libraw1394

version 2.0.4

libraw1394version 2.0.4

Table of Contents

Chapter 1. Introduction

The Linux kernel's IEEE 1394 subsystem provides access to the raw 1394 bus through the raw1394 module. This includes the standard 1394 transactions (read, write, lock) on the active side, isochronous stream receiving and sending and dumps of data written to the FCP_COMMAND and FCP_RESPONSE registers. raw1394 uses a character device to communicate to user programs using a special protocol.

libraw1394 was created with the intent to hide that protocol from applications so that

1. the protocol has to be implemented correctly only once.
2. all work can be done using easy to understand functions instead of handling a complicated command structure.
3. only libraw1394 has to be changed when raw1394's interface changes.

To fully achieve the goals (especially 3) libraw1394 is distributed under the LGPL (Lesser General Public License - see file COPYING.LIB for more information.) to allow linking with any program, be it open source or binary only. The requirements are that the libraw1394 part can be replaced (relinked) with another version of the library and that changes to libraw1394 itself fall under LGPL again. Refer to the LGPL text for details.

Chapter 2. Short Introduction into IEEE 1394

IEEE 1394 in fact defines two types of hardware implementations for this bus system, cable and backplane. The only one described here and supported by the Linux subsystem is the cable implementation. Most people not familiar with the standard probably don’t even know that there is something else than the 1394 cable specification.

If you are familiar with CSR architectures (as defined in ISO/IEC 13213 (ANSI/IEEE 1212)), then you already know quite a bit of 1394, which is a CSR implementation.

2.1. Bus Structure

The basic data structures defined in the standard and used in this document are the quadlet (32 bit quantity) and the octlet (64 bit quantity) and blocks (any quantity of bytes). The bus byte ordering is big endian. A transmission can be sent at one of multiple possible speeds, which are 100, 200 and 400 Mbit/s for the currently mostly used IEEE 1394a spec and up to 3.2 Gbit/s in the recently finalized 1394.b standard (these speeds are also referred to as S100, S200, ...).

A 1394 bus consists of up to 64 nodes (with multiple buses possibly being connected, but that is outside of the scope of this document and not completely standardized yet). Each node is addressed with a 16 bit address, which is further divided into a 10 bit bus ID and a 6 bit local node number, the so-called physical ID. The physical IDs are completely dynamic and determined during the bus reset. The highest values for both are special values. Bus ID equal to 1023 means "local bus" (the bus the node is connected to), physical ID equal to 63 means "all nodes" (broadcast).

The local bus ID 1023 is the only one that can be used unless IEEE 1394.1 bridge portals to more buses were available. Therefore the node IDs have to be given as (1023<<6) | phy_ID. (This is also true if libraw1394 runs at a host which contains multiple 1394 bus adapters. The local ID 1023 is valid on each of these buses. The Linux host itself is no IEEE 1394.1 bridge.)

Each node has a local address space with 48 bit wide addressing. The whole bus can thus be seen as a linear 64 bit address space by concatenating the node ID (most significant bits) and local address (least significant bits). libraw1394 treats them separately in function arguments to save the application some fiddling with the bits.

Unlike other buses there aren’t many transactions or commands defined, higher level commands are defined in terms of addresses accessed instead of separate transaction types (comparable to memory mapped registers in hardware). The 1394 transactions are:

- read (quadlets and blocks)
- write (quadlets and blocks)

- lock (some atomic modifications)

There is also the isochronous transaction (the above three are called asynchronous transactions), which is a broadcast stream with guaranteed bandwidth. It doesn't contain any address but is distinguished by a 6 bit channel number.

The bus view is only logical, physically it consists of many point-to-point connections between nodes with every node forwarding data it receives to every other port which is capable of the speed the transaction is sent at (thus a S200 node in the path between two S400 nodes would limit their communication speed to S200). It forms a tree structure with all but one node having a parent and a number of children. One node is the root node and has no parents.

2.2. Bus Reset

A bus reset occurs whenever the state of any node changes (including addition and removal of nodes). At the beginning a root node is chosen, then the tree identification determines for every node which port is connected to a parent, child or nothing. Then the SelfID phase begins. The root node sends a SelfID grant on its first port connected to a child. If that is not a leaf node, it will itself forward the grant to its first child. When a leaf node gets a grant, it will pick the lowest physical ID not yet in use (starting with 0) and send out a SelfID packet with its physical ID and more information, then acknowledge the SelfID grant to its parent, which will send a grant to its next child until it configured all its children, then pick a physical ID itself, send SelfID packet and ack to parent.

After bus reset the used physical IDs are in a sequential range with no holes starting from 0 up to the root node having the highest ID. This also means that physical IDs can change for many or all nodes with the insertion of a new node or moving the role of root to another node. In libraw1394 all transactions are tagged automatically with a generation number which is increased in every bus reset and transactions with an obsolete generation will fail in order to avoid targetting the wrong node. Nodes have to be identified in a different way than their volatile physical IDs, namely by reading their globally unique ID (GUID) contained in the configuration ROM.

2.3. Transactions

The packets transmitted on the bus are acknowledged by the receiving end unless they are broadcast packets (broadcast writes and isochronous packets). The acknowledge code contains an error code, which either signifies error, success or packet pending. In the first two cases the transaction completes, in the last a response packet will follow at a later time from the targetted node to the source node (this is called a split transaction). Only writes can succeed and complete in the ack code, reads and locks require a response. Error and packet pending can happen for every transaction. The response packets contain a response code (rcode) which signifies success or type of error.

For read and write there are two different types, quadlet and block. The quadlet types have all their

payload (exactly one quadlet) in the packet header, the block types have a variable length data block appended to the header. Programs using libraw1394 don't have to care about that, quadlet transactions are automatically used when the data length is 4 bytes and block transactions otherwise.

The lock transaction has several extended transaction codes defined which choose the atomic operation to perform, the most used being the compare-and-swap (code 0x2). The transaction passes the data value and (depending on the operation) the arg value to the target node and returns the old value at the target address, but only when the transaction does not have an error. All three values are of the same size, either one quadlet or one octlet.

In the compare-and-swap case, the data value is written to the target address if the old value is identical to the arg value. The old value is returned in any case and can be used to find out whether the swap succeeded by repeating the compare locally. Compare-and-swap is useful for avoiding race conditions when accessing the same address from multiple nodes. For example, isochronous resource allocation is done using compare-and-swap, as described below. Since the old value is always returned, it more efficient to do the first attempt with the reset value of the target register as arg instead of reading it first. Repeat with the returned old value as new arg value if it didn't succeed.

2.4. Bus Management

There are three basic bus service nodes defined in IEEE 1394 (higher level protocols may define more): cycle master, isochronous resource manager and bus manager. These positions are contended for in and shortly after the bus reset and may all be taken by a single node. A node does not have to support being any of those but if it is bus manager capable it also has to be iso manager capable, if it is iso manager capable it also has to be cycle master capable.

The cycle master sends 8000 cycle start packets per second, which initiate an iso cycle. Without that, no isochronous transmission is possible. Only the root node is allowed to be cycle master, if it is not capable then no iso transmissions can occur (and the iso or bus manager have to select another node to become root and initiate a bus reset).

The isochronous resource manager is the central point where channel and bandwidth allocations are stored. A bit in the SelfID shows whether a node is iso manager capable or not, the iso manager capable node with the highest ID wins the position after a bus reset. Apart from containing allocation registers, this one doesn't do much. Only if there is no bus manager, it may determine a cycle master capable node to become root and initiate a bus reset.

