ZIO: THE ULTIMATE LINUX I/O FRAMEWORK

Alessandro Rubini (University of Pavia, Italy)
Juan David González Cobas, Tomasz Włostowski (CERN)
Federico Vaga (GNUDD)
Simone Nellaga (University of Pavia, Italy)

Abstract

ZIO (standing for "The Ultimate I/O" Framework) was developed for CERN with the specific needs of physics labs in mind, which are poorly addressed in the mainstream Linux kernel.

ZIO provides a framework for industrial, high-bandwidth, high-channel count I/O device drivers (digitizers, function generators, timing devices like TDCs) with performance, generality and scalability as design goals.

Among its features, it offers abstractions for

- both input and output channels, and channel sets
- run-time selection of trigger types
- run-time selection of buffer types
- sysfs-based configuration
- char devices for data and metadata
- a socket interface (PF_ZIO) as alternative to char devices

In this paper, we discuss the design and implementation of ZIO, and describe representative cases of driver development for typical and exotic applications: drivers for the FMC (FPGA Mezzanine Card, see [1]) boards developed at CERN like the FMC ADC 100Msps digitizer, FMC TDC timestamp counter, and FMC DEL fine delay.

MOTIVATION AND REQUIREMENTS

The initial motivation behind the development of ZIO arose in 2011, after a careful analysis of Comedi and IIO, the I/O frameworks existing at that time in the staging area of the Linux kernel source tree. It was clear that both alternatives were not suitable, generic nor complete enough for the needs of data acquisition systems like the ones at CERN or other physics laboratory facilities, where high performance and diversity of available hardware impose stringent conditions. For example, block transfers, output, fine timestamping or mmap/DMA were absent in IIO and definitely required.

As announced in [2], the design and development of a Linux kernel framework for I/O better suited to the needs of physics laboratories was initiated, with these aims in mind:

- digital and analog input and output;
- one-shot and streaming (buffered) data acquisition or waveform play;
- high resolution (under 1ns) timestamping of data blocks;
- generic coverage of resolution, sampling rate, data sizes, calibration, offset and gain parameters;
- pluggable buffer and trigger types;
- low overhead;
- support for DMA;
- bit grouping in digital I/O;
- clean design conforming to Linux kernel practice, with the intention to integrate in the mainstream kernel.

Development and Release Timeline

Alessandro Rubini and Federico Vaga started ZIO development in October 2011, as part of their collaboration with CERN BE/CO/HT section [3], within the Open Hardware initiative. Its ongoing development can be followed in the project page at the Open Hardware Repository, `http://www.ohwr.org/projects/zio`.

By May 2012, ZIO was mature enough to make its real-world debut in the experimental setting of the LNGS neutrino speed measurements [4]. At around the same time, development of the PF_ZIO network type was started by Simone Nellaga [5].

The first official release of ZIO v1.0 appeared in January 2013. Contributions have continued ever since by the three core developers, adding features, test benches, fixing bugs and the ongoing development of PF_ZIO.

HOW ZIO WORKS

As a Linux kernel I/O framework, ZIO has to provide the following basic abstractions:

- a view of I/O devices and their features,
- a mechanism to pipe data from user space applications to the raw hardware and vice versa,
- a model of the data it handles, with associated metadata,
- an appropriate interface to user space to access it all in a uniform way

The following sections explain how ZIO addresses the above abstractions in its own peculiar ways.

HOW ZIO SEES I/O DEVICES

Devices, Buffers and Triggers

I/O peripheral devices transfer data (dealt with by ZIO in so-called data blocks, to be described later) through channels. Input or output actions can happen in response

ISBN 978-3-95450-139-7

to events related to external signals, timing, or even self-timing of the device; in addition, some kind of buffer has to be in place to host the transferred data. According to this view, ZIO deals with abstractions for

devices which correspond to a specific I/O peripheral
triggers software objects which provoke I/O events when a condition they are prepared (or *armed*) to respond to occurs
buffers which take care of holding data passed back and forth

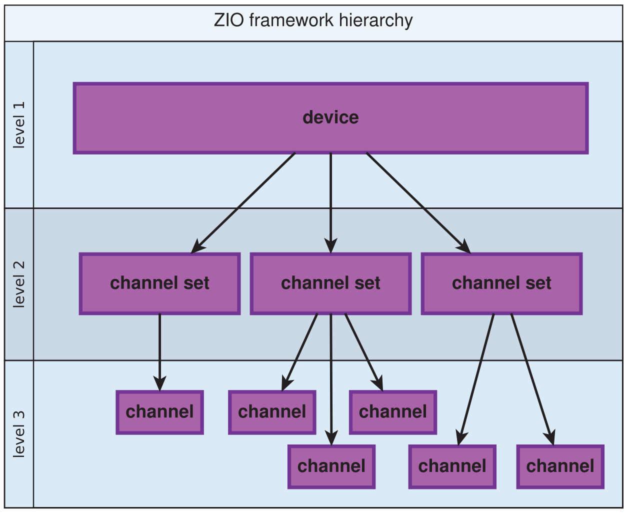


Figure 1: The ZIO framework hierarchy.

