# LIBST 2.0 API DESIGN

An API & design for portable software supporting low-latency operating system bypass over high-performance network topologies

Monika ten Bruggencate
George Lamb
Anthony Voellm
Jeff Howard
David Powell
Bryan Bush

V8.27.2001A
Copyright © 2001, Silicon Graphics, Inc.
Printed on Monday, August 27, 2001

# TABLE OF CONTENTS


1 INTRODUCTION ........ 3

2 LIBST API OBJECT MODEL ........ 4

3 LIBST API DETAILED DESIGN ........ 6
- 3.1 Interface Object ........ 6
- 3.2 Connector Object ........ 7
- 3.3 Bufxrange Object ........ 10
- 3.4 MX object ........ 11
- 3.5 Utility Routines ........ 12

4 SAMPLE PROGRAM ........ 14

5 LIBST.H ........ 17

6 CODING CONVENTIONS ........ 21
- 6.1 Error Handling Strategy ........ 21
- 6.2 Input Parameter Checking ........ 21

7 GLOSSARY ........ 22

8 REFERENCES ........ 23

# 1 INTRODUCTION

This document presents the design for libst 2.0, a software technology that provides low latency access to high-speed network hardware via the Schedule Transfer Protocol (STP). Libst achieves low latency by delivering a safe means to access network hardware while bypassing expensive operating system context switches.

The combination of Libst's operating system (OS) bypass approach with STP requires the user to manage both memory and network resources. Libst includes a set of objects that model the software components inside the operating system and the physical network devices. The need to carefully manage memory and network resources means libst client code is more complicated than traditional socket based codes that rely on the kernel for connection and memory management. However, there are significant latency benefits for using libst.

# 2 LIBST API OBJECT MODEL

The API object model, shown in Figure 1, is the interface for using libst. The implementation is object inspired where naming conventions and structures in C act as C++ methods and classes.

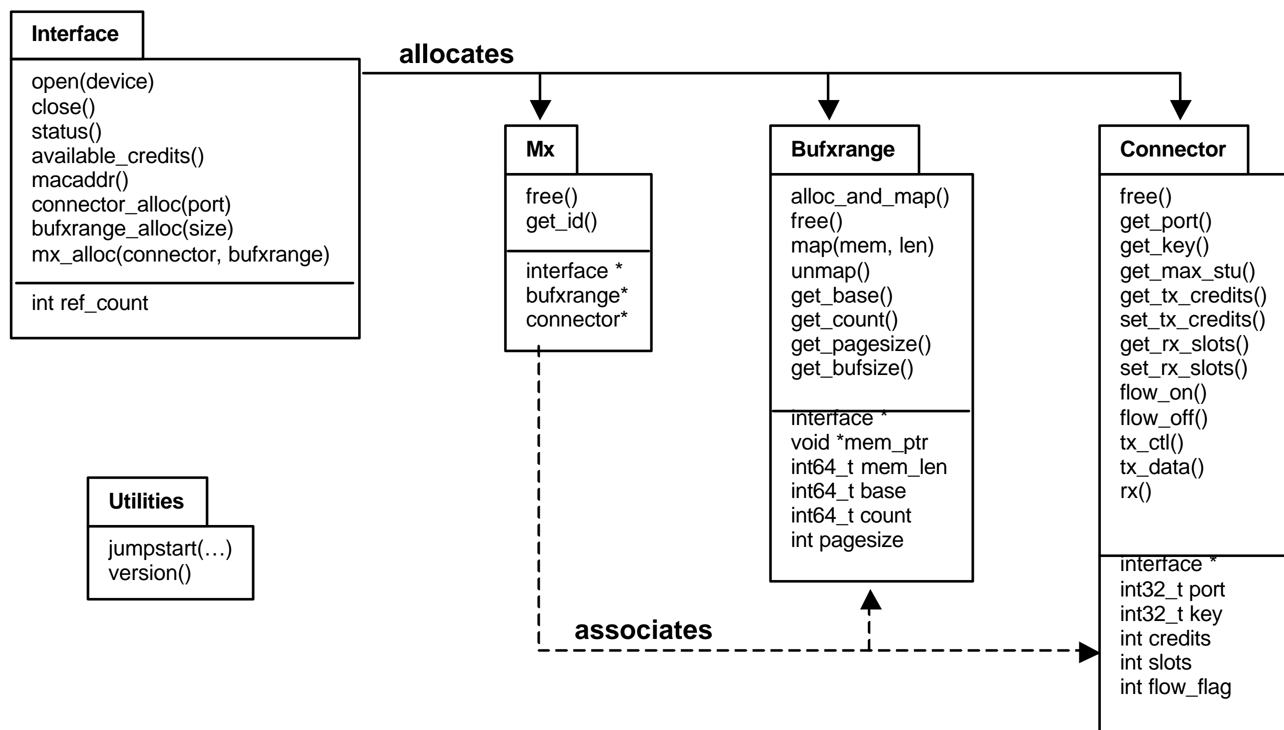


*Figure 1. API Object Mode*

*l*

The four objects are the Interface, Connector, Bufxrange, and Mx. The library also includes several utility routines.

An *interface* represents a physical device. It provides open and close methods. An application can open the same device multiple times, but all interfaces created refer to the same physical device. Similarly, different applications can open the same device and will be manipulating the same physical resource. A system may have multiple devices.