The bus manager has more responsibilities: power management (calculate power provision and consumption on the bus and turn on disabled nodes if enough power is available), bus optimization (calculate an effective gap count, optimize the topology by selecting a better positioned node for root) and some registers relevant to topology (topology map containing the SelfIDs of the last reset and a speed map, which is obsoleted in IEEE 1394a). The bus manager capable nodes contend for the role by

doing a lock transaction on the bus manager ID register in the iso manager, the first to successfully complete the transaction wins the role.

2.5. Isochronous Transmissions

Nodes can allocate a channel and bandwidth for isochronous transmissions at the iso manager to broadcast timing critical data (e.g. multimedia streams) on the bus. However these transmissions are unreliable, there is no guarantee that every packet reaches the intended recipients (the software and hardware involved also take iso packets a bit more lightly). After a cycle start packet, the isochronous cycle begins and every node can transmit iso packets, however only one packet per channel is allowed. As soon as a gap of a certain length appears (i.e. no node sends anymore), the iso cycle ends and the rest of the time until the next cycle start is reserved for asynchronous packets.

The channel register on the iso manager consists of 64 bits, each of which signifies one channel. A channel can be allocated by any node by doing a compare-swap lock request with the new bitmask. Likewise the bandwidth can be allocated by doing a lock request with the new value. The bandwidth register contains the remaining time available for every iso cycle. Since you allocate time, the maximum data you are allowed to put into an iso packet depends on the speed you will send at.

On every bus reset, the resource registers are resetted to their initial values (all channels free, all bandwidth minus some amount set aside for asynchronous communication available), this has to happen since the isochronous manager may have moved to another node. Isochronous transmissions may continue with the old allocations for 1000ms. During that time, the nodes have to reallocate their resources and no new allocations are allowed to occur. Only after this period new allocations may be done, this avoids nodes losing their allocations over a bus reset.

libraw1394 does not provide special functions for allocating iso resources nor does it clean up after programs when they exit. Protocols exist that require the first node to use some resources to allocate it and then leave it for the last node using it to deallocate it. This may be different nodes, so automatic behaviour would be very undesirable in these cases.

Chapter 3. Data Structures and Program Flow

3.1. Overview

The 1394 subsystem in Linux is divided into the classical three layers, like most other interface subsystems in Linux. The in-kernel subsystem consists of the ieee1394 core, which provides basic services like handling of the 1394 protocol (converting the abstract transactions into packets and back), collecting information about bus and nodes and providing some services to the bus that are required to be available for standards conformant nodes (e.g. CSR registers). Below that are the hardware drivers, which handle converting packets and bus events to and from hardware accesses on specific 1394 chipsets.

Above the core are the highlevel drivers, which use the services provided by the core to implement protocols for certain devices and act as drivers to these. raw1394 is one such driver, however it is not specialized to handle one kind of device but is designed to accept commands from user space to do any transaction wanted (as far as possible from current core design). Using raw1394, normal applications can access 1394 nodes on the bus and it is not neccessary to write kernel code just for that.

raw1394 communicates to user space like most device drivers do, through device files in /dev. It uses a defined protocol on that device, but applications don't have to and should not care about that. All of this is taken care of by libraw1394, which provides a set of functions that convert to and from raw1394 protocol packets and are a lot easier to handle than that underlying protocol.

3.2. Handles

The handle presented to the application for using libraw1394 is the raw1394handle_t, an opaque data structure (which means you don't need to know its internals). The handle (and with it a connection to the kernel side of raw1394) is obtained using `raw1394_new_handle()`. Insufficient permissions to access the kernel driver will result in failure of this function, among other possibilities of failure.

While initializing the handle, a certain order of function calls have to be obeyed or undefined results will occur. This order reflects the various states of initialization to be done:

1. `raw1394_new_handle()`
2. `raw1394_get_port_info()`
3. `raw1394_set_port()`

3.3. Ports

A computer may have multiple 1394 buses connected by having multiple 1394 chips. Each of these is called a port, and the handle has to be connected to one port before it can be used for anything. Even if no nodes are connected to the chip in question, it forms a complete bus (with just one node, itself).

A list of available ports together with some information about it (name of the hardware, number of connected nodes) is available via `raw1394_get_port_info()`, which is to be called right after getting a fresh handle. The user should be presented with a choice of available ports if there is more than one. It may be good practice to do that even if there is only one port, since that may result from a normally configured port just not being available, making it confusing to be dropped right into the application attached to a port without a choice and notion of anything going wrong.

The choice of port is then reported using `raw1394_set_port()`. If this function fails and errno is set to ESTALE, then something has changed about the ports (port was added or removed) between getting the port info and trying to set a port. It is required that the current port list is fetched (presenting the user with the choice again) and setting the port is retried with the new data.

After a successful `raw1394_set_port()`, the get and set port functions must not be used anymore on this handle. Undefined results occur if you do so. To make up for this, all the other functions are allowed now.

3.4. The Event Loop

All commands in libraw1394 are asynchronous, with some synchronous wrapper functions for some types of transactions. This means that there are two streams of data, one going into raw1394 and one coming out. With this design you can send out multiple transactions without having to wait for the response before you can continue (sending out other transactions, for example). The responses and other events (like bus resets and received isochronous packets) are queued, and you can get them with `raw1394_loop_iterate()` or `raw1394_loop_iterate_timeout()` (which always returns after a user-specified timeout if no raw1394 event has occurred).

This forms an event loop you may already know from similar systems like GUI toolkits. `raw1394_loop_iterate()` gets one message from the event queue in raw1394, processes it with the configured callback functions and returns the value returned by the callback (so you can signal to the main loop from your callback; the standard callbacks all return 0). It normally blocks when there are no events and always processes only one event. If you are only receiving broadcast events like isochronous packets you thus have to set up a loop continuously calling the iterate function to get your callbacks called.

Often it is necessary to have multiple event loops and combine them, e.g. if your application uses a GUI toolkit which also has its own event loop. In that case you can use `raw1394_get_fd()` to get the file

descriptor used for this handle by libraw1394. The fd can be used to for `select()` or `poll()` calls together with the other loop's fd. (Most toolkits, like GTK and Qt, have special APIs for integrating file descriptors into their own event loops).

If using `poll()`, you must test for POLLIN and POLLPRI events. If using `select()`, you must test for both read and exception activity.

If any of these conditions trigger, you should then call `raw1394_loop_iterate()` to pick up the event. `raw1394_loop_iterate()` is guaranteed not to block when called immediately after select() or poll() indicates activity. After the first call you continue the main event loop. If more events wait, the `select()`/`poll()` will immediately return again.

You can also use the fd to set the O_NONBLOCK flag with `fcntl()`. After that, the iterate function will not block anymore but fail with errno set to EAGAIN if no events wait. These are the only legal uses for the fd returned by `raw1394_get_fd()`.

There are some functions which provide a synchronous wrapper for transactions, note that these will call `raw1394_loop_iterate()` continuously until their transaction is completed, thus having implicit callback invocations during their execution. The standard transaction functions have names of the form `raw1394_start_xxx`, the synchronous wrappers are called `raw1394_xxx`.

3.5. Handlers

There are a number of handlers which can be set using the appropriate function as described in the function reference and which libraw1394 will call during a `raw1394_loop_iterate()`. These are:

- tag handler (called for completed commands)
- bus reset handler (called when a bus reset happens)
- iso handler (called when an iso packet is received)
- fcp handler (called when a FCP command or response is received)

The bus reset handler is always called, the tag handler for every command that completes, the iso handler and fcp handler are only called when the application chooses to receive these packets. Handlers return an integer value which is passed on by `raw1394_loop_iterate()` (only one handler is called per invocation), `0` is returned without a handler in place.