The standard ZIO distribution provides ready-to-use examples of these three concepts. In particular, generic RAM-based buffer types are provided based on `kmalloc` and `vmalloc`. Basic trigger types that come with stock ZIO are:

transparent trigger activated by software read/write request or by self-timing devices
kernel timer activated periodically
high resolution timer periodic or one-shot
external activated by external interrupt

Additionally, add-on drivers can register their own trigger type, so for example an ADC card can provide data-driven triggers for a scope-like application. An example is the FMC ADC driver developed by Federico Vaga which we will reference later.

Channel Sets

The core concept of the device model of ZIO is the *channel set* (cset for short): a set of channels of identical characteristics that is associated with a trigger instance. Such an instance is a software object prepared to react to a particular type of trigger event and provoke the actual I/O action. Trigger instances are, as one could expect, instantiations of trigger types. The consequence is that all channels in a cset are always affected as a whole by a particular I/O event. A device can contain various csets, which gives rise to the hierarchical structure of Fig. 1.

In addition, each channel in a cset gets a buffer instance of a buffer type associated with the cset.

To summarize, csets contain channels whose I/O events are driven by a trigger instance of a particular type, i.e., reacting to a particular class of trigger events; the cset channels are homogeneous and have buffers, all belonging to the cset buffer type. The relationships are displayed in Fig. 2.

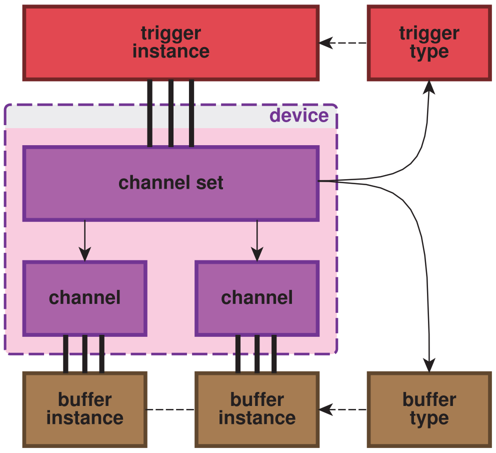


Figure 2: A ZIO channel set (cset).

HOW ZIO SEES DATA FLOW: THE DATA PIPELINE

In ZIO, each I/O event is the transfer of a data block, which consists of zero or more samples. The path a data block traverses between the I/O peripheral and the user space application is depicted in Fig. 3.

The lifetime of a data block during a transfer is easy to understand from the picture. In a `read` call, the buffer receives a `retr_block` request that it honours if an input block is already buffered. Otherwise, the buffer requests the trigger with a `pull_block` to notify the trigger that a transfer should proceed; most often, this method is absent and the trigger acts by itself. The actual I/O is initiated when the trigger calls `raw_io`, its completion being notified to the trigger by a `data_done` callback.

Output occurs in a dual manner, the roles of all methods being identical or parallel to the input case.

For the many particular corner cases that may arise we refer to the user manual [6].

It is interesting to note the following

- the input and output pipelines are symmetric.
- ZIO components have a simple core interface. Buffers provide methods for the allocation, storage, retrieval and freeing of the active data block. Triggers, on their side, communicate with the peripheral driver with `raw_io` and `data_done` methods; in addition, explicit read/write requests are communicated to triggers via `pull_block` and `push_block` methods.

