

ZIO User Manual

February 2012
A kernel framework for laboratory I/O

Alessandro Rubini and Federico Vaga for CERN

Table of Contents

Introduction	1
1 General Concepts	1
1.1 Definitions	1
1.2 Supported Devices	2
1.3 Future Developments	3
2 ZIO data model	3
2.1 The Block	4
2.2 The ZIO Pipeline	5
2.3 The Control Structure	6
2.4 The Time Stamp	9
2.5 The Data	9
3 Accessing ZIO from User Space	9
3.1 Details of Char Device Policies	10
3.2 User Space Utilities	10
4 Accessing ZIO from Kernel Space	10
5 Internals	10
5.1 The Generic Object Head	10
5.2 The Device	10
5.3 The Trigger	11
5.4 The Buffer	13
5.5 The Attributes	14
6 Available Modules	14
6.1 Devices	14
6.2 Triggers	14
6.3 Buffers	14
7 Writing ZIO Modules	15
7.1 Locking Policies	15
7.2 Writing a Device	15
7.3 Writing a Trigger	15
7.4 Writing a Buffer	15
Index	16

Introduction

ZIO is meant 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.

The version at time of this writing is known to compile and run with kernels 2.6.34 onwards.

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 *block* 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 not forbidden, 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 5.2 [The Device] , page 10 . |
| Trigger | The trigger is code concerned with firing I/O, in response to some event. ZIO supports both software triggers and hardware triggers; in the latter case the software module is used to configure to the device and retrieve status information. See Section 5.3 [The Trigger] , page 11 . |
| 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 (but ZIO buffers are not constrained to). See Section 5.4 [The Buffer] , page 13 . |

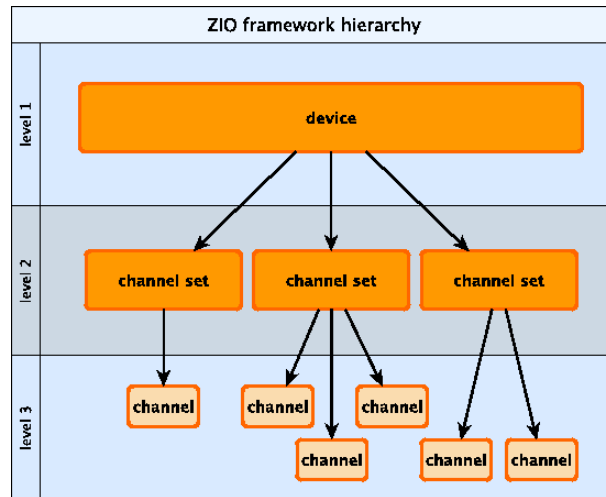
The ZIO driver manages a *device*, which in turn may be either a chip or a PCB (or something else). The driver is the *probe unit*, i.e. a batch of I/O peripherals that are plugged and unplugged as a whole, usually based on some bus, like PCI, USB or SPI.

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, like gain and offset).

The channel is the basic I/O entity, usually associated with a single socket on some backplane.

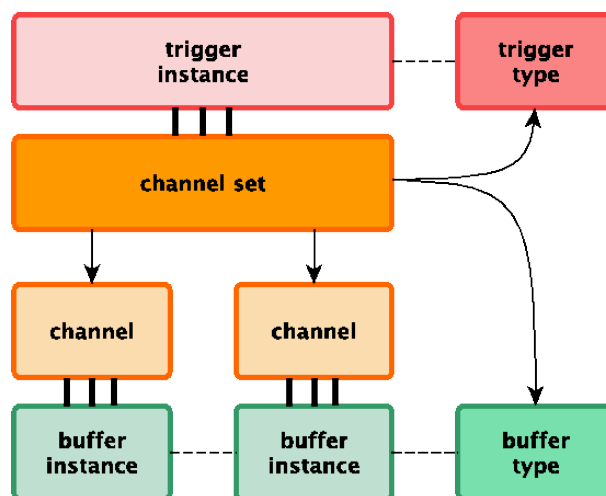
The next figure depicts ZIO's layered structure.



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; ZIO creates an instance of the trigger type for each cset using it. 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.