The tag handler case is a bit special since the default handler is actually doing something. Every command that you start can be given an unsigned long tag which is passed untouched to the tag handler when the event loop sees a completed command. The default handler expects this value to be a pointer to a raw1394_reqhandle structure, which contains a data pointer and its own callback function pointer. The callback gets the untouched data pointer and error code as arguments. If you want to use tags that are not raw1394_reqhandle pointers you have to set up your own tag handler.

3.6. Generation Numbers

libraw1394 and the kernel code use generation numbers to identify the current bus configuration and increment those on every configuration change. The most important generation number is stored per connected 1394 bus and incremented on every bus reset. There is another number managed by raw1394 which identifies global changes (like a complete port being added or removed), which is used for the `raw1394_set_port()` function to make sure you don't use stale port numbers. This is done transparently to you.

The bus generation number is more relevant for your work. Since nodes can change IDs with every bus reset, it is very likely that you don't want to send a packet you constructed with the old ID before you noticed the bus reset. This does not apply to isochronous transmissions, since they are broadcast and do not depend on bus configuration. Therefore every packet is automatically tagged with the expected generation number, and it will fail to send if that does not match the number managed in the kernel for the port in question.

You get the current generation number through the bus reset handler. If you don't set a custom bus reset handler, the default handler will update the generation number automatically. If you set your own handler, you can update the generation number to be used through `raw1394_update_generation()` directly in the handler or later.

3.7. Error and Success Codes

libraw1394 returns the ack/rcode pair in most transaction cases. The rcode is undefined in cases where the ack code is not equal to ack_pending. This is stored in a type raw1394_errcode_t, from which the ack and rcode parts can be extracted using two macros.

With the function `raw1394_errcode_to_errno()` it is possible to convert this to an errno number that conveys roughly the same meaning. Many developers will find that easier to handle. This is done automatically for the synchronous read/write/lock wrapper functions, i.e. they return 0 for success and a negative value for failure, in which case they also set the errno variable to the appropriate code. The raw ack/rcode pair can then still be retrieved using `raw1394_get_errcode()`.

Chapter 4. Isochronous Transmission and Reception

4.1. Overview

Isochronous operations involve sending or receiving a constant stream of packets at a fixed rate of 8KHz. Unlike raw1394's asynchronous API, where you "push" packets to raw1394 functions at your leisure, the isochronous API is based around a "pull" model. During isochronous transmission or reception, raw1394 informs your application when a packet must be sent or received. You must fulfill these requests in a timely manner to avoid breaking the constant stream of isochronous packets.

A raw1394 handle may be associated with one isochronous stream, either transmitting or receiving (but not both at the same time). To transmit or receive more than one stream simultaneously, you must create more than one raw1394 handle.

4.2. Initialization

When a raw1394 handle is first created, no isochronous stream is assocated with it. To begin isochronous operations, call either `raw1394_iso_xmit_init()` (transmission) or `raw1394_iso_recv_init()` (reception). The parameters to these functions are as follows:

handler is your function for queueing packets to be sent (transmission) or processing received packets (reception).

buf_packets is the number of packets that will be buffered at the kernel level. A larger packet buffer will be more forgiving of IRQ and application latency, however it will consume more kernel memory. For most applications, it is sufficient to buffer 2000-16000 packets (0.25 seconds to 2.0 seconds maximum latency).

max_packet_size is the size, in bytes, of the largest isochronous packet you intend to handle. This size does not include the isochronous header but it does include the CIP header specified by many isochronous protocols.

channel is the isochronous channel on which you wish to receive or transmit. (currently there is no facility for multi-channel transmission or reception).

speed is the isochronous speed at which you wish to operate. Possible values are RAW1394_ISO_SPEED_100, RAW1394_ISO_SPEED_200, and RAW1394_ISO_SPEED_400.

irq_interval is the maximum latency of the kernel buffer, in packets. (To avoid excessive IRQ rates, the low-level drivers only trigger an interrupt every irq_interval packets). Pass -1 to receive a default value that should be suitable for most applications.

mode for `raw1394_iso_recv_init()` sets whether to use packet-per-buffer or buffer-fill receive mode. Possible values are RAW1394_DMA_DEFAULT (bufferfill on ohci1394), RAW1394_DMA_BUFFERFILL, and RAW1394_DMA_PACKET_PER_BUFFER.

If `raw1394_iso_xmit/recv_init()` retuns successfully, then you may start isochronous operations. You may not call `raw1394_iso_xmit/recv_init()` again on the same handle without first shutting down the isochronous operation with `raw1394_iso_shutdown()`.

Note that `raw1394_iso_xmit_init()` and `raw1394_iso_recv_init()` involve potentially time-consuming operations like allocating kernel and device resources. If you intend to transmit or receive several isochronous streams simultaneously, it is advisable to initialize all streams before starting any packet transmission or reception.

4.3. Stopping and Starting

Once the isochronous operation has been initialized, you may start and stop packet transmission with `raw1394_iso_xmit/recv_start()` and `raw1394_iso_stop()`. It is legal to call these as many times as you want, and it is permissible to start an already-started stream or stop an already-stopped stream. Packets that have been queued for transmission or reception will remain queued when the operation is stopped.

`raw1394_iso_xmit/recv_start()` allow you to specify on which isochronous cycle number to start transmitting or receiving packets. Pass -1 to start immediately. This parameter is ignored if isochronous transmission or reception is already in progress.

`raw1394_iso_xmit_start()` has an additional parameter, prebuffer_packets, which specifies how many packets to queue up before starting transmission. Possible values range from zero (start transmission immediately after the first packet is queued) up to the total number of packets in the buffer.

Once the isochronous operation has started, you must repeatedly call `raw1394_loop_iterate()` as usual to drive packet processing.

4.4. Receiving Packets

Raw1394 maintains a fixed-size ringbuffer of packets in kernel memory. The buffer is filled by the low-level driver as it receives packets from the bus. It is your application's job to process each packet,

after which the buffer space it occupied can be re-used for future packets.

The isochronous receive handler you provided will be called from `raw1394_loop_iterate()` after each packet is received. Your handler is passed a pointer to the first byte of the packet's data payload, plus the packet's length in bytes (not counting the isochronous header), the cycle number at which it was received, the channel on which it was received, and the "tag" and "sy" fields from the isochronous header. Note that the packet is at this point still in the kernel's receive buffer, so the data pointer is only valid until the receive handler returns. You must make a copy of the packet's data if you want to keep it.

The receive handler is also passed a "packet(s) dropped" flag. If this flag is nonzero, it means that one or more incoming packets have been dropped since the last call to your handler (usually this is because the kernel buffer has completely filled up with packets or a bus reset has occurred).

4.5. Transmitting Packets

Similar to reception, raw1394 maintains a fixed-size ringbuffer of packets in kernel memory. The buffer is filled by your application as it queues packets to be sent. The buffer is drained by the hardware driver as it transmits packets on the 1394 bus.

