High-speed Data Acquisition
using the
Linux Industrial IO framework

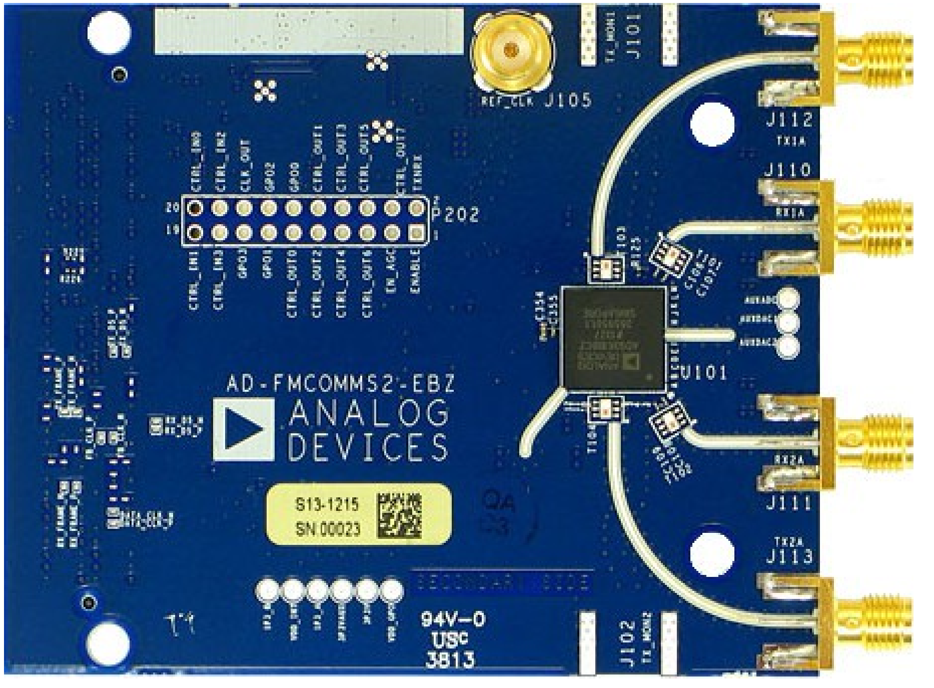
Lars-Peter Clausen, Analog Devices

What is High-Speed

- > ~100k samples per second
- Applications
  - RF communication, Software Defined Radio, Direct RF
  - Radar
  - Ultrasound
  - Measuring equipment, Spectrum analyzer
  - Usually NOT: Power monitoring, HID