A *connector* is allocated from an interface. A connector represents a dedicated port on a physical device that can be used to communicate with other connectors. The total number of connectors that can be allocated across all applications on the system is limited to the number of physical ports on the devices.. Connectors are the primary objects used for communication.

A *bufxrange* is allocated from an interface. A bufxrange represents a range of bufx's, where a bufx is a reference to a page in memory. Bufx's are a physical resource assigned to a device by the kernel. A user allocated memory region can be used for libst transmit and receive operations after it has been mapped to a bufxrange. Calls to allocate and manipulate a bufxrange require kernel intervention.

An *mx* is allocated from an interface to associate a connector to a bufxrange. An mx is necessary to correctly route incoming messages on an ST port (connector) to allocated memory (bufxrange.) Mx's are also used for authentication.

These four objects provide the infrastructure to access kernel services and physical devices. The API assumes that the user will properly allocate and align memory before mapping it to a bufxrange. It also assumes that the user understands the ST protocol and setting up ST headers for data transfers.

# 3 LIBST API DETAILED DESIGN

This section presents the detailed design for the Libst API. It includes the methods for each object along with the algorithms and design decisions. The structure definitions are device specific and can only be referenced through an opaque (void *) structure. The definitions of the structures are left to the implementer. Each parameter used for setting up and communicating schedule transfer information is explicitly sized using ctype.h definitions.

## 3.1 INTERFACE OBJECT

The interface object models a physical device. From a developer's perspective the interface is a combination of a device file and a network interface. The interface design must address the following issues:

1. Specifying and opening a device. There are several ways to describe the interface:

- node in the hardware graph: /hw/gsn/0/device
- device file: /dev/gsn0
- configured interface: gsn0, gsn1
- network address associated with an interface: quartz-g0

2. Managing resources: The interface is designed as a factory object that is responsible for allocating connectors, bufxranges, and mx's. Each of these is associated with a kernel resource. Once the interface is closed, the kernel will automatically free all associated resources, regardless of whether the user intended them to be freed. The interface needs to track kernel resource allocation and delay freeing them until the user explicitly frees them. The chosen design is to implement reference counting as follows:

```
on connector_alloc, bufxrange_alloc, or mx_alloc -> ref_ct++
on connector_free, bufxrange_alloc, or mx_free -> ref_cnt--
on close -> close_request = true
on close, connector_free, bufxrange_free, or mx_free
   if ref_cnt == 0 and close_request == true, then close
```

### 3.1.1 INTERFACE STRUCTURE

The interface is an opaque type:

```
typedef void * st_interface_t;
```

### 3.1.2 INTERFACE MANAGEMENT

The interface object provides the following management functions:

- st_interface_t st_open (char *interface)

  This is the interface constructor. It returns an interface or NULL on error. The interface string is a node in the hardware graph, a device file, a network interface, or a network address.

- int st_close (st_interface_t interface)

  This is the interface destructor. It sets the close_request flag and checks the reference count. If the reference count is zero, it unmaps the device, and closes the fd.

- int st_interface_status (st_interface_t interface)

  This returns the status of the interface: ST_INTERFACE_ERROR or ST_INTERFACE_USABLE.

- int st_interface_available_credits(st_interface_t interface, unit16_t num_credits[ST_MAX_VC])

  This returns the number of tx credits available in the general tx credit pool on the interface for virtual channels (VC's) 0 through ST_MAX_VC - 1. The numbers are obtained at a particular instant and may change at any time as connectors are being allocated, and in turn take tx credits out of the general pool, are being freed and in turn put tx credits back into the general pool or as the tx credits assigned to a connector change via calls to st_connector_set_tx_credits().

- st_macaddr_t st_macaddr (st_interface_t interface, const char *hostname);

  This returns the MAC address of a host. It uses gethostbyname to find the host address then uses HARP (or some other IP to ULA mapping protocol) to determine the MAC address. The MAC address of a host must be known to transmit ST messages.

  - int st_interface_rx_alignment(st_interface_t interface);

  This returns the required transmit alignment value for the interface.

  - int st_interface_tx_alignment(st_interface_t interface);

  This returns the required receive alignment value for the interface.

### 3.1.3 Interface Factory Methods

The interface object provides the following factory methods. Each method will acquire the appropriate kernel resources and return an initialized libst object on success or NULL on error.

- st_connector_t st_connector_alloc (st_interface_t interface, int *requested_hw_port);

  This allocates and returns a connector or NULL. The requested hardware port must be in the range [11, 4096] or ST_ANY_PORT in which case the assigned port is returned.

- st_bufxrange_t st_bufxrange_alloc (st_interface_t interface, int num_bufxs);

  This allocates and returns a bufxrange or NULL. The number of bufx's, num_bufxs, in the bufxrange must be sufficient to span the memory region to later be mapped to the bufxrange. Bufx's can refer to various size pages. All pages in a bufxrange must be of the same size.

- st_mx_t st_mx_alloc (st_interface_t interface, st_connector_t connector, st_bufxrange_t bufxrange);

  This allocates and returns an mx or NULL. The connector and bufxrange must have been allocated from the interface. Note that if the bufxrange or connector is freed prior to freeing the mx, the mx becomes invalid. If an mx is successfully allocated, it contains a valid id that can be inserted into an ST header.