The isochronous transmit handler you provided will be called from `raw1394_loop_iterate()` whenever there is space in the buffer to queue another packet. The handler is passed a pointer to the first byte of the buffer space for the packet's data payload, pointers to words containing the data length in bytes (not counting the isochronous header), "tag" and "sy" fields, and the isochronous cycle number at which this packet will be transmitted. The handler must write the packet's data payload into the supplied buffer space, and set the values pointed to by "len", "tag", and "sy" to the appropriate values. The handler is permitted to write any number of data bytes, up and including to the value of max_packet_size passed to `raw1394_iso_xmit_init()`.

Note: If you passed -1 as the starting cycle to `raw1394_iso_xmit_init()`, the cycle number provided to your handler will be incorrect until after one buffer's worth of packets have been transmitted.

The transmit handler is also passed a "packet(s) dropped" flag. If this flag is nonzero, it means that one or more outgoing packets have been dropped since the last call to your handler (usually this is because the kernel buffer has gone completely empty or a bus reset has occurred).

4.6. Shutting down

When the isochronous operation has finished, call `raw1394_iso_shutdown()` to release all associated resources. If you don't call this function explicitly, it will be called automatically when the raw1394 handle is destroyed.

Chapter 5. Function Reference

raw1394_iso_xmit_init

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_iso_xmit_init` — initialize isochronous transmission

Synopsis

```
int raw1394_iso_xmit_init (raw1394handle_t handle, raw1394_iso_xmit_handler_t
handler, unsigned int buf_packets, unsigned int max_packet_size, unsigned
char channel, enum raw1394_iso_speed speed, int irq_interval);
```

Arguments

handle

libraw1394 handle

handler

handler function for queueing packets

buf_packets

number of isochronous packets to buffer

max_packet_size

largest packet you need to handle, in bytes (not including the isochronous header)

channel

isochronous channel on which to transmit

speed

speed at which to transmit

irq_interval

maximum latency of wake-ups, in packets (-1 if you don't care)

Description

Allocates all user and kernel resources necessary for isochronous transmission. Channel and bandwidth allocation at the IRM is not performed.

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_recv_init

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_iso_recv_init` — initialize isochronous reception

Synopsis

```
int raw1394_iso_recv_init (raw1394handle_t handle, raw1394_iso_recv_handler_t
handler, unsigned int buf_packets, unsigned int max_packet_size, unsigned
char channel, enum raw1394_iso_dma_recv_mode mode, int irq_interval);
```

Arguments

handle

libraw1394 handle

handler

handler function for receiving packets

buf_packets

number of isochronous packets to buffer

max_packet_size

largest packet you need to handle, in bytes (not including the isochronous header)

channel

isochronous channel to receive

mode

bufferfill or packet per buffer mode

irq_interval

maximum latency of wake-ups, in packets (-1 if you don't care)

Description

Allocates all user and kernel resources necessary for isochronous reception.

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_multichannel_recv_init

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_iso_multichannel_recv_init` — initialize multi-channel iso reception

Synopsis

```
int raw1394_iso_multichannel_recv_init (raw1394handle_t handle,
raw1394_iso_recv_handler_t handler, unsigned int buf_packets, unsigned int
max_packet_size, int irq_interval);
```

Arguments

handle

libraw1394 handle

handler

handler function for receiving packets

buf_packets

number of isochronous packets to buffer

max_packet_size

largest packet you need to handle, in bytes (not including the isochronous header)

irq_interval

maximum latency of wake-ups, in packets (-1 if you don't care)

Description

Allocates all user and kernel resources necessary for isochronous reception.

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_recv_listen_channel

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_iso_recv_listen_channel` — listen to a specific channel in multi-channel mode

Synopsis

```
int raw1394_iso_recv_listen_channel (raw1394handle_t handle, unsigned char
channel);
```

Arguments

handle

libraw1394 handle

channel

the channel to start listening

Description

listen/unlisten on a specific channel (multi-channel mode ONLY)

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_recv_unlisten_channel

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_iso_recv_unlisten_channel — stop listening to a specific channel in multi-channel mode

Synopsis

```
int raw1394_iso_recv_unlisten_channel (raw1394handle_t handle, unsigned char channel);
```

Arguments

handle

libraw1394 handle

channel

the channel to stop listening to

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_recv_set_channel_mask

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_iso_recv_set_channel_mask — listen or unlisten to a whole bunch of channels at once

Synopsis

```
int raw1394_iso_recv_set_channel_mask (raw1394handle_t handle, u_int64_t mask);
```

Arguments

handle

libraw1394 handle

mask

64-bit mask of channels, 1 means listen, 0 means unlisten, channel 0 is LSB, channel 63 is MSB

Description

for multi-channel reception mode only

Returns

0 on success, -1 on failure (sets errno)

raw1394_iso_xmit_start

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_iso_xmit_start` — begin isochronous transmission

Synopsis

```
int raw1394_iso_xmit_start (raw1394handle_t handle, int start_on_cycle, int
prebuffer_packets);
```

Arguments

handle

libraw1394 handle

start_on_cycle

isochronous cycle number on which to start (-1 if you don't care)

prebuffer_packets

number of packets to queue up before starting transmission (-1 if you don't care)

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_recv_start

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_iso_recv_start — begin isochronous reception

Synopsis

```
int raw1394_iso_recv_start (raw1394handle_t handle, int start_on_cycle, int
tag_mask, int sync);
```

Arguments

handle

libraw1394 handle

start_on_cycle

isochronous cycle number on which to start (-1 if you don't care)

tag_mask

mask of tag fields to match (-1 to receive all packets)

sync

not used, reserved for future implementation

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_xmit_write

LINUX

Name

raw1394_iso_xmit_write — alternative blocking-write API for ISO transmission

Synopsis

```
int raw1394_iso_xmit_write (raw1394handle_t handle, unsigned char * data,
unsigned int len, unsigned char tag, unsigned char sy);
```

Arguments

handle

libraw1394 handle

data

pointer to packet data buffer

len

length of packet, in bytes

tag

tag field

sy

sync field

Description

write style API - do NOT use this if you have set an xmit_handler if buffer is full, waits for more space UNLESS the file descriptor is set to non-blocking, in which case xmit_write will return -1 with errno = EAGAIN

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_xmit_sync

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_iso_xmit_sync — wait until all queued packets have been sent

Synopsis

```
int raw1394_iso_xmit_sync (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_recv_flush

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_iso_recv_flush — flush all already received iso packets from kernel into user space

Synopsis

```
int raw1394_iso_recv_flush (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Description

If you specified an irq_interval > 1 in iso_recv_init, you won't be notified for every single iso packet, but for groups of them. Now e.g. if irq_interval is 100, and you were just notified about iso packets and after them only 20 more packets arrived, no notification will be generated (20 < 100). In the case that you know that there should be more packets at this moment, you can call this function and all iso packets which are already received by the kernel will be flushed out to user space.

Returns

0 on success or -1 on failure (sets errno)

raw1394_iso_stop

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_iso_stop — halt isochronous transmission or reception

Synopsis

```
void raw1394_iso_stop (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

raw1394_iso_shutdown

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_iso_shutdown — clean up and deallocate all resources for isochronous transmission or reception

Synopsis

```
void raw1394_iso_shutdown (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

raw1394_read_cycle_timer

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_read_cycle_timer — get the current value of the cycle timer

Synopsis

```
int raw1394_read_cycle_timer (raw1394handle_t handle, u_int32_t *
cycle_timer, u_int64_t * local_time);
```

Arguments

handle

libraw1394 handle

cycle_timer

buffer for Isochronous Cycle Timer

local_time

buffer for local system time in microseconds since Epoch

Description

Simultaneously reads the cycle timer register together with the system clock.

