The Industrial I/O subsytem

Jonathan Cameron, jic23@cam.ac.uk

March 3, 2009

DRAFT: May well have chunks of completely gibberish.

This document is intended to provide an overview of the structure and inner workings of the Industrial I/O subsytem. It is worth noting right from the start that this subsystem is still very much in development and not all functionality we hope to include in the future will be described by this document. For example we will not discuss the as yet non existent support for output devices.

Why the name? Basically it was suggested on lkml and is the best option anyone has come up with yet. Other suggestions welcomed though obviously we want to get this fixed before going for a mainline merge.

1 Motivation for yet another subsystem

The decision to investigate a new subsystem came out of discussions on LKML about where to put drivers for the numerous, typically embedded, sensors that people increasingly want to connect to linux devices. Here we are interested in sensors connected to busses such as I2C or SPI or more chip specific interface which would typically be implemented using the generic gpio framework.

Both I2C and SPI have now been well supported by the linux kernel for some time with new bus controllers being added all the time.

The set of requirements being discussed were:

1. Simple grouping of drivers offering similar functionality both within the kernel source tree and as a sysfs class.
2. Consistent set of sysfs files to read data directly from sensors and control sensor configuration
3. Event interface to allow events such as threshold breaches as supported by the hardware to be communicated up to userspace.
4. Triggered capture of data.
5. Low latency means of ensuring few if any samples are missed even at several ksamples/sec on typical embedded hardware. (ring buffer)

The obvious options for were to place such drivers were:

i2c chips and equivalent In general grouping devices by connectivity is a bad idea. In the case of many of the sensors being considered the hardware provides more than one interface option anyway. The I2C chips directory is being deliberately emptied for exactly the reason that the drivers within it should be elsewhere.

hwmon Some of the sensors being considered might well fit within hwmon; ADC's are simply voltage measurement devices. However, many applications of these sensors require data rates well in excess of those for which hwmon was designed. Hwmon's purpose is to provide access to devices monitoring sensors related to machine health which typically need only be updated at a few hz or less.

input This is the existing subsystem which is closest to meeting the requirements. Input supports for events being passed up to userspace and relatively high rate data capture. However it is designed to anonymize the data in a way that would make extensive support of device specific elements tricky. Input is interested in what the event was, not typically where it came from. A large amount of the IIO code is directly based on that within the input core.

The conclusion of that discussion was that there was no particularly suitable place for such drivers and they were of sufficient general interest to motivate a new subsystem more closely focussed on these sorts of sensor chips.

Thanks to all those who contributed to that discussion, in particular Jean Delvare and Dmitry Torokhov for there comments on input and hwmon (amongst others).

1.1 Typical sensors at which IIO is directed

Whilst is difficult to pick a small group of sensors from which to considerable what is desirable functionality, the selection of a reasonably widely varying example set does provide motivation for different elements of an initial prototype. Obviously a certain amount of bias in the sensor selection is due to which chips come in convenient prototyping forms.

The sensors selected were:

- Maxim MAX1363 ADC: Simple i2c ADC with some event detection features. Data capture occurs on demand.
- ST Microelectronics LIS3L02DQ Accelerometer: Dual SPI/I2C interfaced accelerometer. A number of different event detectors and continuously captures data with new data being available signaled via a data ready pin.
- VTI SCA3000-E05 Accelerometer: Hardware ring buffer equiped accelerometer with a number of more unusual event interfaces.

2 Key System Elements

Figure 1 gives a simple overview of the key elements of IIO and some of their more important interfaces to userspace. Throughout this explanation we will assume we are dealing with the first device to be registered with IIO.

