ZIO User Manual (version-1.0)

January 2013
A kernel framework for laboratory I/O

Alessandro Rubini and Federico Vaga for CERN (BE-CO-HT)

Table of Contents

Introduction

ZIO plans to be "the ultimate I/O framework". It is being developed on the open hardware repository at `http://www.ohwr.org/projects/zio`.

The framework is meant to offer a flexible interface for the development of input and output drivers for very-high-bandwidth devices with high-definition time stamps and a uniform meta-data representation.

The version at time of this writing is known to compile and run with kernels 2.6.32 onwards. We also maintain a *git* branch for 2.6.24, since this is the version we are running in some production laboratories.

1 General Concepts

While the design is pretty stable and we don't plan to introduce any serious change in the code that affects our users, there are some research ideas that we are evaluating and experimenting with. Thus, we want to define from the start both the words that we are using within the framework and the ideas that will be introduced at a later time.

1.1 Definitions

The ZIO framework is designed to move *data blocks*, or just *blocks* for short (not in italic from now on). A block is a sequence of zero or more data samples with associated meta-information. Blocks containing a single data sample are not expected to be common, as our use case is concerned with devices that input or output several thousand samples in a single shot, with a hardware-defined data rate and a single time-stamp marking the beginning of the event. Blocks containing zero data items are allowed, because they can be used to pass meta-information without associated data (e.g., TDC and DTC devices, described later).

A block can flow in either the input or output direction. The meta-information about a block is stored in a *control structure* or just *control* for short (again, not in italic in this document).

The ZIO code base is designed around three main items:

Device
: A ZIO device describes one specific I/O peripheral. See Section 6.3 [The Peripheral Driver], page 22.

Trigger
: The trigger is code concerned with arming the I/O event, in response to some event. Peripheral devices with no internal timing will input or output data when the trigger is armed; peripherals that are self-timed are supported by means of the default trigger, without the need to write custom trigger code for each of them. See Section 6.4 [The Trigger], page 26.

Buffer
: A buffer stores blocks, either input blocks generated by a trigger or output blocks generated by some injecting code. One end of the buffer is always connected to a trigger, and the other end is usually connected to a char device, though this is not mandated by the framework. See Section 6.5 [The Buffer], page 28.

The ZIO driver manages a *device*, which in turn may be a chip, a card or an external equipment connected through a field bus. The driver is the *probe unit*, i.e. a batch of I/O peripherals that are plugged and unplugged as a whole, usually based on some bus, like PCI, USB or SPI.

Devices include I/O channels of one or more type. A PCI board for example may have both analogue input and digital output, while an SPI chip will usually offer only one channel type. A ZIO device, thus, is described as a collection of *channel-sets*, or *csets* for short – again, in roman font throughout this document.

A cset is a group of channels usually associated with a wire connected to some backplane of the computer. All channels within a cset must be alike: same sample size and same configuration options (i.e., configuration values may vary across channels in the same cset, but all of them must feature the same set of parameters).

The channel is the basic I/O entity, usually associated with a single electrical connection on some backplane. The layers are displayed in Figure 1.1

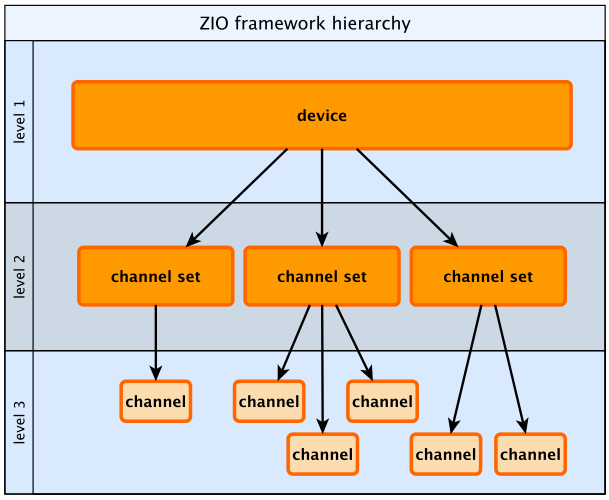


Figure 1.1

The cset is the most important object for ZIO devices. Each cset is associated with a trigger, and I/O events affects all channels in the cset – although you can disable individual channels as needed.

Each cset has a *current_trigger* attribute, which defines a trigger type; for each cset using that trigger type, ZIO creates an instance of the trigger. Each cset has a *current_buffer* attribute as well. ZIO creates a buffer instance for each channel in the cset. Thus, each channel owns a buffer instance, but of the same type across the cset.

Figure Figure 1.2 shows a cset, the trigger and buffer types it refers to an the instances it is using. A cset has one trigger instance overall and one buffer instance for each channel.

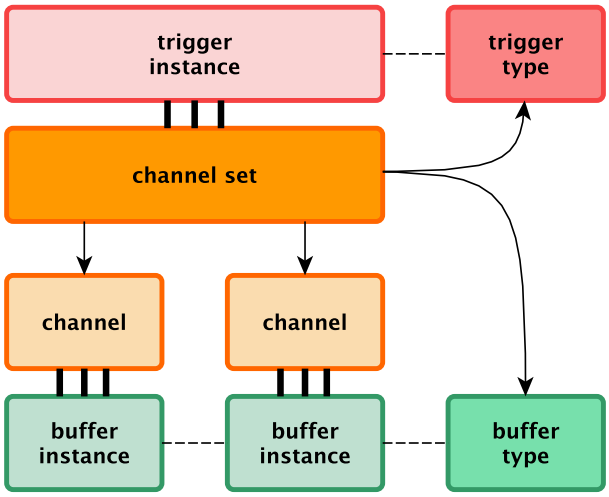


Figure 1.2

1.2 Supported Devices

This is a quick list of devices or device types that are supported in the current version of ZIO. They are mainly examples, because the real devices we are working with are developed as external packages. Such external packages include a 100Ms ADC, a Fine-Delay module and a TDC (both with 10ps resolution).

zero device
: The zero device is a software-driven input and output device, it is used for demonstration and stress-testing. It behaves like '`/dev/zero`', '`/dev/null`' and similar devices, but it inputs and outputs ZIO blocks.

loop device
: The device is a collection of csets and character devices connected in pairs. This is another software aid for stress testing and looking at the internals. Requiring no hardware, it's a good tool during development of the core. You can loop blocks from ZIO to ZIO, from ZIO to a char device or from a char device to ZIO.

mini device
: This is another software-only device we offer. It is designed to be as simple as possible, to help new developers who are learning how to write ZIO drivers. It registers one cset with one channel, by default, but module parameters allow to create several csets and several instances of the device. With it you can also stress-test the ZIO bus abstraction, and interaction with *udev*. Each channel simply returns one sample in each block, which represents a binary `struct timespec` that marks when the input event happened.

gpio device
: The gpio device acts on an array of input and output GPIO pins (up to 8 pins for input, up to 8 for output). Each cset has one channel only. The driver relies on the Linux GPIO abstraction, so it can run in any system that has registered GPIO pins. The configuration is set with module parameters.

SPI devices
: Some ADC devices connected to the SPI bus are supported. The driver supports AD7887 and AD7887 with a single source file; it relies on the Linux SPI subsystem.

uart device
: We are working on a *ldisc* device driver, which however is not included in this release yet. It is our test-bed for interleaved support, that will be included in later versions.

external drivers
: As said, there are a number of external device drivers that rely on the ZIO framework. They are not part of this distribution because they live in their own projects, and are quite specialized. We have at least a *fine-delay* module, a *TDC* and a fast *ADC*. What we and our mates wrote is available in `ohwr.org`, but there may be other drivers that we still ignore about.

This distribution also includes a few buffers and triggers. They are not listed here as they are not properly device drivers. The buffers are generic RAM-based buffers (we offer *kmalloc* and *vmalloc*, the latter supports *mmap* as well), and the triggers are a set of generic software-driven triggers: kernel timer (periodic), high-resolution timer (both periodic and one-shot), external-interrupt, and the transparent trigger which is selected by default. With the transparent trigger, I/O is driven by either the user who initiates read and writes, or by the device itself if it claims to be self-timed.

For more details see Section 7.2 [Available Buffers], page 30 and Section 7.1 [Available Triggers], page 29.

1.3 Future Developments

Before going to the details of what is implemented now, we'd like to list the ideas we are working on, because they will reach the ZIO framework at some time in the future. Some of them are already in *beta* stage, available from *git* branches in our `ohwr` repository.

- Supporting an `input_device` buffer type

  A buffer that injects input events to the Linux input subsystem, without requiring any change in other ZIO modules.

- Writing a new PF_ZIO network type.

  The protocol family `PF_ZIO` allows to see the whole set of I/O peripheral devices as a network. Such network can be internal to a single host, or it can span multiple hosts by exchanging frames of type `ETH_P_ZIO`. With `PF_ZIO` each channel has a network address, with the *control* acting as link-level protocol header. This allows applications to open only a single socket (or a few of them) instead of one or two char devices for each channel being used. This is already available, and is being stabilized. We presented it at the *Embedded Linux Conference Europe* in November 2012.

- Interleaved channels

  A number of high-throughput peripherals interleave data from their channels. We have no support for interleaved I/O in the master branch, but the 100Ms ADC driver already uses interleaving internally. We are working to add support for this in ZIO, in a portable and future-proof way. The implementation will rely on TLV records (see Section 4.2.3 [TLV in the Control], page 15).

- Defining the "zio_interface" data structure.

  An *interface* layer may abstract char device support out of the ZIO core, so users will be able to use different interfaces; with interfaces, the PF_ZIO protocol or a link with the input subsystem can be a first-level alternative to char devices instead of being a buffer type that doesn't prevent creation of char devices.

- Generalized timestamping.

  We are soon going to add capabilities to timestamp the internal ZIO pipeline. This allows to measure the overhead of the pipeline (driver, trigger and buffer) independently of which objects are being used – so you can timestamp during production as well. The feature will be a compile-time option, that can be enabled with a *sysfs* attribute on individual csets.

2 The ZIO Pipeline

During input and output operations, ZIO blocks travel across a pipeline of software modules. Hoping to make the representation clearer, we chose to stick to a consistent mnemonic color model for our figures. Unfortunately, some older figures still need to be converted to this color model, and we apologize for this.

