libraw1394

version 0.9

libraw1394: version 0.9

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 --FIXME--), then you already know most 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, from the slowest 100 Mbit/s over 200 and 400 Mbit/s up to 3.2 Gbit/s in the future (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 having a local address space with 48 bit wide addressing. 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 ID. The highest value for both is a special value. Bus ID equal to 1023 means "local bus" (the bus the node is connected to), node ID equal to 63 means "all nodes" (broadcast).

The whole bus can thus be seen as a linear 64 bit address space by concatenating the node address (most significant bits) and and node address (least significant bits). libraw1394 treats them separately in function arguments to save the application some fiddling with the bits. The node IDs are completely dynamic and determined during the bus reset.

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 node ID not yet in use (starting with 0) and send out a SelfID packet with its node 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 node ID itself, send SelfID packet and ack to parent.

After bus reset the used node IDs are in a sequential range with no holes starting from 0 with the root node having the highest ID. This also means that node 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 node 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. libraw1394 contains a function to convert ack/rcode pairs into errno codes which convey roughly the same meaning. This is done automatically for the synchronous read/write/lock wrapper functions, i.e. they return a negative value for failure and a standard error code can be found in the global variable errno.

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. 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 a certain (FIXME) amount of time. 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 subsystem consists of the 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 node (e.g. CSR registers). Below that are the hardware drivers, which handle converting packets and bus events to and from hardware accesses.

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 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()`.

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 (testing for read availability) together with the other loop's fd, some event loops also allow to add other fds to their own set (GTK's event loop does). If these trigger on the libraw1394 fd, you can call `raw1394_loop_iterate()` once and it is guaranteed that it will not block since at the very least one event waits. 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.

Chapter 4. Function Reference

raw1394_new_handle

Name

`raw1394_net_handle` — create new handle

Synopsis

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

Description

Creates and returns a new handle. 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.

Return Value

Returns the created handle or `NULL` when initialization fails. In the latter case `errno` either contains some OS specific error code or `0` if the error is that libraw1394 and raw1394 don't support each other's protocol versions.

raw1394_destroy_handle

Name

`raw1394_destroy_handle` — deallocate handle

Synopsis

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

Arguments

handle

handle to be deallocated

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_get_fd

Name

`raw1394_get_fd` — get the communication file descriptor

Synopsis

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

Arguments

`handle`

handle of which the fd is to be returned from

Description

Returns the fd used for communication with the raw1394 kernel module. 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 block behaviour in `raw1394_loop_iterate()`. It must not be used for anything else.

Return Value

The fd of the communication stream. Invalid fds may be returned before a port was set using `raw1394_set_port()`.

raw1394_(get|set)_userdata

Name

`raw1394_get_userdata, raw1394_set_userdata` — associate user data with a handle

Synopsis

```
void *raw1394_get_userdata(raw1394handle_t handle);
void raw1394_set_userdata(raw1394handle_t handle);
```

Arguments

handle

handle associated with the user data

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. This can be useful when multiple handles are used, so that callbacks can identify the handle.

Return Value

`raw1394_get_userdata()` returns the void pointer associated with the handle.