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



1.2 Supported Devices

This is a quick list of devices or device types that are supported in the current version of ZIO. This is a cursory overlook, for some of them more information is provided in specific sections of [Chapter 6 \[Available Modules\]](#), page 14.

zero device

The zero device is a software-driven input and output device, it is used for demonstration and stress-testing.

gpio device

The gpio device acts on an array of input and output GPIO pins. Again, we think it's a useful example.

uart device

ZIO includes a line discipline device, that can be connected to a serial port or a *pty*.

SPI devices

Some ADC devices connected to the SPI bus are supported, it should be easy to add more.

RAM-based buffers

The distributions includes generic buffers based on system RAM. See [Section 6.3 \[Available Buffers\]](#), page 14.

Various software-driven triggers

This version includes a few software-driven trigger: kernel timer, external interrupt (GPIO or otherwise) and a transparent trigger to be used when the device must run continuously with it's own I/O data rate. See [Section 6.2 \[Available Triggers\]](#), page 14.

1.3 Future Developments

The following list outlines future developments which are being studied and/or already tested.

- Defining a ZIO bus type

Using a bus type, we'll be able to instantiate several card/chips of the same kind, within a single host. Initially we decided to avoid it for simplicity.

- Writing `input_device` for the kernel

A buffer type may inject input events to the Linux input subsystem, and we feel it's an easy and interesting feature to offer.

- Writing an Ethernet ZIO buffer

This buffer will be able to send input blocks as network frames and receive output blocks in network frames. The buffer will register an *eth* device, with an own MAC address, so frames may be exchanged with remote hosts, either as UDP packets or raw frames.

- Writing a new PF_ZIO network type.

The protocol family PF_ZIO will be used both as *ethertype* for raw frames and as a new address family (AF_ZIO) where each channel has a network address, with the *control* acting as link-level protocol header. This will allow applications to open only a single socket (or a few of them) instead of one or two char devices for each channel being used.

- Defining the “zio_interface” data structure.

The *interface* will abstract char device support out of the ZIO core, so users will be able to use different interfaces; the PF_ZIO protocol will therefore be able to use hardware-specific buffers when the device requires them.

2 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.

2.1 The Block

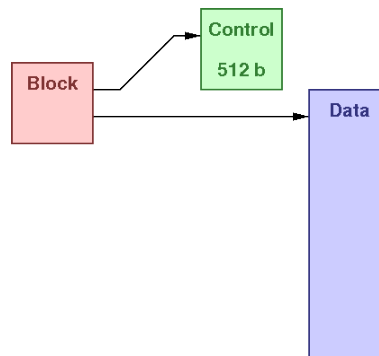
Our typical use case handles several samples at the same time. We need to manage thousands or even millions of samples at the same time, because I/O is performed at hardware level (within the FPGA) and the driver only passes data around.

For this reason ZIO defines a block structure. All data transfers within ZIO happen in the context of a block, which is the atomic data item.

The block structure is defined in `zio-buffer.h` in the following way:

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

A block may be depicted as shown in the next figure:



The meaning of the fields is:

`ctrl_flags`

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

`data`

The pointer to actual data associated with this block.

`datalen`

The length of the memory area pointed-to by `data` above. May be zero for TDC or DTC devices.

`uoff`

User offset. This field is used when data is being consumed or produced in the context of a stream-like interface, like a chat device is.

We chose to coalesce the pointer to control and the `cdone` bit in a single field 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 shown `ctrl` is a pointer and `cdone` is a bit. The bit is used to keep track of whether a stream-like interface already managed the control structure or not. A typical user is the `read` file operation. See [Chapter 3 \[Accessing ZIO from User Space\]](#), page 9.