The color model for all of our objects is shown in Figure 2.1.

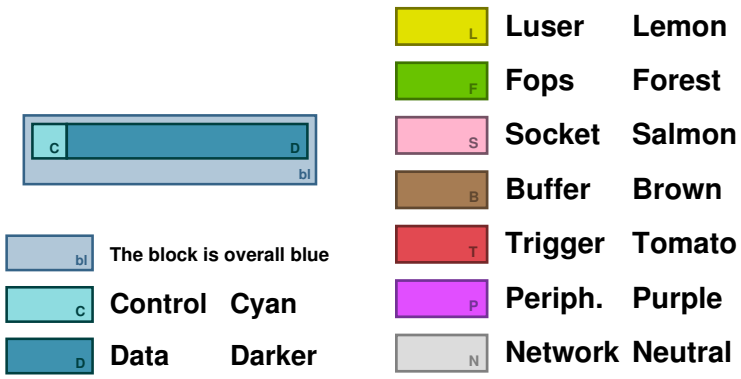


Figure 2.1

The ZIO pipeline connects user space with the peripheral device. The main objects involved are the file operations in the character device, the buffer where blocks are stored, the trigger that sequences the actual data transfers and the peripheral driver that is concerned with hardware operations when the trigger fires.

The whole pipeline is depicted in the Figure 2.2, where the path of the blue blocks is shown for both input and output. The black arrows represent function calls.

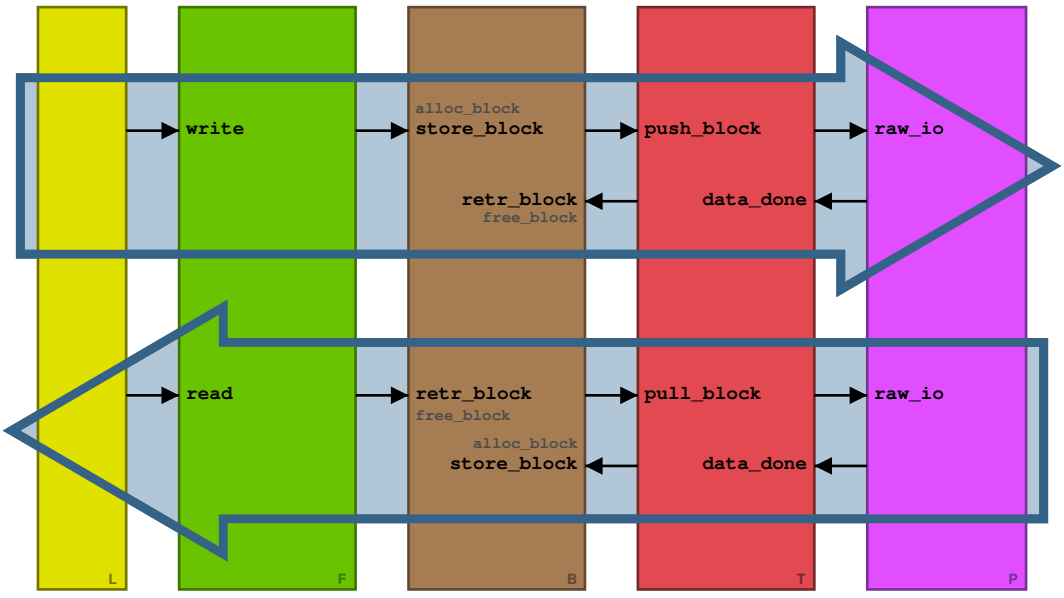


Figure 2.2

User space makes system calls, which are implemented by the `file_operations` in the current buffer. Generic functions are implemented in `chardev.c` and exported as `zio_generic_file_operations` (currently all buffers included in the distribution are using these operations). This may slightly change in the future if we introduce the *interface* concept, but the change will not

affect buffers besides the removal of their `f_op` field. See Section 1.3 [Future Developments], page 4 about the *interface* idea.

The file operations code calls `store_block` and `retr_block` in the proper buffer instance, the buffer in turn communicates with the trigger, and the trigger refers to the device for the raw data transfer.

2.1 All the Steps in the Pipeline

This sections details all the steps a block takes from user space to the device and back. You should get acquainted with the various step when writing a ZIO module or debugging it.

When you load the device driver, each of its csets receives an instance of a trigger, and each channel in the cset receives an instance of a buffer; the preferred trigger type and the preferred buffer type are selected by the cset or the device, otherwise default ones are used. When the user changes the trigger or buffer type for a cset, the operation is atomic: it either succeeds or it fails. This design ensures that all steps of the pipeline are always present; we always use `bi` as name for the pointer to the current *buffer instance* and `ti` as name for the pointer to the current *trigger instance*.

When data is flowing, operation of the pipeline is quite straightforward, but there are always some corner cases to be dealt with, so the description below is sometimes longer than expected, but it covers all cases.

Let's start with the input direction:

- When a user program calls *read*, *poll* or *select* on a ZIO character device, it means it is interested in data. The code in the ZIO file operation calls `bi->retr_block`, which may succeed or fail. If it fails, the device is reported as not readable; the process may be added to a wait queue or not, according to standard Unix semantics. If `retr_block` succeeds, the block just retrieved is managed by the *file operations* code until it is exhausted and eventually freed by calling `bi->free_block`.
- The `retr_block` method lives in the buffer. If the buffer is already hosting at least one block, it returns the first block in its FIFO structures, and the loop closes. If not, the function checks `ti->pull_block`. If the pointer is NULL, the buffer can only wait for the trigger to autonomously produce a block; otherwise it calls the function to request an input event from the trigger.
- Most trigger types do not implement `pull_block`: a time-driven trigger or an event-driven trigger fire by themselves, whether or not the user is reading. But if the trigger implements the method, then it means it wants to know when the user is asking for input data: such a trigger may thus return an incomplete block, or just use *that* event to fire acquisition. In any case, `pull_block` does not return anything, it just informs the trigger about the request.
- The trigger is involved with driving actual input events, in any way it wants. At some time, or at some event, it knows it wants input to happen, so we say the trigger is *armed*. When the trigger is armed, all enabled channels in the cset should have a valid *active block*, allocated by the trigger. (Actually, allocation may fail, so some channels may have no active block when the trigger fires). Then, the trigger notifies the event to the relevant cset by calling `device->raw_io(cset)`. The *transparent* trigger arms the device at `pull_block` time, and each other trigger in its own specific way.
- After the trigger is *armed*, the device is in charge. If the device is self-timed (like a TDC or another data-driven device), the trigger can remain armed for a while. If the device has no internal timing, actual input begins immediately. Input may complete synchronously or asynchronously; for example reading GPIO pins is synchronous, but if the device initiates DMA it needs to wait for an interrupt that happens at a later time. The `raw_io` device

method returns 0 if input completed synchronously, and it returns `-EAGAIN` to report that data will be ready at a later time. In the former case the system knows the trigger has fired, in the latter case ZIO knows the trigger is still armed for this cset. Return values other than 0 and `-EAGAIN` represent a real error, and ZIO knows the trigger is not armed.

- At some time in the future, the device that reported `-EAGAIN` is done with the input operation. It thus calls `zio_trigger_data_done()` (that may rely on *ti->data_done* if the trigger defines it). When this happens (or if `raw_io` completed synchronously), the respective trigger changes its status to un-armed; the active block for each channel in the cset is stored to the respective buffer instance, by calling *bi->store_block*. According to its own semantics, the trigger can re-arm the trigger immediately, or it may not.
- At *data-done* time, ZIO raises an *alarm* bit for each channel that is enabled by has no active block. The alarm is persistent until cleared by writing to *sysfs*.
- The buffer, in its `store_block` method, enqueues the block. If there are already other blocks in the buffer, the loop is over. If, instead, this is the first block, the buffer awakes its own wait queue, because there may be a user-space program waiting for such data.

The sequence of events for output is similar, but not in every detail:

- When a user program calls *write* on a ZIO character device, the data builds up into a block, which has been allocated by calling `bi->alloc_block`. Select for writing is supported as well, for processes that need it: when allocation of a block fails the char device is reported as not writable, and the process may sleep on a wait queue, according to Unix semantics. The details of how the control and the data are filled are not described here (see Section 5.1 [Details of Char Device Policies], page 18). When the block is full, the code calls `bi->store_block`, that never fails.
- The buffer method `store_block` takes hold of the new block. If there are already blocks in its FIFO structure, the new block is enqueued and nothing more is done. If this is the first block, the buffer calls `ti->push_block`. If the push succeeds, the buffer remains empty (and will try *push* again next time). If the push fails, the block is enqueued.
- As said, the trigger may accept or refuse the push. In general, it should refuse the push if it already is getting hold of a block for the current channel, and it should accept the push if no block is pending. A double-buffering trigger can be implemented by properly writing *push*.
- After accepting the push, the trigger may, or may not, arm the trigger, according to its own trigger policies. For example, the transparent trigger only arms when every enabled channel has an active block – because the output event happens for the whole cset at once.
- When the trigger module wants output to be performed, it turns into *armed* state, and calls `device->raw_io(cset)`. Similarly to the input case described earlier, this method can perform I/O synchronously and return 0 to mean "success", or it can perform I/O asynchronously and return `-EAGAIN`. In the latter case the trigger remains in *armed* state. Other return values are considered errors, and the trigger is not considered as being armed.
- At some time in the future, the device that reported `-EAGAIN` is done with hardware output, so it calls `zio_trigger_data_done()` (that may rely on *ti->data_done*). When this happens (or if `raw_io` completed synchronously), the trigger instance becomes un-armed again, and it frees the blocks for the cset by calling `bi->free_block`. In general, the trigger should call `bi->retr_block` in order to be ready to re-arm the trigger.
- If the buffer is empty, it is not able to satisfy the `retr_block` request. As a consequence, the next time this buffer receives a block from user space it calls `ti->push` to restart a loop that has been broken. If the buffer was completely full, this `retr_block` makes some space available, and the buffer awakes the associated wait queue.

