers.

Some key aspects of infrared data communication include the following:

- There is an international standard for all IR communication devices and software.
- IR communication requires a “line-of-sight” between devices. This means that two devices must have a clear path between them when communicating.
- IR communication is directional (communicating devices must be aimed at one another).
- Infrared data does not penetrate solid objects (walls, ceilings, etc.).
- IR communication does not require a license from the Federal Communications Commission (FCC).

Using Infrared Technology

If it works over a wire or cable, it will work over an infrared data link.

That has been the de facto motto for developers of infrared communication technology. Infrared technology provides a means to move data between computers and other devices—once only a chore for cables, interface cards, and interface software.

Applications

Some applications of infrared communication include the following:

- printing files
- synchronizing electronic telephone books and schedulers
- synchronizing files between portables and desktop computers
- exchanging data files between portable computers and network nodes
- exchanging data files between desktop computers and personal digital assistants (PDAs)
- exchanging business cards between handheld computers
- sending and receiving faxes or email directly from a notebook PC through a cellular or traditional public telephone
- storing bank records from ATM machines
- accessing home entertainment, security, and automated environmental control devices

IrDA—An Industry Standard

The Infra-red Data Association (IrDA) is an international group of component manufacturers, original equipment manufacturers, and hardware and software companies that promotes an industry standard for infrared communications. The association was founded in 1993.

IrDA's stated mission is to "create an interoperable, low-cost, low-power, half-duplex, serial data interconnection standard that supports a walk-up, point-to-point user model adaptable to a broad range of computing, communications, and consumer devices."

For More Information

IrDA
P.O. Box 3883
Walnut Creek, CA 94598
510-943-6546 (voice)
510-943-5600 (fax)
www.irda.org (WWW address)

Setting Industry Standards

The IrDA specifications consist of a series of documents that list the industry standards for the physical layer, link access, and link management. There are also documents specifying communications, transport, and object exchange protocols, which are all optional features of the IrDA standard. In addition, IrDA has specifications for a minimal IrDA protocol, called IrDA Lite, as well as plug and play extensions.

Some of the specifications at the physical layer include peak wavelength, cone size, and pulse rise and fall time. Some specifications at the data link layer include baud rates supported, data size, transceiver and receiver window sizes supported, and controls for primary/secondary role exchange.

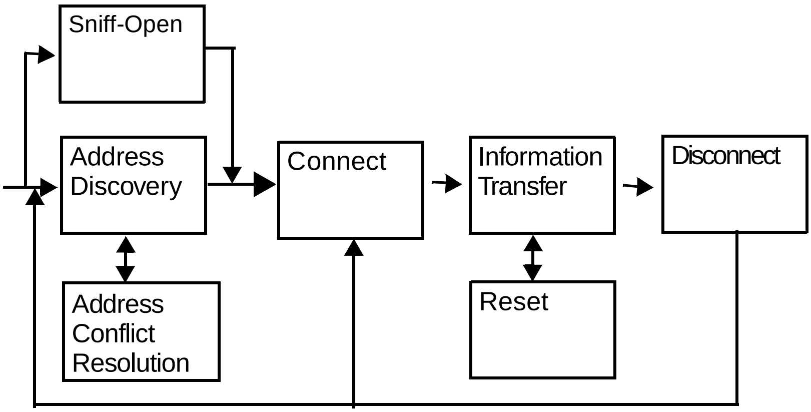
IrDA-Compliant System Design Basics

IrDA standards specify both the physical and data-link layer protocols for infrared (IR) data transfer. The protocols are based on a pre-existing half-duplex protocol developed by Hewlet Packard. In 1994, IrDA issued its first IR data transmission standard—for a 10-foot distance and an initial throughput rate of 115.2 Kpbs. Since that time, new standards have been issued for faster transfer rates. An IR data link basically operates in the following manner:

1. A device attempts to connect to another device—either by automatic detection or by direct user request.
2. Following specific media access rules, the initiating device (**primary**) sends connection request information to the other device (such as device address, data rate, data size limits, etc.).
3. The responding device assumes the **secondary** role. After obeying the media access rules, it returns its address and capabilities information.
4. The primary and secondary devices then change the data rate and other parameters to a common set defined during the information transfer.
5. The primary then confirms the link with the secondary.
6. The two devices are now connected and data is transferred between them under complete control of the primary.

The life cycle of an IR communication link is illustrated in Figure 1-1 below.

Figure 1-1 Life Cycle of an IrDA Link



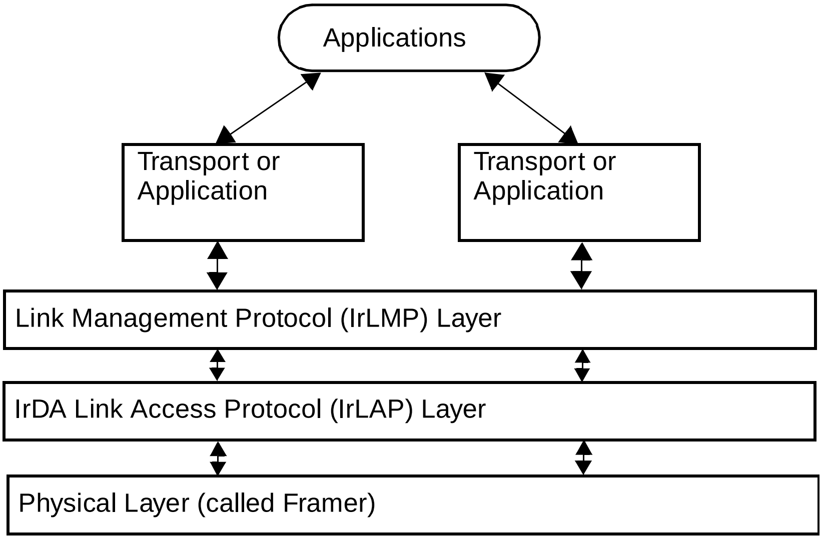
Basic IrDA System Components

The IrDA Serial Infrared Data Link Standard (version 1.1) describes a protocol stack with the following three mandatory components:

- the physical layer
- the infrared (IR) link access protocol layer
- the IR link management protocol layer

Optional sections include high-speed extensions (1.15 and 4.0 Mb/sec), and various flow control mechanisms. A standard IrDA protocol stack is represented in Figure 1-2.

Figure 1-2 The IrDA Protocol Stack


Applications
Transport or Application
Transport or Application
Link Management Protocol (IrLMP) Layer
IrDA Link Access Protocol (IrLAP) Layer
Physical Layer (called Framer)

The Physical Layer

The physical link layer is based on the common asynchronous serial port. It is a half-duplex system with the maximum UART (universal asynchronous receiver/transmitter) based data rate of 115.2 kbps. The data rate, however, can be programmed from software to a lower data rate to match slower devices. The IR LED peak wavelength ranges from 0.85 µm to 0.90 µm.

Note

The IrDA Serial Infrared Data Link Standard (version 2.0) defines non-UART environments with high speed (1.15 and 4.0 M/sec) options.

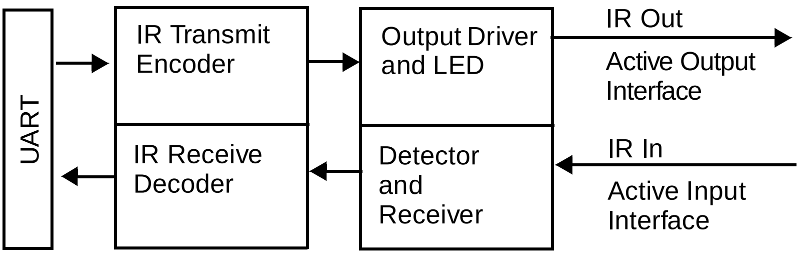
The physical hardware consists of the following two elements:

- encoder/decoder (which performs the IR transmit encoder and IR receiver decoder functions)
- IR transducer (which includes the output driver and IR emitter for transmitting and the receiver/detector)

The encoder/decoder interfaces to the UART, which is already present in most devices.

During a link, the output from a standard asynchronous serial character stream from the UART is encoded—with “0” represented by a pulse and “1” represented by no pulse. This pulse stream forms the input to the driver. The IR emitter then converts the electrical pulses to IR energy.

Figure 1-3 Typical IrDA Hardware Implementation



Note

For more information, see IrDA (Infrared Data Association) *Serial Infrared Physical Layer Link Specification*, Version 1.1, October 17, 1995.

The IR Link Access Protocol (IrLAP) Layer

The IR Link Access Protocol (IrLAP) layer uses services provided by the physical layer (below it) and provides services to the layer above it. IrLAP establishes the IR media access rules and manages various procedures, such as discovery, negotiation, and information exchange. IrLAP is adapted from HDLC (high-level data link control), an existing asynchronous data communications standard.

Note

For more information, see IrDA (Infrared Data Association) *Serial Infrared Link Access Protocol (IrLAP) Specification*, Version 1.1, June 16, 1996.

The IR Link Management Protocol (IrLMP) Layer

The IR Link Management Protocol (IrLMP) layer uses services provided by the data link layer (IrLAP) and provides services to higher layers (for example transport entities and applications).

The IrLMP layer consists of two components—the Link Management Information Access Service (LM-IAS) and the Link Management Multiplexer (LM-MUX). LM-IAS maintains an information base so that other IrDA stations can inquire what services are offered on the device. LM-MUX provides multiple data link connections over the single connection provided by IrLAP.

Note

For more information, see IrDA (Infrared Data Association) *Link Management Protocol Specification*, Version 1.1, January 23, 1996.

Link Management Multiplexer (LM-MUX)

Whereas IrLAP (the bottom layer of IrDA's protocol stack) provides an infrared data connection between two devices, the Link Management Multiplexer (LM-MUX) provides multiple data link connections over IrLAP. LM-MUX operates in one of two modes—exclusive mode or multiplexed mode. Exclusive mode allows only one active connection. The multiplexed mode allows several connections to actively share the same underlying IrLAP connection.

WARNING

When operating LM-MUX in multiplexed mode, the IrLAP flow control is NOT a suitable mechanism for providing flow control. Deadlock problems can result. In this case, additional flow control must be provided by upper layers of the protocol stack or by the applications.

Information Access Service (LM-IAS)

The Link Management Information Access Service (LM-IAS) maintains information about the services provided by a particular IR device. It also allows the device to remotely access the information base on another device. This feature allows different IR devices to "describe" themselves to each other—providing essential parameters necessary to establish a connection. An IR device's information base contains a number of objects, each associated with a set of attributes. For example, `Device` is a mandatory object and has the attributes `DeviceName` (an ASCII string) and `IrLMPSupport` (IrLMP version number, IAS support, and LM-MUX support).

Optional IrDA-compliant Features

The physical layer, IrLAP, and IrLMP are the only mandatory layers in the IrDA standard. These layers provide the basic necessities for an infrared data link. The IrDA design, however, is open-ended, and there are many options for the upper layer. Some of these options provide flow control, serial and parallel port emulation, object exchange, as well as many others. Most of these features make adopting and developing new application programs easier. Some of these options are briefly described below.

IrTP and TinyTP

IrTP and TinyTP are transport protocols. They provide flow control functions to segment and reassemble data. This additional flow control is necessary when LM-MUX is in the multiplexed mode. While in multiplex mode, LM-MUX does not provide flow control and does not indefinitely buffer data from the IrLAP layer. Without additional flow control, LM-MUX may discard data when a client is unable to accept it.

IrCOMM

IrCOMM is a protocol that emulates pre-existing wired serial and parallel ports. This allows devices without IR capabilities (such as an older model personal computer) to be easily converted for IR communication use.

IrOBEX

IrOBEX (infrared object exchange) is a binary protocol that enables devices to exchange data in a simple and spontaneous manner. IrOBEX is being developed by members of IrDA, but the object exchange function is not limited to infrared communications.

IrOBEX allows ubiquitous communications between portable devices or devices in dynamic environments. It does this as a “Squirt” or “Slurp” application. In one case, a desktop PC “Squirts” a file to a printer. In another use, an industrial computer “Slurps” diagnostic information from a piece of factory floor machinery.

IrOBEX consists of two major parts: an object model with information that describes the objects, and a session protocol that provides a structure for the exchange of information between devices.

IrDA Lite—Minimum Implementation Standards

An important feature of the IrDA protocol is that not all features included in the specifications are mandatory for IrDA compliance. This has led to the development of IrDA Lite. The IrDA Lite specification defines the minimum standards for IrDA implementation. For example, one of the IrDA standards not required under IrDA Lite (at the IrLAP layer) is no support for optional NDM features including sniffing, connectionless UI frames, and TEST frames. Another is to ignore all frames that do not have the broadcast address.

This flexibility in IrDA Lite reduces the complexity and code size of the IrDA protocols while maintaining the capability to communicate with existing IrDA devices. This lets devices with simple data communication needs implement IrDA protocols with a minimum of complexity in a small amount of RAM and ROM.

The IrDA standards allow developers to implement as much or as little of the IrDA Lite specification as desired. Therefore, it is possible to use as few or as many features described in IrDA Lite as needed.

Note

For more information, including a complete list of minimum standard to implement IrDA Lite, see IrDA (Infrared Data Association) *Minimum IrDA Protocol Implementation*, Version 1.0, November 7, 1996.

Chapter 2: Using JetBeam™ for OS-9

This chapter describes JetBeam for OS-9 and includes the following sections:

- **Installing JetBeam for OS-9**
- **Loading JetBeam for OS-9**
- **Overview of JetBeam for OS-9**
- **JetBeam for OS-9 Protocol Module**
- **Framer device driver**
- **Using JetBeam for OS-9 at the Application Level**
- **JetBeam for OS-9 Applications**
- **IrDA Library**

Installing JetBeam for OS-9

Before Installing the JetBeam Software

Before you install JetBeam™ for OS-9 onto your host computer, you must have first installed either ***Enhanced OS-9 for Embedded Systems*** onto your host computer.

Also, you need to make sure that your Host and Target systems meet the Hardware and Software Requirements of JetBeam™ for OS-9.

Host Requirements for JetBeam for OS-9

- One Megabyte of free disk space for JetBeam for OS-9.
- Windows 95/98 or Windows NT 4.0
- Enhanced OS-9 for Embedded Systems for StrongARM or SH-3
- Microware Hawk

Target Requirements for JetBeam for OS-9

SoftStax 2.2 and JetBeam for OS-9 are compatible with the following hardware and software:

- Reference board for SH-3 processor family (Hitachi SH7709SE01, Hitachi SH7709ASE01, or Hitachi EBX7709)
- Microware OS-9 with SoftStax enabled

Installing the JetBeam Software

Run the installer program from the CD to install JetBeam for OS-9 into the `mwos` directory of your host system.

After Installing the JetBeam Software

When the installer program is run, a list of the files and modules that are placed into the `mwos` directory is generated and stored in the following file:

```
mwos\jetbeam_<current version>.log
```

For example, if you install **JetBeam for OS-9** Version 2.1, the list of files and modules added to the `mwos` directory and its subdirectories would be found in the file `jetbeam_v2.1.log` in the `mwos` root directory.

Loading JetBeam for OS-9

Modules for Enabling IrDA

For JetBeam to work, the following modules need to be loaded into the module directory: ir1, spirlite, irdalite, sp1110ir or sp7709ir. The drivers are sp1110ir (driver for the SA-11x0), sp7709ir (driver for the SH-3), and irdalite, the framer descriptor is ir1, and spirlite is the IrDA Lite™ stack.