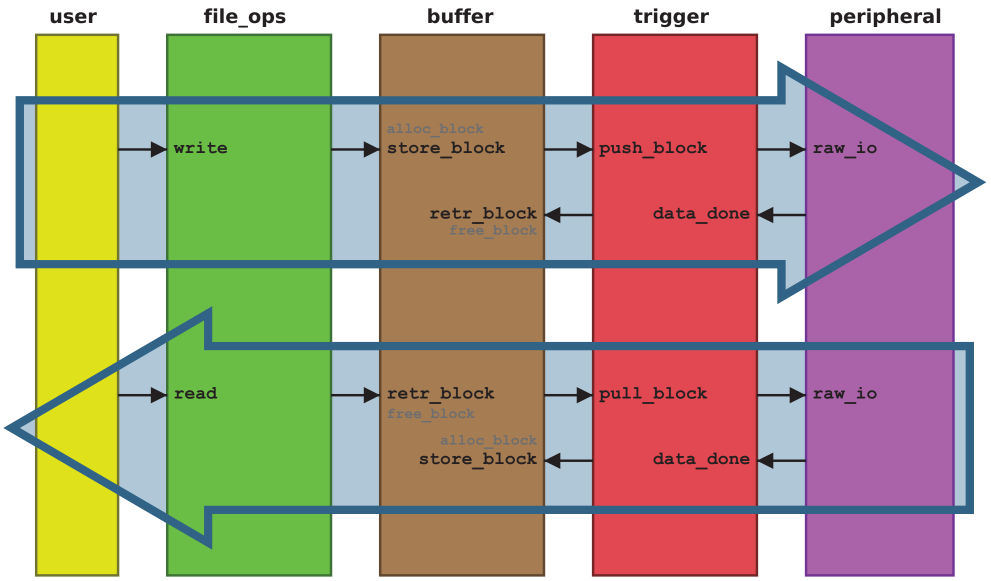


Figure 3: The ZIO data pipeline.

- the overall flow is very simple: in normal conditions, all that is involved is `store_block` and `retr_block` for the buffers, and `raw_io` and `data_done` for the triggers. In addition `push_block` and `pull_block` can explicitly initiate transfers, or restart them when they stopped because of data starvation.

HOW ZIO SEES DATA: THE DATA MODEL

The most original and fertile idea in ZIO is its data model, based on a structure named *block*. A block consists of two parts

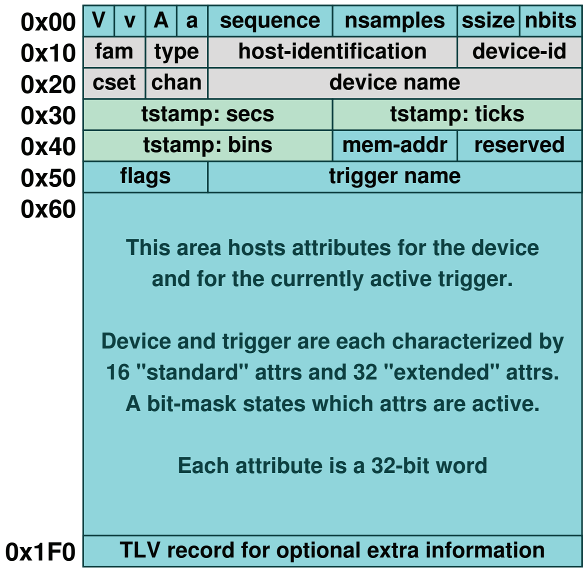
- metadata associated with the I/O event that produced the data, in the form of a fixed-size structure called a *control* in ZIO jargon.
- the actual data, a possibly huge number of samples, in the form of a payload completely transparent to ZIO.

It is interesting to take a look at the layout of the control data structure, as depicted in Fig. 4.

The structure is defined to be always 512 bytes long. The two most important sub-structures in the control are a high-resolution time stamp (green in the figure), which can be either software- or hardware-generated, and a complete, world-unique, identification of the channel this block belongs to (gray in the figure). Such data structure is used as socket address in PF_ZIO, so applications can deterministically `sendto()` and `recvfrom()` in the ZIO network.

The leading fields identify the version of the structure, the sequence number and size of this block, as well as alarm bits to persistently report errors in the stream for this channel (alarm conditions are reset by bitwise sysfs writes).

The bulk of the control structure then lists attributes for both the current channel and the current trigger. In this way, a ZIO block carries the complete metadata information together with the data, offering a flexible transport interface where only the endpoints of the ZIO pipeline are concerned with the inner details of the specific device or trigger.



0x00	V	v	A	a	sequence	nsamples	ssize	nbits
0x10	fam		type		host-identification		device-id	
0x20	cset		chan		device name			
0x30	tstamp: secs					tstamp: ticks		
0x40	tstamp: bins					mem-addr	reserved	
0x50	flags				trigger name			
0x60	This area hosts attributes for the device and for the currently active trigger. Device and trigger are each characterized by 16 "standard" attrs and 32 "extended" attrs. A bit-mask states which attrs are active. Each attribute is a 32-bit word							
0x1F0	TLV record for optional extra information							

Figure 4: The ZIO control structure.

HOW ZIO IS SEEN FROM USER SPACE

ZIO interfaces with user space by means of character devices, two per channel: one for control and one for data. The most common data flow is depicted in Fig. 5. In any case, the user can choose not to read or write either the control or the data; the semantics allows an application, for example, to ignore the metadata if it knows it is acquiring a stream of similar data blocks. More complex and demanding data flows are possible, allowing access to mmapped data using the vmalloc buffer type.

