# Linux Misc-devices Documentation

The kernel development community

Jul 14, 2020

# CONTENTS

This documentation contains information for assorted devices that do not fit into other categories.

Table of contents

CHAPTER

ONE

# KERNEL DRIVER EEPROM

Supported chips:

- Any EEPROM chip in the designated address range

  Prefix: 'eeprom'

  Addresses scanned: I2C 0x50 - 0x57

  Datasheets: Publicly available from:

  Atmel (www.atmel.com), Catalyst (www.catsemi.com), Fairchild (www.fairchildsemi.com), Microchip (www.microchip.com), Philips (www.semiconductor.philips.com), Rohm (www.rohm.com), ST (www.st.com), Xicor (www.xicor.com), and others.

| Chip | Size (bits) | Address |
|---|---|---|
| 24C01 | 1K | 0x50 (shadows at 0x51 - 0x57) |
| 24C01A | 1K | 0x50 - 0x57 (Typical device on DIMMs) |
| 24C02 | 2K | 0x50 - 0x57 |
| 24C04 | 4K | 0x50, 0x52, 0x54, 0x56 (additional data at 0x51, 0x53, 0x55, 0x57) |
| 24C08 | 8K | 0x50, 0x54 (additional data at 0x51, 0x52, 0x53, 0x55, 0x56, 0x57) |
| 24C16 | 16K | 0x50 (additional data at 0x51 - 0x57) |
| Sony | 2K | 0x57 |
| Atmel | 34C02B 2K | 0x50 - 0x57, SW write protect at 0x30-37 |
| Catalyst | 34FC02 2K | 0x50 - 0x57, SW write protect at 0x30-37 |
| Catalyst | 34RC02 2K | 0x50 - 0x57, SW write protect at 0x30-37 |
| Fairchild | 34W02 2K | 0x50 - 0x57, SW write protect at 0x30-37 |
| Microchip | 24AA52 2K | 0x50 - 0x57, SW write protect at 0x30-37 |
| ST | M34C02 2K | 0x50 - 0x57, SW write protect at 0x30-37 |

**Authors:**

- Frodo Looijaard <frodol@dds.nl>,
- Philip Edelbrock <phil@netroedge.com>,
- Jean Delvare <jdelvare@suse.de>,
- Greg Kroah-Hartman <greg@kroah.com>,
- IBM Corp.

## 1.1 Description

This is a simple EEPROM module meant to enable reading the first 256 bytes of an EEPROM (on a SDRAM DIMM for example). However, it will access serial EEPROMs on any I2C adapter. The supported devices are generically called 24Cxx, and are listed above; however the numbering for these industry-standard devices may vary by manufacturer.

This module was a programming exercise to get used to the new project organization laid out by Frodo, but it should be at least completely effective for decoding the contents of EEPROMs on DIMMs.

DIMMS will typically contain a 24C01A or 24C02, or the 34C02 variants. The other devices will not be found on a DIMM because they respond to more than one address.

DDC Monitors may contain any device. Often a 24C01, which responds to all 8 addresses, is found.

Recent Sony Vaio laptops have an EEPROM at 0x57. We couldn’t get the specification, so it is guess work and far from being complete.

The Microchip 24AA52/24LCS52, ST M34C02, and others support an additional software write protect register at 0x30 - 0x37 (0x20 less than the memory location). The chip responds to “write quick” detection at this address but does not respond to byte reads. If this register is present, the lower 128 bytes of the memory array are not write protected. Any byte data write to this address will write protect the memory array permanently, and the device will no longer respond at the 0x30-37 address. The eeprom driver does not support this register.

## 1.2 Lacking functionality

- Full support for larger devices (24C04, 24C08, 24C16). These are not typically found on a PC. These devices will appear as separate devices at multiple addresses.
- Support for really large devices (24C32, 24C64, 24C128, 24C256, 24C512). These devices require two-byte address fields and are not supported.
- Enable Writing. Again, no technical reason why not, but making it easy to change the contents of the EEPROMs (on DIMMs anyway) also makes it easy to disable the DIMMs (potentially preventing the computer from booting) until the values are restored somehow.