WARNING

Do not load both the sp1110ir and the sp7709ir drivers onto the same reference board. They are processor specific and will work only with their designated processor. The sp1110ir driver is only for the SA-11x0 and the sp7709ir driver is only for the SH-3. The driver irdalite is processor independent.

These modules can be loaded either through Hawk or they can be included in the bootfile with the Configuration Wizard. To load the modules into memory from Hawk, use the **Load** command found under the **Target** menu. Each file will need to be loaded individually.

To include the modules into a bootfile with the Configuration Wizard, the following tasks need to be performed:

- The names of the modules with their search paths need to be entered into the user.ml file.
- A new bootfile or ROM image needs to be created with the IrDA modules included as user modules.

Modifying the user.ml file

To modify the `user.ml` file, you need to open the file in Configuration Wizard.

Step 1. Start the wizard with the configuration you used to create your bootfile.

Step 2. Make sure the Advanced Mode is selected and click OK to open the advanced mode window.

Step 3. Open the user.ml file by selecting `Sources->PORT->User` from the menu bar.

Enter the following paths and file names for the IrDA modules, save your changes and close the file. The paths you should use are:

For StrongARM

```
../../../../../CMDS/BOOTOBJS/SPF/spirlite
../../../../../CMDS/BOOTOBJS/SPF/irdalite
../../../CMDS/BOOTOBJS/SPF/ir1
../../../CMDS/BOOTOBJS/SPF/sp1110ir
```

For SH-3

```
../../../../../CMDS/BOOTOBJS/SPF/spirlite
../../../../../CMDS/BOOTOBJS/SPF/irdalite
../../../CMDS/BOOTOBJS/SPF/ir1
../../../CMDS/BOOTOBJS/SPF/sp7709ir
```

Adding the modules to the Bootfile or ROM image

Once you have made the changes to the `user.ml` file in Configuration Wizard. You need to build a new bootfile or ROM image with the new IrDA modules included. Do the following steps to add the IrDA modules to the bootfile:

Step 1. Open the Master Builder screen by clicking on the **Build Images** button. The **Build Images** button is labeled with a **BI**.

Step 2. Click on the **User Modules** check box. This will tell Configuration Wizard to add the modules listed in the `user.ml` file.

Step 3. Rebuild your bootfile or ROM image. The ROM image is built by selecting the **Coreboot + Bootfile** option.

Step 4. Load your bootfile or ROM image onto the reference board and reboot the board.

For More Information

If you forgot how to load the ROM image or the bootfile onto your reference board, go to your reference board's board guide for more information on how to do this step.

Step 5. After the board boots, type `mdir` at the shell prompt and check the module listing. You will see the following modules in the listing: `ir1`, `irdalite`, `spirlite`, and either `sp1110ir` or `sp7709ir`.

Once the modules are loaded, the IrDA communication is enabled on the OS-9 target.

Overview of JetBeam for OS-9

JetBeam for OS-9 is an extended implementation of IrDA Lite that provides infrared communication between OS-9-based devices and other devices compliant with the Infrared Data Association (IrDA) standards. The package contains a primary and secondary protocol stack that includes an IrDA protocol API for accessing the stack; the `irda.l` library, which contains all API functions; and uses the stacked protocol file (SPF) manager for file management.

The basic components of the system include the following:

- **IrDA Lite Protocol Module**—The IrDA Lite protocol module, `spirlite,` contains three layers and is responsible for establishing connections, managing multiple connections, and providing information flow control.
- **JetBeam (IrDA) Protocol API**—This API contains the function calls for developing and running IR applications.
- **Framer Device Driver**—this is the hardware driver level of the protocol. The Framer communicates with the rest of the stack through the Framer API.
- **Applications**—There is one application: QBeam. QuickBeam provides object exchange facilities.

JetBeam for OS-9 is a development kit for products using IR communication. The protocol stack, the APIs, and the device drivers provided let you develop applications to manage your IR connections.

JetBeam for OS-9 uses a standard primary/secondary device configuration. Primary and secondary devices and their respective attributes are determined by the application. Primary devices initiate and control IR communications while secondary devices react and respond to commands from the primary device. In all IR communications, one device assumes the primary role and the other device assumes the secondary role.

To use the system, the developer uses the function calls provided in `irda.l` to access the protocol stack and develop applications for IR communication.

IrDA Lite and JetBeam for OS-9

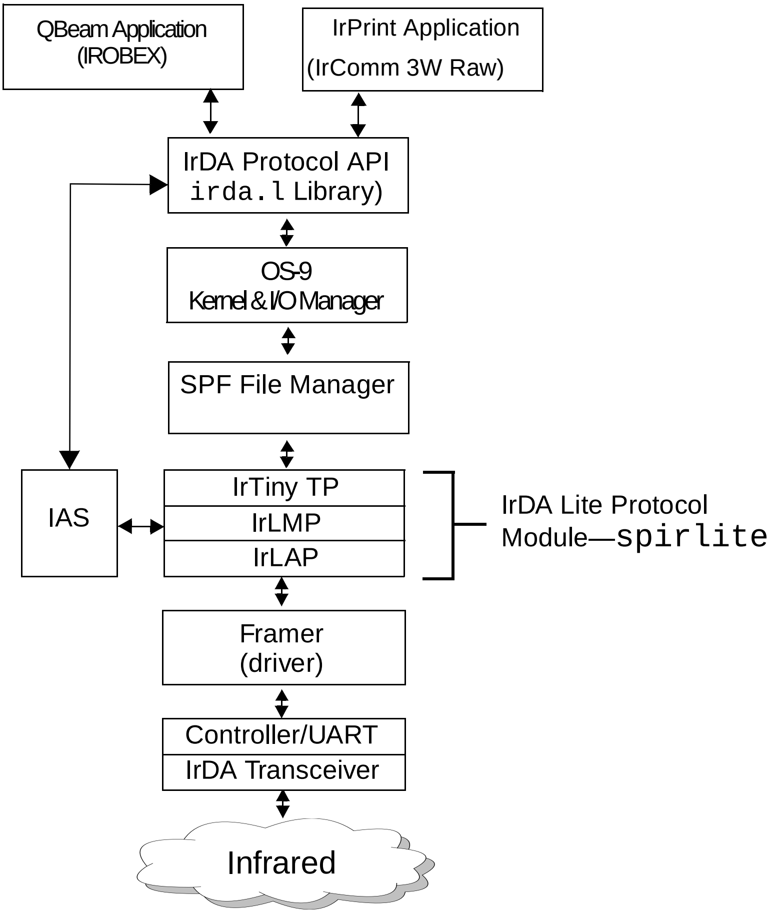
IrDA standards provide written specifications for a variety of infrared communications devices and functions. Some of these functions are mandatory and some are optional.

An IrDA standard, called IrDA Lite, specifies the minimum requirements for an IrDA-compliant device. An IrDA Lite system requires only the minimum, mandatory functions for its devices. In addition, the IrDA Lite specification allows you to implement as few or as many of the optional functions as desired. This lets devices with simple data communication needs implement IrDA protocols with a minimum of complexity in a small amount of RAM and ROM. The JetBeam for OS-9 uses IrDA Lite, with extended negotiation features and a code size of approximately 30 Kbytes.

JetBeam for OS-9 System Design

The components of JetBeam for OS-9 are displayed in the following figure and described in the following sections.

Figure 2-1 JetBeam for OS-9 System



JetBeam for OS-9 Protocol Module

The JetBeam for OS-9 protocol module, called `spirlite`, is based on minimum standards for IR communication described by the Infra-red Data Association. This standard, called IrDA Lite™, employs three IrDA protocol layers: IrLAP, IrLMP, and Tiny TP. These layers manage connections between devices as well as information flow control. The `spirlite` module supports the following extensions to enhance performance:

- IrLAP negotiation supporting data rates from 9600 bps to 115.2 kbps
- IrLAP negotiation of maximum packet size from 64 to 2048 bytes

The `spirlite` module interfaces with applications above it and is a standard SoftStax protocol driver under DPIO with support for all the required protocol driver calls and entry points. The module also interfaces with the hardware below it through the Framer device driver.

Access to the `spirlite` module is gained via the JetBeam Protocol API. Applications can use this API by linking to the API library, `irda.l`. The `spirlite` module also can access the Framer directly using setstat and getstat calls, and send and receive data using the appropriate driver entry points.

Data packets passed between the protocol module and the Framer are stored in mbufs, and standard mbuf calls are used for inter-driver communications. A SoftStax descriptor, `ir1`, provides initialized logical unit values for the IrDA SoftStax protocol driver. It incorporates the standard features of SoftStax descriptors.

The `spirlite` module is a SoftStax protocol driver and must run under the current version of OS-9. It was developed from SPROTO, SoftStax's prototype protocol driver. `spirlite` has the following responsibilities:

- uses the Framer device driver to control the hardware as required for IrDA-compliant communication
- transfers data from the hardware to the applications by packetizing and depacketizing buffers of application data for transfer via IR
- supports JetBeam for OS-9 protocol API calls
- maintains complete hardware independence

For More Information

See ***Using SoftStax*** for more information on SoftStax and the ***SoftStax Porting Guide*** for more information on mbufs.

Intermodule Dependencies

The protocol module is dependent on the following software:

- SoftStax v2.2
- Framer driver modules (`ir` and `sp1110ir` for the SA-11x0 processor; `ir` and `sp7709ir` for the SH-3 processor)
- OS9 Version 2.2 or greater

The protocol module is dependent on the IrDA API and supports library calls that are part of the IRDA library, `irda.l`. The protocol module is also dependent on SoftStax and SysMbuf, and must interface with the framer to transmit and receive data using system mbufs.

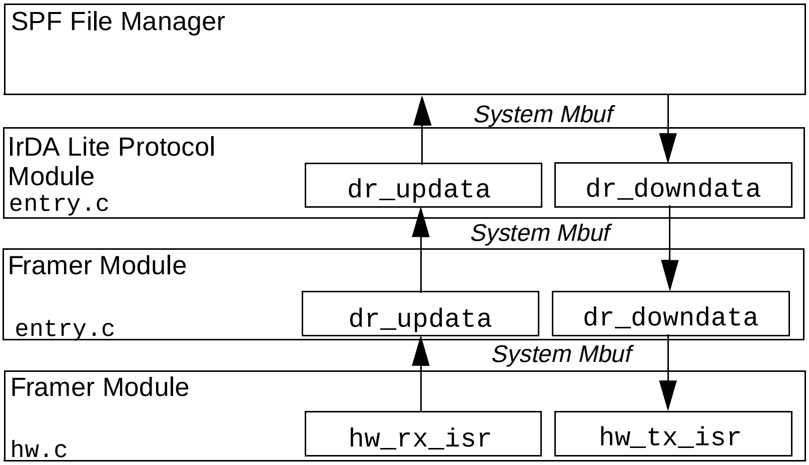
Data Dependencies

Irda.h—IrPacket Header For Mbufs

Every function in the library includes `irda.h`. This header file contains the function prototypes as well as the getstat and setstat codes used in each function.

Each mbuf sent to and received from the framer contains an IrDA header structure called `IrPacket`. The space for the `IrPacket` header is reserved by setting `lustat->lu_txoffset` to the size of an `IrPacket` structure.

Figure 2-2 IR Data Flow



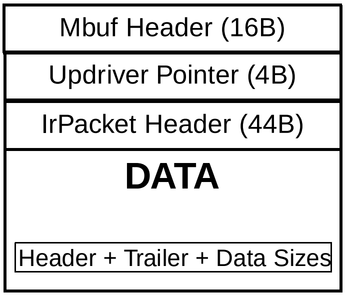
As represented in **Figure 2-2**, all data passed between `spirlite` and the Framer are passed in sysmbufs.

Note

See the ***SoftStax Porting Guide*** for where and how the `lu_txoffset` is used.

Figure 2-8 shows the structure of a SysMBuf. It is followed by definitions of what is contained in the IrPacket header.

Figure 2-3 SysMBuf Structure



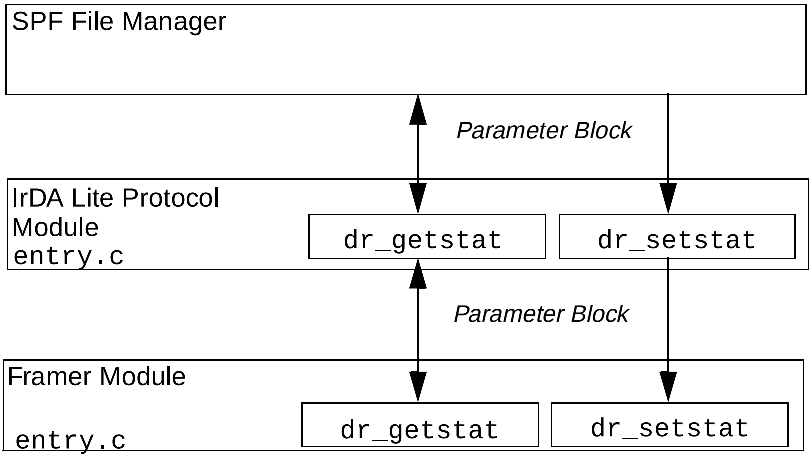
```
/*---------------------------------------------------------------------------
 *
 * Packet Structure for sending IrDA packets.
 */
typedef struct _IrPacket {
    /*========  The Following Are For Internal Use Only =========
     *
     * node
     * origin
     * header
     * headerLen
     *
     *==================================================================*/
    ListEntry  node;
    u_int8*        buff;        /* Pointer to the buffer of data to send */
    Mbuf           mb;          /* Pointer to the mbuf that this structure */
                                /* is contained in */
    void*          origin;      /* Pointer to connection which owns packet */
    u_int16        len;         /* Number of bytes in the buff */
    u_int8         type;        /* Type of the packet */
    u_int8         headerLen;   /* Number of bytes in the header */
    u_int8         more;        /* Flag for TTP SAR more bit */
    u_int8         bufspace[9]; /* space holders for 16 byte boundary */
    u_int8         header[14];  /* Storage for the header */
} IrPacket;
```

Each mbuf passed between the Protocol module and Framer has an `IrPacket` header immediately following the SPF header.

Irda.h—getstats and setstats

The header file `irda.h` contains the getstat and setstat codes defined for the Protocol module to interface with the Framer. All codes are defined in `irda.h`, except `SPF_SS_HWEN` and `SPF_SS_HWDIS`, which are defined in `spf.h`.