We think the overall flow is pretty simple: we offer *pull* and *push* to initiate a new data flow or restart one that stopped for lack of data, but the normal flow involves just *store_block* and *retr_block* for the buffers, *raw_io* and *data_done* for the triggers.

The output design, in particular, is similar to what happens with network devices. A network card accepts pushes by the kernel (the method is called *hard_start_xmit*) until its internal buffers are full; at this point it calls *netif_stop_queue*, and when space is available again it calls *netif_start_queue*.

ZIO output is similar to network transmission, but the ZIO core knows that the trigger can accept only one or two blocks at most; so we chose to not require the trigger to call *stop* and *start* in the buffer, providing instead a clean failure mode for the push, so to reuse the same *retr_block* that is used for the input flow.

2.2 Lifetime of a Data Block

Figure 2.3 and Figure 2.4 show when blocks are allocated and freed in the input and output data paths. Time in both figures is flowing downwards.

The pipeline for output is similar to the input one, but the block is allocated by the implementation of the *write* system call. Note in particular how the buffer works in exactly the same way as in the input case: it just servers `store_block` and `retr_block` requests.

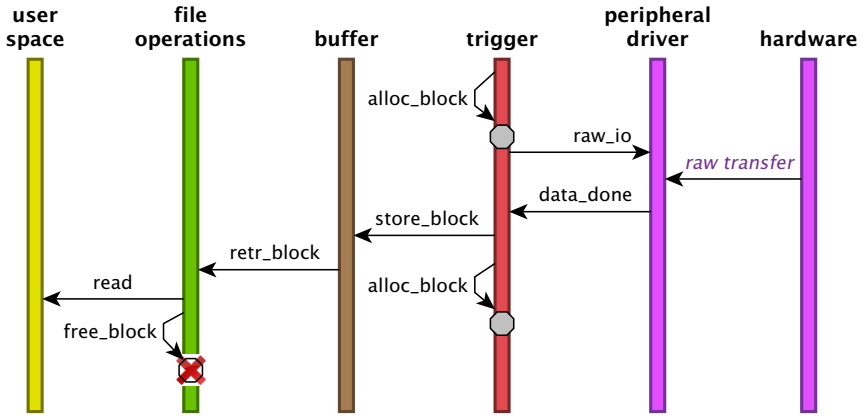


Figure 2.3

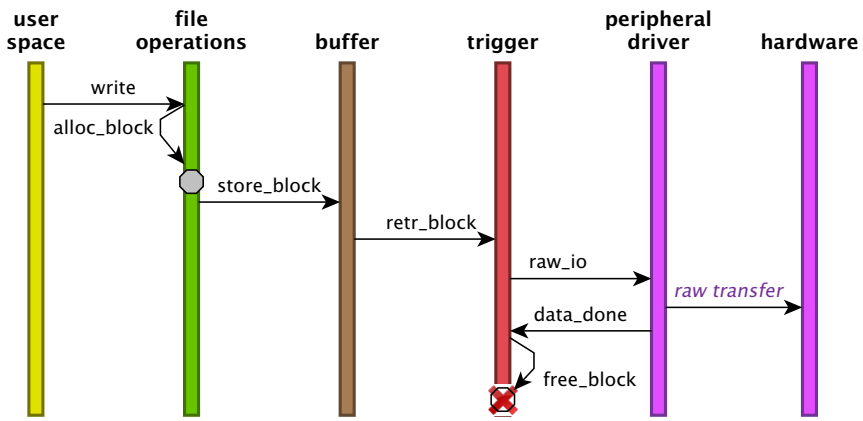


Figure 2.4

3 The Bus Abstraction

The ZIO core module is called `zio.ko` and it creates a new *bus* item in Linux. A *bus* is a software abstraction for kernel-related software modules; it splits the role of the *device* from the role of the *driver*. In order to have a new peripheral working in your system you thus need both items: the *driver* is in charge of any *device* that appears in the system, while the *device* is a data structure that describes the parameters of the specific hardware instance. The two structures are bound by calling a *match* function, which is at the core of the bus abstraction. If the device and the driver match, the driver is asked to manage the new device instance.

Even if no physical *bus* is involved with ZIO, by relying on this software abstraction we are able to deal with several devices of the same type. Our use case involves control systems where more than one instance of the same PCI card is installed in each host. Actually, by distributing our hostless I/O devices using a field-bus, we foresee the need to register up to a few hundred ZIO devices, to be driven by the same driver.

To make an example, let's see how to set up a ZIO module driving a PCI I/O card. Since PCI is a *bus* in Linux, you'll need to register a PCI *driver* to hook to any card present in the system (in this case the associated *device* is created by the PCI controller, when it scan the physical bus). The *probe* function of your PCI driver, then, registers a ZIO *device*. In addition to registering a PCI *driver*, your kernel module also needs to register a ZIO *driver* for your hardware, so that Linux calls its *probe* function for every ZIO *device* that matches such driver.

The ZIO bus is just a software abstraction, so its own *match* function is just comparing the two names: if a device and a driver register the same name string, they match. This is exactly what happens with the *platform bus*, a software-only bus implementation used in every Linux system to take care of peripherals that are directly connected to the computer system, without relying on an enumerated bus.

So, in general, in order to create a ZIO peripheral, you need to register both a *device* and a *driver*, sharing the same name. This is clearly visible in simple modules like `zio-zero` and `zio-gpio`, where the *init* function of the module registers both data structures. Nothing prevents another kernel module from registering a new instance of the *zero* or *gpio* device (or both).

You can experiment with this concept using `zio-mini`, which actually registers more than one *device* associated to its own *driver*.

4 ZIO data model

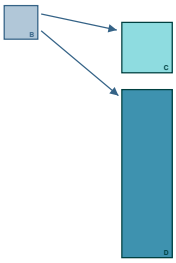
This chapter defines in detail the data model of ZIO. The unconventional data model is the main idea behind the ZIO project, together with the use of a full-featured *bus* abstraction.

4.1 The Block

In the typical ZIO use case we handle several samples at the same time. We need to manage thousands or even millions of samples at the same time, because I/O and time-stamping is performed at hardware level (within an FPGA) while the Linux driver is only concerned with passing data around.

For this reason ZIO defines a structure called *block*. All data transfers within ZIO happen in the context of a block, which is its own atomic data item. A block includes both the actual data and the *meta-data* associated with the input or output event. Inside ZIO, no data ever travels without associated metadata (even if the user is allowed to ignore meta-data and only read or write data). This design choice allows I/O data to be transferred through payload-agnostic transport mechanisms: only the endpoints need to agree about what the actual payload is. More on this later.

The block is made up of two main parts: the data itself, called simply *data* throughout the ZIO framework (including this manual) and the meta-data, called *control* or *ctrl* from now on. The block can be depicted like this:



At C-language level, the block structure is defined in `zio-buffer.h`, because buffers are the main ZIO object involved with storage of blocks. This is the actual definition:

```
struct zio_block {
        unsigned long           ctrl_flags;
        void                    *data;
        size_t                  datalen;
        size_t                  uoff;
};
```

The meaning of the fields is as follows:

`ctrl_flags`
: The field includes a pointer to the control structure and one flag bit, described below.

`data`
: The pointer to actual data associated with this block. May be NULL for TDC or DTC devices.

`datalen`
: The length of the memory area where *data* points. May be zero for TDC or DTC devices.

`uoff`
: *User offset.* This field is used to track partial data, while the block is produced or consumed. This happens, for example, when the user accesses a byte-oriented interface like a char device. The peripheral driver can use the field to track partial data at its own end of the pipeline, but for input blocks the field must be zeroed before storing the block into a buffer.

The flag bit hidden in *ctrl_data* is called `cdone`, and is the counterpart of *uoff*: when a block is being produced or consumed we need to track the status of the control. Unlike data, the control is atomic: no partial controls ever exits, and one bit is enough to track its status within a block.

We chose to coalesce the pointer to control and the `cdone` bit in a single filed in order to avoid wasting bytes and/or break alignment of the structure. The same trick is used in the *rbtree* implementation within the Linux kernel, and it is pretty efficient.

`zio-buffer.h` defines the following macros to access parts of the `ctrl_flags` field:

```
#define zio_get_ctrl(block) ((struct zio_control *)((block)->ctrl_flags & ~1))
#define zio_set_ctrl(block, ctrl) ((block)->ctrl_flags = (unsigned long)(ctrl))
#define zio_is_cdone(block)  ((block)->ctrl_flags & 1)
#define zio_set_cdone(block)  ((block)->ctrl_flags |= 1)
```

As you expected, `ctrl` is a pointer and `cdone` is a single bit. A typical user of this is the *read* file operation. See Chapter 5 [Accessing ZIO from User Space], page 17.

4.2 The Control Structure

The *control* is the container of meta-information used to describe a block of data. Its size is fixed to 512 bytes, with the optional addition of TLV records at the end. Such *type-length-value* records are not used in the current implementation, but they are designed to introduce no incompatibilities on ZIO modules developed externally, when we'll add them to the core. More information about this is in Section 4.2.3 [TLV in the Control], page 15.

The layout of the control is shown in Figure 4.1.

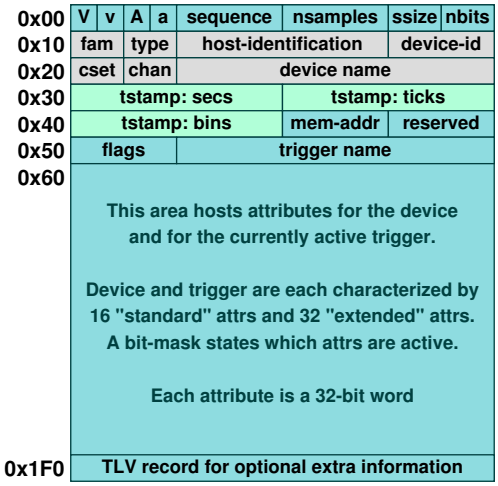


Figure 4.1

The control structure is fixed both to simplify communication with user space and to support generic applications. Such generic tools may perform monitoring or other operations without the need to know device internals. A similar approach is used by the *event* devices in the input subsystem of the Linux kernel.