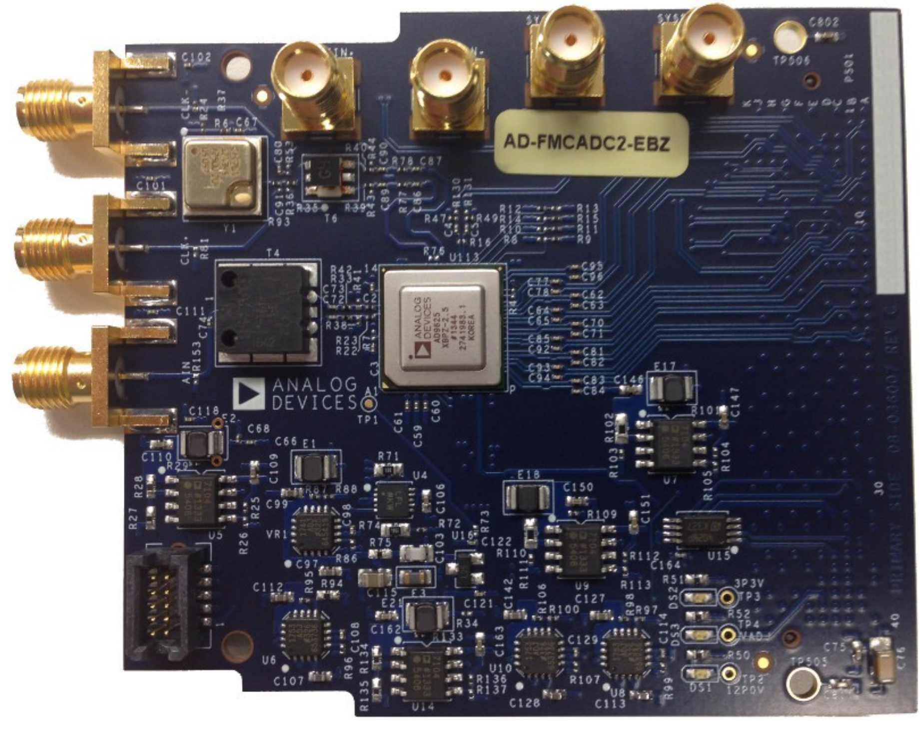
Example I: AD-FMCOMMS2-EBZ

- Software Defined Radio platform
- AD9361 Agile transceiver
- 200 kHz - 56 MHz sample rate
- 2 Channels of RX and TX
  - Each channel a set of 12-bit I and Q data
  - Samples are stored in 16bit words
  - 1 - 450 MB/s in each direction



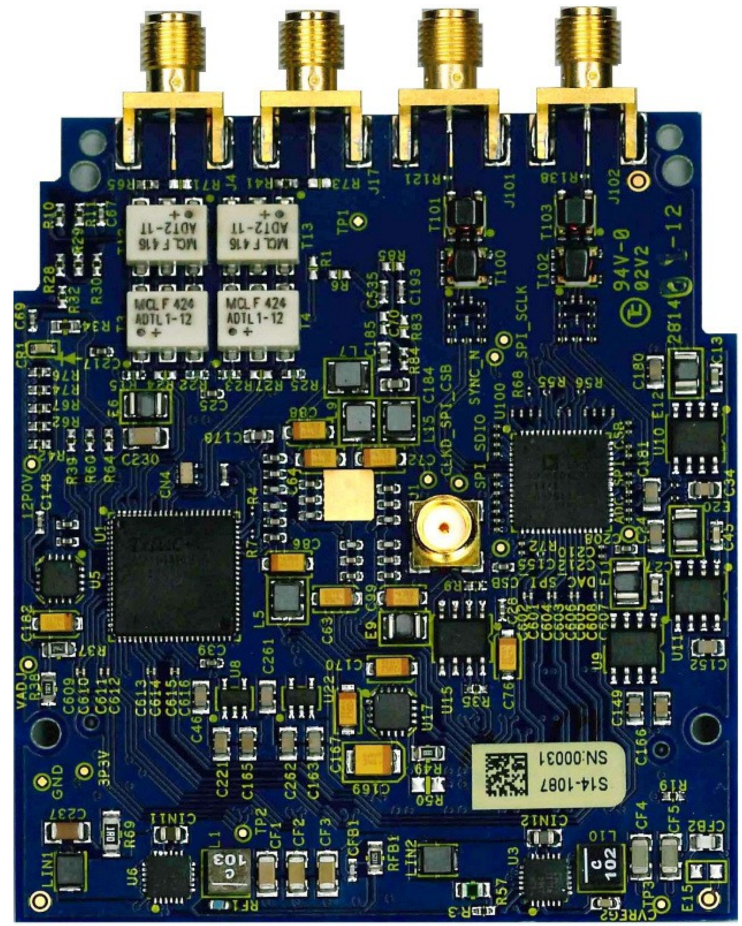
Example II: AD-FMCADC2-EBZ

- AD9625 High-Speed ADC
- JESD204B interface
- 2.5 GHz sample rate
- 12-bit stored in 16-bit word
- 5 GB/s



Example III: DAQ2

- High speed data acquisition board
- AD9680, dual channel 14-bit, 1GSPS ADC
- AD9144, quad-channel 16-bit, 2.8 GSPS DAC
- 4 GB/s receive
- 22.4 GB/s transmit



Why use Linux

- Modern systems are diverse and complex (horizontally and vertically)
  - Many different components from different vendors
  - Same components are used in different solutions
- Wide range of supported hardware
  - Excellent support for additional peripherals
    - Applications processor
    - Storage (SATA, SD, ...)
    - Connectivity (Ethernet, WiFi, USB, …)
    - Human Interface (Graphics, Keyboard, Mouse, …)
    - ...