## 3.2 Connector Object

The connector object models a dedicated hardware port on an interface. Physical devices typically have a fixed number of communication ports that can be shared by applications. Once a connector is allocated, it can be used to send and receive ST messages to and from other connectors. The connector design must address the following issues:

1. ST header setup: the connector is used to obtain information from the kernel necessary to construct an ST header. When the connector is allocated it will prefetch the parameters necessary for setting up schedule transfers and save the latest settings.

2. Data transfers: the connector contains the logic for doing DMA's to and from the device and for managing flow control. The specifics of the flow control logic are dependent on the capabilities of the underlying device. Libst over GSN provides transmit flow control, but it does not provide receive flow control.
3. ST tiling: The connector needs to have sufficient information for tiling. This information is determined when a bufxrange is allocated and exchanged between connectors at setup time by the jumpstart routine. Libst should not make any assumptions about page size and Scheduled Transfer Unit (STU) size. Therefore page size and STU size must be passed to the st_tx_data routine.

### 3.2.1 CONNECTOR STRUCTURE

The connector is an opaque type:

```
typedef void * st_connector_t;
```

### 3.2.2 CONNECTOR MANAGEMENT

The connector object provides the management function:

- int st_connector_free (st_connector_t connector);

  This is the connector destructor. It frees the connector and clears the associated port in the hardware. It also checks the reference count on the interface. If the reference count is zero, it unmaps the device, and closes the fd.

### 3.2.3 CONNECTOR QUERY AND SET ROUTINES

A connector is used to maintain information necessary for setting up ST headers. The following functions return the requested information or those methods which can fail return a negative value on error. The port, key, and maximum STU values all must be exchanged between pairs of communicating connectors prior to starting ST transfers. The st_jumpstart routine is available to exchange these values using another network protocol.

- uint16_t st_connector_get_port (st_connector_t connector);

  This returns the hardware port associated with the connector. The port number must be exchanged with another connector prior to sending or receiving ST messages.

- uint32_t st_connector_get_key (st_connector_t connector);

  This returns the key for a connector. The key is used to certify transmission and reception made on an ST port. It must be exchanged with another connector prior to sending or receiving ST messages.

- uint_t st_connector_get_max_stu (st_connector_t connector);

  This returns the max scheduled transfer unit size in log bytes that can be transferred or received by the connector. It must be exchanged with another connector prior to sending or receiving ST messages.

- uint_t st_connector_get_bufsize(st_connector_t connector);

  This returns the bufsize of the last bufxrange transmitted from by the connector.

- int st_connector_set_src_bufsize(st_connector_t connector, uint_t src_bufsize);

This stores the transmit (or source) bufsize src_bufsize in the connector. The correct source bufsize has to be set in the connector before a message can be transmitted.

- int st_connector_set_bufxrange_and_src_bufsize (st_connector_t connector, st_bufxrange_t bufxrange);
- This sets the bufxrange and source bufsize in the connector. The bufxrange from which a connector is transmitting along with the bufsize of the bufxrange must be set in the connector before the connector starts to transmit messages from the bufxrange. The bufxrange must be mapped for transmit.
- int st_connector_get_tx_credits (st_connector_t connector,
  uint16_t num_tx_credits[ST_MAX_VC]);

  This retrieves the number of tx credits assigned to the connector for each virtual channel. VC 2 is primarily used by libst. There are a fixed number of tx credits per interface and per virtual channel. Every application that opens an interface and allocates connectors will contend for tx credits.

- int st_connector_set_tx_credits (st_connector_t connector,
  uint16_t num_tx_credits[ST_MAX_VC]);

  This sets the number of tx credits assigned to the connector for each virtual channel.

- uint32_t st_connector_get_rx_slots (st_connector_t connector);

  This returns the number of slots assigned to the connector for receive. The number of rx slots is shared across all virtual channels.

- int st_connector_set_rx_slots (st_connector_t connector,
  uint32_t num_rx_slots);

  This sets the number of slots assigned to the connector for receive.

- int st_connector_flow_on (st_connector_t connector);

  This turns transmit flow control on (on by default). Without transmit flow control, the user can exceed the physical limits on tx credits and have transmissions dropped.

- int st_connector_flow_off (st_connector_t connector);

  This turns transmit flow control off.

### 3.2.4 CONNECTOR TX AND RX ROUTINES

A connector is used to transmit and receive ST headers and data. The following communication functions are available:

- int st_tx_ctl (st_connector_t connector, st_macaddr_t macaddr,
  st_header_t *hdr, uchar_t opt_payload[OPT_PAYLOAD_SIZE]);

  This transmits the control message encoded in hdr from the connector to the given MAC address. If opt_payload is non-NULL it must point to a buffer from which the first 32 bytes are transmitted as option payload along with the control message.

- int st_tx_data(st_connector_t connector, st_macaddr_t macaddr,
  st_header_t *hdr, st_bufxrange_t bufxrange, uint64_t len,
  uint32_t offset, uint_t stusize, uint_t rem_bufsize);

  This transmits the data message encoded in hdr from the connector to the given MAC address. The data to be transmitted is mapped by bufxrange and its length in bytes is specified by len, starting from the offset in the buffer. Rem_bufsize is the bufsize (pagesize) of the mapped buffer

on the remote connector. Stusize is the STU size expected at the remote connector. The ST tiling algorithm needs both of these parameters when sending messages.