Format of *cycle_timer*, from MSB to LSB: 7 bits cycleSeconds (seconds, or number of cycleCount rollovers), 13 bits cycleCount (isochronous cycles, or cycleOffset rollovers), 12 bits cycleOffset (24.576 MHz clock ticks, not provided on some hardware). The union of cycleSeconds and cycleCount is the current cycle number. The nominal duration of a cycle is 125 microseconds.

Returns

the error code of the ioctl, or 0 if successful.

raw1394_get_errcode

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_get_errcode` — return error code of async transaction

Synopsis

```
raw1394_errcode_t raw1394_get_errcode (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Description

Some macros are available to extract information from the error code, `raw1394_errcode_to_errno` can be used to convert it to an errno number of roughly the same meaning.

Returns

the error code of the last `raw1394_read`, `raw1394_write`, `raw1394_lock`. The error code is either an internal error (i.e. not a bus error) or a combination of acknowledge code and response code, as appropriate.

raw1394_errcode_to_errno

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_errcode_to_errno` — convert libraw1394 errcode to errno

Synopsis

```
int raw1394_errcode_to_errno (raw1394_errcode_t errcode);
```

Arguments

errcode

the error code to convert

Description

The error code as retrieved by `raw1394_get_errcode` is converted into a roughly equivalent errno number and returned. `0xdead` is returned for an illegal errcode.

It is intended to be used to decide what to do (retry, give up, report error) for those programs that aren't interested in details, since these get lost in the conversion. However the returned errnos are equivalent in source code meaning only, the associated text of e.g. `perror` is not necessarily meaningful.

Returns

`EAGAIN` (retrying might succeed, also generation number mismatch), `EREMOTEIO` (other node had internal problems), `EPERM` (operation not allowed on this address, e.g. write on read-only location), `EINVAL` (invalid argument) and `EFAULT` (invalid pointer).

raw1394_new_handle

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_new_handle` — create new handle

Synopsis

```
raw1394handle_t raw1394_new_handle ( void);
```

Arguments

`void`

no arguments

Description

Creates and returns a new handle which can (after being set up) control one port. It is not allowed to use the same handle in multiple threads or forked processes. It is allowed to create and use multiple handles, however. Use one handle per thread which needs it in the multithreaded case.

The default device node is /dev/raw1394, but one can override the default by setting environment variable RAW1394DEV. However, if RAW1394DEV points to a non-existant or invalid device node, then it also attempts to open the default device node.

Returns

the created handle or `NULL` when initialization fails. In the latter case errno either contains some OS specific error code or EPROTO if libraw1394 and raw1394 don't support each other's protocol versions.

raw1394_destroy_handle

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_destroy_handle` — deallocate handle

Synopsis

```
void raw1394_destroy_handle (raw1394handle_t handle);
```

Arguments

`handle`

handle to deallocate

Description

Closes connection with raw1394 on this handle and deallocates everything associated with it. It is safe to pass `NULL` as handle, nothing is done in this case.

raw1394_new_handle_on_port

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_new_handle_on_port — create a new handle and bind it to a port

Synopsis

```
raw1394handle_t raw1394_new_handle_on_port (int port);
```

Arguments

port

port to connect to (same as argument to raw1394_set_port)

Description

Same as raw1394_new_handle, but also binds the handle to the specified 1394 port. Equivalent to raw1394_new_handle followed by raw1394_get_port_info and raw1394_set_port. Useful for command-line programs that already know what port they want. If raw1394_set_port returns ESTALE, retries automatically.

The default device node is /dev/raw1394, but one can override the default by setting environment variable RAW1394DEV. However, if RAW1394DEV points to a non-existant or invalid device node, then it also attempts to open the default device node.

Returns

the new handle on success or NULL on failure

raw1394_busreset_notify

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_busreset_notify` — Switch off/on busreset-notification for handle

Synopsis

```
int raw1394_busreset_notify (raw1394handle_t handle, int off_on_switch);
```

Arguments

`handle`

libraw1394 handle

`off_on_switch`

RAW1394_NOTIFY_OFF or RAW1394_NOTIFY_ON

Returns

0 on success or -1 on failure (sets errno)

raw1394_get_fd

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_get_fd` — get the communication file descriptor

Synopsis

```
int raw1394_get_fd (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Description

This can be used for `select`/`poll` calls if you wait on other fds or can be integrated into another event loop (e.g. from a GUI application framework). It can also be used to set/remove the O_NONBLOCK flag using `fcntl` to modify the blocking behaviour in `raw1394_loop_iterate`. It must not be used for anything else.

Returns

the fd used for communication with the raw1394 kernel module or -1 on failure (sets errno).

raw1394_set_userdata

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_set_userdata` — associate user data with a handle

Synopsis

```
void raw1394_set_userdata (raw1394handle_t handle, void * data);
```

Arguments

handle

libraw1394 handle

data

user data (pointer)

Description

Allows to associate one void pointer with a handle. libraw1394 does not care about the data, it just stores it in the handle allowing it to be retrieved at any time with `raw1394_get_userdata`. This can be useful when multiple handles are used, so that callbacks can identify the handle.

raw1394_get_userdata

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_get_userdata` — retrieve user data from handle

Synopsis

```
void * raw1394_get_userdata (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Returns

the user data pointer associated with the handle using `raw1394_set_userdata`.

raw1394_get_local_id

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_get_local_id` — get node ID of the current port

Synopsis

```
nodeid_t raw1394_get_local_id (raw1394handle_t handle);
```

Arguments

`handle`

libraw1394 handle

Returns

the node ID of the local node connected to which the handle is connected. This value can change with every bus reset.

raw1394_get_irm_id

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_get_irm_id` — get node ID of isochronous resource manager

Synopsis

```
nodeid_t raw1394_get_irm_id (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Returns

the node ID of the isochronous resource manager of the bus the handle is connected to. This value may change with every bus reset.

raw1394_get_nodecount

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_get_nodecount — get number of nodes on the bus

Synopsis

```
int raw1394_get_nodecount (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Description

Since the root node always has the highest node ID, this number can be used to determine that ID (it's LOCAL_BUS|(count-1)).

Returns

the number of nodes on the bus to which the handle is connected. This value can change with every bus reset.

raw1394_get_port_info

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_get_port_info` — get information about available ports

Synopsis

```
int raw1394_get_port_info (raw1394handle_t handle, struct raw1394_portinfo *
pinf, int maxports);
```

Arguments

`handle`

libraw1394 handle

`pinf`

pointer to an array of struct raw1394_portinfo

`maxports`

number of elements in *`pinf`*

Description

Before you can set which port to use, you have to use this function to find out which ports exist.

If your program is interactive, you should present the user with this list to let them decide which port to use if there is more than one. A non-interactive program (and probably interactive ones, too) should provide a command line option to choose the port. If *`maxports`* is `0`, *`pinf`* can be `NULL`, too.

Returns

the number of ports and writes information about them into *pinf*, but not into more than *maxports* elements.

raw1394_set_port

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_set_port — choose port for handle

Synopsis

```
int raw1394_set_port (raw1394handle_t handle, int port);
```

Arguments

handle

libraw1394 handle

port

port to connect to (corresponds to index of struct raw1394_portinfo)

Description

This function connects the handle to the port given (as queried with raw1394_get_port_info). If successful, raw1394_get_port_info and raw1394_set_port are not allowed to be called afterwards on this handle. To make up for this, all the other functions (those handling asynchronous and isochronous transmissions) can now be called.

Returns

0 for success or -1 for failure with errno set appropriately. A possible failure mode is with errno = ESTALE, in this case the configuration has changed since the call to raw1394_get_port_info and it has to be called again to update your view of the available ports.

raw1394_reset_bus

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_reset_bus — initiate bus reset

Synopsis

```
int raw1394_reset_bus (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Description

This function initiates a bus reset on the connected port. Usually this is not necessary and should be avoided, this function is here for low level bus control and debugging.