The control is designed to offer the full meta-information needed to describe a block. In includes the unique global identification of the channel, as well as the name and parameters for the current trigger. Applications can thus pass around a block without knowing what it is; knowledge about device and trigger details can be concentrated in a single place, without the need to spread information to all actors in the input/output pipeline.

By using a control structure (i.e., zio blocks in their entirety) ZIO users can perform both offline data generation for output channels and offline data analysis for input channels. A program may prepare an output waveform in advance and ask generic tools to deliver it to the device; similarly, acquisition may be performed by generic tools that concentrate data from a set if I/O computers to a data center where programs that know the hardware can digest all data blocks.

The control structure is defined in `zio-user.h` because it must be accessed by both kernel and user space. The definition includes a few sub-structures, to ease logical grouping of information. This is the current definition of the control block, version 1.0:

```
struct zio_control {
        uint8_t major_version;
        uint8_t minor_version;
        uint8_t zio_alarms;     /* alarms are persistent, until somebody */
        uint8_t drv_alarms;     /* clears them writing to a sysfs attribute */

        /* byte 4*/
        uint32_t seq_num;       /* block sequence number */
        uint32_t nsamples;      /* number of samples in this data block */
```

```
        /* byte 12 */
        uint16_t ssize;         /* sample-size for each of them, in bytes */
        uint16_t nbits;         /* sample-bits: number of valid bits */

        /* byte 16 */
        struct zio_addr addr;

        /* byte 48 */
        struct zio_timestamp tstamp;

        /* byte 72 */
        uint32_t mem_offset;    /* position in mmap buffer of this block */
        uint32_t reserved;      /* possibly another offset, or space for 64b */
        uint32_t flags;         /* endianness etc */

        /* byte 84 */
        /* Each data block is associated with a trigger and its features */
        char triggername[ZIO_OBJ_NAME_LEN];

        /* byte 96 */
        struct zio_ctrl_attr attr_channel;
        struct zio_ctrl_attr attr_trigger;

        /* byte 496 */
        struct zio_tlv tlv[1];
        /* byte 512: we are done */
};
```

This is the meaning of all the fields. Please note that some of the fields are not yet being filled by the core.

`major_version`
`minor_version`
: The version is currently 1.0, as defined by `ZIO_MAJOR_VERSION` and `ZIO_MINOR_VERSION` (in `zio.h`). The version is used to ensure all actors agree on the meaning of the fields. We warrant that all minor version changes will be compatible (for example, 1.1 may have new flags that can be ignored by older code). Any incompatible change will force an upgrade of the major number, but we plan no such change in the foreseeable future – we had enough experience with the 0.x versions already, and we are confident we covered all needs that may arise in the future.

`zio_alarms`
`drv_alarms`
: Two masks of active alarms. Alarms can be set by any module in the pipeline, and are persistent until cleared by means of a *sysfs* attribute. ZIO alarms are defined by the core, and we currently support "lost block" and "lost trigger".

`seq_num`
: Block sequence number. It starts at 0 when a channel is created, and it is incremented at each I/O event (so the first block is number 1. The value 0 is reserved as a signal that the sequence number is not used by the entity that generated the block (e.g., user space, for output blocks).

`nsamples`
`ssize`
`nbits`
: Number of samples, sample size in bytes, and number of bits in each sample. The fields are used to describe the data itself. Sample size and sample bits are both needed because we may have something like 5-bit samples aligned in 32-bit words.

The number of bits is a device attribute: drivers may allow to change them at run-time. The sample size is however immutable for the lifetime of a cset. An input driver can reduce `nsamples` in the current control, in order to return a shorter block than requested (the value is re-initialized every time a trigger is armed).

`addr`

The address is a 32-byte data structure that allows to uniquely identify the channel associated to this data block. It is described separately in Section 4.2.1 [ZIO Address], page 14.

`struct zio_timestamp tstamp`

The timestamp associated with this input or output event. For input, the trigger must fill it, and it may be software-generated or hardware-generated. For output, some triggers use it and some don't (for example, and external-irq trigger won't use pre-set timestamps, but TDC devices use it to fire output pulses at specified times). The internals of this structure are defined later, in Section 4.2.2 [The Time Stamp], page 15.

`mem_offset`

If the buffer supports *mmap*, this is the memory offset of the data in the buffer's storage area. Please note that the *data* pointer in the block structure is still valid: kernel users should refer to `block->data`, while user space can use `ctrl->mem_offset` while relying on *mmap*. We have no clear ruses, yet, about how to use of *mmap* for output.

`flags`

The flags specify features of the data block. See below for currently defined flags.

`triggername`

Name of the current trigger. For input, the name is filled by ZIO, for output, ZIO ignores it (so it can be left blank). To change the trigger you should write to `current_trigger` in *sysfs*.

`attr_channel`
`attr_trigger`

Attributes for the channel and the trigger. Each structure is 200 bytes long and includes both standard and extended attributes. The attributes are described in Section 6.6 [The Attributes], page 29.

`tlv`

The TLV record is normally zeroed. Special csets may need more information than what the standard control can host, and in this case this field is used. See Section 4.2.3 [TLV in the Control], page 15 for more information.

The size of the control structure is exactly 512 bytes, as defined at the inception of the ZIO project, and this is verified by a compile-time check, to ensure nobody changes this in error while working on a new version. Actually, the check already triggered during development: a proposed layout failed because of different alignment constraints on different architectures. In the future, we'll support control structures bigger than 512 bytes with no incompatibilities in already-written code. See Section 4.2.3 [TLV in the Control], page 15.

We currently define the following flags, in `zio-user.h`:

`ZIO_CONTROL_LITTLE_ENDIAN`
`ZIO_CONTROL_BIG_ENDIAN`

These are used to identify how data is written in this block. For input channels, data is produced in native endianness; for output channels the applications must

convert everything to native endianness (i.e., ZIO doesn't fix data in kernel space). Both these flags are endian-agnostic, so a endian-aware application may read the `flags` field as a 32-bit integer, and apply endian conversion to all other fields if needed.

`ZIO_CONTROL_MSB_ALIGN`
`ZIO_CONTROL_LSB_ALIGN`

The flags specify where the active sample bits (`nbits`) are placed within the bytes of a sample (`ssize`). Both flags are not 0; if neither is set the alignment is unspecified.

4.2.1 ZIO Address

The `zio_addr` structure is designed to uniquely identify a channel. Its main purpose is to allow generalized routing of the blocks to the output channels, as well as to trace the source of any input data block.

The chosen layout for `zio_addr` is matching `sockaddr_zio`, the addressing structure used for the `PF_ZIO` implementation, a network protocol to route I/O blocks.

The address of each channel is also available as a binary file in *sysfs*, called `address`.

This is the definition of `zio_addr`:

```
struct zio_addr {
        uint16_t sa_family;
        uint8_t host_type;      /* 0 == local, 1 == MAC, ... */
        uint8_t filler;
        uint8_t hostid[8];      /* MAC or other info */
        uint32_t dev_id;        /* Driver-specific id */
        uint16_t cset;          /* index of channel-set within device */
        uint16_t chan;          /* index of channel within cset */
        char devname[ZIO_OBJ_NAME_LEN];
};
```

The fields have the following meaning:

`sa_family`

Currently unused in the control, this is meant to be `AF_ZIO` (in network byte order) in all contexts where this structure is called `sockaddr_zio`. By having the same layout in the control block we simplify conversions between the I/O world and the networking world.

`host_type`
`filler`
`hostid`

The first two bytes are used as *class* identifiers for the following `hostid`. Currently we only support type 0, which means *local delivery*; all other bytes are expected to be 0 as well. When `PF_ZIO` will route frames over Ethernet, we'll be able to use `hostid` as a MAC address. We chose to use one byte only for `host_type` to avoid endianness problems; if needed, `filler` can be used as a subclass for some of the future host types.

`dev_id`
`cset`
`chan`
`devname`

These four fields allow to uniquely identify the channel within the current host. The `devname` is a 12-byte string that identifies the driver producing or consuming data; `dev_id` is a driver-specific field used to identify different instances of the same driver, `cset` and `chan` are the indexes of cset and channel within the device.

The `dev_id` field is used differently by different drivers: software-only drivers like `zio-zero` and `zio-mini` just count starting from zero, while PCI cards use the *bus* and *defvn* identifiers as geographical numbering.

For input, all fields are set by ZIO; for output they are ignored if data reaches ZIO through char devices. The network interface of the framework will use the fields to route data to the channel associated with the output block.

4.2.2 The Time Stamp

Time stamps in ZIO are represented by `struct zio_timestamp`, defined in `zio-user.h` because it is shared with user space. It is is made of three 64-bit fields:

```
struct zio_timestamp {
        uint64_t secs;
        uint64_t ticks;
        uint64_t bins;
};
```

The meaning of the three fields is channel-specific, in order to cater for any hardware requirements without imposing conversions in kernel space. We have, however, some suggestions for use of the fields in a consistent way:

`secs`

The field should be used to host a TAI value, i.e. the number of seconds since Jan 1, 1970. The field is 64-bits wide to preserve alignment, to survive 2038 and to allow for a different choice of the Epoch, if needed. When ZIO uses software timestamping, the field hosts the `tv_sec` value of a `struct timespec`.

`ticks`

The field should be used to host a nanosecond count, if this choice makes sense for the hardware at hand. If you need to use a scalar nanosecond value, without a separate *seconds* component, this is where to host the value, leaving `secs` zeroed.

`bins`

This field should be used for any high-precision number as used in the hardware. It may also be the only non-zero field, for example if the hardware timestamp is taken from a custom counter unrelated to the standard Epoch.

For example, *White Rabbit* devices use hardware timestamps made up of three 32-bit values: seconds, nanoseconds with a granularity of 8ns and phase offsets as 12-bit fractional bit withing the 8ns tick. In this case, ZIO timestamps will store the seconds in `secs`, the nanoseconds in `ticks`, as a multiple of 8, and the phase in the `bins` field. This setup allows WR-unaware ZIO users to still get most of the information, while our application can combine *ticks* and *bins* to get the full resolution provided by the hardware.