- int st_rx(st_connector_t connector, st_macaddr_t *macaddr,
  st_header_t *hdr, uchar_t opt_payload[OPT_PAYLOAD_SIZE],
  struct timeval *timeout);

  This retrieves the next ST message received by the connector. This can be either a data or a control message. The MAC address, ST header, and option payload parameters are filled in with the appropriate information from the received message. Macaddr and opt_payload can be NULL, causing the corresponding data to not be extracted from the received message. If no message is available, st_rx blocks until the amount of time specified by timeout has elapsed or, if timeout is NULL.

## 3.3 BUFXRANGE OBJECT

The bufxrange object models and manages the memory used in ST transfers. The user allocates a bufxrange and maps a user data buffer from which or into which data will transmitted or received. The bufxrange prepares the user data buffer for ST transfers.

The user application steps required to prepare a buffer are:

1. allocate the memory
2. align the memory
3. optionally mpin() the memory
4. allocate enough bufx's from the interface to span the memory
5. associate the memory with the bufxrange
6. for receive buffers, allocate an mx to associate the connector with the bufxrange

Following a series of transfers, these steps return resources to the kernel:

7. free the MX
8. disassociate the bufxrange
9. free the bufxrange
10. if mpin()'d in #3, munpin() the memory
11. free the memory

The user is responsible for steps 1, 2, and 11. The bufxrange object supports steps 3, 4, 5 and 8, 9, and 10. The mx object supports steps 6 and 7.

### 3.3.1 BUFXRANGE STRUCTURE

The bufxrange is an opaque type:

```
typedef void * st_bufxrange_t;
```

### 3.3.2 BUFXRANGE MANAGEMENT

The bufxrange object provides the following management functions:

- st_bufxrange_t st_bufxrange_alloc_and_map ( st_interface_t interface, int rw_flags,
  void *mem_ptr, uint64_t mem_len, uint_t bufsize);

  This combines bufxrange allocation with mapping the bufxrange to the memory region specified by its address mem_ptr and length in bytes mem_len. The bufxrange is mapped in chunks of

bufsize and is mapped for transmit or receive or both depending on rw_flags (ST_BUFX_MAP_TX or ST_BUFX_MAP_RX or ST_BUFX_MAP_TX | ST_BUFX_MAP_RX, respectively). This function is slightly less efficient than executing the allocation and map calls separately but has the advantage of computing and allocating the minimum number of bufx's to span the memory region. This prevents the bufxrange from being reused with a larger memory region at a later time.

- int st_bufxrange_free(st_bufxrange_t bufxrange);
  This frees the bufxrange.

### 3.3.3 BUFXRANGE QUERY ROUTINES

The following functions obtain certain information about a bufxrange. This information is necessary for setting up ST data transfers.

- uint32_t st_bufxrange_get_base (st_bufxrange_t bufxrange);
  This returns the base bufx, i.e. first bufx, of the bufxrange.
- uint_t st_bufxrange_get_count (st_bufxrange_t bufxrange);
  This returns the number of bufx's in the bufxrange. Each bufx corresponds to a page of memory.
- uint_t st_bufxrange_get_pagesize (st_bufxrange_t bufxrange);
  This returns the size in log bytes of each page mapped by the bufxrange.
- uint_t st_bufxrange_get_bufsize (st_bufxrange_t bufxrange);
  This returns the size in log bytes of the bufsize of each bufx in the bufxrange.

### 3.3.4 BUFXRANGE MAPPING ROUTINES

These functions map and unmap a bufxrange.

- int st_bufxrange_map (st_bufxrange_t bufxrange, int rw_flags,
  void *mem_ptr, uint64_t mem_len, uint_t bufsize);
  This maps a user memory region specified by its address mem_ptr and length in bytes mem_len into a bufxrange. The bufxrange must have sufficient bufx's to span the memory region. The page will be mapped according to the following rx_flags

```
ST_BUFX_MAP_RX
ST_BUFX_MAP_TX
(ST_BUFX_MAP_RX | ST_BUFX_MAP_TX)
```

  and the given bufsize.
- int st_bufxrange_unmap (st_bufxrange_t bufxrange);
  This unmaps a bufxrange. The bufxrange can then be remapped to a different buffer.

## 3.4 MX OBJECT

An mx associates a bufxrange mapped for receive with a connector which receives from the bufxrange. The local host A will allocate an mx for the bufxrange, and then pass the mx to a remote host B (typically via an ST control message) from which it expects to receive ST transfers at the connector. The remote

host B will set the mx in those ST data headers destined for the connector and bufxrange on host A to identify where the data is to be received. This allows ST to write directly into user allocated buffers.

### 3.4.1 MX STRUCTURE

The mx is an opaque type:

```
typedef void *st_mx_t;
```

### 3.4.2 MX MANAGEMENT ROUTINES

The mx supports the following methods:

- int st_mx_free (st_mx_t mx);
  This frees an mx and disassociates it from the bufxrange and connector.
- uint16_t st_mx_get_id(st_mx_t mx);
  This returns the id of the mx if the mx is valid and ST_MX_INVALID otherwise. The id can be inserted into an ST header.

## 3.5 UTILITY ROUTINES

Libst includes several utility functions that can be used to query the library and to simplify using it.

### 3.5.1 JUMPSTART

The jumpstart function provides a single call to exchange the minimal information necessary for ST communications. It's used in client-server programs that have prepared a connector and mapped a bufxrange. Jumpstart will open a socket to exchange the information. Parameters being exchanged are transformed into network byte order prior to transmission.