Returns

0 for success or -1 for failure with errno set appropriately

raw1394_reset_bus_new

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_reset_bus_new — Reset the connected bus (with certain type).

Synopsis

```
int raw1394_reset_bus_new (raw1394handle_t handle, int type);
```

Arguments

handle

libraw1394 handle

type

RAW1394_SHORT_RESET or RAW1394_LONG_RESET

Returns

0 for success or -1 for failure

raw1394_loop_iterate

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_loop_iterate — get and process one event message

Synopsis

```
int raw1394_loop_iterate (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Description

Get one new message through handle and process it with the registered message handler. Note that some other library functions may call this function multiple times to wait for their completion, some handler return values may get lost if you use these.

Returns

`-1` for an error or the return value of the handler which got executed. The default handlers always return zero.

raw1394_set_bus_reset_handler

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_set_bus_reset_handler` — set bus reset handler

Synopsis

```
bus_reset_handler_t raw1394_set_bus_reset_handler (raw1394handle_t handle,
bus_reset_handler_t new_h);
```

Arguments

handle

libraw1394 handle

new_h

pointer to new handler

Description

Sets the handler to be called on every bus reset to *new_h*. The default handler just calls raw1394_update_generation.

Returns

the old handler or NULL on failure (sets errno)

raw1394_get_generation

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_get_generation — get generation number of handle

Synopsis

```
unsigned int raw1394_get_generation (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Description

The generation number is incremented on every bus reset, and every transaction started by raw1394 is tagged with the stored generation number. If these don't match, the transaction will abort with an error. The generation number of the handle is not automatically updated, `raw1394_update_generation` has to be used for this.

Returns

the generation number associated with the handle or UINT_MAX on failure.

raw1394_update_generation

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_update_generation` — set generation number of handle

Synopsis

```
void raw1394_update_generation (raw1394handle_t handle, unsigned int
generation);
```

Arguments

`handle`

libraw1394 handle

`generation`

new generation number

Description

This function sets the generation number of the handle to *gen*. All requests that apply to a single node ID are tagged with this number and abort with an error if that is different from the generation number kept in the kernel. This avoids acting on the wrong node which may have changed its ID in a bus reset.

You should call this within your bus reset handler with an incremented value.

raw1394_set_tag_handler

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_set_tag_handler` — set request completion handler

Synopsis

```
tag_handler_t raw1394_set_tag_handler (raw1394handle_t handle, tag_handler_t
new_h);
```

Arguments

handle

libraw1394 handle

new_h

pointer to new handler

Description

Sets the handler to be called whenever a request completes to *new_h*. The default handler interprets the tag as a pointer to a struct raw1394_reqhandle and calls the callback in there.

Care must be taken when replacing the tag handler and calling the synchronous versions of the transaction functions (i.e. `raw1394_read`, `raw1394_write`, `raw1394_lock`) since these do pass pointers to struct raw1394_reqhandle as the tag and expect the callback to be invoked.

Returns

the old handler or NULL on failure (sets errno)

raw1394_set_arm_tag_handler

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_set_arm_tag_handler` — set the async request handler

Synopsis

```
arm_tag_handler_t raw1394_set_arm_tag_handler (raw1394handle_t handle,
arm_tag_handler_t new_h);
```

Arguments

`handle`

libraw1394 handle

`new_h`

pointer to new handler

Description

Set the handler that will be called when an async read/write/lock arm_request arrived. The default action is to call the arm_callback in the raw1394_arm_reqhandle pointed to by arm_tag.

Returns

old handler or NULL on failure (sets errno)

raw1394_set_fcp_handler

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_set_fcp_handler` — set FCP handler

Synopsis

```
fcp_handler_t raw1394_set_fcp_handler (raw1394handle_t handle, fcp_handler_t
new_h);
```

Arguments

`handle`

libraw1394 handle

`new_h`

pointer to new handler

Description

Function Control Protocol is defined in IEC 61883-1.

Sets the handler to be called when either FCP command or FCP response registers get written to *`new_h`*. The default handler does nothing. In order to actually get FCP events, you have to enable it with `raw1394_start_fcp_listen` and can stop it with `raw1394_stop_fcp_listen`.

Returns

the old handler or NULL on failure (sets errno)

int

LINUX

Kernel Hackers ManualAugust 2009

Name

`int` — This is the general request handler

Synopsis

```
typedef int ( * req_callback_t);
```

Arguments

`req_callback_t`

This is the general request handler

Description

It is used by the default tag handler when a request completes, it calls the callback and passes it the data pointer and the error code of the request.

int

LINUX

Kernel Hackers ManualAugust 2009

Name

`int` — This is the general arm-request handle

Synopsis

```
typedef int ( * arm_req_callback_t);
```

Arguments

arm_req_callback_t

This is the general arm-request handle

Description

(arm = address range mapping) It is used by the default arm-tag handler when a request has been received, it calls the arm_callback.

raw1394_arm_register

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_arm_register` — register an AddressRangeMapping

Synopsis

```
int raw1394_arm_register (raw1394handle_t handle, nodeaddr_t start, size_t
length, byte_t * initial_value, octlet_t arm_tag, arm_options_t
access_rights, arm_options_t notification_options, arm_options_t
client_transactions);
```

Arguments

handle

libraw1394 handle

start

identifies addressrange

length

identifies addressrange

initial_value

pointer to buffer containing (if necessary) initial value NULL means undefined

arm_tag

identifier for arm_tag_handler (usually pointer to raw1394_arm_reqhandle)

access_rights

access-rights for registered addressrange handled by kernel-part. Value is one or more binary or of the following flags - ARM_READ, ARM_WRITE, ARM_LOCK

notification_options

identifies for which type of request you want to be notified. Value is one or more binary or of the following flags - ARM_READ, ARM_WRITE, ARM_LOCK

client_transactions

identifies for which type of request you want to handle the request by the client application. for those requests no response will be generated, but has to be generated by the application. Value is one or more binary or of the following flags - ARM_READ, ARM_WRITE, ARM_LOCK For each bit set here, notification_options and access_rights will be ignored.

Description

ARM = Adress Range Mapping

Returns

0 on success or -1 on failure (sets errno)

raw1394_arm_unregister

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_arm_unregister` — unregister an AddressRangeMapping

Synopsis

```
int raw1394_arm_unregister (raw1394handle_t handle, nodeaddr_t start);
```

Arguments

handle

libraw1394 handle

start

identifies addressrange for unregistering (value of start have to be the same value used for registering this adressrange)

Returns

0 on success or -1 on failure (sets errno)

raw1394_arm_set_buf

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_arm_set_buf — set the buffer of an AdressRangeMapping

Synopsis

```
int raw1394_arm_set_buf (raw1394handle_t handle, nodeaddr_t start, size_t
length, void * buf);
```

Arguments

handle

libraw1394 handle

start

identifies addressrange

length

identifies addressrange

buf

pointer to buffer

Description

This function copies *length* bytes from user memory area *buf* to one ARM block in kernel memory area with start offset *start*.