## 1.3 Use

After inserting the module (and any other required SMBus/i2c modules), you should have some EEPROM directories in `/sys/bus/i2c/devices/*` of names such as “0-0050”. Inside each of these is a series of files, the eeprom file contains the binary data from EEPROM.

CHAPTER

TWO

# IBM VIRTUAL MANAGEMENT CHANNEL KERNEL DRIVER (IBMVMC)

**Authors** Dave Engebretsen <engebret@us.ibm.com>, Adam Reznechek <adreznec@linux.vnet.ibm.com>, Steven Royer <seroyer@linux.vnet.ibm.com>, Bryant G. Ly <bryantly@linux.vnet.ibm.com>,

## 2.1 Introduction

Note: Knowledge of virtualization technology is required to understand this document.

A good reference document would be:

https://openpowerfoundation.org/wp-content/uploads/2016/05/LoPAPR_DRAFT_v11_24March2016_cmt1.pdf

The Virtual Management Channel (VMC) is a logical device which provides an interface between the hypervisor and a management partition. This interface is like a message passing interface. This management partition is intended to provide an alternative to systems that use a Hardware Management Console (HMC) - based system management.

The primary hardware management solution that is developed by IBM relies on an appliance server named the Hardware Management Console (HMC), packaged as an external tower or rack-mounted personal computer. In a Power Systems environment, a single HMC can manage multiple POWER processor-based systems.

### 2.1.1 Management Application

In the management partition, a management application exists which enables a system administrator to configure the system's partitioning characteristics via a command line interface (CLI) or Representational State Transfer Application (REST API's).

The management application runs on a Linux logical partition on a POWER8 or newer processor-based server that is virtualized by PowerVM. System configuration, maintenance, and control functions which traditionally require an HMC can be implemented in the management application using a combination of HMC to hypervisor interfaces and existing operating system methods. This tool provides a

subset of the functions implemented by the HMC and enables basic partition configuration. The set of HMC to hypervisor messages supported by the management application component are passed to the hypervisor over a VMC interface, which is defined below.

The VMC enables the management partition to provide basic partitioning functions:

- Logical Partitioning Configuration
- Start, and stop actions for individual partitions
- Display of partition status
- Management of virtual Ethernet
- Management of virtual Storage
- Basic system management

### 2.1.2 Virtual Management Channel (VMC)

A logical device, called the Virtual Management Channel (VMC), is defined for communicating between the management application and the hypervisor. It basically creates the pipes that enable virtualization management software. This device is presented to a designated management partition as a virtual device.

This communication device uses Command/Response Queue (CRQ) and the Remote Direct Memory Access (RDMA) interfaces. A three-way handshake is defined that must take place to establish that both the hypervisor and management partition sides of the channel are running prior to sending/receiving any of the protocol messages.

This driver also utilizes Transport Event CRQs. CRQ messages are sent when the hypervisor detects one of the peer partitions has abnormally terminated, or one side has called H_FREE_CRQ to close their CRQ. Two new classes of CRQ messages are introduced for the VMC device. VMC Administrative messages are used for each partition using the VMC to communicate capabilities to their partner. HMC Interface messages are used for the actual flow of HMC messages between the management partition and the hypervisor. As most HMC messages far exceed the size of a CRQ buffer, a virtual DMA (RMDA) of the HMC message data is done prior to each HMC Interface CRQ message. Only the management partition drives RDMA operations; hypervisors never directly cause the movement of message data.

### 2.1.3 Terminology

**RDMA** Remote Direct Memory Access is DMA transfer from the server to its client or from the server to its partner partition. DMA refers to both physical I/O to and from memory operations and to memory to memory move operations.