Figure 2-4 dr_getstat and dr_setstat Entry Points



In addition, `irda.h` contains all structures common to the Protocol module and the Framer. The following structure is used for all getstat and setstat calls to the Framer from the Protocol module.

```
/***** Structure for parameter block********************/
typedef struct ss_irda_pb {
    spf_ss_pb   spb;
    u_int8      speed;  /* set to bit number for speed */
    u_int8      setmedia;  /* Flag to indicate media busy (1 or 0) */
    u_int8      turnindex;  /* Minimum turn aroundc time index */
    u_int8      numbofs;  /* Number of extra bofs to be sent */
    u_int16     mbufsize;  /* The mbuf size to be supported */
    u_int8      getmedia;   /* The returning media busy flag (1 or 0) */
    u_int8      getstatus;  /* Returning the status of the framer */
    IrFramerCapabilities*  capabilities; /*pointer to the capabilities struct*/
    IrDeviceList*     deviceList;     /* Discovery result structure */
    IrDeviceAddr   condevaddr;     /* Address of the device to connect to */
    u_int8      data_pending;  /* Status of data pending on all paths */
    u_int8      connecttype;   /* IrLMP = 0, Tiny TP = 1 */
    u_int8     rlsap;      /* Remote lsap to connect to */
    u_int8     lsap;   /* Local lsap of the connection opened */
    u_int8*    connectdata;  /* Pointer to connection data */
    u_int16    datalen;  /* Length of the connection data */
    u_int16    lis_time;  /*Time in 1/100ths of second to listen. 0 is infinite*/
    process_id    appid;  /* Process id of application */
    IrIasQuery*   token;  /* Ias query request and results */
    IrIasObject*  iasobject;  /* Obj to add or remove in ias */
    u_int8*   deviceInfo;   /* Device info to be sent on discovery */
    u_int8    deviceInfoLen;  /* Length of the deviceInfo string */
    u_int16   regsig;     /* Signal to be registered in this path */
} ss_irda_pb, *Ss_irda_pb;
```

irda.h—spf.h and mbuf.h

The header file `SPF.h` includes data structures used by the JetBeam for OS-9 Protocol API and also includes the protocol code for IrDA. The header file `mbuf.h` is required for the API calls passing an mbuf pointer or structure.

Module interface

The JetBeam for OS-9 API is the application writer’s access to the `spirlite` protocol stack and the Framer device driver. This API supports all function calls to enable IR communication. It also supports all additional calls necessary for exercising and testing the IrDA compliancy of `spirlite` and the Framer device driver.

As an interface to the IrDA protocol module, the API function calls must take the standard route of providing a set of function calls to the users, which are ultimately presented as setstats or getstats to `spirlite`.

Note

The JetBeam for OS-9 Protocol API is documented in **Chapter 3: The Jetbeam Library Function Reference**.

irda.l

The JetBeam for OS-9 Protocol API is the interface for applications to communicate over an infrared link. Thus the JetBeam for OS-9 Protocol API interfaces with applications and the protocol modules.

Counterpoint Systems Foundry Design Notes

Design Assumptions

- The JetBeam for OS-9 Protocol API contains function calls that allow an application to establish and maintain communication between two IR devices. Also it exercises the required setstat and getstat calls that the Protocol module needs to perform these tasks.
- One or more paths may be opened to the driver. The logical unit static storage is allocated on a per path basis, per path storage.
- The driver is designed under DPIO (dual ported I/O). Therefore, OS-9 calls may be employed by using the conv_lib.

Note

Drivers designed under DPIO will work with OS-9 for 68K as well as the other ports of OS-9 (Enhanced OS-9 for SuperH for example).

The driver implements an extended IrDA Lite with the following capabilities:

- negotiated data rates from 9600 bps to 115.2 kbps
- maximum window size of 1
- maximum turn-around time of 500 msec
- data sizes from 64 to 2048 bytes

Intermodule dependencies

The JetBeam for OS-9 Protocol API is dependent on the following modules:

- Staked Protocol File Manager (SPF)
- Protocol module (`spirlite`)

Framer device driver

The JetBeam for OS-9 Framer device driver is a standard SoftStax (SPF) device driver under DPIO with support for all the required calls and entry points. This driver provides a hardware interface to `spirlite` through the Framer interface. A descriptor accompanies the Framer device driver and follows standard SPF descriptor standards. The Framer device driver controls the hardware to send and receive data.

The Framer device driver serves as the primary interface to the hardware's UART port that supports IR. The driver is responsible for all hardware configuration and communication. The Framer device driver must also support getstat and setstat calls from `spirlite`.

Functional Requirements

The driver is responsible for all hardware configuration and communication and has the following requirements:

- initialize the needed hardware for protocol communications
- transfer data to and from the IO port through interrupt-driven procedures
- reset the hardware at termination

The Framer device driver must be implemented as a SoftStax (SPF) device driver and must run under the current version of OS-9. The standard SPF entry points must be implemented.

For More Information

See ***Using SoftStax***.

IrDA specific getstats and setstats must also be supported to communicate with `spirlite,` and the driver must use standard MBUFs for data transfer.

The Framer device driver must support the physical link layer specifications as outlined in the IrDA Serial Infrared Physical Layer Link Specification, and provided all the services required by Link Access Protocol Specification with the limitations imposed by an extended IrDA Lite. Services such as CRC generation and checking, framing, character stuffing, and the baud rates outlined in the specification are supported.

Using JetBeam for OS-9 at the Application Level

JetBeam for OS-9 is a development kit for products using IR communication. The protocol stack, the APIs, and the device drivers provided enable you to develop applications to manage your IR connections.

JetBeam for OS-9 is both a primary and secondary application development package. A secondary device is created by adding a service and listening for a command. A primary device is created by the discovering and connecting processes.

The system uses a primary/secondary device configuration. Primary and secondary devices and their respective attributes are determined by the application.

- Primary devices control IR communications, initiate connections, issue commands, and respond to commands.
- Secondary devices provide services and react and respond to commands from the primary device. The data flow from the secondary device is controlled by the primary device.

To develop an application for IR communication, use the function calls provided in `irda.l`.

The following sections provide examples of using JetBeam for OS-9 in primary and secondary modes and illustrate which function calls are used and how to use them.

For More Information

Source code examples for a primary (IRPRINT) and a secondary (GENOA) IR device using JetBeam for OS-9 are in the following location:

```
\mwos\SRC\SPF\UTILS\IRDA\
```

IRPRINT (Primary Device)

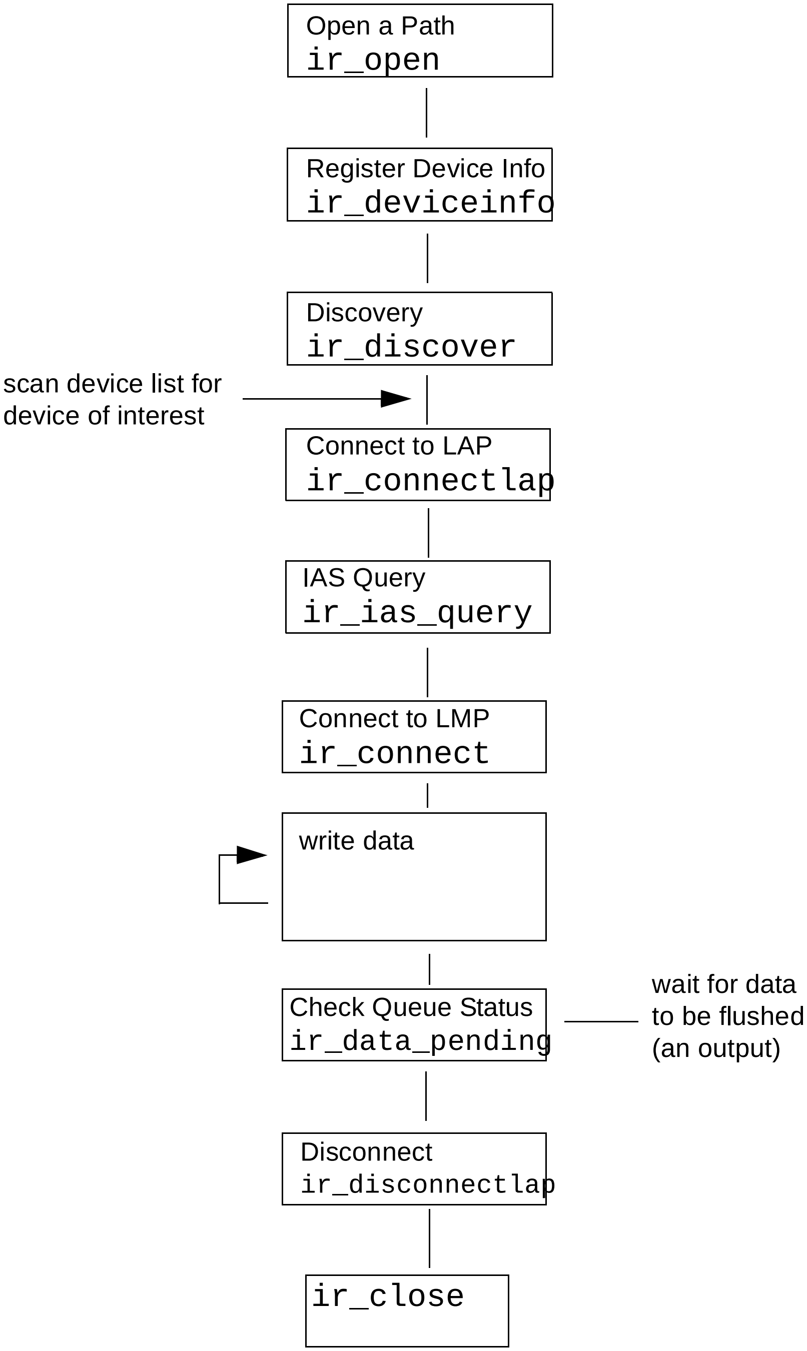
This particular application prints a data file from the primary device (OS-9 reference platform) to a secondary device (laser printer) using infrared communication. In this case, the secondary device was set up first. The secondary device was written to provide a “service”—printing. After the device and the service were defined, the primary device was designed around how printing was to be provided.

Note

The function calls used by the sample applications below are documented in **Chapter 3: The Jetbeam Library Function Reference**.

IRPRINT is an application that sends data—over IR media—from a primary device to a generic secondary device. Following is a description of IRPRINT, including a communication flow diagram and a description of the function calls used.

Figure 2-5 Primary Device Communication Flow Diagram



Primary Device Function Calls

`ir_open`

This command opens a path for infrared communication.

`os_gs_popt`

After `ir_open`, the application uses the OS-9 command `os_gs_popt`, which changes the path option of the specific path. It also makes the `ir_write` command (described below) non-blocking (see `ir_write`, below).

`ir_deviceinfo`

Sets the device information. Information is sent to the remote during the discovery process.

`ir_discover`

This command attempts to find a remote IR device.

`ir_connectlap`

This command establishes a LAP connection between two IR devices.

`ir_ias_query`

This command queries the remote device about a particular object. The application should check the return code (IrIasQuery.retCode) to make sure the object is found. If an integer is returned, the remote LSAP is determined. The remote LSAP is defined in irda.h

`ir_connect`

This command connects the primary device to the secondary device through the remote LSAP connection.

`ir_write`

This command sends data from the buffer to the remote device. Data is sent in one of two modes, blocking or non-blocking.

In blocking mode, the device waits until the transmission is successfully completed and then returns.

In non-blocking mode, an application initiates a transmission and returns without waiting for the result.

`ir_disconnectlap`

This command disconnects LAP or returns an error saying it wasn’t cancelled.

`ir_close`

This command closes the path between the primary and secondary device.

Signals

The primary device also registers signals using `ir_status_regsig`. The following table lists and describes the signals used

.

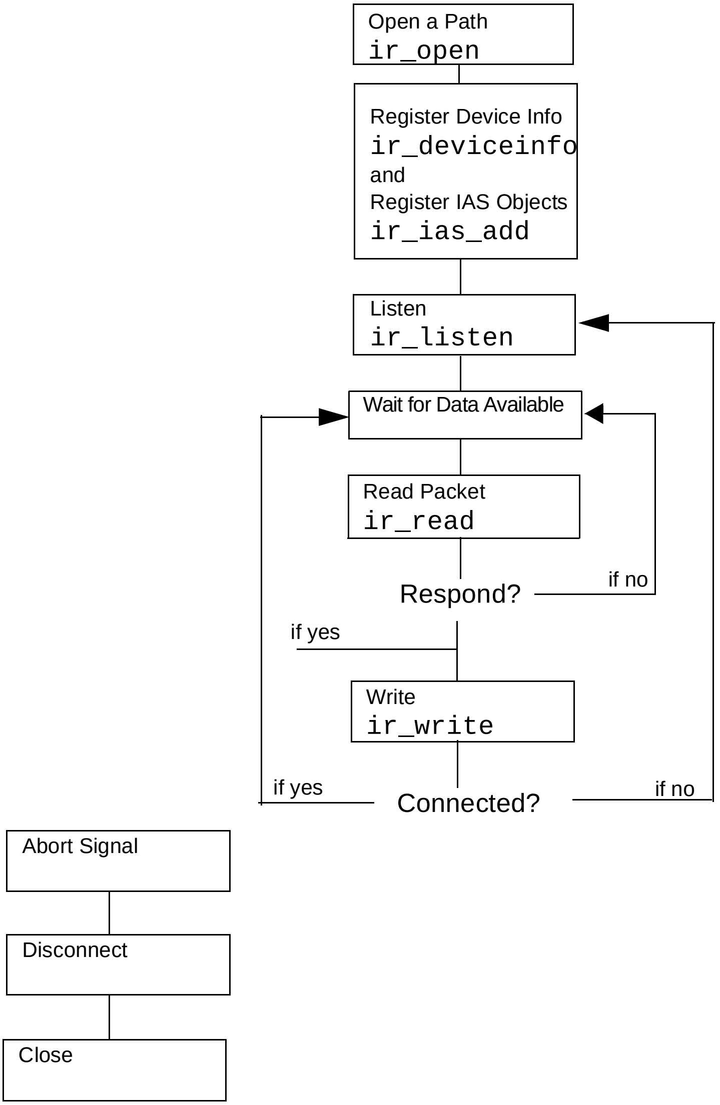
Table 2-1 Signals used by ir_status_regsig

Signal	Description
`IR_SIG_DATA_AVAIL`	triggered when the read queue has data for the specific path
`IR_SIG_NO_PROGRESS`	triggered when there is no response from the remote for a defined period of time
`IR_SIG_LINK_OKAY`	triggered when the remote responds to the primary within a defined period of time
`IR_SIG_LAP_DOWN`	triggered when LAP is disconnected
`IR_SIG_LAP_CONNECT`	triggered when LAP is connected

GENOA (Secondary Device - SH-3 only)

GENOA is a generic secondary device that responds to general primary device commands over IR media. Following is a description of GENOA, including a general communication flow diagram and a description of the function calls used.

Figure 2-6 Secondary Device Communication Flow Diagram



Secondary Device Function Calls

`ir_open`

This command opens a path for infrared communication at the LMP layer. A “slot number” or LSAP number is automatically assigned to the path that is opened.

`ir_ias_add`

This command adds an object to the device. An object is a description of the device’s services. You must attach an LSAP number to the object (the LSAP number is not automatically assigned). Use the LSAP number returned by the `ir_open` call. This associates a path to an object so that information flows through the same path.

`ir_deviceinfo`

This command attaches information about the device. `deviceinfo` is an information string sent through with the discovery packet.

`ir_listen`

This command enables the secondary device to listen for commands from the primary device on a particular path.

Wait for Data Available

At this stage, the secondary device is waiting for a signal from the primary device that will determine its future action(s).

Read Packet (data available)

At this stage, there is data in the read queue.

`ir_read`

This command reads data from the read queue.

Check to see if data needs to be returned

At this stage, the secondary device checks the data just read to determine whether the primary expects a response.

`ir_write`

This command writes data.

Abort

Three commands terminate the connection. They include `SIGINT`, `SIGQUIT`, and `SIGHUP`.

`ir_disconnectlap`

This command closes LAP.

`ir_close`

This command closes the path.

`LAP_DOWN`

At this stage, the primary device disconnects LAP and listens again for a new connection.

Other calls used by the secondary device include the following:

`ir_status_regsig`

This command registers a signal with the protocol stack. The protocol stack sends a signal back to the application when the corresponding event occurs. If the event is already occurring at the time of signal registration, the signal is sent back immediately. Signals are registered per path.