Returns

0 on success or -1 on failure (sets errno)

raw1394_arm_get_buf

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_arm_get_buf — get the buffer of an AdressRangeMapping

Synopsis

```
int raw1394_arm_get_buf (raw1394handle_t handle, nodeaddr_t start, size_t
length, void * buf);
```

Arguments

handle

libraw1394 handle

start

identifies addressrange

length

identifies addressrange

buf

pointer to buffer

Description

This function copies *length* bytes from one ARM block in kernel memory area with start offset *start* to user memory area *buf*

Returns

0 on success or -1 on failure (sets errno)

raw1394_echo_request

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_echo_request — send an echo request to the driver

Synopsis

```
int raw1394_echo_request (raw1394handle_t handle, quadlet_t data);
```

Arguments

handle

libraw1394 handle

data

arbitrary data; raw1394_loop_iterate will return it

Description

the driver then send back the same request. raw1394_loop_iterate will return data as return value, when it processes the echo.

Returns

0 on success or -1 on failure (sets errno)

raw1394_wake_up

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_wake_up` — wake up raw1394_loop_iterate

Synopsis

```
int raw1394_wake_up (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Description

(or a blocking read from the device file). actually this calls raw1394_echo_request with 0 as data.

Returns

0 on success or -1 on failure (sets errno)

raw1394_phy_packet_write

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_phy_packet_write` — send physical request

Synopsis

```
int raw1394_phy_packet_write (raw1394handle_t handle, quadlet_t data);
```

Arguments

handle

libraw1394 handle

data

the contents of the packet

Description

examples of physical requests are linkon, physicalconfigurationpacket, etc.

Returns

0 on success or -1 on failure (sets errno)

raw1394_start_phy_packet_write

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_start_phy_packet_write — initiate sending a physical request

Synopsis

```
int raw1394_start_phy_packet_write (raw1394handle_t handle, quadlet_t data,
unsigned long tag);
```

Arguments

handle

libraw1394 handle

data

the contents of the packet

tag

data to identify the request to completion handler

Description

examples of physical requests are linkon, physicalconfigurationpacket, etc.

Returns

0 on success or -1 on failure (sets errno)

raw1394_start_read

LINUX

Name

`raw1394_start_read` — initiate a read transaction

Synopsis

```
int raw1394_start_read (raw1394handle_t handle, nodeid_t node, nodeaddr_t
addr, size_t length, quadlet_t * buffer, unsigned long tag);
```

Arguments

`handle`

libraw1394 handle

`node`

target node ID

`addr`

address to read from

`length`

amount of bytes of data to read

`buffer`

pointer to buffer where data will be saved

`tag`

data to identify the request to completion handler

Description

This function starts the specified read request. If *`length`* is `4` a quadlet read is initiated and a block read otherwise.

The transaction is only started, no success of the transaction is implied with a successful return of this function. When the transaction completes, a `raw1394_loop_iterate` will call the tag handler and pass it the tag and error code of the transaction. *`tag`* should therefore be set to something that uniquely identifies this transaction (e.g. a struct pointer casted to unsigned long).

Returns

0 on success or -1 on failure (sets errno)

raw1394_start_write

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_start_write` — initiate a write transaction

Synopsis

```
int raw1394_start_write (raw1394handle_t handle, nodeid_t node, nodeaddr_t
addr, size_t length, quadlet_t * data, unsigned long tag);
```

Arguments

handle

libraw1394 handle

node

target node ID

addr

address to write to

length

amount of bytes of data to write

data

pointer to data to be sent

tag

data to identify the request to completion handler

Description

This function starts the specified write request. If *length* is 4 a quadlet write is initiated and a block write otherwise.

The transaction is only started, no success of the transaction is implied with a successful return of this function. When the transaction completes, a raw1394_loop_iterate will call the tag handler and pass it the tag and error code of the transaction. *tag* should therefore be set to something that uniquely identifies this transaction (e.g. a struct pointer casted to unsigned long).

Returns

0 on success or -1 on failure (sets errno)

raw1394_start_lock

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_start_lock — initiate a 32-bit compare-swap lock transaction

Synopsis

```
int raw1394_start_lock (raw1394handle_t handle, nodeid_t node, nodeaddr_t
addr, unsigned int extcode, quadlet_t data, quadlet_t arg, quadlet_t *
result, unsigned long tag);
```

Arguments

handle

libraw1394 handle

node

target node ID

addr

address to read from

extcode

extended transaction code determining the lock operation

data

data part of lock parameters

arg

arg part of lock parameters

result

address where return value will be written

tag

data to identify the request to completion handler

Description

This function starts the specified lock request. The transaction is only started, no success of the transaction is implied with a successful return of this function. When the transaction completes, a `raw1394_loop_iterate` will call the tag handler and pass it the tag and error code of the transaction. *tag* should therefore be set to something that uniquely identifies this transaction (e.g. a struct pointer casted to unsigned long).

Returns

0 on success or -1 on failure (sets errno)

raw1394_start_lock64

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_start_lock64` — initiate a 64-bit compare-swap lock transaction

Synopsis

```
int raw1394_start_lock64 (raw1394handle_t handle, nodeid_t node, nodeaddr_t
addr, unsigned int extcode, octlet_t data, octlet_t arg, octlet_t * result,
unsigned long tag);
```

Arguments

handle

libraw1394 handle

node

target node ID

addr

address to read from

extcode

extended transaction code determining the lock operation

data

data part of lock parameters

arg

arg part of lock parameters

result

address where return value will be written

tag

data to identify the request to completion handler

Description

This function starts the specified lock request. The transaction is only started, no success of the transaction is implied with a successful return of this function. When the transaction completes, a `raw1394_loop_iterate` will call the tag handler and pass it the tag and error code of the transaction. *tag* should therefore be set to something that uniquely identifies this transaction (e.g. a struct pointer casted to unsigned long).

Returns

0 on success or -1 on failure (sets errno)

raw1394_start_async_stream

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_start_async_stream` — initiate asynchronous stream

Synopsis

```
int raw1394_start_async_stream (raw1394handle_t handle, unsigned int channel,
unsigned int tag, unsigned int sy, unsigned int speed, size_t length,
quadlet_t * data, unsigned long rawtag);
```

Arguments

handle

libraw1394 handle

channel

the isochronous channel number to send on

tag

data to be put into packet's tag field

sy

data to be put into packet's sy field

speed

speed at which to send

length

amount of data to send, in bytes

data

pointer to data to send

rawtag

data to identify the request to completion handler

Description

Passes custom tag. Use pointer to raw1394_reqhandle if you use the standard tag handler.

Returns

0 on success or -1 on failure (sets errno)

raw1394_start_async_send

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_start_async_send` — send an asynchronous packet

Synopsis

```
int raw1394_start_async_send (raw1394handle_t handle, size_t length, size_t
header_length, unsigned int expect_response, quadlet_t * data, unsigned long
rawtag);
```

Arguments

handle

libraw1394 handle

length

the amount of bytes of data to send

header_length

the number of bytes in the header

expect_response

indicate with a 0 or 1 whether to receive a completion event

data

pointer to data to send

rawtag

data to identify the request to completion handler

Description

This starts sending an arbitrary async packet. It gets an array of quadlets consisting of header and data (without CRC in between). Header information is always in machine byte order, data (data block as well as quadlet data in a read response for data quadlet) shall be in big endian byte order. expect_response indicates, if we expect a response (i.e. if we will get the tag back after the packet was sent or after a response arrived). length is the length of the complete packet (header_length + length of the data block). The main purpose of this function is to send responses for incoming transactions, that are handled by the application. Do not use that function, unless you really know, what you do! Sending corrupt packet may lead to weird results.

Returns

0 on success or -1 on failure (sets errno)

raw1394_read

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_read` — send async read request to a node and wait for response.

Synopsis