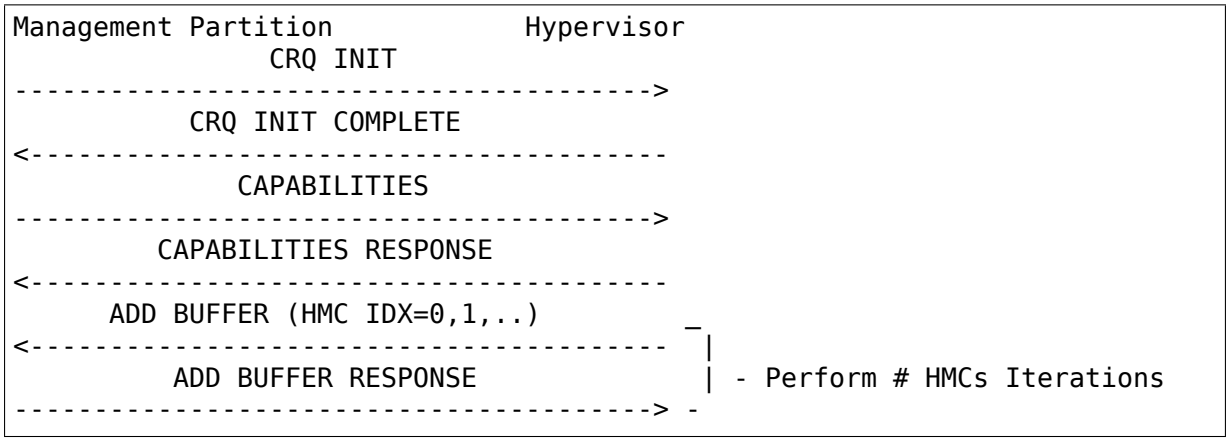
**CRQ** Command/Response Queue a facility which is used to communicate between partner partitions. Transport events which are signaled from the hypervisor to partition are also reported in this queue.

## 2.2 Example Management Partition VMC Driver Interface

This section provides an example for the management application implementation where a device driver is used to interface to the VMC device. This driver consists of a new device, for example /dev/ibmvmc, which provides interfaces to open, close, read, write, and perform ioctl's against the VMC device.

### 2.2.1 VMC Interface Initialization

The device driver is responsible for initializing the VMC when the driver is loaded. It first creates and initializes the CRQ. Next, an exchange of VMC capabilities is performed to indicate the code version and number of resources available in both the management partition and the hypervisor. Finally, the hypervisor requests that the management partition create an initial pool of VMC buffers, one buffer for each possible HMC connection, which will be used for management application session initialization. Prior to completion of this initialization sequence, the device returns EBUSY to open() calls. EIO is returned for all open() failures.




### 2.2.2 VMC Interface Open

After the basic VMC channel has been initialized, an HMC session level connection can be established. The application layer performs an open() to the VMC device and executes an ioctl() against it, indicating the HMC ID (32 bytes of data) for this session. If the VMC device is in an invalid state, EIO will be returned for the ioctl(). The device driver creates a new HMC session value (ranging from 1 to 255) and HMC index value (starting at index 0 and ranging to 254) for this HMC ID. The driver then does an RDMA of the HMC ID to the hypervisor, and then sends an Interface Open message to the hypervisor to establish the session over the VMC. After the hypervisor receives this information, it sends Add Buffer messages to the management partition to seed an initial pool of buffers for the new HMC connection. Finally, the hypervisor sends an Interface Open Response message, to indicate that it is ready for normal runtime messaging. The following illustrates this VMC flow:

```
Management Partition            Hypervisor
              RDMA HMC ID
---------------------------------------->
            Interface Open
---------------------------------------->
              Add Buffer                  _
<----------------------------------------  |
          Add Buffer Response               | - Perform N Iterations
----------------------------------------> -
        Interface Open Response
<----------------------------------------
```