Why use Linux

- No need to reinvent the wheel
  - Leverage existing solutions
  - Focus on solving the problem
  - Reduces development cost and time to market

What is IIO

- Industrial Input/Output framework
  - Not really just for Industrial IO
  - All non-HID IO
  - ADC, DAC, light, accelerometer, gyro, magnetometer, humidity, temperature, rotation, angular momentum, ...
- In the kernel since v2.6.32 (2009)
- Moved out of staging/ in v3.5 (2012)
- ~200 IIO device drivers (v3.17)
  - Many drivers support multiple devices

Why use IIO

- Distinction between high-speed and low-speed is fuzzy
- Large parts of the existing infrastructure can be reused
  - E.g. configuration and description API
  - High-speed only needs a new transport mechanism for data
- Allows sharing of (existing) userspace tools

Traditional IIO data flow - Kernel

- Driver registers trigger handler
- Trigger handler is called for each sample
- Trigger handler reads data and passes it to the IIO core

```
static irqreturn_t ad7266_trigger_handler(int irq, void *p)
{
    …
    spi_read(st->spi, st->data.sample, 4);
    iio_push_to_buffers_with_timestamp(indio_dev, &st->data,
                pf->timestamp);
    iio_trigger_notify_done(indio_dev->trig);
    …
}


  …
  ret = iio_triggered_buffer_setup(indio_dev, &iio_pollfunc_store_time,
          &ad7266_trigger_handler, &iio_triggered_buffer_setup_ops);
  …
```

Kernel – Issue I

- One interrupt per sample
  - Large overhead
  - Limits the samplerate to a few kSPS

Kernel – Issue II

- Multiple memory copies per sample
  - *iio_push_to_buffers*()
    - Sample demuxing
  - Peripheral access (SPI/I2C/USB/...)
- Impacts performance for larger data sets

Traditional IIO data flow - Userspace

- Sample data is transferred between userspace and kernelspace by write()/read()

```
...
fd = open("/dev/iio:device4", O_RDONLY);
...
while (...) {
    read(fd, buf, sizeof(buf));
    process_data(buf);
}
...
```

Userspace – Issue I

- *read()/write()* does a memory copy
- Impacts performance for larger data sets

Issues - Summery

- One interrupt per sample
  - Limits the maximum sample rate to a few 100 kSPS
- Using *read()/write()* requires a memory copy

Design goals for the new API

- Reduce number of interrupts
- Reduce number of memory copy operations

Solution I - Blocks

- Group multiple samples into a “block”
- Only generate one interrupt per block
  - Reduces management overhead
- Size of one block should be configurable
  - Allows application to make tradeoffs between latency and management overhead

Solution II – DMA + mmap()

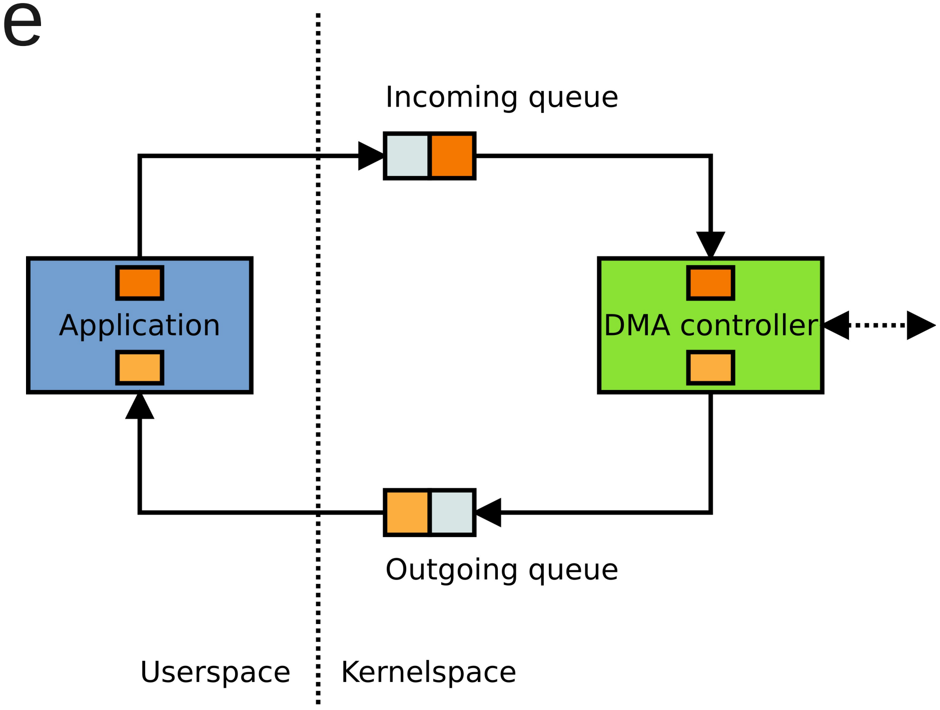
- Use DMA to transfer data from peripheral to memory
- Use *mmap()* to make the memory accessible from userspace