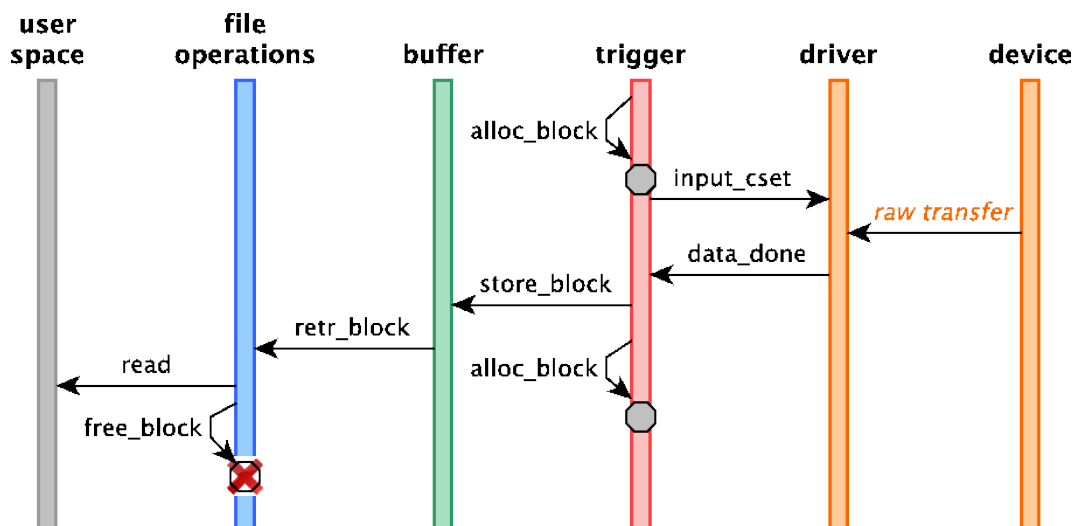
2.2 The ZIO Pipeline

During input and output operations, ZIO blocks travel across a pipeline of objects.

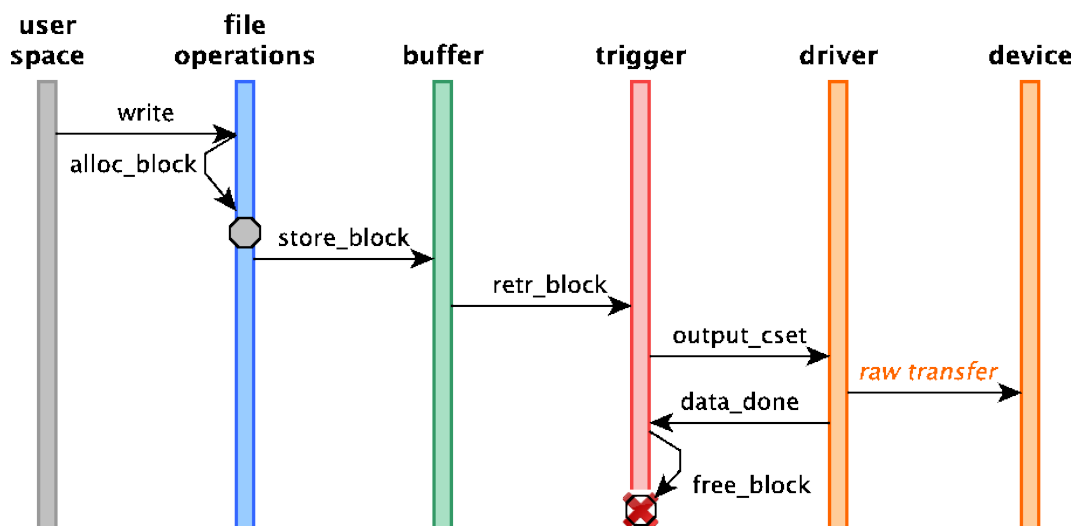
User space makes system calls, which are implemented by the `file_operations` in the active buffer. Generic functions are implemented in `zio-cdev.c` and exported as `zio_generic_fops` (currently all buffers included in the distribution are using these operations). This will slightly change when we the *interface* concept will be introduced, but the change will not affect buffers (only, they won't have a `f_op` pointer any more). See [Section 1.3 \[Future Developments\]](#), page 3 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.

The next figure shows the input pipeline, showing where a block is allocated and then freed. Time in the figure is flowing downwards.



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



2.3 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 and won't change in future ZIO releases (in some case we may concatenate several control structures, but this will be self-described).

The structure is fixed in order to simplify communication with user space, and allow generic applications to 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.