### 2.2.3 VMC Interface Runtime

During normal runtime, the management application and the hypervisor exchange HMC messages via the Signal VMC message and RDMA operations. When sending data to the hypervisor, the management application performs a write() to the VMC device, and the driver RDMA's the data to the hypervisor and then sends a Signal Message. If a write() is attempted before VMC device buffers have been made available by the hypervisor, or no buffers are currently available, EBUSY is returned in response to the write(). A write() will return EIO for all other errors, such as an invalid device state. When the hypervisor sends a message to the management, the data is put into a VMC buffer and an Signal Message is sent to the VMC driver in the management partition. The driver RDMA's the buffer into the partition and passes the data up to the appropriate management application via a read() to the VMC device. The read() request blocks if there is no buffer available to read. The management application may use select() to wait for the VMC device to become ready with data to read.

```
Management Partition            Hypervisor
                MSG RDMA
---------------------------------------->
                SIGNAL MSG
---------------------------------------->
                SIGNAL MSG
<----------------------------------------
                MSG RDMA
<----------------------------------------
```

### 2.2.4 VMC Interface Close

HMC session level connections are closed by the management partition when the application layer performs a close() against the device. This action results in an Interface Close message flowing to the hypervisor, which causes the session to be terminated. The device driver must free any storage allocated for buffers for this HMC connection.

```
Management Partition            Hypervisor
             INTERFACE CLOSE
---------------------------------------->
```

(continues on next page)

(continued from previous page)

```
        INTERFACE CLOSE RESPONSE
<----------------------------------------
```

## 2.3 Additional Information

For more information on the documentation for CRQ Messages, VMC Messages, HMC interface Buffers, and signal messages please refer to the Linux on Power Architecture Platform Reference. Section F.

CHAPTER

THREE

# KERNEL DRIVER ICS932S401

Supported chips:

- IDT ICS932S401

  Prefix: 'ics932s401'

  Addresses scanned: I2C 0x69

  Datasheet: Publicly available at the IDT website

Author: Darrick J. Wong

## 3.1 Description

This driver implements support for the IDT ICS932S401 chip family.

This chip has 4 clock outputs–a base clock for the CPU (which is likely multiplied to get the real CPU clock), a system clock, a PCI clock, a USB clock, and a reference clock. The driver reports selected and actual frequency. If spread spectrum mode is enabled, the driver also reports by what percent the clock signal is being spread, which should be between 0 and -0.5%. All frequencies are reported in KHz.

The ICS932S401 monitors all inputs continuously. The driver will not read the registers more often than once every other second.

## 3.2 Special Features

The clocks could be reprogrammed to increase system speed. I will not help you do this, as you risk damaging your system!

CHAPTER

FOUR

# KERNEL DRIVER ISL29003

Supported chips:

- Intersil ISL29003

Prefix: 'isl29003'

Addresses scanned: none

Datasheet: http://www.intersil.com/data/fn/fn7464.pdf

Author: Daniel Mack <daniel@caiaq.de>

## 4.1 Description

The ISL29003 is an integrated light sensor with a 16-bit integrating type ADC, I2C user programmable lux range select for optimized counts/lux, and I2C multi-function control and monitoring capabilities. The internal ADC provides 16-bit resolution while rejecting 50Hz and 60Hz flicker caused by artificial light sources.

The driver allows to set the lux range, the bit resolution, the operational mode (see below) and the power state of device and can read the current lux value, of course.

## 4.2 Detection

The ISL29003 does not have an ID register which could be used to identify it, so the detection routine will just try to read from the configured I2C address and consider the device to be present as soon as it ACKs the transfer.