  => No memory copy necessary

  => De-muxing in userspace for free

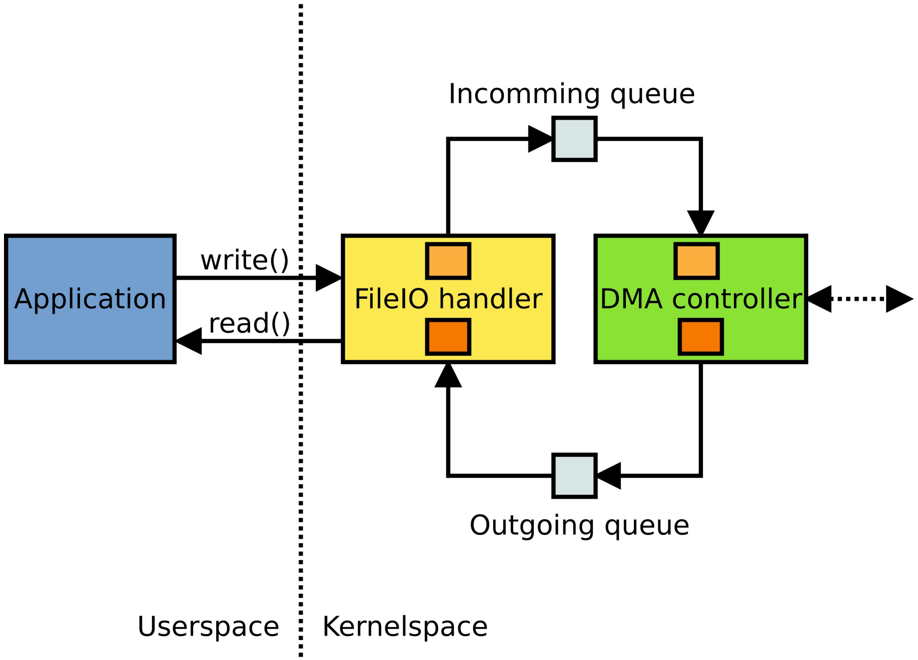
New DMA based capture flow

1. Application allocates blocks
2. Enqueues them in the incoming queue
3. DMA controller processes it
4. Puts it in the outgoing queue
5. Application dequeues it
6. Application processes it
7. goto 2



DMA capture flow – FileIO Mode

- Kernel controls block flow
- Implements read()/write() interface
- Compatibility with existing applications



Userspace ABI

- 5 ioctls
  - IIO_BLOCK_ALLOC_IOCTL
  - IIO_BLOCK_FREE_IOCTL
  - IIO_BLOCK_QUERY_IOCTL
  - IIO_BLOCK_ENQUEUE_IOCTL
  - IIO_BLOCK_DEQUEUE_IOCTL
- 2 structs
  - struct iio_buffer_block_alloc_req
  - struct iio_buffer_block
- mmap()

IIO_BLOCK_ALLOC_IOCTL

- Creates and allocates new blocks
- Can be called multiple times to allocate blocks of different sizes
- After allocation the blocks are owned by the application

```
struct iio_buffer_block_alloc_req {
	__u32 type;
	__u32 size;
	__u32 count;
	__u32 id;
};

int ioctl(int fd,
    IIO_BLOCK_ALLOC_IOCTL,
    struct iio_buffer_block_req *);
```

IIO_BLOCK_FREE_IOCTL

- Frees all previously allocated blocks
  - Necessary to keep block ids contiguous

```
int ioctl(int fd, IIO_BLOCK_FREE_IOCTL);
```

IIO_BLOCK_QUERY_IOCTL

- Gets the current state of a block from the kernel

```
struct iio_buffer_block {
	__u32 id;
	__u32 size;
	__u32 bytes_used;
	__u32 type;
	__u32 flags;
	union {
		__u64 offset;
	} data;
	__u64 timestamp;
};

#define IIO_BUFFER_BLOCK_FLAG_TIMESTAMP_VALID (1 << 0)
#define IIO_BUFFER_BLOCK_FLAG_CYCLIC (1 << 1)

int ioctl(int fd, IIO_BLOCK_QUERY_IOCTL,
    struct iio_buffer_block *);
```

IIO_BLOCK_ENQUEUE_IOCTL

- Transfers ownership of a block from the application to the kernel
- Kernel passes block to the driver for DMA transfer setup when IIO buffer is enabled