The control is depicted in the next figure:

```

Control Structure

/* byte 0 */
uint8_t major_version;
uint8_t minor_version;
uint8_t more_ctrl; /* number of further ctrl, for interleaved */
uint8_t alarms; /* set by channel, persistent, write 1 to clr */

/* byte 4 */
uint32_t seq_num; /* block sequence number */
uint32_t flags; /* endianness etc, see below */
uint32_t nsamples; /* number of samples in this data block */

/* byte 16 */
uint16_t ssize; /* sample-size for each of them, in bytes */
uint16_t sbits; /* sample-bits: number of valid bits */
uint16_t cset_i; /* index of channel-set within device */
uint16_t chan_i; /* index of channel within cset */

/* byte 24 */
uint64_t hostid[8]; /* Macaddress or whatever unique */

/* byte 32 */
struct timespec timestamp;

/* byte 56 */
uint32_t mem_offset; /* position in mmap buffer of this block */
uint32_t reserved; /* possibly another offset, or space for 64b */

/* byte 64 */
/* The control block includes what device the data belongs to */
char devname[ZIO_DEV_NAME_LEN];

/* byte 76 */
/* Each data block is associated with a trigger and its features */
char triggername[ZIO_DEV_NAME_LEN];

/* byte 88 */
struct zio_ctrl_attr attr_channel;
struct zio_ctrl_attr attr_trigger;

/* byte 488 */
uint8_t _fill_end[ZIO_CONTROL_SIZE - 488];

```

The control is designed to offer the full meta-information needed to describe the block. It includes names and parameters for both the device and the current trigger. Applications can thus pass around a block without knowing what it is; thus, 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 and offline data analysis. 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 of 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. This is the detail of it:

```

struct zio_control {
    /* byte 0 */
    uint8_t major_version;
    uint8_t minor_version;
    uint8_t more_ctrl; /* number of further ctrl, for interleaved */
    uint8_t alarms; /* set by channel, persistent, write 1 to clr */

    /* byte 4 */
    uint32_t seq_num; /* block sequence number */
    uint32_t flags; /* endianness etc, see below */
    uint32_t nsamples; /* number of samples in this data block */

    /* byte 16 */
    uint16_t ssize; /* sample-size for each of them, in bytes */
    uint16_t sbits; /* sample-bits: number of valid bits */
    uint16_t cset_i; /* index of channel-set within device */
    uint16_t chan_i; /* index of channel within cset */

    /* byte 24 */

```

```

uint8_t hostid[8];      /* Macaddress or whatever unique */

/* byte 32 */
struct zio_timestamp tstamp;

/* byte 56 */
uint32_t mem_offset;   /* position in mmap buffer of this block */
uint32_t reserved;    /* possibly another offset, or space for 64b */

/* byte 64 */
/* The control block includes what device the data belongs to */
char devname[ZIO_OBJ_NAME_LEN];

/* byte 76 */
/* Each data block is associated with a trigger and its features */
char triggername[ZIO_OBJ_NAME_LEN];

/* byte 88 */
struct zio_ctrl_attr attr_channel;
struct zio_ctrl_attr attr_trigger;

/* byte 488 */
uint8_t __fill_end[ZIO_CONTROL_SIZE - 488];
};

```

This is the meaning of all the fields. Some of them are not currently being filled by the code and the description here doesn't always reflect the code status. We plan to detail the status of each of them soon – while auditing and completing support for it.

major_version
minor_version

The version is currently 0.4, 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. Since version 1, we ensure that all minor version changes will be compatible (i.e., new fields are there but they may be ignored) and incompatible changes will force an upgrade of the major number – for example, if the layout of attributes changes for some reason.

more_ctrl

This field, not used in the current code, can state that this data block is described by more than one control structure. It represents the number of additional control structures (all 512 byte long) that are used in this block. The field is always 0 with current code but we won't upgrade the version number when using it; so applications should be ready for it to be greater than 0 and act accordingly.

alarms

A mask of active alarm. They should be cleared by writing 1 to them. They are not currently in use, and the mechanism to clear them in `sysfs` is designed but not implemented.

seq_num

Block sequence number. It starts at 1 and can be checked to know about lost blocks for buffer overflows. 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).

flags

The flags specify features of the data block. See below.

nsamples
ssize
sbits

Number of samples, sample size in bytes, and sample bits. 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.

`cset_i`
`chan_i`

The index of the cset and the channel within the device

`hostid`

Identification of the host where this device/cset/channel tuple is hosted. This is not currently set nor read by ZIO, but will be soon added to easily support multi-host environments with centralized data generation or consumptions, ensuring proper routing. The meaning of the field is application-specific; it may be a MAC address or whatever. We plan to add a default *hostid* value for input channels, set as a module parameter.

`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 `externa-irq` trigger won't use pre-set timestamps). The internals of this structure are defined later.

`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 valid nonetheless.

`devname`

`triggername`

Name of the device this control refers to and of the trigger currently active. For input, the names are filled by ZIO, for output they may be used by applications to route packets or switch triggers, but ZIO ignores them when the block is fed to output character devices.