`ir_status_unregsig`

This command cancels the request to be notified of an event.

`ir_close`

This command closes the path between the application and device.

`os_gs_ready`

This is an OS-9 command. It returns the number of bytes available to read with `ir_read`.

JetBeam for OS-9 Applications

IrPrint application

The IrPrint application is a primary role application that prints a file over IR. If a file exists on a random-block storage device, IrPrint opens the file, reads its contents, and transmits the data over IR to a printer device. IrPrint is passed command line parameters of file names.

Intermodule dependencies

The IrPrint application is dependent on the following modules:

- SPF File Manager
- Protocol Module
- Framer

Data dependencies

The IrPrint application is dependent on the following:

- `irda.h`
- `irda.l`

IrPrint, like other JetBeam for OS-9 applications, is dependent on `irda.h` and `irda.l`. The application must include `irda.h` for definitions, structures, and prototypes. The application must also link to `irda.l` to be able to use any of the IR Communication Pak API functions.

Module interface

The IrPrint application takes file names on the command line as its parameters. The IrPrint application is used as follows:

```
irprint [-?] [<filename> <filename> ...]
   -? - Display usage
   <filename> - File to print over IR
```

Design assumption

IrPrint assumes that all modules it depends on are loaded into memory.

Design Sub-entities

IrPrint comprises four primary functions. They are described in the following sections and include:

- `error_code printer_connect(void);`
- `error_code printer_disconnect(void);`
- `error_code print_file(char *printfile);`
- `error_code wait_for_data(void);`

When initiated, IrPrint's logical flow is to connect to a printer, send the file data, wait for the data to be delivered, and disconnect from the printer. If more than one file name is passed on the command line, IrPrint will loop on the number of files and send the data of each file.

printer_connect()

`printer_connect connects to the printer service of the IR printer device.`

This function:

1. opens an IR communications path (`ir_open`)
2. gets the path options (`_os_gs_popt`)
3. sets the asynchronous mode (`_os_ss_popt`)
4. discovers an IR device (`ir_discover`)
5. establishes a LAP connection (`ir_connectlap`)
6. sends a query for printer services (`ir_ias_query`)
7. connect to the printer service (`ir_connect`)

printer_disconnect()

`printer_disconnect` disconnects from the printer service that `print_connect` established.

This function:

1. disconnects the LAP connection (`ir_disconnectlap`)
2. closes the IR communications path (`ir_close`)

print_file()

`print_file` opens a print file and transfers its contents to the connected IR printer device.

This function:

1. opens a print file (`fopen`)
2. reads its contents (`fread`)
3. transfers the data (`ir_write`)
4. closes the print file (`fclose`)

Steps 2 and 3 are repeated until the entire contents of the print file is read and transferred.

wait_for_data()

wait_for_data() waits for all data to be transferred to the IR print devices. This must be done before the connection is closed. If the connection is closed before the data is transferred, an incomplete file will be received at the printer device. (Asynchronous mode only.)

This function:

1. sleeps for a while (_os_sleep)
2. checks if any data is pending (ir_data_pending)

Steps 1 and 2 are repeated until no data is pending on the IR communications path.

Genoa application

The Genoa application is a secondary role application that provides printer services to IrPrint and allows for protocol stack testing with the Genoa Test Suite. The Genoa Test Suite is a set of tests performed over IR to test IrDA compliance. This test suite is provided by Genoa Technology, Inc. and its current version is 1.4.

Intermodule dependencies

The Genoa application is dependent on the following modules:

- SPF File Manager
- Protocol Module
- Framer

Data dependencies

The Genoa application is dependent upon the following:

- `irda.h`
- `irda.l`

Genoa, like other JetBeam for OS-9 applications, is dependent on `irda.h` and `irda.l`. The application must include `irda.h` for definitions, structures, and prototypes. The application must also link to `irda.l` to be able to use any of the IR Communication Pak API functions.

Module interface

The Genoa application takes no command line parameters.

Design Sub-entities

Genoa comprises three primary functions. They are described in the following sections and include the following:

- `error_code test_connect(void);`
- `error_code test_disconnect(void);`
- `error_code genoa_test(void);`

When initiated, Genoa's logical flow is to connect to the protocol stack, provide a printer service, wait for an IR connection, read and send data, and disconnect from the protocol stack.

test_connect()

`test_connect` connects to the protocol stack and provides a printer service for connecting IR devices.

This function:

1. opens an IR communications path (`ir_open`)
2. sets the device information (`ir_deviceinfo`)
3. adds the printer service (`ir_ias_add`)

test_disconnect()

`test_disconnect` disconnects from the protocol stack.

This function:

1. disconnects the LAP connection (`ir_disconnectlap`)
2. closes the IR communications path (`ir_close`)

genoa_test()

`genoa_test` listens for IR device connections and sends and receives data as needed.

This function:

1. registers the LAP connect signal (`ir_status_regsig`)
2. registers the data available signal (`ir_status_regsig`)
3. listens for an IR device connection (`ir_listen`)
4. registers the LAP disconnect signal (`ir_status_regsig`)
5. if data available signal is received, checks for data (`_os_gs_ready`)
6. if data is present, reads all data (`ir_read`)
7. if an iframe request is received, sends iframe(s) (`ir_write`)
8. registers the data available signal (`ir_status_regsig`)
9. registers the LAP connect signal (`ir_status_regsig`)

While a LAP connection exists, steps 5-8 are repeated to receive and send all IR data. Step 3-9 are repeated indefinitely. The Genoa application is aborted by sending the `SIGINT` or `SIGQUIT` signal to the application.

IrTest application

IrTest is a testing application that manually invokes the Protocol API functions. IrTest is dependent on SPF and Protocol and Framer modules to communicate over infrared. It assumes that all necessary modules are loaded into memory.

IrTest is a menu-driven program that calls each of the IrDA Protocol API functions when a path is opened. The purpose of this application is twofold:

- provide an example of how to use each API function
- provide individual API function testing

The user interface of IrTest displays each API function that is supported when the protocol or test stack is opened. If any parameters are needed to perform the API function, you are prompted to enter the additional data.

Test Stack API Functions

The following menu of supported API functions is displayed when the test stack is opened:

1. Open a path

2. Read data from the opened path

w. Write data to the opened path

4. Set baud rate

5. Set num extra bofs

6. Set turn-around time

7. Set mbuf size

8. Set media status

9. Get media status

0. Get framer status

a. Get capabilities

q. Quit

Menu option 1: `open_path()`

This function opens an infrared communications path. This function:

- asks you to select a type of connection to open (LMP or TinyTP).
- opens the infrared path (`ir_open`)

Menu option 2: `read_data()`

This function reads data from the opened path. This function:

- checks if any data is available to be read from the opened path (`_os_gs_ready`).
- asks you to input a number of bytes of data to be read.
- reads the specified number of bytes (`ir_read`).

Menu option W: `write_data()`

This function writes data to the opened path. This function:

- asks you to input a number of bytes to be written.
- writes the specified number of bytes to the opened path (`ir_write`).

Menu option 4: `set_baud()`

This function sets the baud rate of the framer. This function:

- asks you to select the appropriate baud rate (9600, 19200, 38400, 57600, or 115200).
- sets the framer to the specified baud rate (`irfrmr_ss_baud`).

Menu option 5: `set_bofs()`

This function sets the number of additional bofs to be sent during packet transmission. This function:

- asks you to input the number of additional bofs to be sent.
- sets the number of bofs to the specified number (`irfrmr_ss_bofs`).

Menu option 6: `set_turntime()`

This function sets the link turn-around time delay index. This function:

- asks you to select the link turn around time delay index.
- sets the turn around time delay index to the specified value (`irfrmr_ss_turnindex`).

Menu option 7: `set_mbufsize()`

This function sets the size of the mbuf that SPF should allocate when data packets are transmitted. This function:

- asks you to select the size of the mbuf (64, 128, 256, 512, 1024, or 2048 bytes).
- sets the size of the mbuf to the specified value (`irfrmr_ss_mbufsize`).

Menu option 8: `set_media()`

This function sets the media busy state for the Framer. This function:

- asks you to select the appropriate media busy state (busy or not busy).
- sets the media busy state to the specified value (`irfrmr_ss_media_busy`).

Menu option 9: get_media()

This function gets the media busy status from the Framer (`irfrmr_gs_media_busy`).

Menu option 0: `get_frmstat()`

This function gets the Framer’s receive status byte (`irfrmr_gs_frm_status`).

Menu option a: `get_cap()`

This function gets the Framer’s capabilities data structure (`irfrmr_gs_frm_caps`).

Protocol Stack API Functions

The following menu of supported API functions is displayed when the protocol stack is opened:

1. Open a path

2. Read data from the opened path

w. Write data to the opened path

c. Connect to IrLap

d. Do a discovery

e. Disconnect IrLap

f. Connect to device

g. Register a signal

h. Unregister a signal

i. Ias lpt query

l. Listen

n. Set device info

r. Ias lpt register

s. Signal wait

u. Ias lpt unregister

q. Quit

Menu option 1: `open_path()`

This function opens an infrared communications path. This function:

- asks you to select a type of connection to open (LMP or TinyTP).
- opens the infrared path (`ir_open`)

Menu option 2: `read_data()`

This function reads data from the opened path. This function:

- checks if any data is available to be read from the opened path (`_os_gs_ready`).
- asks you to input a number of bytes of data to be read.
- reads the specified number of bytes (`ir_read`).

Menu option W: `write_data()`

This function writes data to the opened path. This function:

- asks you to input a number of bytes to be written.
- writes the specified number of bytes to the opened path (`ir_write`).

Menu option c: `irlap_connect()`

This function starts an IrLAP connection on the opened path (`ir_connectlap`).

Menu option d: `discovery()`

This function performs a discovery process on the opened path (`ir_discover`).

Menu option e: `irlap_disconnect()`

This function disconnects an IrLAP connection from the opened path (`ir_disconnectlap`).

Menu option f: `connect_device()`

This function starts an IrLMP or TTP connection. This function:

- asks you to input a remote lsap of the device to which you wish to connect.
- requests an IrLMP or TinyTP connection (`ir_connection`).

Menu option g and h: `reg_signal()`

This function registers or unregisters a signal to be sent to the application by the infrared communications driver. Whether the signal is to be registered or unregistered depends on the menu option selected in the protocol testing menu. This function:

- asks you to select the signal to be (un)registered (`IR_SIG_DATA_AVAIL`, `IR_SIG_NO_PROGRESS`, `IR_SIG_LINK_OKAY`, `IR_LAP_DOWN`, or `IR_SIG_LAP_CONNECT`).
- (un)register the signal (`ir_status_regsig`, or `ir_status_unregsig`).

Menu option i: `ias_query()`

This function starts an LPT query process (`ir_ias_query`).

Menu option l: `listen()`

This function listens for a LMP or TTP connection as another device tries to connect to the device listening (`ir_listen`).

Menu option n: set_info()

This function sets the device information (ir_deviceinfo).

Menu option r: ias_register()

This function adds an LPT object to the IAS database (ir_ias_add).

Menu option s: signal_wait()

This function waits for the signal(s) that the user registered through the "Register a signal" menu option. This function:

- asks you to input how many signals the program should wait for.
- waits for the specified number of signals.

Menu option u: ias_unregister()

This function removes the LPT object from the IAS database (ir_ias_remove).

IrDA Library

The JetBeam for OS-9 library contains the function calls used by applications to interface to the JetBeam for OS-9 protocol module—`spirlite`—and the Framer device driver.

Note

These functions are discussed in detail in **Chapter 3: The Jetbeam Library Function Reference**.

The JetBeam for OS-9 library contains the code to implement all the functions supported in each of the APIs—JetBeam for OS-9 protocol API and Framer interface, and for the applications provided. The header file `irda.h` is also required. This header file contains the prototypes for all of the functions, all irda data structures, and all the irda getstat and setstat codes.

The Library follows the standard programming practices for assembling the library function calls sources in: `mwos\SRC\DPIO\SPF\LIB\IRDA.` The library object code is in `mwos\OS9000\ARMV4\LIB\irda.l` (for the SA-1110) or `mwos\OS9000\SH3\LIB\irda.l` (for the SH-3).

Chapter 3: The Jetbeam Library Function Reference

This chapter contains all function calls for the JetBeam for OS-9 library in alphabetical order. When using these functions in source code, be sure to include the `irda.h` header file.

Note

Some descriptions mention error checking. This is in reference to errors that are flagged only when error checking is enabled.

ir_close()

Close a Path to an IR Device

Syntax

```
error_code ir_close(path_id path)
```

Description

`ir_close()` closes a path to a device.

Parameters

`path`	The path number to close.

Non-Fatal Errors

`SUCCESS`	The operation is successful.

Libraries

`irda.l`

ir_connect()

Request an IrLMP or TinyTP Connection

Syntax

```
error_code ir_connect(path_id path, u_int8 rlsap,
                      u_int8* data, u_int16* len)
```

Description

`ir_connect()` requests an IrLMP or TinyTP connection to an LSAP.

Parameters

`path`	The path to perform the LMP or TTP connection.
`rlsap`	The number of the remote lsap.
`data*`	A pointer to the data string sent with the connect packet. If there is connection data from the other side, the memory that this pointer points to will contain that data when the function returns.
`len*`	A pointer to the length of the data string being sent. If there is connection data from the other side, the memory that is pointed to will contain the length of the data when the function returns.

Non-Fatal Errors

`SUCCESS`	The operation is successful.
`IR_REQ_FAILED`	Connection is busy because it is already connected (error check only):
`IR_NO_LAP`	Operation failed; no IrLAP connection.
`IR_LAP_DISCON`	LAP connection disconnected during transaction.

Libraries

`irda.l`

ir_connectlap()

Initiate an IrLAP Connection

Syntax

```
error_code ir_connectlap(path_id path,
                         IrDeviceAddr addr);
```

Description

`ir_connectlap()` starts an IrLAP connection to the secondary device.

Parameters

`path`	The path to perform the IrLAP connection.
`addr`	The 32-bit address of the device to which you are connecting. Returned by ir_discover().

Non-Fatal Errors

`SUCCESS` The operation is successful.

`IR_MEDIA_BUSY` The operation failed to start because the IR media is busy. Media busy is caused by one of the following reasons:

- Other devices are using the IR medium.
- An IrLAP connection already exists.
- A discovery process is in progress.

Libraries

`irda.l`

ir_data_pending()

Obtain Current Write Queue Status

Syntax

```
error_code ir_data_pending(path_id path,
                           u_int8* data_pending)
```

Description

`ir_data_pending()` determines whether data is queued.

Parameters

`path`	The path used to perform the data pending query.
`data_pending*`	Pointer to the data pending status field to be returned.

Non-Fatal Errors

`SUCCESS`	The operation is successful.

data pending status field:

`IR_STATUS_NO_DATA`	No data is queued on any path’s write queue for the IR driver.
`IR_STATUS_MULTI_DATA`	Data queued on the requested path’s and other path’s write queues in the IR driver.
`IR_STATUS_PP_DATA`	Data queued only on the requested path’s write queue in the IR driver.
`IR_STATUS_OTHER_DATA`	Data queued only on other path’s write queues in the IR driver.

Libraries

`irda.l`

ir_deviceinfo()

Set Device Information

Syntax

```
error_code ir_deviceinfo(path_id path,
                         u_int8* deviceInfo,
                         u_int8 deviceInfoLen)
```

Description

`ir_deviceinfo()` set the device information for the secondary device. The device information is stored in the application. The IR stack driver saves the pointer to that information. This pointer is used to access the device information and respond with it over infrared during a discovery.