- int st_jumpstart (int protocol, char *host, int ulp_port,
  ```
                      uint16_t loc_port,     uint32_t loc_key,
                      uint_t loc_bufsize,  uint_t loc_stusize,
                      uint16_t *rem_port,    uint32_t *rem_key,
                      uint_t *rem_bufsize, uint_t *rem_stusize);
  ```
  Protocol is the socket type desired for jumpstarting. SOCK_STREAM is typical. Host is a hostname or IP address for the server side. The server should set host to NULL. ULP_port is an upper level protocol port selected by the user. The server side will bind to the ulp_port and the client side will connect to it.

  Port and key parameters are required for control and data messages. They can be extracted from the connector.

  Bufsize and stusize are required for data communications. The max stusize is determined from the connector. A smaller stusize can be used if desired, so long as the device supports it. Bufsize is the page size of the memory region that was mapped into a bufxrange. It can be determined from the bufxrange.

  Both the client and server must call jumpstart, the client should be delayed to allow the server socket to bind before calling jumpstart.

### 3.5.2 VERSION

The version function returns a string describing the libst version.

- const char *st_version (void);

  The string returned is of the form:

  SGI ST Bypass Library (N32_M4)    compiled_date    compiled_time

# 4 SAMPLE PROGRAM

```
/* Link: cc libst_example.c -lst -o libst_example         */
/* run: libst_example -c&; sleep 1; libst_example <host>  */
/* Output: Received (Hello libst!)                        */

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <libst.h>

#define ST_VC_0                  0
#define ST_VC_2                  2
#define LOGBUFSIZE              14
#define BUFLEN     (1<<LOGBUFSIZE) /* 16k */

int main(int argc, char *argv[])
{
    st_interface_t interface;
    st_connector_t connector;
    st_bufxrange_t bufxrange;
    st_mx_t mx;
    sthdr_data_t datahdr;
    sthdr_rts_t rtshdr;
    sthdr_cts_t ctshdr;
    st_macaddr_t macaddr;
    char consumer, *buffer, *hostname, *message = "Hello libst!";
    int port, rc;
    uint16_t remote_port;
    uint32_t remote_key;
    uint_t remote_bufsize, remote_stusize;
    uchar_t opt_payload[ST_OPT_PAYLOAD_SIZE];

    /* not checking return codes to make the example simple */

    consumer = (argc > 1) && (strcmp(argv[1],"-c")==0);

    if (consumer) {
      port = 500;
      hostname = NULL;
    } else {
      port = 501;
      hostname = argv[1];
    }

    interface = st_open("gsn0");
    connector = st_connector_alloc(interface, &port);

    /* request connection ... */

    st_jumpstart(SOCK_STREAM, hostname, 4500,
                 st_connector_get_port(connector),
                 st_connector_get_key(connector),
                 LOGBUFSIZE, LOGBUFSIZE,
                 &remote_port, &remote_key,
                 &remote_bufsize, &remote_stusize);
```

```
/* allocate and map BUFXs */

buffer = memalign(BUFLEN, BUFLEN);
bufxrange = st_bufxrange_alloc_and_map(interface,
                                       (ST_BUFX_MAP_RX | ST_BUFX_MAP_TX),
                                       buffer, BUFLEN, 0);

/* restrict the bufxrange legal for tx to what is really needed */

rc = st_connector_set_bufxrange_and_src_bufsize(connector, bufxrange);
if (rc < 0) {
    printf("Failed to set bufxrange for tx.\n");
    return -1;
}


if (consumer) {
  /* Consumer */
  mx = st_mx_alloc(interface, connector, bufxrange);

  /* Receive RTS */
  st_rx(connector, &macaddr, (st_header_t *)&rtshdr, NULL, NULL);

  ctshdr.OpFlags = ST_CTS | ST_VC_0;
  ctshdr.Blocksize = LOGBUFSIZE;
  ctshdr.I_Port = remote_port;
  ctshdr.R_Port = st_connector_get_port(connector);
  ctshdr.I_Key = remote_key;
  ctshdr.R_Mx = st_mx_get_id(mx);
  ctshdr.R_Bufx = st_bufxrange_get_base(bufxrange);
  ctshdr.R_Offset = 0;
  ctshdr.B_num = 0;
  ctshdr.I_id = rtshdr.I_id;
  ctshdr.R_id = 29;

  /* Send CTS */
  st_tx_ctl(connector, macaddr, (st_header_t *)&ctshdr, opt_payload);

  /* Receive DATA */
  st_rx(connector, &macaddr, (st_header_t *)&datahdr, NULL, NULL);

  buffer[strlen(message)] = '\0';
  printf("Received (%s)\n", buffer);
} else {
  /* Producer */

  macaddr = st_macaddr(interface, hostname);

  strcpy(buffer, message);

  /* send RTS */
  rtshdr.OpFlags = ST_RTS | ST_VC_0;
  rtshdr.CTS_req = 1; /* only one outstanding CTS at a time */
  rtshdr.R_Port = remote_port;
  rtshdr.I_Port = st_connector_get_port(connector);
  rtshdr.R_Key = remote_key;
  rtshdr.Max_Block = LOGBUFSIZE;
  rtshdr.tlen = BUFLEN;
```

```
        rtshdr.I_id = 28;

        st_tx_ctl(connector, macaddr, (st_header_t *)&rtshdr, opt_payload);

        /* Receive CTS */
        st_rx(connector, &macaddr, (st_header_t *)&ctshdr, NULL, NULL);

        /* Send Data */
        datahdr.OpFlags = ST_DATA | ST_LAST | ST_VC_2;
        datahdr.STU_num = 0;
        datahdr.R_Port = remote_port;
        datahdr.I_Port = st_connector_get_port(connector);
        datahdr.R_Key = remote_key;
        datahdr.R_Mx = ctshdr.R_Mx;
        datahdr.R_Bufx = ctshdr.R_Bufx;
        datahdr.R_Offset = ctshdr.R_Offset;
        datahdr.Sync = 0;
        datahdr.B_num = ctshdr.B_num;
        datahdr.R_id = ctshdr.R_id;

        st_tx_data(connector, macaddr, (st_header_t *)&datahdr,
                 bufxrange, strlen(message), 0, LOGBUFSIZE, LOGBUFSIZE);
    }

    return 0;
}
```