4.2.3 TLV in the control

The control structure already includes a TLV record at the end. This is currently unused, but we'll need to use TLV records in future versions. This section describes our plans and how (actually, how little) the introduction of TLV will affect developers of external ZIO modules.

The need for extending the control comes from interleaved acquisition. Some input or output devices exchange buffers where data is interleaved between the channels, but each channel has its own attributes, such as gain and offset. We may use a cset with a single interleaved channel, but this has two problems: on one side 32 device attributes may not be enough for all channels, and on the other we need a way to describe how physical channels are mapped to the interleaved data set.

The initial idea was to use several control structures attached to a single block, one control per interleaved channel. Actually, moving around 512 bytes to convey what may be just one attribute (4 bytes) seems overkill. Besides, being restricted to the "control" as it is now is not future-proof.

After serious discussion considering real use cases, we found that the best approach is extending the control with a TLV structure. This gives flexibility and generality at the same time: any user who ignores some type number can just use the length to skip over the data (but still carry everything to others if the specific user is only one step in a communication channel).

Thus, the last 16 bytes of the control, which were unused before version 1.0 of the data structure, are the first *lump* of a TLV chain.

A TLV record is defined like this, in `zio-user.h`:

```
struct zio_tlv {
        uint32_t type;          /* low-half is globally assigned */
        uint32_t length;        /* number of lumps, including this one */
        uint8_t payload[8];
};
```

While 32 bits for the type and length may seem overkill, the choice is meant to ensure the payload is 8-byte aligned, to prevent inconsistencies between 32-bit and 64-bit hosts.

The *type* is split into globally-assigned and locally-assigned numbers. We reserve to define new types in the low half of the range in future versions of ZIO, while our users are free to use any new TLV they need in passing special meta-data throughout the ZIO pipeline.

Type 0 is the terminator. All TLV sequences must end with a *lump* of 16 zeroes. The unextended control, with its trailing zeroes, is thus already TLV-compliant.

Type 1 is "read more". Its length is always 1 lump and its content is a 32-bit count of how many bytes of TLV follow after this lump (the remaining 4 bytes are unused). Thus, when a control structure is augmented with trailing TLV data, it must feature this *read more* lump. The consumer of the control is thus able to read all the trailing data with a single system call.

Effect on device drivers

When we'll introduce TLV support in ZIO core, we'll simply add a *controlsize* field to the cset structure. So ZIO drivers can state the overall size of the control and exchange extra metadata with user space. If you are not using this feature your code will not be affected and will work perfectly after recompilation under the new ZIO data structures.

Current drivers will need no modification, because the core knows a zeroed *controlsize* field means 512 bytes.

Buffer modules won't be affected, as long as they use the provided ZIO helpers to allocate the control structure. If buffer modules need to allocate the control in a special way, they should call `zio_control_size(cset)` to be future-proof. This for example applies to the `PF_ZIO` buffer, which needs to store the control inside the socket buffer it allocates through the network subsystem.

Generic trigger drivers won't be affected because they only look at trigger attributes in the base control.

Effect on user space

User space code designed to work with a specific driver won't be affected by the change, because the control size for its own driver is not going to change. Current code will continue working as well as future code that doesn't need to use TLV records. Needless to say, user-space code designed to work with a device that uses TLV must be aware of such TLV records.

On the other hand, generic user space tools that use the ZIO abstraction to work with any I/O peripherals need to take care of TLV extensions using the suggested ZIO protocol. Such protocol already works with the unextended control. Clearly, generic code that ignores metadata and only manages data, is not affected.

For the input direction, generic code that reads a ZIO control device must work like this:

- Read 512 bytes (`sizeof zio_control`);
- Check the first TLV lump, which is guaranteed to be type 0 or 1;
- If it is 1 (*read-more*), read the whole trailing part of the control.

For the output direction, generic code receives metadata from elsewhere and sends it to the device. If metadata and data are concatenated, because they both are streamed through some other communication mechanism, generic code needs to follow the same procedure described above.

This is how parser code should be implemented. But, again, please remember that most of the time your control will be unextended and you can just ignore TLV extensions:

```
struct zio_tlv *tlv = ctrl->tlv;

while (tlv->type != 0) {
      switch(tlv->type) {
      case ZIO_TLV_READ_MORE: /* 1 */
            read( <source>, tlv + 1, tlv->payload32);
            break;

      case ....
      }
      tlv += tlv->length;
}
```

The code above assumes native endianness, which always applies when working on the host where devices live. If you scan remote blocks, you should convert all values using the endianness information from `ctrl->flags`.

4.3 The Data

The data pointed-to by a control structure is just opaque payload for ZIO. For output, only the device driver needs to make sense of it (or it may just pass it hardware without ever knowing what it is); for input, the final destination of the data will use it according to the device/cset/channel it originated from, as well as the information about sample size, bits and alignment that is found in the control structure.

In practice, only the endpoints of the pipeline need to know what the data is (usually relying on the metadata provided in the control structure. Everything else in ZIO works with blocks without knowing what is the payload at hand.

5 Accessing ZIO from User Space

ZIO transfers blocks to and from user space using char devices. Each channel is associated with two char devices: one for data and one for metadata.

An alternative implementation, currently in beta stage, is `PF_ZIO`: a protocol family that uses a specific buffer type (in the module `zio-buf-socket.ko`) to offer I/O capabilities through sockets. This is the work of Simone Nellaga, and will be integrated in the master branch of the ZIO repository as soon as it is stabilized.

5.1 Details of Char Device Policies

The default user-space interface of ZIO is based on character devices. The framework registers two devices for each channel: one is used to exchange the control, and the other is used to exchange the data. The name of the device is automatically generated from the name of the driver, its *device_id* field, the cset number and the channel number.

For example, cset 0 of the *zzero* device has three channels, and they appear as follows:

```
spusa.root# ls -l /dev/zio/zzero-0000-0-*
crw------- 1 root root 250, 0 Nov 30 13:12 /dev/zio/zzero-0000-0-0-ctrl
crw------- 1 root root 250, 1 Nov 30 13:12 /dev/zio/zzero-0000-0-0-data
crw------- 1 root root 250, 2 Nov 30 13:12 /dev/zio/zzero-0000-0-1-ctrl
crw------- 1 root root 250, 3 Nov 30 13:12 /dev/zio/zzero-0000-0-1-data
crw------- 1 root root 250, 4 Nov 30 13:12 /dev/zio/zzero-0000-0-2-ctrl
crw------- 1 root root 250, 5 Nov 30 13:12 /dev/zio/zzero-0000-0-2-data
```

The exact name, unfortunately, depends on the version of *udev* you are running (older versions did not make the directory */dev/zio* and just created the devices in */dev*).

The role of the two devices is as represented in figure Figure 5.1. The meta-data and data are strictly ordered in time, but applications can choose to read either one or both. For input, if you only read one of the devices, the other items are discarded; the framework marks both devices as readable when a block is available, and users can choose what to read – after you read at least some of the data, the next read from the control device discard trailing data and returns the next control. For output, if you only write data the default control is used; if you write the control you replace the default control for the following data writes. (This is not yet implemented in the current release, although we have a beta version).

If the channel is a zero-size device, user space must write only control blocks. This is how the DTC devices work, and *tools/test-dtc* shows how to do that.

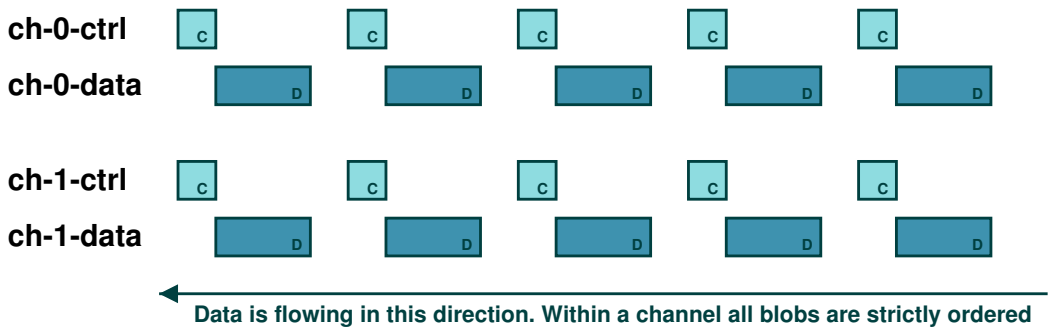


Figure 5.1

This split allows simple programs to just ignore the metadata and retrieve unadorned data in the usual way: *cat*, *dd*, *hexdump* are all good tools for simple diagnostics of both input and output. More refined applications that need to deal with meta-data can afford the extra burden of opening a second device. Example tools to access ZIO data are described in Section 5.2 [User Space Utilities], page 19.

You can think of this approach as what you normally do with industrial food items. The envelope describes the specific content (including the timestamp, name, item size and item count of each package), and you can choose to read the envelope or not; similarly, you can choose to eat the food or not, and sometimes you make the choice only after reading the envelope.

Different buffer types can offer different functionalities; for example the *vmalloc* buffer type supports *mmap* by means of the `mem_offset` field in the control. Applications can *mmap* the

whole buffer; later they can read the control and just access the data through a pointer. This is represented in Figure 5.2.

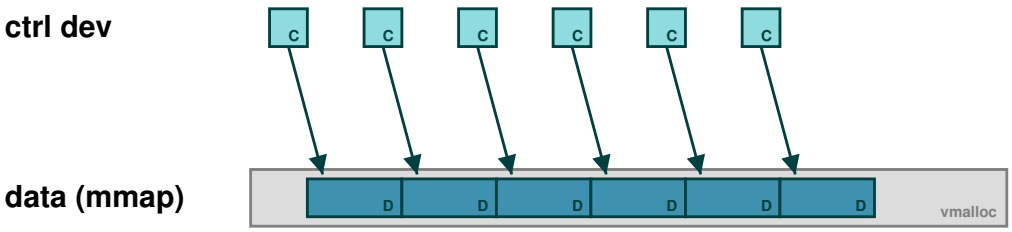


Figure 5.2

An attribute of the *vmalloc* buffer, called `merge-data` can turn it into a real circular buffer: a buffer instance, if so configured, can merge (stick together) several data blocks, in order to immediately release some of the control structures. The user may activate the attribute if the application knows it won't need meta-data for every block: if you know you acquire a continuous stream at 1kHz, for example, time-stamping the first sample may be enough, because further data samples are self-timed.