Parameters

`path`	The path to perform the IrLAP connection.
`deviceInfo*`	A pointer to the device information.
`deviceInfoLen`	The length of the device information.

Non-Fatal Errors

`SUCCESS`	The operation is successful.

Libraries

`irda.l`

ir_disconnectlap()

Disconnect an IrLAP Connection

Syntax

```
error_code ir_disconnectlap(path_id path)
```

Description

`ir_disconnectlap()` disconnects an IrLAP connection.

Parameters

`path`	The path to perform the IrLAP disconnection.

Non-Fatal Errors

`SUCCESS`	The operation is successful.
`IR_NO_LAP`	The operation failed because no IrLAP connection exists.

Libraries

`irda.l`

ir_discover()

Initiate the Ir Discovery Process

Syntax

```
error_code ir_discover(path_id path,
                       IrDeviceList* deviceListPT)
```

Description

ir_discover() starts a discovery process. The calling process is put to sleep until the discovery is completed. A pointer to a IrDeviceList must be passed to this library. The calling program must have space allocated for the structure because the driver will memcopy the values into memory starting at the IrDeviceList pointer.

Parameters

path	The path to perform the discovery.
deviceListPT*	Pointer to the IrDeviceList structure allocated in the application.

Non-Fatal Errors

SUCCESS

The operation completed successfully. The discovered items are contained in the IrDeviceList pointer. Defined in irda.h.

IR_MEDIA_BUSY

Operation failed; media is busy. Media busy is caused by one of the following reasons:

- Other devices are using the IR medium.
- A discovery process is already in progress.
- An IrLAP connection exists.
- This is the third discovery in a row.

Libraries

`irda.l`

ir_ias_add()

Add an Object to the IAS Database

Syntax

```
error_code ir_ias_add(path_id path,
                      IrIasObject* iasobject)
```

Description

ir_ias_add() adds an IAS object to the IAS database. The object is not copied so memory for the object must exist for as long as the object is in the data base. The IAS database only allows objects with unique class names. The error checking version checks for this. Class names and attributes names must not exceed IR_MAX_IAS_NAME. Also, attribute values must not exceed IR_MAX_IAS_ATTR_SIZE. This function is only available if IR_IAS_ADD_REMOVE is defined.

The lsap value in the IrIasObject must be set to the value returned by the ir_open call for this object to be added on the correct lsap.

Parameters

path	The path to perform the IAS add.
iasobject*	A pointer to the IrIasObject structure allocated in the application.

Non-Fatal Errors

SUCCESS	The operation is successful.
IR_REQ_FAILED	Operation failed for one of the following reasons: • No space in the data base (see irconfig.h to increase the size of the IAS database). • Class name already exists (error check only).

- Object attributes violate IrDA Lite rules (attribute name exceeds `MAX_IAS_NAME` or attribute value exceeds `IR_MAX_IAS_ATTR_SIZE`, error check only).
- The class name exceeds `IR_MAX_IAS_NAME` (error check only).

Libraries

`irda.l`

ir_ias_query()

Query the IAS Database

Syntax

```
error_code ir_ias_query(path_id path,
                        IrIasQuery* token)
```

Description

ir_ias_query() starts an IAS query process by the primary device. A pointer to an IrIasQuery structure must be supplied. The IrIasQuery structure must be allocated in the application. Each pointer in the structure must also have allocated space. The desired query is supplied by the application in the structure fields. See irda.h for the structure elements. The calling process is put to sleep until the query is completed.

Parameters

path	The path id to perform the query.
token*	A pointer to the IrIasQuery structure allocated in the application.

Non-Fatal Errors

SUCCESS — The operation completed successfully. The query results are contained in the IrIasQuery structure. This structure is defined in irda.h.

IR_REQ_FAILED — The operation failed for one of the following reasons (Error check only):

- The query exceeds IR_MAX_QUERY_LEN.
- The resultBuffSize field of the IrIasQuery is 0.
- A query is already in progress.

`IR_NO_LAP`	The operation failed because there is no IrLAP connection.
`IR_LAP_DISCON`	The IrLAP connection was disconnected during the transaction.

Libraries

`irda.l`

ir_ias_remove()

Remove an Object From the IAS Database

Syntax

```
error_code ir_ias_remove(path_id path,
                         IrIasObject* iasobject)
```

Description

`ir_ias_remove()` removes an IAS object from the IAS database. This function is only available if `IR_IAS_ADD_REMOVE` is defined.

Parameters

`path`	The path id to perform the IAS remove.
`iasobject*`	A pointer to the `IrIasObject` structure allocated in the application.

Non-Fatal Errors

`SUCCESS`	The operation is successful.
`IR_REQ_FAILED`	The object was not in the database.

Libraries

`irda.l`

ir_listen()

Listen for LMP or TinyTP Connection

Syntax

```
error_code ir_listen(path_id path, u_int8* data,
u_int16* len, u_int16 lis_time);
```

Description

During `ir_listen()` the secondary listens for an LMP or Tiny TP connection. The primary tries to connect to the device that is listening.

Parameters

`path`
The path id on which to listen.

`data*`
A pointer to the data string to send with the connect packet. If there is connection data from the other side, the memory that this pointer points to will contain that data when the function returns.

`len*`
A pointer to the length of the data string to be sent. If there is connection data from the other side, the memory that this pointer points to will contain the length of the data when the function returns.

`lis_time`
The time in 1/100ths of a second that the driver is to listen for an LMP or Tiny TP connection. If no connection is started within this time, then the application wakes up. If 0 is used, the application will sleep until an LMP or Tiny TP connection is started.

Non-Fatal Errors

SUCCESS	An LMP or Tiny TP connection is established.
IR_REQ_FAILED	The operation failed because no LMP or Tiny TP connection was requested during the listen time. The operation will also fail if the other side requests a disconnect while a connect response is being sent
IR_LAP_DISCON	The IrLAP connection was disconnected during the transaction.

Libraries

```
irda.l
```

ir_open()

Open a Path to an IR Device

Syntax

```
error_code ir_open(path_id* path, char* driver,
                   u_int8 contype, u_int8* LSAP)
```

Description

`ir_open()` opens a path to an IR device, returns the LSAP number associated to the path within the stack.

Parameters

`path*`	A pointer to the path id for the path to open.
`driver*`	A pointer to the string that contains the name of the descriptor to use for the path.
`contype`	A 0 or 1 indicating an LMP or Tiny TP connection, respectively.
`LSAP*`	A pointer to the LSAP variable where `ir_open` stores the LSAP number associated with the path.

Non-Fatal Errors

`EOS_DEVBSY`	There is no capabilities getstat in the framer.
`IR_REQ_FAILED`	The stack has been opened with a connection type different than this current open request. Wait until the stack has been closed before trying to connect with this new type.
`SUCCESS`	The operation is successful.

Libraries

`irda.l`

ir_read()

Read Data

Syntax

```
error_code ir_read(path_id path, u_char* buf,
                   u_int32* readamount)
```

Description

ir_read() reads data from an infrared communications path.

Parameters

path	The path id from which data is read.
buf*	A pointer to the application memory that can store the requested number of bytes to read.
readamount*	A pointer to the number of bytes to read. This value is set to the actual amount read when the function returns.

Non-Fatal Errors

SUCCESS	The operation is successful.

Any error codes that an _os_read function may return.

Libraries

irda.l

ir_status_regsig()

Register a Signal

Syntax

```
error_code ir_status_regsig(path_id path,
                            u_int16 signalcode)
```

Description

`ir_status_regsig()` registers a signal to be sent to the application by the infrared communications driver.

Signals

Signals must be registered with the stack for the stack to send them to the application. Each signal will only be triggered once for each registration. The following is a list of stack initiated signals and when and why they are sent.

If an application has multiple paths open to the stack and one of the paths triggers a signal (and the signal is registered on that path), the signal will be sent to the application. The application will not know which path triggered the signal. This condition is really only relevant for the data available. The application can then just check each receive queue to determine which path sent the signal.

`IR_SIG_DATA_AVAIL`

Every time that application data is received this signal will be triggered to be sent. The signal will only be received by the application for which the data is being sent to.

`IR_SIG_NO_PROGRESS`

This signal is sent when the IrLAP link has not had any progress for the threshold time which is current set at 3 seconds. All connections using IrLAP should receive this signal. Since a registered signal is only sent once only the first connection using IrLAP will be signaled.

`IR_SIG_LINK_OKAY`
If an IrLAP connection has been in a state of no progress and is now functioning again this signal should be sent to all connections. Since a registered signal is only sent once only the first connection using IrLAP will be signaled.

`IR_SIG_LAP_DOWN`
This signal indicates that the IrLAP connection has gone down. This could be because the other side disconnected, the connection time-out was surpassed w/no progress (currently 40 seconds), or this side disconnected the link. This signal should be sent to all connections. Since a registered signal is only sent once only the first connection using IrLAP will be signaled.

`IR_SIG_LAP_CONNECT`
This signal indicates that an IrLAP connection has come up. The requesting connection will be signaled.

Parameters

`path`
The path id on which to register the signal.

`signalcode`
The signal code to register. Acceptable values are contained in `irda.h`. Some values follow. `IR_SIG_DATA_AVAIL`, `IR_SIG_NO_PROGRESS`, `IR_SIG_LINK_OKAY`, `IR_SIG_LAP_DOWN`, `IR_SIG_LAP_CONNECT`.

Non-Fatal Errors

`SUCCESS`	The operation is successful.
`IR_BAD_INPUT`	The signal code value is out of range.

Libraries

`irda.l`

ir_status_unregsig()

Unregister a Signal

Syntax

```
error_code ir_status_unregsig(path_id path,
                              u_int16 signalcode)
```

Description

`ir_status_unregsig()` unregisters a signal in the infrared communications driver. This signal will no longer be sent to the application.

Signals

Signals must be registered with the stack for the stack to send them to the application. Each signal will only be triggered once for each registration. The following is a list of stack initiated signals and when and why they are sent.

If an application has multiple paths open to the stack and one of the paths triggers a signal (and the signal is registered on that path), the signal will be sent to the application. The application will not know which path triggered the signal. This condition is really only relevant for the data available. The application can then just check each receive queue to determine which path sent the signal.

`IR_SIG_DATA_AVAIL` Every time that application data is received this signal will be triggered to be sent. The signal will only be received by the application for which the data is being sent to.

`IR_SIG_NO_PROGRESS` This signal is sent when the IrLAP link has not had any progress for the threshold time which is current set at 3 seconds. All connections using IrLAP should receive this signal. Since a registered signal is only sent once only the first connection using IrLAP will be signaled.

`IR_SIG_LINK_OKAY`

If an IrLAP connection has been in a state of no progress and is now functioning again this signal should be sent to all connections. Since a registered signal is only sent once only the first connection using IrLAP will be signaled.

`IR_SIG_LAP_DOWN`

This signal indicates that the IrLAP connection has gone down. This could be because the other side disconnected, the connection timeout was surpassed w/no progress (currently 40 seconds), or this side disconnected the link. This signal should be sent to all connections. Since a registered signal is only sent once only the first connection using IrLAP will be signaled.

`IR_SIG_LAP_CONNECT`

This signal indicates that an IrLAP connection has come up. The requesting connection will be signaled.

Parameters

`path`

The signal is unregistered from this path.

`signalcode`

The signal code to unregister. Acceptable values are contained in `irda.h`. Some values follow:

```
IR_SIG_DATA_AVAIL
IR_SIG_NO_PROGRESS
IR_SIG_LINK_OKAY
IR_SIG_LAP_DOWN
IR_SIG_IRLAP_CONNECT
```

Non-Fatal Errors

`SUCCESS`	The operation is successful.
`IR_BAD_INPUT`	The signal code value is out of range.

Libraries

`irda.l`

ir_write()

Write Data

Syntax

```
error_code ir_write(path_id path, u_char* buf,
                    u_int32* sendamount)
```

Description

ir_write() writes data to an infrared communications path.

Parameters

path	The path to which data is written.
buf*	A pointer to the application memory that contains the data to write.
sendamount*	A pointer to the number of bytes to write. This value is set to the actual number of bytes written when the function returns.

Non-Fatal Errors

SUCCESS	The operation is successful.
EOS_WRITE	The write to the infrared communications path failed for one of the following reasons: • An IR communications path has not been opened successfully. • Packet exceeds the maximum size (error check only). • An IrLAP connection is not active.
R_REQ_FAILED	The operation failed because the other side requested an LMP/Tiny TP disconnect while the packet of data was waiting to be sent.

`EOS_DEVBSY` This error is only received if the descriptor for the infrared communications stack has the asynchronous bit set. This error means that the write queue is full and the unwritten data must be resent. The pointer to the actual number of bytes written allows the application to determine how much of the write buffer to retransmit.

Any error codes that an `_os_write` function may return.

Libraries

`irda.l`

Chapter 4: OBEX Library Calls

Overview

The OBEX library code provides the IrDA protocol stack with OBEX functionality. OBEX is not a driver. OBEX accesses the IrDA driver through the IrDA library calls. Since OBEX runs in the application or user space, a signal handler function is necessary for OBEX functions to be informed of stack events. Currently this signal handler is maintained in the OBEX library code. Applications that use OBEX will not be able to register their own signal handler with the intercept call. A signal handler can be registered by using the signal call instead.

Error Codes Specific To The OBEX library

These error codes are currently defined in `obex.h`. These errors are generated only with the OBEX library.

`OB_SERV_OPEN`	Server already open.
`OB_CLNT_OPEN`	Client already open.
`OB_NOT_CONNC`	No Client OBEX connection.
`OB_PENDING`	Operation is in progress.
`OB_CLIENT_BUSY`	Client has some other operation in progress.
`OB_SERVER_BUSY`	Server has some other operation in progress.
`OB_HEADER_ERR`	Error while trying to build the OBEX header.

OBEX Header Construction Functions

OBEXH_Build1Byte()

Build a 1 Byte Style Header

Syntax

```
error_code OBEXH_Build1Byte(u_int8 cs, ObexHeaderType Type,
                            u_int8 Value)
```

Description

Build a 1 byte style header.

Parameters

`cs`	Flag that indicates whether the header is created for a client or server. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.
`Type`	`OBEXH_`type of header to build.
`Value`	Byte value to place in header.

Error Codes

`SUCCESS`	Operation completed successfully.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.
`OB_HEADER_ERR`	Header exceeded buffer size.

Libraries

`obex.l`

OBEXH_ Build4Byte()

Build a 4 Byte Style Header

Syntax

```
error_code OBEXH_Build4Byte(u_int8 cs, ObexHeaderType Type,
                            u_int32 Value)
```

Description

Build a 4 byte style header.

Parameters

`cs`	Flag that indicates whether the header is created for a client or server. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.
`Type`	OBEXH_ type of header to build.
`Value`	4 byte value to place in header.

Error Codes

`SUCCESS`	The operation completed successfully.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.
`OB_HEADER_ERR`	Header exceeded buffer size.

Libraries

`obex.l`

OBEXH_BuildByteSeq()

Build a Style Header

Syntax

```
error_code OBEXH_BuildByteSeq(u_int8 cs, ObexHeaderType Type,
                              u_int8 *Value, u_int16 Len)
```

Description

Build a Byte Sequence style header.