# 5 LIBST.H

```
/***************************************************************************
 *                                                                         *
 *               Copyright (C) 2001, Silicon Graphics, Inc                 *
 *                                                                         *
 *  These coded instructions, statements, and computer programs  contain  *
 *  unpublished  proprietary  information of Silicon Graphics, Inc., and  *
 *  are protected by Federal copyright law.  They  may  not be disclosed  *
 *  to  third  parties  or copied or duplicated in any form, in whole or  *
 *  in part, without the prior written consent of Silicon Graphics, Inc.  *
 *                                                                         *
 ***************************************************************************/

#ifndef _LIBST_H
#define _LIBST_H

#ifdef __cplusplus
extern "C" {
#endif


#include <sys/time.h>
#include <netinet/st.h>


/* The following codes and structs from st.h are used:
 *  ST Opcodes
 *  ST Bit Flags
 *  ST Bufx Mapping Flags (for TX or RX in map, unmap, mx_alloc, mx_free)
 *  sthdr_general_t
 */

/* Opaque object definitions and typedefs */

typedef void * st_interface_t;       /* a device interface       */
typedef void * st_connector_t;       /* a communication port     */
typedef void * st_bufxrange_t;       /* a bufx range             */
typedef void * st_mx_t;               /* an mx ties a connector to bufxrange for rx */
typedef uint64_t  st_macaddr_t;

typedef sthdr_general_t  st_header_t;

#define ST_ANY_PORT            -1

#define ST_OPT_PAYLOAD_SIZE     32

/* Interface status return codes
 */
#define ST_INTERFACE_ERROR       1
#define ST_INTERFACE_USABLE      2

/* MX invalid id
 */
#define ST_MX_INVALID   ((uint16_t)(-1))

/* Receive function return codes
   The defined constants must have positive values since they express
   successful completion of the receive function.
 */
#define ST_OPTION_PAYLOAD_RECEIVED    1
#define ST_APPEND_DATA_RECEIVED       2

/* Interface management
 *
 *  An interface connects to a physical device and manages the
 *  resources available on that device.  Each connector or bufxrange
 *  allocated by the interface is reference counted, so the interface
 *  will not be closed until all allocated resources are freed.
 */
st_interface_t st_open(char *interface_name);   /* /dev, address, or ifname */
```

```
void      st_close(st_interface_t interface);
int       st_interface_status(st_interface_t interface);
int       st_interface_available_credits(st_interface_t interface,
                                  uint16_t num_tx_credits[ST_MAX_VC]);
int       st_interface_rx_alignment(st_interface_t interface);
int       st_interface_tx_alignment(st_interface_t interface);

st_macaddr_t  st_macaddr(st_interface_t interface, const char *hostname);

/* Connector management
 *
 *  A connector is analagous to a specific port on an interface used
 *  for communication.  It's used to transmit and recieve to/from
 *  bufxranges
 */
st_connector_t  st_connector_alloc(st_interface_t interface,
                                  int *requested_hw_port);
void      st_connector_free(st_connector_t connector);

/* Connector query routines--all return the values or < 0 on error */
uint16_t  st_connector_get_port(st_connector_t connector);
uint32_t  st_connector_get_key(st_connector_t connector);

          /* the log bytes of the max stu that can be transmitted */
uint_t    st_connector_get_max_stu(st_connector_t connector);

          /* the log bytes of the bufsize from the last data tx */
uint_t    st_connector_get_bufsize(st_connector_t connector);

int       st_connector_get_tx_credits(st_connector_t connector,
                                  uint16_t num_tx_credits[ST_MAX_VC]);
int       st_connector_set_tx_credits(st_connector_t connector,
                                  uint16_t num_tx_credits[ST_MAX_VC]);

uint32_t  st_connector_get_rx_slots(st_connector_t connector);
int       st_connector_set_rx_slots(st_connector_t connector,
                                  uint32_t num_rx_slots);

int       st_connector_set_src_bufsize(st_connector_t connector, uint_t src_bufsize);
int       st_connector_set_bufxrange_and_src_bufsize(st_connector_t connector,
                                                  st_bufxrange_t bufxrange);

          /* flow control is on by default */
int       st_connector_flow_on(st_connector_t connector);
int       st_connector_flow_off(st_connector_t connector);


/* Bufx range management
 *
 *  A bufx range is a list of bufx's managed by an interface.  Each
 *  bufx corresponds to a pointer to a page (or partial page if
 *  bufsizes are used) in physical memory.  The bufx range must be
 *  large enough to span the memory set up for transmission by libst.
 *  The library can automatically calculate the necessary size, or the
 *  user can request a specific number.  This is typically done when
 *  the user will map several different regions with the same bufx
 *  range.
 *
 *  Note: if bufsize = 0, then the memory pagesize will be used
 */

st_bufxrange_t  st_bufxrange_alloc(st_interface_t interface, int num_bufxs);
st_bufxrange_t  st_bufxrange_alloc_and_map(st_interface_t interface,
                                         int rw_flags,  void *mem_ptr,
                                         uint64_t mem_len, uint_t bufsize);