`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 describe in [Section 5.5 \[The Attributes\], page 14](#).

`__fill_end`

This filler ensures that the size of the control structure is 512 bytes. ZIO includes a compile-time check for the size of the control structure, to ensure this field has the right length even if someone changes the structure (and version number).

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 (**sbits**) are placed within the bytes of a sample (**ssize**). Both flags are not 0; if neither is set the alignment is unspecified.

2.4 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 made of 3 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 an UTC or TAI value, i.e. the number of seconds since Jan 1, 1970. The field is 64-bits wide to preserve alignment and to allow of a different choice of the epoch, if needed. When ZIO uses software timestamping for input channels, 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.

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 without a second or nanosecond component, if it fits in 64 bits.

For example, *White Rabbit* devices use hardware timestamps made up of three 32-bit values: seconds, nanoseconds with a granularity of 8ns or 16ns and phase offsets as picoseconds within the 8ns or 16ns interval. In this case, ZIO timestamps will store the seconds in `secs`, the nanoseconds in `ticks` (with 0 in the 3 or 4 LSB) and the phase in the `bins` field.

2.5 The Data

The data pointed-to by a control structure is just opaque data for ZIO. For output, it's the device that makes sense of it (or just passes it hardware without knowing what it is); for input, it's the final destination of the data that will use it according to the device/cset/channel it originated from, and the information about sample size, bits and alignment found in the control structure.

3 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.

3.1 Details of Char Device Policies

Still to be written. Feel free to express your interest in this section to the mailing list.

3.2 User Space Utilities

Still to be written. Feel free to express your interest in this section to the mailing list.

The utilities, anyways, are found in the *tools* subdirectory of the source repository.

4 Accessing ZIO from Kernel Space

Still to be written. Feel free to express your interest in this chapter to the mailing list.

5 Internals

This chapter details the design ideas and data structures that make up the three main ZIO objects. The section about devices includes the description of csets and channels.

5.1 The Generic Object Head

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

5.2 The Device

A *device* is the description of a complete I/O peripheral device (or board). A device 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 (they 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. This allow a device with an hardware-internal trigger to used that as soon as it is initialized, instead of requiring the user to select it. 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 5.5 \[The Attributes\]](#), page 14.

```
const struct zio_device_operations *d_op
```

The device operations are used to request input or output from/to a specific cset in the device. **Note:** we are going to remove them and have a `raw_io` function for each cset instead.

We won't describe further details of `struct zio_device` at this point, because some details may still change in the near future.

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.

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

`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 are expected to share the sample size in the data blocks.

`unsigned long flags`

Currently only the type: `ZCSET_TYPE_DIGITAL` or `ZCSET_TYPE_ANALOG`. The flag has only informative value.

`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 will allocate the array of channels when the device is registered, by replicating this template and updating the *index* value for each of them.

`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 (before removing) the channel array. They may useful to resp. setup and release the `priv_d` field.

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.

5.3 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 using a trigger type, and uses an instance of that type. Different csets can use different trigger types, because the `current_trigger` is an attribute of each cset. When the trigger fires, 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 structure used in the device.

```
const struct zio_trigger_operations *t_op
```

The trigger operations, expanded 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_cset *cset);
};
```

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 trigger instance structure, which is usually part of a larger structure that the instance itself will recover with the macro `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 (returns 0) or not (returns `-EBUSY`). This because an output trigger has only one pending data transfer. When the block is consumed, the trigger will 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 is called by the device. 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 `data_`

done later in the current trigger instance. For output csets, `data_done` frees the blocks and prepares new blocks for the next trigger event; for input, `data_done` pushes material to the buffers.

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.

change_status

abort

To be documented.

5.4 The Buffer

The buffer interface in ZIO allows to select between different allocation techniques and memory access. By splitting the buffer to a separate ZIO object, the framework allows drivers with special allocation needs to define their own hardware-specific buffer.

Each cset is using a buffer type, for which an instance exists for each channel in the cset. Data transfers only happen if a channel is enabled, so different buffers instances in the cset may at times host a different number of blocks, but this is not the usual case.

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 function 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. This pointer will be removed when the *interface* idea is implemented.

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 program accessing the buffer. The operations are associated 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 the the `open` method is called when *mmap* happens. 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,
        struct zio_control *ctrl,
        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 will usually be part of a bigger structure used internally by the buffer implementation, using the `container_of` macro to access it from the `zio_bi` pointer.