Parameters

`cs`	Flag that indicates whether the header is created for a client or server. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.
`Type`	OBEXH_type of header to build.
`*Value`	Buffer to copy into header.
`Len`	Length of data pointed to by `Value`.

Error Codes

`SUCCESS`	The operation completed successfully.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.
`OB_HEADER_ERR`	The header exceeded buffer size.

Libraries

`obex.l`

OBEXH_BuildUnicode()

Build a Unicode Style Header

Syntax

```
error_code OBEXH_BuildUnicode(u_int8 cs, ObexHeaderType Type,
                              u_int8 *Value, u_int16 Len)
```

Description

Build a Unicode style header.

Parameters

cs	Flag that indicates whether the header is created for a client or server. Acceptable values are OBEX_SERVER or OBEX_CLIENT.
Type	OBEXH_ type of header to build.
*Value	Buffer to copy into header.
Len	Length of data pointed to by Value.

Error Codes

SUCCESS	Operation completed successfully.
IR_BAD_INPUT	The flag value is out of the acceptable range.
OB_HEADER_ERR	Header exceeded buffer size.

Libraries

obex.l

OBEXH_FlushHeaders()

Flush Header Buffer

Syntax

```
error_code OBEXH_FlushHeaders(u_int8 cs)
```

Description

Flush header buffer of unsent headers.

Parameters

`cs`	Flag indicating whether a client or server is to flush its buffer. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.

Error Codes

`SUCCESS`	Operation completed successfully.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.

Libraries

`obex.l`

OBEX Header Deconstruction Functions

OBEXH_GetHeader1Byte()

Get Value of 1 Byte Style Header

Syntax

```
error_code OBEXH_GetHeader1Byte(u_int8 cs, u_int8 *value)
```

Description

Returns the value of a 1 byte style header.

Parameters

`cs`	Flag that indicates whether the header is created for a client or server. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.
`*value`	Pointer to the one byte header.

Error Codes

`SUCCESS`	Operation completed successfully.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.

Libraries

`obex.l`

OBEXH_GetHeader4Byte()

Get Value of a 4 Byte Style Header

Syntax

```
error_code OBEXH_GetHeader4Byte (u_int8 cs, u_int32 *value)
```

Description

Returns the value of a 4 byte style header.

Parameters

`cs`	Flag that indicates whether the header is created for a client or server. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.
`*value`	Pointer to the four byte header.

Error Codes

`SUCCESS`	Operation completed successfully.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.

Libraries

`obex.l`

OBEXH_GetHeaderAvailLen()

Return Header Length

Syntax

```
error_code OBEXH_GetHeaderAvailLen(u_int8 cs, u_int16 *len)
```

Description

Returns the currently available (to copy) header length. For use only with Byte Sequence and Unicode headers.

Parameters

`cs`	Flag that indicates whether the header is created for a client or server. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.
`*len`	Pointer to the length.

Error Codes

`SUCCESS`	Operation completed successfully.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.

Libraries

`obex.l`

OBEXH_GetHeaderBuff()

Get Pointer to Current Headers Buffer

Syntax

```
error_code OBEXH_GetHeaderBuff(u_int8 cs, u_int8 **header)
```

Description

Returns a pointer to the current headers buffer. For use only with Byte Sequence and Unicode headers.

Parameters

`cs`	Flag that indicates whether the header is created for a client or server. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.
`**header`	Pointer to the buffer pointer.

Error Codes

`SUCCESS`	Operation completed successfully.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.

Libraries

`obex.l`

OBEXH_GetHeaderType()

Return Type of Current Header

Syntax

```
error_code OBEXH_GetHeaderType(u_int8 cs, ObexHeaderType *type)
```

Description

Returns the type of the current header.

Parameters

cs	Flag that indicates whether the header is created for a client or server. Acceptable values are OBEX_SERVER or OBEX_CLIENT.
*type	Pointer to the OBEX header type.

Error Codes

SUCCESS	Operation completed successfully.
IR_BAD_INPUT	The flag value is out of the acceptable range.

Libraries

obex.l

OBEXH_GetTotalHeaderLen()

Get Total Length of Current Header

Syntax

```
error_code OBEXH_GetTotalHeaderLen(u_int8 cs, u_int16 *len)
```

Description

Returns the total length of the current header. For use only with Byte Sequence and Unicode headers.

Parameters

`cs`	Flag that indicates whether the header is created for a client or server. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.
`*len`	Pointer to the length.

Error Codes

`SUCCESS`	Operation completed successfully.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.

Libraries

`obex.l`

OBEX General functions

OBEX_Abort()

Abort the Current Process

Syntax

```
error_code OBEX_Abort(u_int8 cs, ObexRespCode Resp)
```

Description

Abort the process that is being performed. This may be either a client or server process.

Parameters

`cs`	A flag that specifies either client or server. Acceptable values are OBEX_SERVER or OBEX_CLIENT.
`Resp`	Reason for abort. Only used with server.

Error Codes

`SUCCESS`	Operation completed successfully. The process will be aborted during the next OBEX transmission.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.
`IR_REQ_FAILED`	The abort failed. Server or client process may not have been opened or no request in process.

Libraries

`obex.l`

OBEX_Close()

Close an OBEX Server or Client

Syntax

```
error_code OBEX_Close(u_int8 cs)
```

Description

Close either an OBEX server or client.

Parameters

`cs`	A flag that specifies either client or server process. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.

Error Codes

`SUCCESS`	Operation completed successfully. The specified OBEX process will close immediately. If no other process is using the IrDA stack then the stack will be closed also.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.
`IR_REQ_FAILED`	The specified process does not exist.

Libraries

`obex.l`

OBEX_ClientConReq()

Request a Connection for a Client

Syntax

```
error_code OBEX_ClientConReq(void)
```

Description

Attempt to setup a Tiny TP connection to an OBEX Server. Completion or failure of the operation is reported by calling the application client callback function with the appropriate event.

Parameters

None

Error Codes

`SUCCESS`	Operation completed successfully.
`IR_REQ_FAILED`	Operation failed.
`IR_NO_LAP`	Operation failed because there is no IrLAP connection.
`IR_LAP_DISCON`	The IrLAP connection was disconnected during the transaction.
`IR_MEDIA_BUSY`	Operation failed because the media is busy.

Libraries

`obex.l`

OBEX_Connect()

Create an OBEX Connection

Syntax

```
error_code OBEX_Connect(void)
```

Description

Perform the OBEX connect operation. If it is to be executed over a Tiny TP connection then `OBEX_ClientConReq()` must be executed first.

Parameters

None

Error Codes

`OB_PENDING`	Operation was successfully started. Completion will be signaled via a call to the application client callback function.
`IR_REQ_FAILED`	Operation was not started because the client is currently executing another operation.

Libraries

`obex.l`

OBEX_Disconnect()

Terminate an OBEX Connection

Syntax

```
error_code OBEX_Disconnect()
```

Description

Disconnect the client OBEX connection. If a server connection exists then this function will not disconnect IrLAP.

Parameters

None

Error Codes

`OB_PENDING`	Operation was successfully started. Completion will be signaled via a call to the application client callback function.
`IR_REQ_FAILED`	Operation failed because the server has a connection up. When the server is complete it will disconnect the link.
`OB_NOT_CONNC`	Operation failed because there is no client connection.

Libraries

`obex.l`

OBEX_DiscReq()

Disconnect the IrLAP Connection

Syntax

```
error_code OBEX_DiscReq(u_int8 force)
```

Description

Disconnect the IrLAP connection. If a server connection exists then this function will not disconnect IrLAP unless force is equal to TRUE.

Parameters

`force`	TRUE means disconnect IrLAP even if the server connection exists. FALSE means only disconnect IrLAP if server connection is disconnected.

Error Codes

`SUCCESS`	Operation successfully disconnected IrLAP.
`IR_REQ_FAILED`	Operation failed because the server has a connection up. Need to try again when the server is finished.
`OB_NOT_CONNC`	Operation failed because there is no client connection.

Libraries

`obex.l`

OBEX_Get()

Get the Specified File

Syntax

```
error_code OBEX_Get(ObStoreHandle ClientObj)
```

Description

Start a process to get the file specified. The received object is stored at the location specified by the OBSTORE library. This process can only be initiated by the client. The server uses this filename to obtain the object to send to the client.

Parameters

`ClientObj`	A pointer to a filename that describes the object to get.

Error Codes

`SUCCESS`	The process has been started successfully and the results will be signaled via a call to the application client callback function.
`IR_REQ_FAILED`	Operation failed.
`OB_NOT_CONNC`	Operation failed because there is no client connection.

Libraries

`obex.l`

OBEX_GetAbortReason()

Get the Reason for the Abort

Syntax

```
error_code OBEX_GetAbortReason(ObexAbortReason *reasoncode)
```

Description

Return the reason for the current abort indication. This call is only valid during calls to `ObClientProcessApp()` with the event `OBCE_ABORTED`.

Parameters

`*reasoncode`	The reason for the abort will be contained at this pointer location. See `ObexAbortReason` in `obex.h`.

Error Codes

`SUCCESS`	Operation completed successfully.

Libraries

`obex.l`

OBEX_Open()

Open an OBEX Server or Client

Syntax

```
error_code OBEX_Open(char* driver, u_int8 cs, void *appfunc)
```

Description

Open either an OBEX server or client. This function calls `ir_open()`. Any error that is caused by `ir_open()` will be returned by this function to the calling application.

Parameters

`*driver`	A pointer to the descriptor of the driver to use. Normally this will just be used to specify different framers.
`cs`	Flag indicating a client or server to be opened. Acceptable values are `OBEX_SERVER` or `OBEX_CLIENT`.
`*appfunc`	The pointer to OBEX callback function for this OBEX path. This function should never make calls to the `Process_OBEX` function. The `Process_OBEX` function calls this function to indicated events. This function can have one parameter of type `ObServerEvent`.

Error Codes

`SUCCESS`	Operation completed successfully. The process will be aborted during the next OBEX transmission.
`IR_BAD_INPUT`	The flag value is out of the acceptable range.
`OB_SERV_OPEN`	The request to open a server failed because a server is already open.

`OB_CLNT_OPEN`	The request to open a client failed because a client is already open.

Libraries

`obex.l`

OBEX_Put()

Put a File

Syntax

```
error_code OBEX_Put(ObStoreHandle ClientObj)
```

Description

Start a process to put the file specified. Currently the file must exist on the ram disk /r0 or /dd under the directory qbinbox. This process can only be initiated by the client. The server uses this filename to store the object that it receives.

Parameters

`ClientObj`	A pointer to a filename that describes the object to put.

Error Codes

`SUCCESS`	The process has been started successfully and the results will be signalled via a call to the application client callback function.
`IR_REQ_FAILED`	Operation failed.
`OB_NOT_CONNC`	Operation failed because there is no client connection.

Libraries

`obex.l`

OBEX_ServAccept()

Accept Server Operation

Syntax

```
error_code OBEX_ServAccept(ObStoreHandle Obsh)
```

Description

Accept the current server operation.

Parameters

`Obsh`	Valid store handle.

Error Codes

`IR_BAD_INPUT`	Incorrect values in input parameters. Possible that the set path command could not fit into one OBEX Packet.
`SUCCESS`	

Libraries

`obex.l`

OBEX_ServGetPathFlags()

Get Path Flags

Syntax

```
error_code OBEX_ServGetPathFlags(ObexSetPathFlags *flags)
```

Description

Retrieve the flags associated with the SetPath request. This function is valid during a OBSE_SETPATH_START indication.

Parameters

`*flags`	Pointer to variable that returns the flags. See `ObexSetPathFlags` in `obex.h` for values.

Error Codes

```
SUCCESS
```

Libraries

```
obex.l
```

OBEX_SetPath()

Set the Specified Path

Syntax

```
error_code OBEX_SetPath(u_int8 flags)
```

Description

Start a process to set the specified path on the device that is being communicated with over IR.

Parameters

flags	Flags associated with set path. See `ObexSetPathFlags` in `obex.h` for values.

Error Codes

`OB_PENDING`	Operation was successfully started. Completion will be signaled via a call to the application client function.
`IR_BAD_INPUT`	Incorrect values in input parameters. Possible that the set path command could not fit into one OBEX Packet.

Libraries

`obex.l`

Process_OBEX()

Allows OBEX Processes to Run

Syntax

```
error_code Process_OBEX(void)
```

Description

Allows OBEX processes to run. Do not call this function in any of the registered OBEX callback functions. Calling this function from there will cause re-entrancy problems in the OBEX libraries. This function processes one Tiny TP packet of data for both the server and client if both are open and running. If this function is not called, the OBEX protocol will not respond to the other side since OBEX is a user state process that is run by the application using the OBEX libraries. This function performs both IrDA registration and read calls to the IrDA stack. Therefore, any errors produced will be returned to the calling application.

Parameters

None

Error Codes

`SUCCESS`	One pass at the opened OBEX protocols finished.

Libraries

`obex.l`

Chapter 5: OBSTORE Library Calls

Overview

The OBSTORE library code provides an OBEX user with the flexibility to change where and how objects are stored and retrieved.

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

The structure `ObStoreEntry` is used only in the OBSTORE library, `obstore.l`. This structure may be modified to met the needs of the implementation.

OBSTORE_AppendNameAscii()

Append ASCII String

Syntax

```
void OBSTORE_AppendNameAscii(ObStoreHandle obs,  u_int8 *name,
                             u_int16 len)
```

Description

Appends the ASCII string to the object name. Called when a client put or get is initiated.

Parameters

`obs`	pointer to object store entry.
`name`	pointer to ASCII string to append.
`len`	length in bytes of string to append.

Return Values

void

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_AppendNameUnicode()

Append Unicode String

Syntax

```
ObexRespCode OBSTORE_AppendNameUnicode(ObStoreHandle obs,
                                       u_int8 *name,
                                       u_int16 len)
```

Description

Append the unicode string to the object name. Called when a object name is received over IR. This function may be called more than once if the file name is sent in multiple TTP packets.

Parameters

`obs`	Pointer to the object store entry.
`*name`	Pointer to Unicode string to append.
`len`	Length in bytes of string to append.

Return Values

`ObexRespCode`	User defined. Values such as `OBRC_SUCCESS` and `OBRC_UNAUTHORIZED` are suggested.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_AppendPathUnicode()

Append Path to Current Path

Syntax

```
ObexRespCode OBSTORE_AppendPathUnicode(u_int8 *path,
                                       u_int16 len,
                                       u_int8  first)
```

Description

Append the path provided to the current OBEX Server path. The path provided is in Unicode. Called by the server during a set path operation when a name is received. The parameter first is used so the name can be sent in multiple Tiny TP packets.

Parameters

`*path`	Unicode path string.
`len`	length of the path in bytes.
`first`	If set, this is the first packet for the name. If not set, the name is being continued form a previous packet.

Return Values

ObexRespCode	User defined. Values such as `OBRC_SUCCESS` and `OBRC_UNAUTHORIZED` are suggested.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_CancelPath()

Cancel a Path Update

Syntax

```
void OBSTORE_CancelPath(void)
```

Description

Cancel a path update. Called whenever a set path operation fails for any reason.

Parameters

None

Return Values

Void

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_Close()

Close the Object Store Entry

Syntax

```
ObexRespCode OBSTORE_Close(ObStoreHandle obs)
```

Description

Close the object store entry. The protocol is now done with this object. Called when a client or server get or put completes. It is also called when a client put or server get is aborted.

Parameters

`obs`	Pointer to the object store entry.