void      st_bufxrange_free(st_bufxrange_t bufxrange);

/* Bufx query routines--all return the values or < 0 on error */
uint32_t  st_bufxrange_get_base (st_bufxrange_t bufxrange);     /* kernel index of 1st bufx */
uint_t    st_bufxrange_get_count(st_bufxrange_t bufxrange);     /* number of bufx's */
uint_t    st_bufxrange_get_pagesize (st_bufxrange_t bufxrange); /* pagesize (log bytes) */
uint_t    st_bufxrange_get_bufsize (st_bufxrange_t bufxrange);  /* bufsize (log bytes) */
```

```
/* Memory mapping
 *
 *  Memory regions are mapped to bufxes prior to transmission.  The
 *  bufx range must be large enough to span the memory region.  A
 *  memory region can be mapped:
 *    ST_BUFX_MAP_RX, ST_BUFX_MAP_TX, or (ST_BUFX_MAP_RX | ST_BUFX_MAP_TX)
 *
 *  Note: if bufsize = 0, then the bufxrange will map the memory on pages
 */
int       st_bufxrange_map(st_bufxrange_t bufxrange,  int rw_flags,
                           void *mem_ptr,  uint64_t mem_len,
                           uint_t bufsize);
int       st_bufxrange_unmap(st_bufxrange_t bufxrange);

/* MX allocation
 *
 *  An mx identifies a connector with a bufx range.  Note that both the
 *  connector and bufx must have been allocated from the same interface
 */
st_mx_t  st_mx_alloc(st_interface_t interface,
                     st_connector_t connector,
                     st_bufxrange_t bufxrange);
void     st_mx_free(st_mx_t mx);
uint16_t st_mx_get_id(st_mx_t mx);

/* Communication routines
 *
 */
int    st_tx_ctl (st_connector_t connector,  st_macaddr_t macaddr,
                  st_header_t *hdr, uchar_t opt_payload[ST_OPT_PAYLOAD_SIZE]);

int    st_tx_data(st_connector_t connector,  st_macaddr_t macaddr,
                  st_header_t *hdr,  st_bufxrange_t bufxrange,
                  uint64_t len,      uint32_t offset,
                  uint_t stusize,  uint_t rem_bufsize);

int    st_rx(st_connector_t connector, st_macaddr_t *macaddr, st_header_t *hdr,
             uchar_t opt_payload[ST_OPT_PAYLOAD_SIZE],
             struct timespec *timeout);

/* Utility Routines
 *
 * Jumpstart routine
 *  This will jumpstart a connection by using a socket to exchange the
 *  minimal information necessary to use libst.  Both the client and
 *  server must call jumpstart.  Note: set host = NULL for the server
 * side, and delay the client before jumpstarting
 */
int st_jumpstart(int type, char *host, int ulp_port,
                uint16_t loc_port,   uint32_t loc_key,
                uint_t loc_bufsize,  uint_t loc_stusize,
                uint16_t *rem_port,  uint32_t *rem_key,
                uint_t *rem_bufsize, uint_t *rem_stusize);

const char *st_version(void);

/* ST Header manipulation routines
 * These are used to manipulate bit fields in the st_header OpFlags
 *    opcode  is 5 bits 11-15
 *    func    is 3 bits 8-10:
 *    flags   is 6 bits 2-7:
 *    channel is 2 bits 0-1:
 * Sample usage to set the bit fields:
 *     ST_HDR_SET_OPCODE(&hdr, ST_OP_NOP);
 *     ST_HDR_SET_FLAGS(&hdr, ST_FLAG_LAST);
 * Sample usage to get the bit fields, note that opcode is being assigned:
 *     ST_HDR_GET_OPCODE(hdr, opcode);
 *
 *
 * The following macros set/get a bit field in/from an ST header.
 * The parameters used are:
 *    val: a value to be assigned to the bit field/filled in from the
```

```
 *         bit field
 *    mask: a 16 bit mask set with 1s for the length of the field
 *          example: a 5 bit field is 0x001f = 0000 0000 0001 1111
 *    shift: the start (rightmost) position of the field within the header
 *    leftbit: the end (leftmost) bit of the field within the header
 *    len: the length of the field
 */

/* ST_HDR_CLEAR_BITS uses mask to clear bits from the field starting at shift */

#define ST_HDR_CLEAR_BITS(hdr, mask, shift)  ((hdr)->OpFlags) &= ~( mask << shift)

/* ST_HDR_SET_BITS will 'or' the value into the field starting at shift */

#define ST_HDR_SET_BITS(hdr, val, shift) ((hdr)->OpFlags) |= (val << shift)

/* ST_HDR_SET_OPFLAG calls CLEAR to nix the bitfield, then uses SET */

#define ST_HDR_SET_OPFLAG(hdr, val, mask, shift)  { \
                   ST_HDR_CLEAR_BITS(hdr, mask, shift); \
                   ST_HDR_SET_BITS(hdr, (val & mask), shift); }

/* ST_HDR_GET_OPFLAG does a left shift/right shift to get val, temp is
 * used in case val is larger than 16 bits.
 */
#define ST_HDR_GET_OPFLAG(hdr, val, leftbit, len) {  \
                   uint16_t temp = (hdr)->OpFlags << (15 - (leftbit)); \
                   val = temp >> (16 - (len)); }

/* ST_HDR_SET_XXX define the mask and shift for bitfield XXX and set
 * it to the specified value.
 */
#define ST_HDR_SET_OPCODE(hdr, opcode)   ST_HDR_SET_OPFLAG(hdr, opcode, 0x001f, 11)
#define ST_HDR_SET_FUNC(hdr, func)       ST_HDR_SET_OPFLAG(hdr, func, 0x0007, 8)
#define ST_HDR_SET_FLAGS(hdr, flags)     ST_HDR_SET_OPFLAG(hdr, flags, 0x003f, 2)
#define ST_HDR_SET_CHANNEL(hdr, channel) ST_HDR_SET_OPFLAG(hdr, channel, 0x0003, 0)

/* ST_HDR_GET_XXX define the leftbit and shift for bitfield XXX and
 * save it's value into val.
 */
#define ST_HDR_GET_OPCODE(hdr, val)   ST_HDR_GET_OPFLAG(hdr, val, 15, 5)
#define ST_HDR_GET_FUNC(hdr, val)     ST_HDR_GET_OPFLAG(hdr, val, 10, 3)
#define ST_HDR_GET_FLAGS(hdr, val)    ST_HDR_GET_OPFLAG(hdr, val,  7, 6)
#define ST_HDR_GET_CHANNEL(hdr, val)  ST_HDR_GET_OPFLAG(hdr, val,  1, 2)


#ifdef __cplusplus
}
#endif


#endif /* _LIBST_H */
```