By setting the attribute you don't break the ZIO data model because whenever a new data block arrives, the buffer sticks it to the previous data by releasing the new control and increasing the block size of the previous block. When a block enters an empty buffer, its own control is preserved, but its own `nsamples` field may be later increased when another data block is merged to it. The *control* + *data* abstraction thus still works towards user space. This situation is shown in Figure 5.3.

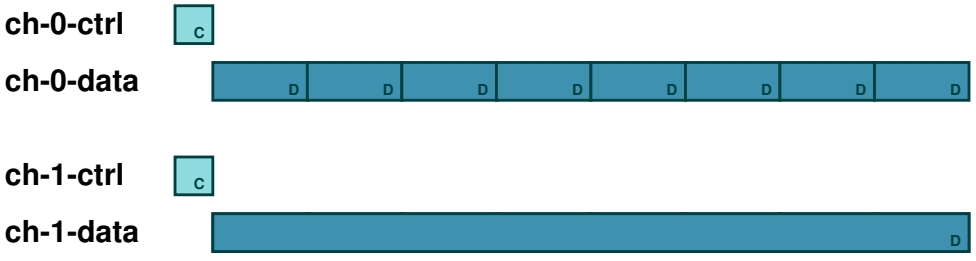


Figure 5.3

5.2 User Space Utilities

The distribution includes a few device-independent tools in the *tools* subdirectory.

5.2.1 zio-dump

The most important one is `zio-dump`: a program that reads pairs of ZIO devices and prints to *stdout* both the meta-data and the data it finds.

The following example shows the output of *zio-dump* reading three pairs of devices, that refer to the three channels of cset 0 of *zzero*. The first channel returns zeroes, the second is random and the third is sequential; the current trigger is the timer, with a 2-second period.

(Actually, the current version of *zio-dump* prints more information from the control block, including alarms, but we had no time to update this section).

```
spusa.root# ./tools/zio-dump /dev/zio/zzero-0000-0*
Ctrl: version 1.0, trigger timer, dev zzero-0000, cset 0, chan 0
Ctrl: seq 603, n 16, size 1, bits 8, flags 01000001 (little-endian)
Ctrl: stamp 1354278753.419850926 (0)
Data: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Ctrl: version 1.0, trigger timer, dev zzero-0000, cset 0, chan 1
Ctrl: seq 603, n 16, size 1, bits 8, flags 01000001 (little-endian)
Ctrl: stamp 1354278753.419850926 (0)
Data: 4b 01 7f a7 a5 69 fa 6a e0 90 b2 53 89 de 2e 46

Ctrl: version 1.0, trigger timer, dev zzero-0000, cset 0, chan 2
Ctrl: seq 603, n 16, size 1, bits 8, flags 01000001 (little-endian)
Ctrl: stamp 1354278753.419850926 (0)
Data: a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af

Ctrl: version 1.0, trigger timer, dev zzero-0000, cset 0, chan 0
Ctrl: seq 604, n 16, size 1, bits 8, flags 01000001 (little-endian)
Ctrl: stamp 1354278755.419925778 (0)
Data: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Ctrl: version 1.0, trigger timer, dev zzero-0000, cset 0, chan 1
Ctrl: seq 604, n 16, size 1, bits 8, flags 01000001 (little-endian)
Ctrl: stamp 1354278755.419925778 (0)
Data: 7f 49 eb df 1e 32 94 3b 2f c7 99 cb e4 97 cd 7b

Ctrl: version 1.0, trigger timer, dev zzero-0000, cset 0, chan 2
Ctrl: seq 604, n 16, size 1, bits 8, flags 01000001 (little-endian)
Ctrl: stamp 1354278755.419925778 (0)
Data: b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
```

The following example shows *zio-dump* reading one block only (`-n 1`) from channel 2, showing the attributes as well (`-a`):

```
spusa.root# ./tools/zio-dump -n 1 -a /dev/zio/zzero-0000-0-2*
Ctrl: version 1.0, trigger timer, dev zzero-0000, cset 0, chan 2
Ctrl: seq 686, n 16, size 1, bits 8, flags 01000001 (little-endian)
Ctrl: stamp 1354278919.419898276 (0)
Ctrl: device-std-mask: 0x0001
Ctrl: device-std-0   0x00000008         8
Ctrl: trigger-std-mask: 0x0002
Ctrl: trigger-std-1   0x00000010        16
Ctrl: trigger-ext-mask: 0x0001
Ctrl: trigger-ext-0   0x000007d0      2000
Data: d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
```

The attributes above show that the block size is 16 samples and the `ms-period` attribute of this instance is 2000. Each attribute for the device and the trigger appear as a named file in *sysfs*:

```
spusa.root# ls /sys/bus/zio/devices/zzero-0000/zero-input-8/trigger/
enable  ms-period  ms-phase  name  post-samples  power/  uevent
```

The mapping between names and attribute positions in the control data structure is available by using the proper defines in source files. We'll offer a way to see the mapping from *sysfs* in future versions.

5.2.2 zio-cat-file

An alternative to `zio-dump` is `zio-cat-file`: it receives a single file name on the command line, and copies it to *stdout* using *mmap* if so supported by the device (i.e., but the underlying buffer

type). The program works by opening the ZIO control file associated with the named data file, and by using the meta-information in there to time data accesses. If your buffer is *vmalloc* or any other *mmap*-capable buffer, the tool will memory-map the data device instead of reading it:

```
spusa.root# echo vmalloc > \
     /sys/bus/zio/devices/zzero-0000/zero-input-8/current_buffer
spusa.root# ./tools/zio-cat-file /dev/zio/zzero-0000-0-2-data  3 | od -t x1z
./tools/zio-cat-file: trasferred 3 blocks, 48 bytes, 0.000017 secs
0000000 90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f  >................<
0000020 a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af  >................<
0000040 b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf  >................<
0000060
```

By looking at the source code or using *strace* you can verify how data is retrieved my memory mapping instead of reading.

5.2.3 test-dtc

The simple *test-dtc* program writes control blocks to a control file, specifying timestamps for a DTC device. By default the program uses the name of the *zio-fake-dtc* device that uses an high-resolution timer to create output events at specified times, according to the control structures it receives.

6 Internals

This chapter details the design ideas and data structures that make up the three main ZIO objects, as well as the naming conventions we follow throughout ZIO code.

6.1 Naming Conventions

To avoid confusion in the terminology used in ZIO code and documentation, we list here the types and data structures used in ZIO. For each item we list the structure name (without `struct`), the name (English word) we consistently use in all documentation and the preferred name for pointers to that data structure, used in the code.

Struct name or C type	**Name**	**Var name**	**Description**
zio_device	full device	zdev	The overall I/O device, either a board or a chip.
zio_cset	cset	cset	Part of a device; a group of channels with the same physical features.
zio_channel	channel	chan	Part of a cset; it is the single data stream endpoint, input or output.
zio_device_operations	device operations	d_op	Device-specific methods that act on attributes and data blocks.
zio_buffer_type	buffer type	zbuf	Description of a buffer.
zio_bi	buffer instance	bi	Instance of a buffer type associated to a specific channel.

zio_buffer_operations	buffer operations	b_op	Methods of a buffer type: instance creation and destruction, block allocation etc.
zio_trigger_type	trigger type	trig	Description of a trigger, i.e. a generator of I/O events according to some rules.
zio_ti	trigger instance	ti	Instance of a trigger type associated to a specific cset.
zio_trigger_operations	trigger operations	t_op	Methods of a trigger type. See Section 6.4 [The Trigger], page 26.
void *	data *or* samples	data	A memory area hosting actual I/O data.
zio_control	control	ctrl	A binary data structure used to exchange meta-data with user space.
zio_block	block	block	A container for both data and control. In ZIO, all I/O data is always handled within blocks.
zio_obj_head	head	head	All ZIO objects begin with this sub-structure, to ease common management.
zio_cdev_type	-	-	An enumeration to identify type of a char device (control or data).
zio_f_priv	-	-	Set as private_data in all open files. includes `cdev_type` and `bi`.
zio_attribute_set	attribute set	zattr_set	Attributes of a ZIO object (device, cset, chan, ti, bi)

6.2 The Generic Object Head

All data structures that refer to a ZIO object include a `zio_obj_head` structure. The head collects a few fields that are used across ZIO for object management (including the structure that is used in building the *sysfs* tree).

6.3 The Peripheral Driver

In order to write a ZIO peripheral driver, you need to define all three levels of data structures (device, cset, channel), so this chapter describes the important fields of those structures. It then offers an overview of the registration steps and an example.

Unless you want to know all the details immediately, for a cursory overview you may want to skip the first sections and jump to Section 6.3.4 [The Driver Structure], page 24.

6.3.1 The Device Structure

A *device* is the description of a complete I/O peripheral device (or board).

A device is the *probe-unit* of ZIO: it is made up of channel-sets and may represent a PCI board or an SPI integrated circuit or whatever it makes sense to manage from a single device driver.

The device is primarily a container of csets, but it also host attributes that affect all csets at the same time (the attributes may be defined or not, according to features of the specific physical device).

The most important fields of `struct zio_device` for the developer are:

`struct zio_cset *cset`
`unsigned int n_cset`
: The array of channel sets belonging to this device.

`char *preferred_buffer`
`char *preferred_trigger`
: The device may specify a device-wide default trigger type and/or buffer type. Csets can still make their specific choice, which takes precedence, to easily support kernel modules that register a device-specific trigger. These "preferred" names are meant to be modified by module parameters (see `zio-zero` and other examples). If the fields are NULL or the preferred type is not available in the running instance of ZIO, the system-wide defaults for buffer and trigger type will be used.

`const struct zio_sysfs_operations *s_op`
: The structure includes the `info_get` and `conf_set` methods that act on ZIO attributes. See Section 6.6 [The Attributes], page 29.