Return Values

`ObexRespCode`	User defined. Values such as `OBRC_SUCCESS`, `OBRC_UNAUTHORIZED` and `OBRC_NO_CONTENT` are suggested.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_Create()

Creates an Object

Syntax

```
ObexRespCode OBSTORE_Create(ObStoreHandle obs, u_int8 useInbox)
```

Description

Called during a client get or server put. Must create an object that will have data written to it. OBSTORE_Write() must be able to write to the object created here. Use the ObStoreHandle structure to save a pointer to the object. The useInbox flag is set if the server is performing the create.

Parameters

obs	Return the handle to object store entry or 0 if error.
useInbox	If TRUE create the file from within the inbox. If FALSE create file in initial directory.

Return Values

ObexRespCode	User defined values.

Libraries

obstore.l

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_Delete()

Delete an Object Store Entry

Syntax

```
ObexRespCode OBSTORE_Delete(ObStoreHandle obs)
```

Description

Delete the given object store entry. The protocol is now done with this object. Called when a client get or server put is aborted.

Parameters

`obs`	Pointer to the object store entry.

Return Values

`ObexRespCode`	User defined. Values such as `OBRC_SUCCESS` and `OBRC_NOT_FOUND` are suggested.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_GetNameLenUnicode()

Get Unicode Length of Name

Syntax

```
u_int8 OBSTORE_GetNameLenUnicode(ObStoreHandle obs)
```

Description

Get the Unicode length of the object store's entry's name. The length includes the terminator. Called when a client get or put is initiated.

Parameters

`obs`	Pointer to the object store entry.

Return Values

`u_int8`	Number of bytes in the name.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_GetObjectLen()

Return File Length of Object

Syntax

```
u_int32 OBSTORE_GetObjectLen(ObStoreHandle obs)
```

Description

Return the file length of a created object store entry. Called when a client put or server get is initiated.

Parameters

`obs`	Pointer to the object store entry.

Return Values

`u_int32`	Length of the file.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_Init()

Initializes the Object Store

Syntax

```
u_int8 OBSTORE_Init()
```

Description

Called during the first call to the OBEX library function `OBEX_Open()`. This function should initialize the object store by setting up the inbox where objects are to be stored.

Parameters

None

Return Values

`u_int8`	If TRUE object store was initialized successfully. If FALSE unable to set up the inbox.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_IsDefaultObject()

Is the Default Object Referenced

Syntax

Not Available

Description

Determine if the object store references the default object. Called during a client get operation.

Parameters

`ObStoreHandle`	Pointer to the object store entry.

Return Values

`u_int8`	Return `TRUE` if the ObStoreHandle points to the default object otherwise return FALSE.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_IsOpen()

Determine if Object Store Entry is Open

Syntax

Not Available.

Description

Determine if the object store entry is open. Called during a server get operation.

Parameters

`ObStoreHandle`	Pointer to the object store entry.

Return Values

`u_int8`	Return `TRUE` if the object store entry is open otherwise return `FALSE`.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_New()

Creates a New Object Store Entry

Syntax

```
ObStoreHandle OBSTORE_New()
```

Description

Called whenever a client or server get or put is requested. A blank object store entry is to be returned.

Parameters

None

Return Values

`ObStoreHandle`	Handle of an object store entry. 0 means out of store objects.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_Open()

Open an Object for Reading

Syntax

```
ObexRespCode OBSTORE_Open(ObStoreHandle obs, u_int8 useInbox)
```

Description

Called during a client put or server get. Must open an object that will have data read from it. `OBSTORE_Read()` must be able to read from the object opened here. Use the `ObStoreHandle` structure to save a pointer to the object. If a name is not set, the default object is opened in the default inbox. The `useInbox` flag is set if the server is performing the open.

Parameters

`obs`	Pointer to object store entry
`useInbox`	If TRUE open the file from within the inbox. If FALSE open file in current directory. This parameter may be ignored. It signals a store library to look in different directories such as in the case where the other device sets the path and expects to send and receive objects to and from this new path.

Return Values

`ObexRespCode`	User defined values.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_Read()

Read Data from OBSTORE Object

Syntax

```
ObexRespCode OBSTORE_Read(ObStoreHandle obs, u_int8 *buff,
                          u_int16 len)
```

Description

The data is read from the object that was set up during the `OBSTORE_Open()`. Called whenever the OBEX library needs an object body header for either a client put process or server get process.

Parameters

`obs`	Pointer to the object store entry.
`*buff`	Pointer to the storage location for the data read from the object.
`len`	Length in bytes of data to read.

Return Values

`ObexRespCode`	User defined. Values such as `OBRC_SUCCESS` and `OBRC_UNAUTHORIZED` are suggested.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_ReadNameUnicode()

Read Object Name in Unicode

Syntax

```
u_int16 OBSTORE_ReadNameUnicode(ObStoreHandle  obs,
                                u_int8  *buffer,  u_int16  len,
                                u_int8  first)
```

Description

Copy the Object Store entry's name into the provided buffer in Unicode. Called during a client get or put process. The parameter first is used so the name can be sent in multiple Tiny TP packets.

Parameters

`obs`	Pointer to the object store entry.
`*buffer`	Pointer to the buffer to receive the name.
`len`	Length of the buffer in bytes.
`first`	If set, this is the first packet for the name. If not set, the name is being continued from a previous packet.

Return Values

len	Return the number of bytes copied including the null terminator.

Libraries

`obstore.l`

WARNING
All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_ResetPath()

Reset Path

Syntax

```
void OBSTORE_ResetPath(void)
```

Description

Called in two places. The first place is when a server connection is started. The second place is during a reset path command in a set path operation that the server receives.

Parameters

None

Return Values

Void

Libraries

```
obstore.l
```

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_SetNameDefault()

Set Object to Default Name

Syntax

```
void OBSTORE_SetNameDefault(ObStoreHandle obs)
```

Description

Set the name of the Object store entry to the default name for the object that is to be received. Set the obs->flags |= OBSEF_DEFAULT_OBJECT. This tells the OBEX protocol to send a default object. This function is called when a client get processes is started and a name is not set or the length of the get name is 0.

Parameters

obs	Pointer to the object store entry.

Return Values

Void

Libraries

obstore.l

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_SetObjectLen()

Set File Length of Entry

Syntax

```
void OBSTORE_SetObjectLen(ObStoreHandle obs, u_int32 len)
```

Description

Set the file length of a created object store entry. Called when the server processes the start of a put operation.

Parameters

`obs`	Pointer to the object store entry.
`len`	Length of file.

Return Values

Void

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_UpdatePath()

Request that Path is Updated

Syntax

```
ObexRespCode OBSTORE_UpdatePath(u_int8 flags,  u_int8 update)
```

Description

Called in two places. If the server receives the `OBS_UPU_BACKUP` flag during a set path operation this function is called. This function is also called once a complete packet is received. The update flag is set in this case.

Parameters

`flags`	SetPath flags to apply.
`update`	Request that the current path be applied. This may be ignored. The other device is requesting that the current path be modified to the path set in `AppendPathUnicode()`.

Return Values

`ObexRespCode`	User defined. Values such as `OBRC_SUCCESS`, `OBRC_NOT_FOUND` and `OBRC_UNAUTHORIZED` are suggested.

Libraries

`obstore.l`

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

OBSTORE_Write()

Write Data to OBSTORE Object

Syntax

```
ObexRespCode OBSTORE_Write(ObStoreHandle obs, u_int8 *buff,
                           u_int16 len)
```

Description

The data is written to the object that was set up during the OBSTORE_Create(). Called whenever a object body header is received for either a client get process or server put process.

Parameters

obs	Pointer to the object store entry.
*buff	Pointer to data to write.
len	Length in bytes of data to write.

Return Values

ObexRespCode	User defined. Values such as OBRC_SUCCESS and OBRC_UNAUTHORIZED are suggested.

Libraries

obstore.l

WARNING

All functions in this library are to be called solely by the OBEX library. None of these calls should be used by an application. These calls are provided for background information only.

Chapter 6: The JetBeam for OS-9 Framers

This chapter describes the functionality and interfaces of framers with JetBeam for OS-9.

Note

A framer—as it relates to infrared communications—is a module that interfaces the IR Communications protocol stack (`spirlite`) to the hardware. Thus, a framer serves the same function as a device driver.

Framer Overview

In infrared communications, a system’s Framer interfaces the protocol stack module to the hardware (typically a UART—universal asynchronous receiver/transmitter). Thus, the Framer module acts as the system’s hardware driver and is responsible for the physical layer requirements specified by the Infra-red Data Association (IrDA).

JetBeam for OS-9 provides one of the following Framers: the `sp1110ir` module for the SA11x0 processors or the `sp7709ir` module for the SH-3 processors. Developers can use this Framer as a guide for porting to other processors or hardware implementations.

This section provides an overview of framers as they are used with the OS-9 for 68K and OS-9 operating systems. Basic framer design, source file relationships, data flow, and basic packet structure are also discussed.

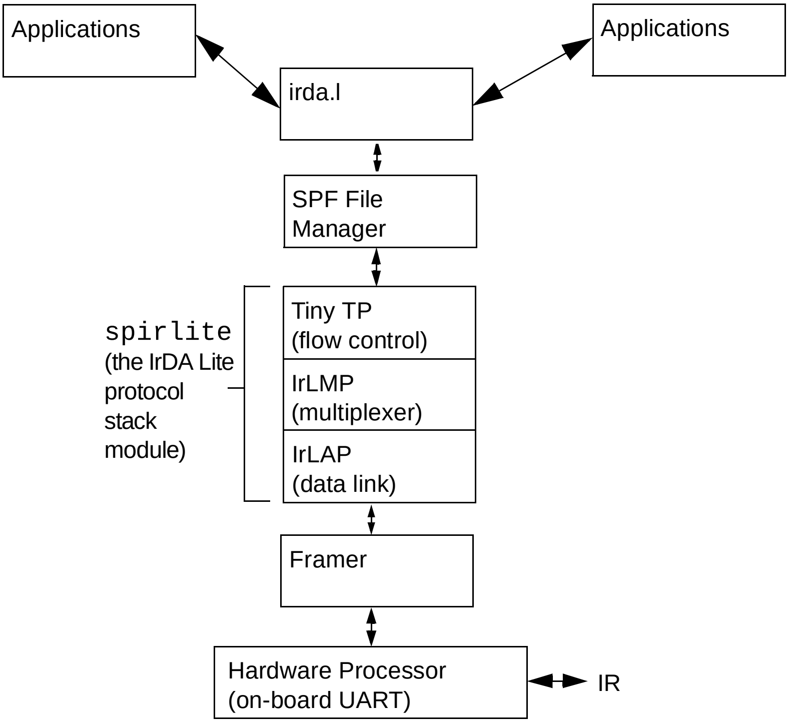
Framer Design

A Framer establishes and maintains a communications path between a processor’s on-board UART (universal asynchronous receiver/transmitter) and `spirlite` (the IrDA protocol stack).

Framers are Serial Infrared (SIR) hardware drivers. Therefore, they are always located on the bottom of an infrared protocol stack. A Framer’s primary purpose is to interface with the hardware and perform framing services for the `spirlite` protocol stack which is opened above it. Framers are a medium for data exchange between `spirlite` and actual IR signals. They are responsible for all hardware configuration and communication.

Framers for the JetBeam for OS-9 are implemented as standard SPF (SoftStax) drivers under DPIO with support for all the required calls/entry points. The following figure shows a general protocol stack configuration for an IrDA Lite implementation.

Figure 6-1 Protocol Stack Configuration for an IrDA Lite Implementation



Note

Framers provided with this release of JetBeam for OS-9 do not support ITEM facilities and use the standard MBUF calls for inter-driver communications. ITEM facilities are not currently supported within the IrDA Lite protocol module, `spirlite`.

As shown in **Figure 6-1 Protocol Stack Configuration for an IrDA Lite Implementation**, Framers depend on SPF and `spirlite`. In JetBeam for OS-9, Framers give `spirlite` the ability to:

- initialize the needed hardware for protocol communications
- transfer data to and from the upper layer protocol stack through the standard SPF entry points
- transfer data to and from the UART IO port through interrupt driven procedures
- service upper layer requests for logical hardware/software configuration and status, for example data rate, buffer size, turn-around time, and the number of BOFs
- provide proper framing requirements, error detection/recovery (CRC), and control byte transparency
- reset the hardware at termination

In addition, JetBeam for OS-9 Framers support the following IrDA default communication requirements:

- Baud rate 9600 bps
- Data size 64 bytes
- Window size 1
- Additional BOFs 0 (set to 10 to meet 10ms minimum turn-around time requirement)
- Maximum turn around time 500ms
- Disconnect/Threshold time 3 seconds

Framers support the following SIR data rates (kbps):

- 9.6
- 19.2
- 38.4
- 57.6
- 115.2

Framers support the following data sizes (bytes):

- 64
- 128
- 256
- 512
- 1024
- 2048

To provide a primary interface to the hardware's UART port (supporting IR), framers must configure one or more devices. For example, these devices may include the following:

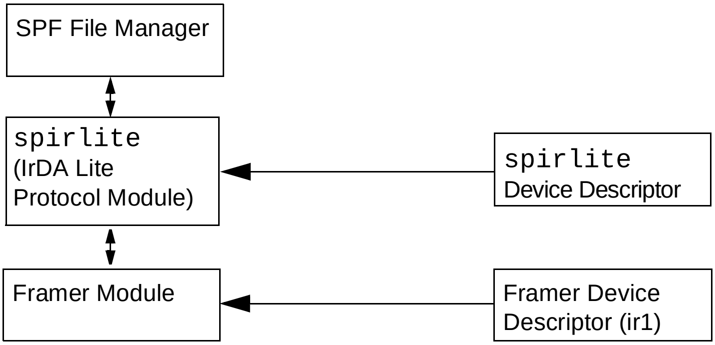
- Parallel Port
- Universal Asynchronous Receiver Transmitter (UART)
- Programmable Interrupt Controller (PIC)
- Count Timers

Intermodule Dependencies

Figure 6-2 displays the JetBeam for OS-9 modules and their relationships. Under the SPF file manager, the two stacked modules—spirlite and Framer—form the overall IrDA Lite protocol stack. This stacking in SoftStax is accomplished by opening a path to the stack by using a list of descriptors. The following path is used In JetBeam for OS-9:

```
/ir1/spirlite
```

Figure 6-2 JetBeam for OS-9 Module Relationships



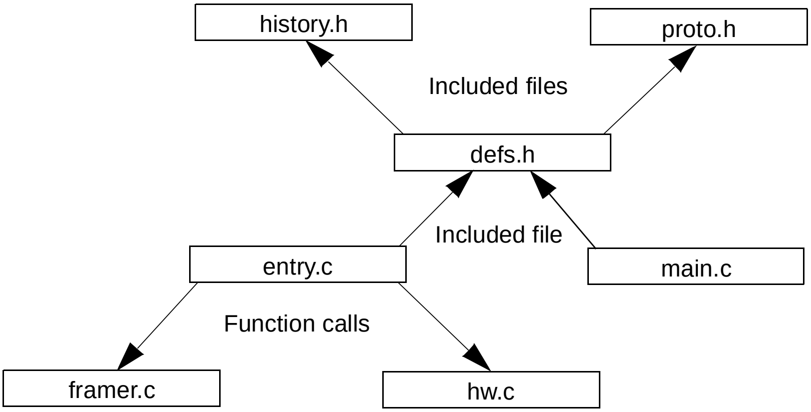
Source File Relationships

The framer module consists of the standard SoftStax (SPF) driver source files, with some additions. The standard files, defs.h, entry.c, history.h, main.c, and proto.h, are maintained. Two files, hw.c and framer.c, are added. The hw.c file contains the source for routines that actually control or configure hardware devices, such as UARTs and PICs. The framer.c file contains all other necessary source code. The two additional source code files make it unnecessary to add routines to entry.c. entry.c should only contain the defined SPF entry points, and reference the other files with function calls when needed. Figure 6-3 illustrates the relationships between these source files.