# 6 CODING CONVENTIONS

The following coding conventions will be standard across the libst source code.

## 6.1 ERROR HANDLING STRATEGY

Error cases will be dealt with according to the following guidelines wherever possible.

1. The library should never call exit(), i.e. the use of exit() should be constrained to the application. The motivation is that exits are difficult to trace when the user does not have access to the source code and needs to rely on error returns while debugging. Also, exits leave the application no chance to recover, error returns do.
2. If there exists a reasonable possibility that a given error may occur, provisions should be made for its interception or handling. Error processing should not cause uncontrolled side-effects. An error caused by erroneous user input should be reported to the user in a manner which fully explains the error.
3. Error returns from functions called within the library will be checked even if the callee functions seem foolproof.
4. A standard format for processing errors will be defined. The standard format should allow for as much detail as necessary to uniquely identify the source of the error without references to external sources.
5. Resources need to be released properly in all error cases.
6. Cookies are clobbered upon freeing the corresponding handle.
7. Errors will return –<ERROR> where <ERROR> is a UNIX error number as defined in error.h

Areas which may require particular attention include making direct write of tx descriptors safer by using the user FIFO tails if possible and by checking function parameters and values in the descriptors.

## 6.2 INPUT PARAMETER CHECKING

Functions which the user has access to, and thus may provide illegal input parameter values to, and which are called infrequently only, e.g. for setup, implement input parameter checking for avoid core dumps etc.

# 7 GLOSSARY

**bufx** Buffer index. An OS-independent, 32-bit parameter identifying the starting address of a data buffer.

**bufsize** The pagesize of an ST buffer. >= 256 bytes <= 2^32.

**tx credit** A credit corresponds to one micropacket's worth of buffer space available in the destination's virtual channel (VC) buffer.

**initiator** The end device that starts a sequence of operations. This is typically a host system, but may also be a non-transparent translator, bridge or router.

**mx** Memory Index, a parameter identifying an area of memory. It associates a bufxrange with a port.

**offset** The data's starting point relative to the start of a Bufx or transfer.

**MAC** Media Access Control.

**option payload** The option payload is 32 bytes of user payload within an STP connection header.

**operation** The procedure defined by the parameters in a Schedule Header, and any payload associated with that Schedule Header. The code in the Schedule Header's "Op" field identifies the operation's name/function.

**port or ST-port** A logical connection within an end device. When the possibility of confusion exists with ULP-Ports the term ST-Port may be used; otherwise the shorter term Port is used.

**rx slot** A space in the end device reserved for a Control operation, or the Schedule Header portion of a Data operation.

**st size** The unit of memory addressed by ST for bufx and offset calculations and expressed as powers of 2.

**STU** Scheduled Transfer Unit.

**ULA** Universal LAN MAC Address. A logical address that uniquely identifies a Source or Destination. The ULA conforms to the 48-bit MAC address specified by the IEEE 802 Overview Standard.

**ULP** Upper-Layer Protocol. The protocol above the link layer. A ULP could be implemented in hardware or software, or could be distributed between the two.

**VC** Virtual channels. One of four logical paths within a single network link.

**virtual connection** A bi-directional logical connection used for scheduled transfers between two end devices. A Virtual Connection contains a logical Control Channel and one or more logical Data Channels in each direction.

# 8 References

http://www.hippi.org/cNEW.html , for a listing of documents related to HIPPI-6400, STP, and HARP.