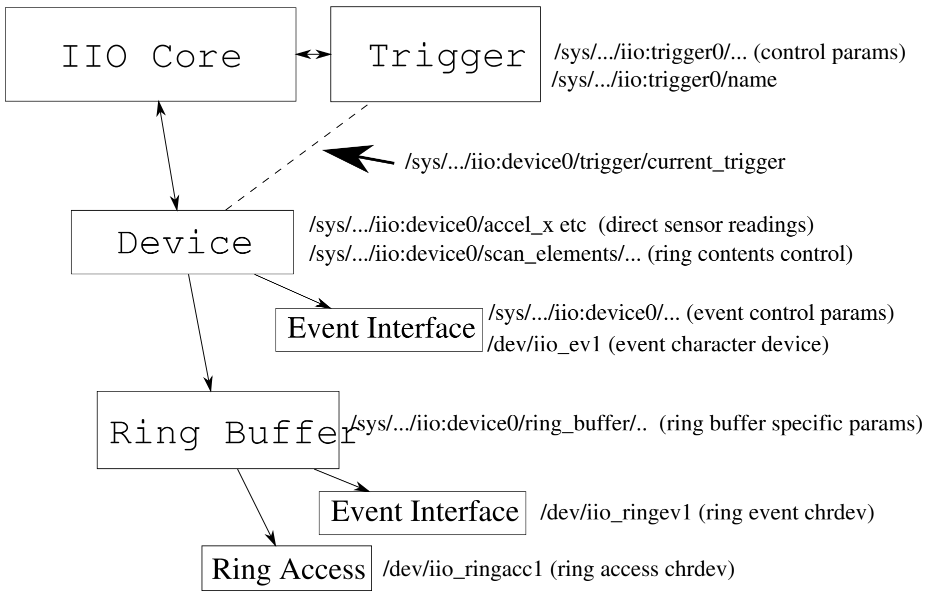
IIO Core
Trigger
/sys/.../iio:trigger0/... (control params)
/sys/.../iio:trigger0/name
/sys/.../iio:device0/trigger/current_trigger
Device
/sys/.../iio:device0/accel_x etc (direct sensor readings)
/sys/.../iio:device0/scan_elements/... (ring contents control)
Event Interface
/sys/.../iio:device0/... (event control params)
/dev/iio_ev1 (event character device)
Ring Buffer
/sys/.../iio:device0/ring_buffer/.. (ring buffer specific params)
Event Interface
/dev/iio_ringev1 (ring event chrdev)
Ring Access
/dev/iio_ringacc1 (ring access chrdev)

Figure 1: IIO element interactions and some userspace interfaces.

2.1 IIO Core

The Industrial I/O core is similar to that provided by input (and several other subsystems) in that drivers register the facilities they provide with the core and it handles allocation of certain resource types (sysfs directories and chrdevs). Also provided by the core is the ability to link together certain elements provided by different drivers.

2.2 Devices

In IIO, the actual sensors are refered to as devices. Any sensor using the subsystem must register itself as a IIO device. In the simplest polling only drivers, this will only require provision of basic attribtues to read from the different sensor channels. Things are a little more complex if ring buffer support and event interfaces are required.

2.3 Event Interfaces

These correspond to character devices used to pass events up to userspace. They typically correspond to underlying hardware interrupts. A good example is a free fall detector on an accelerometer. Each event set has a corresponding character device on which a userspace program may listen for events and typically each event will have control functions, which alongside setting relevant parameters, will allow the generating hardware to be disabled, or failing that the event to simply be dropped transparently.

Many hardware provided events are from a relatively small and predictable set of types. These 'semi-generic' events have formally defined interfaces. If other, uncommon events are to be provided it is up to the driver writer to chose names etc. Commonly occuring events not yet specified via header macros will be added.

2.4 Ring Buffers

Ring buffers fall into two categories, those that are provided by hardware devices and those that are simply useful software structures. The intention is that to userspace these should look as similar as possible.

Every ring buffer will have a pair of character devices. The first is an event interface which is used to notify userspace of ring buffer related events. The second is used to actually read data from the ring buffer itself.

2.4.1 Ring Buffer Event Interface

Whilst most ring buffer events that should be provided to userspace are related to how full the ring buffer is, it is difficult to define a *core* subset of events. For example, if a hardware ring buffer is used, then it may well generate interrupts related to how full the buffer is, where the values these actually take are very much dictated by the hardware implementation rather than what would be ideal in a userspace case. This event interface is very similar to the more generic interface that hardware uses to signal events via an interrupt.

2.5 Triggers

Industrial I/O elements which are able to cause data to be captured to a ring buffer are known as triggers. They may take many forms and may be associated with an IIO device or may simply provide triggering facilities.

A device is associated with a particular trigger by writing the triggers name to

```
/sys/.../iio:device0/trigger/current_trigger
```

This results in up to two functions being added to lists run on the trigger event occuring. These functions must all be capable of running within an interrupt. The two functions are encapsulated within an struct iio_poll_func . There are two lists to allow for the case where a lot of sensors are being triggered by a single trigger and some of them have hardware latches. The first element, poll_fun_immediate is used to set any latches before the more time consumering poll_func_main list. Any driver without a latch will only have a poll_func_main defined.

3 Example of driver initialization

In order to illustrate what the IIO core is actually doing, let us consider what happens on loading of a new IIO device driver. Elements are graded by the type of driver that will make use of them. Those in green will occur for all drivers, orange are only relevant if the device has any hardware event detection elements. Finally, those in red only occur if the device driver is going to utilize a ring buffer.

1. Device Specific Initialization: Conventional device probing and initial configuration.
2. Initialize elements of a struct iio_dev: This tells the core about the functionality this particular driver provides.
3. Register with core: Call iio_device_register () This does the following.
   (a) *Set the device drv_data pointer to point at the struct iio_dev. Actual device specific data can be accessed via the relevant pointers in this structure.*
   (b) *Get a device id. These are unique identifiers, used for things like sysfs naming.*
   (c) *Create a iio : device [.] entry in the /sysfs/industrialio directory and populate it with device type specific attributes.*
   (d) *Register any eventsets . EXPLAIN WHAT THESE ARE.*
   (e) *Initialize any ring buffers.*
   (f) *Register the device as a trigger consumer. This also creates the trigger sub directory to allow the trigger to be specified.*
4. Register any device supplied triggers with the core using iio_trigger_register .