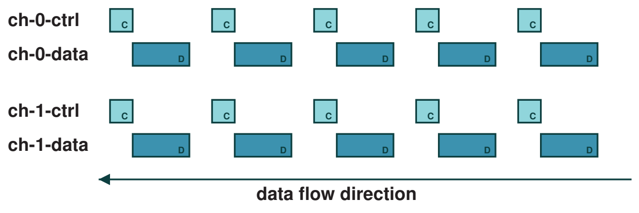


Figure 5: The ZIO control and data streams.

ZIO also offers a rich sysfs interface, with sets of standard and extended attributes assigned to devices, triggers and csets. This results from the basic ZIO design principle of “no `ioctl()`, thanks” for out-of-band device configuration. Apart from the intrinsic interest as an interface, sysfs attributes prove invaluable in testing and debugging. The attributes are automatically mapped to the control structure, so the sysfs and the char device views are consistent.

Last, but not least, PF_ZIO is another development in beta stage that allows to perform I/O through a standard socket interface. Using a single PF_ZIO socket, an application can exchange data blocks with several channels; this is especially useful when collecting events from sensor networks with sporadic data but high cardinality—like neutrino detectors. PF_ZIO supports all three types of socket: stream, datagram and raw.

ZIO IN PRACTICE: FMC ZIO DEVICE DRIVERS

The BE-CO-HT section at CERN has developed a kit of modules described in [7, 8], compliant with the ANSI VITA 57 FMC (FPGA Mezzanine Card) standard [1]. Linux device drivers for the following modules have been developed using the ZIO framework.

- FMC Delay 1ns 4cha (`http://www.ohwr.org/projects/fmc-delay-1ns-8cha`)
- FMC ADC 100M 14b 4cha (`http://www.ohwr.org/projects/fmc-adc-100m14b4cha`)
- FMC TDC 1ns 5ch (`http://www.ohwr.org/projects/fmc-tdc`)

Although both the ADC and the TDC are good examples of devices ZIO was conceived for, it is by an accident of history that the FMC Fine Delay was the first CERN device whose low-level software was entirely ZIO-based.

The first stable release of the ZIO-based FMC ADC driver appeared in July 2013. Interestingly, this module’s particular features (e.g. multi-shot acquisition) required the development of a specific trigger type. Again, the flexibility built into the framework made it adapt smoothly to new requirements.

The FMC TDC drivers are in the final stage of development and will be probably released by the time this paper is published. In this particular case, none of the two developers were members of the original ZIO core development team. They used ZIO to great profit without much difficulty in learning its internals.

CONCLUSIONS

The experience of the FMC drivers development shows that a diversity of devices is handled well within the ZIO framework, even when new features require special types of trigger or buffer.

It might appear that the various abstractions built into ZIO make its learning curve steep; however, as the FMC TDC example shows, developers can quickly get up to speed with it and produce drivers for complex hardware. This is made possible by reference implementations (the FMC Fine Delay driver) and instructive example drivers supplied by the ZIO standard distribution. Moreover, a common interface and set of tools for monitoring and debugging the developed drivers reward the initial learning effort.

REFERENCES

[1] VME International Trade Association, “FPGA Mezzanine Card (FMC) Standard”, `http://www.vita.com/`

[2] J.D. Gonzalez Cobas, S. Iglesias Gonsalvez, J.H. Lewis, J. Serrano, M. Vanga, E.G. Cota, A. Rubini and F. Vaga, “Free and Open Source Software at CERN: Integration of Drivers in the Linux Kernel”, ICALEPCS’2011, Grenoble, October 2011, THCHMUST04, pp.1248-1251 (2011), `http://www.JACoW.org`.

[3] F. Vaga, “Development of an I/O framework within the Linux kernel for high-bandwidth data transfers”, Tesi di Laurea, Facoltà Di Ingegneria, Università degli Studi di Pavia, 2011.

[4] F. Pietropaolo *et al*, “Precision measurement of the neutrino velocity with the ICARUS detector in the CNGS beam”, Journal of High Energy Physics, 2012 (11): 49, pp. 1–21.

[5] S. Nellaga, “Realizzazione di un protocollo di rete per Input/Output industriale”, Tesi di Laurea, Facoltà Di Ingegneria, Università degli Studi di Pavia, 2012.

[6] A. Rubini and F. Vaga, “ZIO User Manual (version 1.0)”, `http://www.ohwr.org/attachments/1896/zio-manual-130121-v1.0.pdf`.

[7] P. Alvarez, M. Cattin, J. H. Lewis, J. Serrano and T. Wlostowski, “FPGA Mezzanine Cards for CERNs Accelerator Control System”, in ICALEPCS’09, p. 376, 2009.

[8] E. Van der Bij, M. Cattin, E. Gousiou, J. Serrano and T. Wlostowski, “ CERN’s FMC Kit”, ICALEPCS’2013 (to appear), WECOCB01; www.JACoW.org