```
int raw1394_read (raw1394handle_t handle, nodeid_t node, nodeaddr_t addr,
size_t length, quadlet_t * buffer);
```

Arguments

handle

libraw1394 handle

node

target node ID

addr

address to read from

length

amount of bytes of data to read

buffer

pointer to buffer where data will be saved

Description

If *length* is 4, a quadlet read request is used.

This does the complete transaction and will return when it's finished. It will call raw1394_loop_iterate as often as necessary, return values of handlers called will be therefore lost.

Returns

0 on success or -1 on failure (sets errno)

raw1394_write

LINUX

Name

`raw1394_write` — send async write request to a node and wait for response.

Synopsis

```
int raw1394_write (raw1394handle_t handle, nodeid_t node, nodeaddr_t addr,
size_t length, quadlet_t * data);
```

Arguments

`handle`

libraw1394 handle

`node`

target node ID

`addr`

address to write to

`length`

amount of bytes of data to write

`data`

pointer to data to be sent

Description

If *`length`* is `4`, a quadlet write request is used.

This does the complete transaction and will return when it's finished. It will call `raw1394_loop_iterate` as often as necessary, return values of handlers called will be therefore lost.

Returns

0 on success or -1 on failure (sets errno)

raw1394_lock

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_lock — send 32-bit compare-swap lock request and wait for response.

Synopsis

```
int raw1394_lock (raw1394handle_t handle, nodeid_t node, nodeaddr_t addr,
unsigned int extcode, quadlet_t data, quadlet_t arg, quadlet_t * result);
```

Arguments

handle

libraw1394 handle

node

target node ID

addr

address to read from

extcode

extended transaction code determining the lock operation

data

data part of lock parameters

arg

arg part of lock parameters

result

address where return value will be written

Description

This does the complete transaction and will return when it's finished. It will call raw1394_loop_iterate as often as necessary, return values of handlers called will be therefore lost.

Returns

0 on success or -1 on failure (sets errno)

raw1394_lock64

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_lock64` — send 64-bit compare-swap lock request and wait for response.

Synopsis

```
int raw1394_lock64 (raw1394handle_t handle, nodeid_t node, nodeaddr_t addr,
unsigned int extcode, octlet_t data, octlet_t arg, octlet_t * result);
```

Arguments

handle

libraw1394 handle

node

target node ID

addr

address to read from

extcode

extended transaction code determining the lock operation

data

data part of lock parameters

arg

arg part of lock parameters

result

address where return value will be written

Description

This does the complete transaction and will return when it's finished. It will call `raw1394_loop_iterate` as often as necessary, return values of handlers called will be therefore lost.

Returns

0 on success or -1 on failure (sets errno)

raw1394_async_stream

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_async_stream` —

Synopsis

```
int raw1394_async_stream (raw1394handle_t handle, unsigned int channel,
unsigned int tag, unsigned int sy, unsigned int speed, size_t length,
quadlet_t * data);
```

Arguments

handle

libraw1394 handle

channel

the isochronous channel number to send on

tag

data to be put into packet's tag field

sy

data to be put into packet's sy field

speed

speed at which to send

length

amount of bytes of data to send

data

pointer to data to send

Returns

0 on success or -1 on failure (sets errno)

raw1394_async_send

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_async_send —

Synopsis

```
int raw1394_async_send (raw1394handle_t handle, size_t length, size_t
header_length, unsigned int expect_response, quadlet_t * data);
```

Arguments

handle

libraw1394 handle

length

the amount of bytes of data to send

header_length

the number of bytes in the header

expect_response

indicate with a 0 or 1 whether to receive a completion event

data

pointer to data to send

Returns

0 on success or -1 on failure (sets errno)

raw1394_start_fcp_listen

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_start_fcp_listen` — enable reception of FCP events

Synopsis

```
int raw1394_start_fcp_listen (raw1394handle_t handle);
```

Arguments

handle

libraw1394 handle

Description

FCP = Function Control Protocol (see IEC 61883-1) Enables the reception of FCP events (writes to the FCP_COMMAND or FCP_RESPONSE address ranges) on *handle*. FCP requests are then passed to the callback specified with `raw1394_set_fcp_handler`.

Returns

0 on success or -1 on failure (sets errno)

raw1394_stop_fcp_listen

LINUX

Kernel Hackers ManualAugust 2009

Name

`raw1394_stop_fcp_listen` — disable reception of FCP events

Synopsis

```
int raw1394_stop_fcp_listen (raw1394handle_t handle);
```

Arguments

`handle`

libraw1394 handle

Description

Stops the reception of FCP events (writes to the FCP_COMMAND or FCP_RESPONSE address ranges) on *`handle`*.

Returns

0 on success or -1 on failure (sets errno)

raw1394_get_libversion

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_get_libversion — Returns the version string

Synopsis

```
const char * raw1394_get_libversion ( void);
```

Arguments

void

no arguments

Description

Instead, typically, one uses 'pkg-config --mod-version libraw1394' Might be useful for an application.

Returns

a pointer to a string containing the version number

raw1394_update_config_rom

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_update_config_rom — updates the configuration ROM of a host

Synopsis

```
int raw1394_update_config_rom (raw1394handle_t handle, const quadlet_t *
new_rom, size_t size, unsigned char rom_version);
```

Arguments

handle

libraw1394 handle

new_rom

a pointer to the new ROM image

size

the size of the new ROM image in bytes

rom_version

the version numer of the current version, not the new

Description

rom_version must be the current version, otherwise it will fail with return value -1.

Returns

-1 (failure) if the version is incorrect, -2 (failure) if the new rom version is too big, or 0 for success

raw1394_get_config_rom

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_get_config_rom — reads the current version of the configuration ROM of a host

Synopsis

```
int raw1394_get_config_rom (raw1394handle_t handle, quadlet_t * buffer,
size_t buffersize, size_t * rom_size, unsigned char * rom_version);
```

Arguments

handle

libraw1394 handle

buffer

the memory address at which to store the copy of the ROM

buffersize

is the size of the buffer, *rom_size*

rom_size

upon successful return, contains the size of the ROM

rom_version

upon successful return, contains the version of the rom

Description

returns the size of the current rom image. *rom_version* is the version number of the fetched rom.

Return

-1 (failure) if the buffer was too small or 0 for success

raw1394_bandwidth_modify

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_bandwidth_modify — allocate or release bandwidth

Synopsis

```
int raw1394_bandwidth_modify (raw1394handle_t handle, unsigned int bandwidth,
enum raw1394_modify_mode mode);
```

Arguments

handle

a libraw1394 handle

bandwidth

IEEE 1394 Bandwidth Alloction Units

mode

whether to allocate or free

Description

Communicates with the isochronous resource manager.

Return

-1 for failure, 0 for success

raw1394_channel_modify

LINUX

Kernel Hackers ManualAugust 2009

Name

raw1394_channel_modify — allocate or release isochronous channel

Synopsis

```
int raw1394_channel_modify (raw1394handle_t handle, unsigned int channel,
enum raw1394_modify_mode mode);
```

Arguments

handle

a libraw1394 handle

channel

isochronous channel

mode

whether to allocate or free

Description

Communicates with the isochronous resource manager.

Return

-1 for failure, 0 for success