6.3.2 The Cset Structure

As said, the cset is a homogeneous set of I/O channels belonging to a single device. All channels in the set have the same physical characteristics. This object is the most important in the ZIO device hierarchy because all data transfers are cset-wide. Each cset includes a pointer to the current trigger and buffer types, which are cset-wide attributes.

The data structure includes two string fields (`zbuf` and `trig`) that name the requested trigger and buffer types for this cset. If the fields are blank or the named object is unavailable, the device-wide or zio-wide defaults are tried in this order.

The most important fields of `struct zio_cset` to be filled or used by the developer are:

`int (*raw_io)(struct zio_cset *cset)`
: This is the function that performs actual I/O or schedules it to be performed when the internal trigger fires. If the function returns 0, the input or output operation is already completed. A return value of `-EAGAIN` means that cset code will call `zio_trigger_data_done()` at a later time. Other return values are used to report real errors.

`void (*stop_io)(struct zio_cset *cset)`
: The function, if present, is called when an armed trigger is aborted. The driver will need to implement this method if it wants to return a partially-filled block. The method is called with the lock taken, and can call `zio_generic_data_done`, which is the locked back-end of `zio_trigger_data_done`).

`unsigned ssize`
: The sample size, in bytes, for each channel in the cset. Different channels may feature a different number of significant bits, but they must use the same number of bytes in the data blocks.

`unsigned long flags`
: A few flags, the most important being the type of cset: `ZIO_CSET_TYPE_DIGITAL`, `ZIO_CSET_TYPE_ANALOG` or `ZIO_CSET_TYPE_TIME` (other types may be added in the future, we reserved for 8 of them), OR'd with `ZIO_DIR_INPUT` or `ZIO_DIR_OUTPUT`.

Then, if you set `ZIO_CSET_SELF_TIMED` for an input cset, the trigger will be immediately armed, so the driver can fill blocks when its time arrives – for self-timed csets you should use the default trigger, which is transparent to user and device actions.

`struct zio_channel *chan_template`
: This points to a channel structure that is used by ZIO as a template. All channels in the cset are homogeneous, so ZIO allocates the array of channels when the cset is registered, by replicating this template and updating the *index* value for each of them. Please refer to existing code for examples.

`unsigned int n_chan`
: The number of channels in this cset.

`struct zio_channel *chan`
: The array of channels, allocated by ZIO at registration time.

`void *priv_d`
: A private pointer for the device, in case it needs it.

`int (*init)(struct zio_cset *cset)`
`void (*exit)(struct zio_cset *cset)`
: The function pointers, if not NULL, are called by ZIO at cset registration and removal time, after allocating and before removing, resp., the channel array. They may useful to channel sets that need to setup and release the `priv_d` field.

6.3.3 The Channel

The channel is the lowest-level object in the ZIO hierarchy. It represents the individual connector of the device, most likely a socket in some backplane of some computer (local or remote, in case *Etherbone* is being used). A channel may also be a software simulation of a data source/sink of some time.

The most important fields of `struct zio_channel` for the user are:

`void *priv_d`
: A private pointer for the device (may be allocated by the `init` function of the cset and released by the corresponding `exit` function).

`void *priv_t`
: Private data for the trigger, that may be used by the trigger during operation. If used, the trigger must allocate it at create time and free it at destroy time.

6.3.4 The Driver Structure

The top-level structure that devices a device driver is similar to what is used with PCI or USB. This is the definition of the `zio_driver`:

```
struct zio_driver {
        const struct zio_device_id      *id_table;
        int (*probe)(struct zio_device *dev);
        int (*remove)(struct zio_device *dev);
        struct device_driver            driver;
};
```

The client module, thus, must specify a table of device types it is able to drive. When a match is found, the ZIO core calls the method *probe*, which can return success (zero) or failure (a negative error code). Similarly, the core calls *remove* when the device disappears or the driver is unloaded. Both methods are optional (if missing, success is assumed).

The `driver` sub-structure must be filled in its *name* and *owner* fields. See Section 6.3.7 [An Example Driver], page 25.

The table of identifiers, then, is defined like this in for ZIO devices:

```
struct zio_device_id {
        char                    name[ZIO_OBJ_NAME_LEN];
        struct zio_device       *template;
};
```

The `name` field is a 12-byte string, the same string that is part of the control block. The `template` is a ZIO device, with the proper fields initialized. The ZIO core replicates such template for each and every device matching the name.

The `id_table` field in `zio_driver` is a pointer to an array of identifiers; the array is terminated with an empty entry, as customary in many Linux subsystems.

6.3.5 Registering a ZIO Driver

By virtue of how the Linux bus abstraction works, registering a ZIO device is a two-step procedure, based on the above data structures.

The client driver must register a `struct zio_driver`, with the associated table of identifiers; when the core finds a match, it allocates a new device instance and calls the *probe* method. Please note that the `zio_device` passed to the *probe* method is not the template, but a copy of it – this is mandatory in order to support several devices of the same type running on the same host.

If, for some reason, you need to save some private data in the device structure, you must do that in the *probe* function for each new device instance – and undo any step in the associated *remove* function.

If your driver is able to drive two or more devices, that are similar in nature but feature a different number of csets or differ in some other detail, you should fill your table with more than one device type, each of them with a different name. This is exemplified by the `ad788x` driver, part of the ZIO distribution.

6.3.6 Registering a ZIO Device

In order to register a device, you need to allocate it from the ZIO core, and then register it by passing a name and a unique `dev_id` value. We agree that this is not completely inutuitive, but it is retained for backward compatibility.

The steps, in practice are:

- Allocate a generic device;
- Fill the *owner* filed with `THIS_MODULE`, for refcounting;
- Register it in the ZIO core, with a name and a unique numeric ID.

This, for example, is how *zio-zero* allocates and registers its device:

```
struct zio_device *zzero_dev;

zzero_dev = zio_allocate_device();
if (IS_ERR(zzero_dev)) {
        err = PTR_ERR(zzero_dev);
        goto out_all;
}
zzero_dev->owner = THIS_MODULE;
err = zio_register_device(zzero_dev, "zzero", 0);
```

When the module is removed, the device must be unregistered and released:

```
zio_unregister_device(zzero_dev);
zio_free_device(zzero_dev);
```

6.3.7 An Example Driver

This is an example of all the data structures need to define a ZIO device driver. The code comes from *zio-mini*:

```
static struct zio_cset zmini_cset[] = {
        {
                ZIO_SET_OBJ_NAME("timespec-in"),
                .raw_io =       zmini_input,
                .flags =        ZIO_DIR_INPUT,
                .n_chan =       1, /* changed at insmod */
                .ssize =        sizeof(struct timespec),
        },
};

static struct zio_device zmini_tmpl = {
        .owner =                THIS_MODULE,
        .cset =                 zmini_cset,
        .n_cset =               ARRAY_SIZE(zmini_cset),
};

/* The driver uses a table of templates */
static const struct zio_device_id zmini_table[] = {
        {"zmini", &zmini_tmpl},
        {},
};

static struct zio_driver zmini_zdrv = {
        .driver = {
                .name = "zmini",
                .owner = THIS_MODULE,
        },
        .id_table = zmini_table,
};
```

With these structures in place, the *init* function of the module just needs to call "`zio_register_driver(&zmini_zdrv)`". Actually, the data structures above are more than half of the complete driver: the only missing bit is the `raw_io` cset method, because triggering, buffering and everything else is managed automatically by the ZIO framework.

This is a very simple case, where the table includes only one entry, the device has only one cset and there's no need for *probe* and *remove*. The seemingly-unneeded extra levels are there in order to be able to support more complex cases, including the common situation when more than one device is driven by a single host.

6.4 The Trigger

Every data exchange, either input or output, is executed in response to an event of some kind. ZIO offers a *trigger* abstraction to describe all such events and configure their activation.

Each cset is connected to a trigger type, and a specific instance of that type. Different csets can use different trigger types, because the `current_trigger` is an attribute of each cset. When the trigger arms, it acts on all the non-disabled channels of the cset.

For input data flows, the trigger receives blocks from the device and stores them in the buffer. For output data flows the trigger retrieves blocks from the buffer and sends them to the device.

When defining a new trigger type, the most important fields of `struct zio_trigger_type` for the programmer are the following ones:

`const struct zio_sysfs_operations *s_op`
: These are the operations used to read and write attributes. It is the same set of operations used in the device data structure.

`const struct zio_trigger_operations *t_op`
: The trigger operations, described below, are the ones that implement the behavior of a trigger type.

The trigger operations are defined by the following structure:

```
struct zio_trigger_operations {
        int                     (*push_block)(struct zio_ti *ti,
                                              struct zio_channel *chan,
                                              struct zio_block *block);
        void                    (*pull_block)(struct zio_ti *ti,
                                              struct zio_channel *chan);

        void                    (*data_done)(struct zio_cset *cset);

        int                     (*config)(struct zio_ti *ti,
                                          struct zio_control *ctrl);

        struct zio_ti *         (*create)(struct zio_trigger_type *trig,
                                          struct zio_cset *cset,
                                          struct zio_control *ctrl,
                                          fmode_t flags);
        void                    (*destroy)(struct zio_ti *ti);
        void                    (*change_status)(struct zio_ti *ti,
                                                 unsigned int status);
        void                    (*abort)(struct zio_ti *ti);
};
```

The detailed meaning of the operations is as follows:

`create`
`destroy`

The operations are called when this trigger type is associated to (resp. de-associated from) a new cset. `create` returns a disabled trigger instance structure, which is usually part of a larger structure, accessed by using `container_of`. Please look at existing triggers for details.

`push_block`

When a buffer has a complete block of data, it can send it to the trigger using `push_block`. The trigger can either accept it (returning 0) or not (returns `-EBUSY`). This happens because an output trigger has only one pending data transfer. When the block is finally consumed, the trigger must call `bi->retr_block` to get the next one: buffering is in the buffer, not in the trigger.

`pull_block`

For input channels, a buffer may call `pull_block`. The trigger may thus fire input directly and later have a block. Most triggers won't support the `pull_block` way of doing input, they will just call `bi->store_block` when a new block is available. In these cases the `pull_block` method can be left `NULL`.