```
alloc_block
free_block
```

The buffer is concerned with memory management, 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.

```
store_block
retr_block
```

The functions simply add a block to an existing buffer instance or ask to retrieve a block out of it. In addition to managing storage according to its own will, 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 `push_block` method of the associated trigger. When `retr_block` asks for a block from an empty input buffer, the method must call `pull_block` in the associated trigger, if not NULL. Please refer to existing implementations for details.

5.5 The Attributes

Still to be written. Feel free to express your interest in this section to the mailing list.

6 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.

The first hardware parts for the real use cases are going to be available during February 2012, so this list will soon increase.

6.1 Devices

Still to be written. Feel free to express your interest in this section to the mailing list.

6.2 Triggers

Still to be written. Feel free to express your interest in this section to the mailing list.

6.3 Buffers

Still to be written. Feel free to express your interest in this section to the mailing list.

7 Writing ZIO Modules

This chapter explains how to write ZIO modules, using existing modules as a working example. This chapter shows how to use the data structures described earlier in this manual.

7.1 Locking Policies

Still to be written. Feel free to express your interest in this section to the mailing list.

7.2 Writing a Device

Still to be written. Feel free to express your interest in this section to the mailing list.

7.3 Writing a Trigger

Still to be written. Feel free to express your interest in this section to the mailing list.

7.4 Writing a Buffer

Still to be written. Feel free to express your interest in this section to the mailing list.

Index

A

AF_ZIO, network address family 3
 alignment of data samples 8
 alloc_block 14
 allocation of blocks 5
 array of channels 11

B

block 1, 4
 block allocation 5
 buffer 1, 13
 buffer, preferred 10
 buffers using system RAM 3
 bus type for ZIO 3

C

channel 1, 11
 channel template 11
 config for triggers 13
 control 1
 control structure 6
 control_structure 6
 create, for buffers 14
 create, for triggers 12
 cset 1, 2, 11
 cset, array of 10
 cset, init and exit functions 11
 csets and buffers 13

D

data block 9
 data model 3
 data_done 12
 destroy, for buffers 14
 destroy, for triggers 12
 development ideas 3
 device 1, 10
 device operations 10
 devices, supported 2
 driver 1

E

endianness of data 8
 Etherbone 11
 Ethernet support inside ZIO 3
 exit function for the cset 11

F

file operations 13
 free_block 14
 future developments 3

G

gpio 3

H

hardware-specific timestamps 9
 host identifier 8

I

index of cset and channel 8
 init function for the cset 11
 input and triggers 11
 input pipeline 5
 input subsystem and ZIO 3
 interface, to abstract file operations 3
 interleaved acquisition 7

K

kernel, supported versions 1

L

line discipline 3

M

meta-information 6
 mmap support 8, 13
 monitoring application 6

N

name of device and trigger 8
 nanoseconds in timestamps 9

O

object head in ZIO 10
 offline data management 6
 output and triggers 11
 output pipeline 5

P

PF_ZIO, network protocol family 3
 pipeline of zio data transfers 5
 preferred buffer and trigger 10
 private pointers in the channel 11
 private pointers in the cset 11
 pull_block 12
 push_block 12

R

RAM-based buffer 3
 retr_block 5
 retr_block 14

S

sample size 7, 11

samples 4
 seconds in timestamps 9
 sequence numbers in control structures 7
 size of control structure 6
 size of data samples 7
 SPI 3
store_block 5
 store_block 14
 sysfs operations 10
 system calls 5

T

timestamp 8, 9
 trigger 1, 11
 trigger in input and output flows 11
 trigger type, defining 12
 trigger, preferred 10
 triggers, software driven 3

U

uart 3

V

version numbers in ZIO 7

virtual memory operations 13

W

White Rabbit 9

Z

zero 3
zio-buffer.h 4
zio-user.h 6
 zio-zero 3
zio_block 4
zio_buffer_operations 13
zio_buffer_type 13
zio_channel 11
zio_control 6
zio_cset 11
zio_device 10
zio_get_ctrl 4
zio_is_cdone 4
zio_set_cdone 4
zio_set_ctrl 4
zio_sysfs_operations 10, 12
zio_timestamp 9
zio_trigger_operations 12