IIO_BLOCK_DEQUEUE_IOCTL

- Transfers ownership of the first completed block from the kernel to the application
- Also gets the current state of the block (like IIO_BLOCK_QUERY_IOCTL)
- Blocks if no completed block is available
  - -EAGAIN if fd is non-blocking

New IIO data flow – Userspace I

```
struct block {
    struct iio_buffer_block block;
    short *addr;
} blocks[4];

struct iio_buffer_block_alloc_req alloc_req;

fd = open("/dev/iio:device4", O_RDONLY);

memset(&alloc_req, 0, sizeof(alloc_req));
alloc_req.size = 0x100000;
alloc_req.count = ARRAY_SIZE(blocks);
ioctl(fd, IIO_BLOCK_ALLOC_IOCTL, &alloc_req);

for (i = 0; i < alloc_req.count; i++) {
    blocks[i].block.id = alloc_req.id + i;
    ioctl(fd, IIO_BLOCK_QUERY_IOCTL, &blocks[i].block);
    blocks[i].addr = mmap(0, blocks[i].block.size, PROT_READ,
            MAP_SHARED, fd, blocks[i].block.data.offset);
    ioctl(fd, IIO_BLOCK_ENQUEUE_IOCTL, &blocks[i].block);
}
```

New IIO data flow – Userspace II

```
...
while (...) {
    ioctl(fd, IIO_BLOCK_DEQUEUE_IOCTL, &block);
    process_data(blocks[block.id].addr);
    ioctl(fd, IIO_BLOCK_ENQUEUE_IOCTL, &block);
}
...
```

Kernel space API

- New callbacks in the iio_buffer_access_funcs struct matching the new IOCTLs

```
struct iio_buffer_access_funcs {
    …
    int (*alloc_blocks)(struct iio_buffer *buffer,
        struct iio_buffer_block_alloc_req *req);
    int (*free_blocks)(struct iio_buffer *buffer);
    int (*enqueue_block)(struct iio_buffer *buffer,
        struct iio_buffer_block *block);
    int (*dequeue_block)(struct iio_buffer *buffer,
        struct iio_buffer_block *block);
    int (*query_block)(struct iio_buffer *buffer,
        struct iio_buffer_block *block);
    int (*mmap)(struct iio_buffer *buffer,
        struct vm_area_struct *vma);
    ...
};
```

struct iio_dma_buffer_ops flow

```
static int hw_submit_block(void *data,
    struct iio_dma_buffer_block *block)
{
    /* Setup hardware for the transfer */
}

static irqreturn_t hw_irq(int irq, void *data)
{
    /* Get handle to completed block */
    ...
    iio_dma_buffer_block_done(block);
    ...
}
```

struct iio_dma_buffer_ops flow

```
static const struct iio_dma_buffer_ops hw_dmabuffer_ops = {
    .submit_block = hw_submit_block,
};

static int hw_probe(...)
{
    …
    buffer = iio_dmabuf_allocate(dev, &hw_dmabuffer_ops,
        priv_data);
    ...
}
```

DMAengine based implementation

- Generic DMAengine API based implementation of the *submit_buffer()* callback
  - Detects capabilities of the DMA controller using *dma_get_slave_caps()*
  - If your DMA controller has a DMAengine driver it works out of the box

Upstream status

- Code mostly ready, but not upstream yet
- Multiple stages
  - Internal infrastructure for generic DMA support
    - Aiming for 3.19
  - Output buffer support
    - Aiming for 3.19-3.20
  - Userspace ABI extensions
    - mmap support, allocate and manage blocks
    - Aiming for 3.21-3.22

Future work

- Componentization
  - Split Converter, PHY and DMA driver
  - Flow graph (media controller API?)
- Zero copy
  - Generic zero copy, e.g. to disk or network
  - *vmsplice(..., SPLICE_F_GIFT)* (?)
- DMABUF support
  - Offloading of buffers to other devices, e.g. accelerators (DSP, GPGPU, FPGA, ...)

Q/A

Further information

- https://github.com/orgs/analogdevicesinc
  - https://github.com/analogdevicesinc/libiio
  - https://github.com/analogdevicesinc/iio-oscilloscope
  - https://github.com/analogdevicesinc/linux
- http://wiki.analog.com/resources/tools-software/linux-software/libiio_internals
- http://analogdevicesinc.github.io/libiio/
- http://wiki.analog.com/resources/tools-software/linux-software/iio_oscilloscope