`data_done`

This method, if defined, is called by the core inside the helper `zio_trigger_data_done()`. I/O in the device is almost always asynchronous, so when asked to transfer a cset, the device will prepare to do it, and will call `zio_trigger_data_done` later. For output csets, `zio_trigger_data_done` frees the blocks; for input, `zio_trigger_data_done` pushes material to the buffers. If the peripheral is self-timed, the function also arms the trigger for the next event. The method, if present, is called while holding the *cset* spin lock, and the trigger still in `ARMED` state (`zio_trigger_data_done` clears the flag only when everything is over with the current trigger event).

`config`

The method is not currently used. The idea is that when a channel is configured by sending it a complete new control structure, this callback allows the trigger to reconfigure itself.

`abort` Abort, if defined, is called when an already-armed trigger event must be aborted. This happens, for example, because event parameters changed (e.g., the block size). The method must call `cset->stop_io` if not NULL and dispose the *active_block* for each channel, setting the pointer to NULL. Please check how it is used in `helpers.c`. The method is called while holding the cset lock and cannot fail nor sleep. See the generic abort implementation for reference.

`change_status`
The method, if defined, is called when the trigger is enabled or disabled. For disable, ZIO calls *abort* beforehand. The method is called while holding the cset lock.

6.5 The Buffer

The buffer interface in ZIO allows to select between different allocation techniques and memory access patterns. By splitting the buffer to a separate ZIO object, the framework allows drivers with special allocation needs to define their own hardware-specific buffer – for example, data samples may need to live in PCI memory.

Each cset is using a buffer type; an instance of that buffer type exists for each channel in the cset. Actual I/O only happens if a channel is enabled, so different buffer instances in the cset may at times host a different number of blocks, but this is not the usual case. When a buffer experiences overflow, a data block is lost, and this is marked in the `zio_alarms` field for the specific channel.

The most important fields of `struct zio_buffer_type`, from the point of view of the developer of anew buffer, are the various operations structures:

`const struct zio_buffer_operations *b_op`
The buffer operations, detailed later, are the functions that define the actual behavior of a buffer.

`const struct file_operations *f_op`
File operations are used to provide user-space access to the buffer. ZIO exports a `zio_generic_fops` structure that will work for most users. We may remove this pointer if the *interface* idea is implemented in the ZIO core.

`const struct vm_operations_struct *v_op`
Buffer types supporting *mmap* must implement the virtual memory operations. They allow to keep track of active uses of the buffer instance and handle page faults for the processes that are accessing the buffer. The operations refer to each `vma` mapped to the char device associated to a buffer instance of this type. Use is exactly like what you do in normal char drivers, with the only difference that the core calls `v_op->open` every time *mmap* happens (because the driver doesn't run its won *mmap* system call back-end. Please refer to the `zio-buf-vmalloc` implementation for details.

The buffer operations are defined as follows:

```
struct zio_buffer_operations {
        struct zio_block *      (*alloc_block)(struct zio_bi *bi,
                                               size_t datalen, gfp_t gfp);
        void                    (*free_block)(struct zio_bi *bi,
                                              struct zio_block *block);

        int                     (*store_block)(struct zio_bi *bi,
                                               struct zio_block *block);
        struct zio_block *      (*retr_block) (struct zio_bi *bi);

        struct zio_bi *         (*create)(struct zio_buffer_type *zbuf,
                                         struct zio_channel *chan);
```

```
        void                    (*destroy)(struct zio_bi *bi);
};
```

This is the specific role of each method in the structure:

`create`
`destroy`
: When ZIO associates a buffer with a new channel, it calls the `create` operation. The returned `zio_bi` structure is usually be part of a bigger structure, accessible using `container_of`. If creation fails, the method must return an `ERR_PTR` value, not NULL.

`alloc_block`
`free_block`
: The buffer is concerned with data storage, so whenever the trigger or the *write* system call need a new block, they ask it to the buffer type. Similarly, the buffer type is asked to release blocks. `alloc_block` must also allocate the control, through the helper `zio_alloc_control`. Such control is filled with the current values for the channel in due time. (For input this copy happens late). On error allocation must return NULL.

`store_block`
`retr_block`
: The functions simply add a block to an existing buffer instance or ask to extract a block out of it. In addition to managing storage according to its own needs, the buffer is requested to make two special actions. When `store_block` inserts the first block in an empty output buffer, the method must call the `ti->push_block`. When `retr_block` is called on an empty input buffer, the method must call `ti->pull_block`, if the function exists. Please refer to existing implementations for details.

6.6 The Attributes

This section is till to be written. Feel free to express your interest in this section to the mailing list. Meanwhile, please refer to current code.

7 Available Modules

The current ZIO repository includes a number of modules for devices triggers and buffers. They are meant to act as test cases, examples and tools to stress-test the code. Some of them are useful in the real world, despite their simple and straightforward design.

Drivers are listed in Section 1.2 [Supported Devices], page 3, so they are not repeated here.

7.1 Triggers

This release of ZIO includes the following trigger types:

`user`
: This is the default trigger, linked in `zio.ko`. It is a transparent trigger: then user space reads an input channel, it arms the input event, and when user space writes to output channels, the event is armed when all channels have been provided their output block. If the device is self-timed (e.g., TDC and DTC devices, or an ADC that streams data at a certain rate) the trigger is armed as soon as possible, to allow the device itself to set its own pace

`timer`

The timer trigger uses a kernel timer and is periodic. While you cannot set it to an absolute time, it includes support to set the *phase*, to be able to schedule in time I/O on several csets. How exactly phase support works is documented in the source code and the commit messages.

`hrt`

High-resolution timer. This trigger can work as both periodic and one-shot. It is configured for absolute times or for some delay in the future (to ease testing with shell scripts). It accepts both scalar nanoseconds (as low-half and high-half values) and seconds + nanoseconds. Another attribute specifies the allowed slack to be used in programming the kernel resource.

`irq`

External-interrupt. The module receives a `irq=` parameter, or a `gpio=` parameter (the latter only available for systems with *libgpio* (and a working `gpio_to_irq` function). You can have several instances of this trigger type, but all of them are bound to the same interrupt. This is mainly used for demonstration purposes.

7.2 Buffers

This release of ZIO includes the following buffer types.

`kmalloc`

This is the default buffer. It allocates blocks by calling *kmalloc*. The buffer size is expressed in number of blocks, and it defaults to 16. You can change it in *sysfs* for each instance.

`vmalloc`

This buffer allocates memory using *vmalloc*, and it supports *mmap*. It also supports the `merge-data` attribute, as described in Section 5.1 [Details of Char Device Policies], page 18. Its size is expressed in kilobytes, and it default to 128. Currently, it cannot be changed, as it still doesn't count the number of active *mmap* users.

There is currently no way to change the buffer size at module load time, but there's nothing preventing it, besides our own time supply.

8 Locking Policies

In order to safely work with ZIO, developers should be aware of the locking policies that are already in place. With this knowledge they can handle their critical sections without over-locking or under-locking in preventing race conditions.

In general, we tried to centralize locking in order to simplify the task of writing ZIO drivers.

The following locks are defined used in ZIO core and data structures:

`zio_status->lock`

This is used internally to take care of all registered devices, csets, trigger types and buffer types. The core hosts a list for each of the item types and a single lock is used for all of them. These lists, anyways, can be modified only when ZIO modules register or unregister themselves.

`zio_device->lock`

This lock is used to serialize every configuration performed through zio attributes. This applies to all configurations pertaining the device and its dependent levels,

including buffer and trigger instances. In other words, all `conf_set` and `info_get` calls are serialized device-wise.

`zio_cset->lock`
: The cset structure includes a lock that is used to serialize access the I/O operations, as well as trigger-related events (i.e. the `ti->flags` bits like `ZIO_TI_ARMED`). For this reason, the field `ti->cset` must be immediately assigned when the trigger instance is created. I/O is serialized by trigger code in the ZIO core: only one trigger event can be pending for each cset, using the ARMED flag. The cset itself, then, can use this same lock to serialize other activities and prevent a trigger to be armed during such activities. While taking this lock interrupts must be disabled, because trigger operations usually happen in interrupt context.

`zio_buffer_type->lock`
`zio_trigger_type->lock`
: Each buffer and trigger type has a lock, which is used by the core when instances are created or destroyed. Thus, such operations are serialized and sub-modules are safe without arranging for their own locking policies.

`zio_bi->lock`
`zio_ti->lock`
: These locks are initialized by the core before calling the respective create function. Buffer and trigger types can use them without declaring a spinlock in their own structures. Buffers distributed with the core use this lock.

From the table above, it's clear how the device lock, used for configuration, is the ZIO spinlock with the widest scope. Even though device, buffer and trigger are registered as different objects, they live on the same peripheral device. Thus, you most often need to serialize configuration on the device as a whole, because configuration parameters are usually stored in hardware registers.

If the trigger and buffer modules are device-specific, they may need to access the device spin lock, too. While this doesn't apply to generic triggers or buffers, taking the device lock won't have bad effects, and the associated overhead is minimal.

If your buffers or triggers request a different kind of locking (e.g., you need to serialize some sections with a scope bigger than the single device), you'll need to arrange for your own locking.

The following lines show how to reach the device lock from the various objects used withing the ZIO framework:

```
zio_device->lock
zio_cset->zdev->lock
zio_channel->cset->zdev->lock
zio_ti->cset->zdev->lock
zio_bi->cset->zdev->lock
```

The device lock is also used to protect the enable/disable bit of device, cset, channel and trigger instance. For this reason, please note that the *abort* and *change_status* trigger operations are called while holding the device lock.

ZIO sub-modules should arrange for any other locking requirements. Buffer modules will typically ensure consistency of the data space within each instance (i.e., concurrency of `store_block` and `retr_block`) (they can use `bi->lock` for this). Trigger modules will need to protect modification of their status flag, like the utility functions `zio_generic_data_done` and `zio_fire_trigger` do (they can use `ti->lock` to this aim). Device modules will need to serialize some of their non-atomic hardware access primitives, in this case by declaring their own locks.

Index

M

N

O

P

R

S

T

U

V

W

Z