For More Information

See SoftStax for standard driver source files.

Figure 6-3 Framer Source File Relationships



In addition to the source code file dependencies, the framer module is also dependent on the following files:

- `/mwos/SRC/DEFS/SPF/spf.h (included in defs.h)`
- `/mwos/SRC/DEFS/SPF/mbuf.h (included in defs.h)`
- `/mwos/SRC/DEFS/SPF/irda.h (included in defs.h)`

SH-3 only

- `/mwos/OS9000/SH3/PORTS/SH7709/SPF/SP7709IR/DEFS /spf_desc.h (device descriptor)`

The framer module is dependent on `spf.h` for common definitions to all SoftStax (SPF) driver modules. The framer module is dependent upon `mbuf.h` for the use of mbufs for data transfer between drivers and the SPF file manager. The framer module is dependent on `irda.h` for all JetBeam for OS-9 definitions and structures defined for use in the IrDA Lite protocol module. The remaining header file, `spf_desc.h`, initializes the device descriptor. It is described below.

Note

spf.h, mbuf.h, and irda.h are described in **Chapter 2:Using JetBeam™ for OS-9**. spf_desc.h is described in the ***Porting SoftStax*** manual.

defs.h

The defs.h file includes all header files used to develop the Framer and is where the Framer's capabilities are defined. Some of the Framer's capabilities to define include IrLAP speed, data size, window size, and minimum turn-around time.

Use defs.h to also define the device-specific driver static storage, spf_drstat, and the device-specific logical unit static storage, spf_lustat. The actual spf_drstat and spf_lustat structures are defined in spf.h.

When the Framer is compiled, the spf_drstat structure is created and initialized using the structure defined in main.c. The spf_lustat structure is logically created but no initial values have yet been assigned to the structure. Once a path is opened to the driver using its descriptor, the structure is initialized. This is because the descriptor stores the copy of the initialized spf_lustat data structure, and on opening the path, the structure gets copied to the Framer's static storage area.

Two macros, SPF_DRSTAT and SPF_LUSTAT, provide additional variables for each driver. The two macros expand the spf_drstat and spf_lustat structures to include useful information for each driver. These expanded variables are called device specific.

The SPF_DRSTAT macro defines the device-specific driver static storage, and the SPF_LUSTAT macro defines the device-specific logical unit static storage.

The SPF_LUSTAT macro includes the device-specific logical unit variables. See the source code provided for the minimum requirements of variables to be implemented and initialized with the SPF_LUSTAT_INIT macro. The SPF_LUSTAT_INIT macro initializes the device-specific logical static storage of the framer. These values get stored in the descriptor, ir1, for initialization when the path is actually opened.

The following table describes the expanded or additional device-specific logical unit variables.

Table 6-1 Device-specific logical unit variables

Variable	Description
lu_dbg and lu_dbg_name	implement a debug data module
lu_vector and lu_priority	install the ISR
lu_cache_cctl and lu_cache_static	support system caching
lu_hwenable	tracks whether the IR hardware receiver or transmitter is enabled
lu_data_updrvr	stores the lu_updrvr that enabled the hardware
lu_rxsize	initially set to 0, but should be initialized to the maximum receive buffer size the framer allows
lu_rxmbuf and lu_txmbuf	store a pointer to the current receiving or transmitting mbuf that the framer has allocated

Table 6-1 Device-specific logical unit variables (continued)

Variable	Description
`lu_pkt_state`	tracks the current receive and transmit states
`lu_pkt_trans`	tracks when to apply transparency
`lu_pkt_cnt`	tracks the current count of bytes being transmitted and received
`lu_pkt_crc`	stores the CRC of the packet being transmitted
`lu_frm_fcstbl`	calculates and verifies every packet's CRC
`lu_frm_caps`	sets the framer's capabilities; this structure is retrieved by `spirlite` to negotiate connection parameters
`lu_frm_busy`	stores the framer's media busy state
`lu_frm_rx`	stores the framer's receiver state
`lu_frm_txbofs`	sets the number of BOF bytes that are sent at the beginning of every transmitted frame
`lu_frm_index`	sets the framer's minimum turn-around time
`lu_sigmbuf`	a statically allocated mbuf used for signaling the protocol module of specific events in the framer

history.h

The history.h file stores all the revision/edition comments as well as the current Framer module and descriptor module edition numbers. This file has the standard Microware Copyright statement followed by the edition history.

This file also contains the Framer's and descriptor's edition and revision values. When compiled, the Framer and descriptor modules take on the following values.

```
/*
**  Edition/Revision Numbers
*/
/* Driver edition number information */
#ifdef SYSEDIT
   _asm("_sysedit: equ 1");
   _asm("_sysattr: equ 0xA000");
#endif
/* Descriptor edition number information */
#ifdef DESC_SYSEDIT
   _asm("_sysedit: equ 1");
   _asm("_sysattr: equ 0x8000");
#endif
```

proto.h

The proto.h file contains all of the Framer's function prototypes. All of the .c source files depend on these function prototypes.

entry.c

The entry.c file contains all of the Framer's standard SPF entry points. SPF and the protocol module depend on these standard entry points to exist. No other functions are added to this file. If additional functions need to be performed, a function call to one of the other source files—hw.c and framer.c—can be done. entry.c also includes the Framer's defs.h file. The entry points are as follows:

- dr_iniz
- dr_term
- dr_getstat
- dr_setstat
- dr_downdata
- dr_updata

main.c

The main.c file contains the spf_drstat structure with its initial variable values. Any of the additional device-specific driver variables (SPF_DRSTAT), defined in defs.h, are also initialized here.

hw.c

This file contains all of the hardware routines the Framer needs to perform. The SPF entry point functions listed in the entry.c section of this document depend on the hardware functions in hw.c to actually configure the hardware, specifically the UART.

Using the hw_function convention, some hw.c functions that the SPF entry points need to call to control the hardware's processor are listed in **Table 6-2**.

Table 6-2 hw.c Functions

Function	Description
hw_iniz	initialize all hardware registers and structures
hw_term	reset hardware registers and structures
hw_ss_en	enable the transmitter and receiver
hw_ss_dis	disable the transmitter and receiver
hw_ss_baud	set the communications baud rate
hw_ss_baud_reg	set the baud rate control registers (StrongARM only)
hw_minta_start	enable timer interrupt
hw_minta_stop	disable timer interrupt
hw_tx	transmit and mbuf
hw_rdy_tx	ready transmit channel
hw_rdy_rx	ready receive channel
hw_rx_data	receive data byte
hw_isr	UART interrupt service routine
hw_rx_isr	receive interrupt service routine
hw_tx_isr	transmit interrupt service routine
hw_tmr_isr	timer interrupt service routine

Note

In the StrongArm SA1100 and SA1110 processor families, there are many different reference boards. The IRDA framer is not only specific to processor, but also to reference board. To make the framer work on a specific board, users may need to modify the code in the function `hw_iniz` located in the file `hw.c` and the macros defined in the file `port.def` located in the directory `mwos/os9000/armv4/ports/board_port_name`. Some GPIO and PPC registers may need to be configured to turn on the IRDA transceiver. Most of the GPIO and PPC registers are defined in `sa11x0.h` and `systype.h`. Users may need to modify these two files to avoid any duplicated definitions.

framer.c

The `framer.c` file contains all of the Framer's functions that are neither entry points (`entry.c`) or hardware configuration routines (`hw.c`). When adding functions to this file, name them according to the `frmr_function` convention. These routines typically maintain the logical structures within the Framer. **Table 6-3** provides an example.

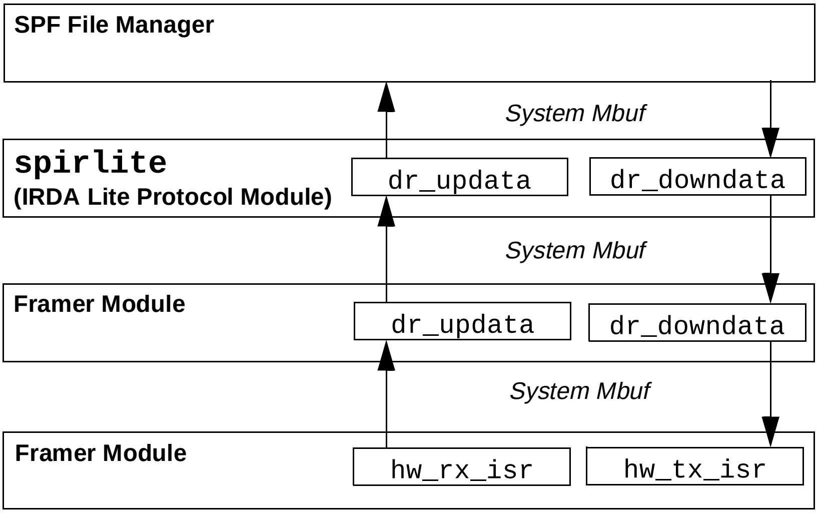
Table 6-3 framer.c Functions

Function	Description
`frmr_rdy_rxbuf`	ready the receive mbuf
`frmr_rdy_txbuf`	ready the transmit mbuf
`frmr_crc`	CRC calculations and checking
`frmr_rx_state`	set the receive state

Data Transfer/Flow

As Figure 6-4 illustrates, data is transferred between modules within the protocol stack through the dr_updata and dr_downdata entry points and as an interrupt service routine. The Framer's interrupt service routines hw_rx_isr and hw_tx_isr are located in hw.c.

Figure 6-4 dr_updata and dr_downdata Entry Points



In most interrupt-driven hardware drivers, an ISR routine—instead of the dr_updata entry point—is used to send data up to the next module within the protocol stack. In the Framer, however, the receive ISR—hw_rx_isr—passes all received data packets to the dr_updata entry point within the framer for additional processing. CRC checking is an example of additional processing that may be performed outside the interrupt context.

When a data packet is delivered upward through a hardware ISR function, data is passed to the SPF receive queue. Some period of time later, after the interrupt service routine, the receive thread of SPF wakes up and delivers the mbufs that are queued on the receive queue. To implement this behavior the DR_FMCALLUP_PKT is called within the hardware ISR.

The following three system mbufs must be passed up to SPF in this manner:

- transmitted mbufs
- received mbufs
- signal mbufs

Each of these mbufs is stored in the device-specific logical unit (lustat) variables `lu_txmbuf`, `lu_rxmbuf`, and `lu_sigmbuf`, respectively. Each of these calls is implemented into the Framer as follows.

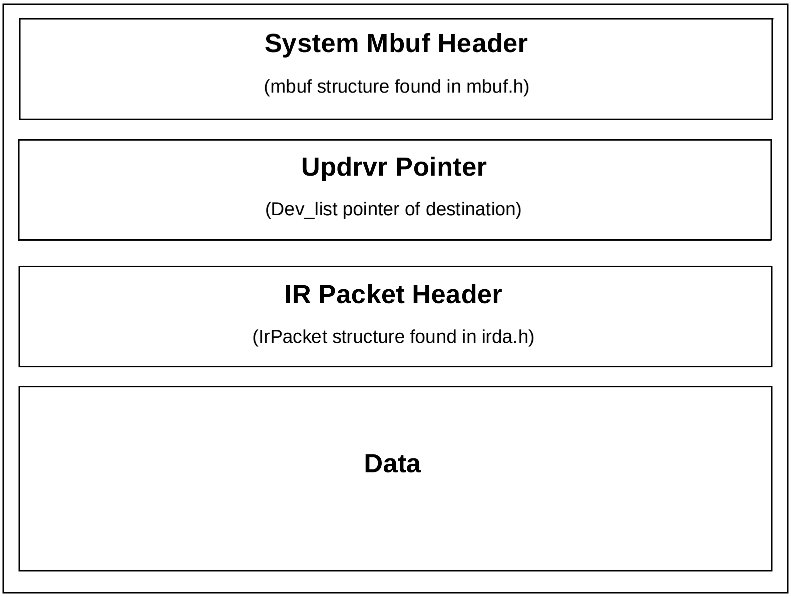
- `DR_FMCALLUP_PKT(deventry,lustat->lu_data_updrvr, Mbuf)&lustat->lu_sigmbuf);`
- `DR_FMCALLUP_PKT(deventry,lustat->lu_data_updrvr, lustat->lu_txmbuf);`
- `DR_FMCALLUP_PKT(deventry,deventry, lustat->lu_rxmbuf);`
- `SMCALL_UPDATA(deventry,lustat->lu_data_updrvr,mb;`

Notice that the transmit and signal mbufs calls reference `lu_data_updrvr`; the receive mbuf call references `deventry`; and in the dr_updata function, the `SMCALL_UPDATA` call references `lu_data_updrvr`. This ensures that the protocol module enabling the Framer receives all data and signal mbufs. In addition, all of the JetBeam for OS-9 system mbufs must have the `updrvr` pointer set in the mbuf itself. **Figure 6-5 JetBeam for OS-9 Mbuf**, shows the location of the `updrvr` pointer. SPF requires that this pointer be valid, otherwise system errors may result.

Framer Packet Structure

The JetBeam for OS-9 modules, which include the protocol module and framer module, use system mbufs to transfer data between each other. SPF requires a system mbuf header structure to be placed at the beginning of each mbuf passed between two modules. Like SPF, the JetBeam for OS-9 modules have a similar requirement. Within every mbuf following the system mbuf header structure is an IR packet header called IrPacket. This structure is used by the protocol stack to maintain per packet information. Framer updates this information while handling the mbuf. **Figure 6-5** outlines the system mbuf structure and orientation.

Figure 6-5 JetBeam for OS-9 Mbuf


System Mbuf Header
(mbuf structure found in mbuf.h)
Updrvr Pointer
(Dev_list pointer of destination)
IR Packet Header
(IrPacket structure found in irda.h)
Data

```
/*------------------------------------------------------
 *
 * Packet Structure for sending IrDA packets.
 */
typedef struct _IrPacket {
    /*===== The Following Are For Internal Use Only ======
    *
    * node
    * origin
    * header
    * headerLen
    *
    *==================================================*/

ListEntry node;
 u_int8*   buff;           /* Pointer to the buffer of data to send */
 Mbuf     mb;       /*Pointer to the mbuf that this structure is contained in */
 void*     origin;         /* Pointer to connection which owns the packet */
 u_int16   len;            /* Number of bytes in the buff */
 u_int8    type;           /* Type of the packet */
 u_int8    headerLen;      /* Number of bytes in the header */
 u_int8    more;           /* Flag for TTP SAR more bit */
 u_int8    bufspace[9];  /* space holders for 16 byte boundary */
 u_int8    header[14];     /* Storage for the header */
} IrPacket;
```

Index

Product Discrepancy Report

To: Microware Customer Support

FAX: 515-224-1352

From:____

Company:____________________________________

Phone:_______________________________________

Fax:___________________Email:_________________

Product Name:

Description of Problem:

____

____

____

____

____

____

____

Host Platform__________________________________

Target Platform________________________________

RadiSys.

MICROWARE SOFTWARE