4 How a typical ring buffer capture configuration works

CLEAN UP

1. Configure the device ring buffer and scan mode. Write required length to ring_buffer /length and selected elements for scan using files in scan_elements directory.

2. Associate a trigger with the iio :device. This is done by writing the unique trigger name to trigger / current_trigger . This will result in the two trigger functions being added to the relevant lists under the trigger.

3. Configure the trigger. This is very trigger type dependant. For periodic triggers, things like configuring the sampling frequency are done using sysfs files in the relevant iio_trig : directory.

4. Enable the trigger using the enable sysfs file in the trig directory.

Once this has happened, a trigger interrupt will first call all the functions in the associated immediate trigger list (typically latch values on sensors that support it) and then call those in the secondary list which will typically schedule interrupt bottom halves to actually perfom the readings as soon as possible. The readings cannot normally be taken in interrupt context as many forms of bus read require sleeping.

5 Questions

Why not do as much of this as possible in userspace? Much of the IIO subsystem and associated drivers could indeed by provided by a fairly complex user library. The principle arguement in favour of such an approach would be that enhanced stability of the kernel as a whole as individual drivers are much less likely to bring it down. There may also be an advantage in reducing the number of times data must be coppied to get a result up to userspace.

However, there are a few issues that make this rather tricky to actually do.

1. In many cases the ADCs form part of more complex devices which already have a support within the kernel. (e.g. PMICs such as those supported by the DA903x mfd driver) and as such any additional interface will require at least some kernel parts. Other places I've seen ADCs that I'd like to have available via a consistent interface, include firewire and usb cameras (aux ADC's and DAC's are common for controlling pan tilt heads etc). Some of these could be implemented in userspace (and quite possibly should be,) but others would require driver changes that will effectively equal those of adding direct support for the subsystem described in these patches.

2. Spidev and i2cdev are fairly restricted interfaces which I have used extensively for initial investigation of sensors. The key thing here is that almost no devices use purely these interfaces. They include interrupt lines etc requiring relatively complex handlers. Thus any driver would probably require at least some kernel space element per chip anyway. Also worth noting that this tends to be the most complex and hence probably error prone part of these drivers. The spidev documentation talks about the limitations of the driver.

   An example of the problems that would be encountered with the current i2c-dev implementation within a full driver would be a control register read on a VTI SCA3000 (i2c) accelerometer. This would require an initial 3 byte write followed by a 1 byte read. In kernel space this whole thing can be set up in a dma capable buffer and done in one go with copies out to userspace only needed if the value is meant to be read by the user.

3. Need to provide kernel space interfaces. Whilst the subsystem as a whole does not currently provide kernel interfaces, the future road map includes providing such a famework. Examples of users within the kernel include things like battery chargers, free fall detectors etc. The systems that use these sometimes require kernel level access which a userspace driver clearly cannot supply.

4. Performance: Thinks like time stamping of data ready signals needs to be as near as possible to the source interrupt. As for the complex question of which is optimal for the ring buffering functionality, this would need thorough optimization of both approaches before it could be addressed. I'm not aware of any similar comparisons but would be grateful if anyone could direct me to any they know of. It is also worth noting here that many applications that will use the ring buffered interface will never actually read the contents of the buffer as they are interested in only restricted regions of data (based on some external trigger).

(Answer original sent in response to a query from Alan Cox on LKML)

Is there enough shared functionality to justify the bulk of this subsystem? This is still at least partly an open question. For drivers that only supply the minimum functionality (sysfs polling of readings only) then all the subsystem provides is a consistent definition of userspace interfaces.

For more advanced drivers, the event system and ring buffers (particularly the trigger based approach to data capture) are the elements that are most consistently useful across a range of devices. As more devices are supported by IIO drivers, other shared elements may become apparent.

Should IIO provide interfaces such as input and hwmon for devices which may have a dual role? This is still very much an open question. Two basic approaches come to mind. Either this can be handled by the IIO subsytem itself with some sort of generic framework supply the functionality required by the various subsytems, or this may be left to the individual drivers which are able to register with several different subsystems.

In the case where IIO provides the functionality, it may be that the requirements are sufficiently different that we end up with an unmaintainable and untestable mess. If the drivers handle the different subsystems they two may in some cases be over complicated.

In conclusion, it may be that as some respondants to discussions on LKML have suggested, that it is more appropriate to maintain more than one driver for a given sensor, allowing them to be tuned to the application of interest. This question will probably only be resolved as the system gets more heavily used.