## 4.3 Sysfs entries

**range:**

| | |
|---|---|
| 0: | 0 lux to 1000 lux (default) |
| 1: | 0 lux to 4000 lux |
| 2: | 0 lux to 16,000 lux |
| 3: | 0 lux to 64,000 lux |

**resolution:**

| | |
|---|---|
| 0: | 2^16 cycles (default) |
| 1: | 2^12 cycles |
| 2: | 2^8 cycles |
| 3: | 2^4 cycles |

**mode:**

| | |
|---|---|
| 0: | diode1’s current (unsigned 16bit) (default) |
| 1: | diode1’s current (unsigned 16bit) |
| 2: | difference between diodes (l1 - l2, signed 15bit) |

**power_state:**

| | |
|---|---|
| 0: | device is disabled (default) |
| 1: | device is enabled |

**lux (read only):** returns the value from the last sensor reading

CHAPTER

FIVE

# KERNEL DRIVER LIS3LV02D

Supported chips:

- STMicroelectronics LIS3LV02DL, LIS3LV02DQ (12 bits precision)
- STMicroelectronics LIS302DL, LIS3L02DQ, LIS331DL (8 bits) and LIS331DLH (16 bits)

**Authors:**

- Yan Burman <burman.yan@gmail.com>
- Eric Piel <eric.piel@tremplin-utc.net>

## 5.1 Description

This driver provides support for the accelerometer found in various HP laptops sporting the feature officially called "HP Mobile Data Protection System 3D" or "HP 3D DriveGuard". It detects automatically laptops with this sensor. Known models (full list can be found in drivers/platform/x86/hp_accel.c) will have their axis automatically oriented on standard way (eg: you can directly play neverball). The accelerometer data is readable via /sys/devices/platform/lis3lv02d. Reported values are scaled to mg values (1/1000th of earth gravity).

Sysfs attributes under /sys/devices/platform/lis3lv02d/:

**position**

- 3D position that the accelerometer reports. Format: "(x,y,z)"

**rate**

- read reports the sampling rate of the accelerometer device in HZ. write changes sampling rate of the accelerometer device. Only values which are supported by HW are accepted.

**selftest**

- performs selftest for the chip as specified by chip manufacturer.

This driver also provides an absolute input class device, allowing the laptop to act as a pinball machine-esque joystick. Joystick device can be calibrated. Joystick device can be in two different modes. By default output values are scaled between -32768 .. 32767. In joystick raw mode, joystick and sysfs position entry have the

same scale. There can be small difference due to input system fuzziness feature. Events are also available as input event device.

Selftest is meant only for hardware diagnostic purposes. It is not meant to be used during normal operations. Position data is not corrupted during selftest but interrupt behaviour is not guaranteed to work reliably. In test mode, the sensing element is internally moved little bit. Selftest measures difference between normal mode and test mode. Chip specifications tell the acceptance limit for each type of the chip. Limits are provided via platform data to allow adjustment of the limits without a change to the actual driver. Seltest returns either "OK x y z" or "FAIL x y z" where x, y and z are measured difference between modes. Axes are not remapped in selftest mode. Measurement values are provided to help HW diagnostic applications to make final decision.

On HP laptops, if the led infrastructure is activated, support for a led indicating disk protection will be provided as /sys/class/leds/hp::hddprotect.

Another feature of the driver is misc device called "freefall" that acts similar to /dev/rtc and reacts on free-fall interrupts received from the device. It supports blocking operations, poll/select and fasync operation modes. You must read 1 bytes from the device. The result is number of free-fall interrupts since the last successful read (or 255 if number of interrupts would not fit). See the freefall.c file for an example on using the device.

## 5.2 Axes orientation

For better compatibility between the various laptops. The values reported by the accelerometer are converted into a "standard" organisation of the axes (aka "can play neverball out of the box"):

- When the laptop is horizontal the position reported is about 0 for X and Y and a positive value for Z
- If the left side is elevated, X increases (becomes positive)
- If the front side (where the touchpad is) is elevated, Y decreases (becomes negative)
- If the laptop is put upside-down, Z becomes negative

If your laptop model is not recognized (cf "dmesg"), you can send an email to the maintainer to add it to the database. When reporting a new laptop, please include the output of "dmidecode"plus the value of /sys/devices/platform/lis3lv02d/position in these four cases.

## 5.3 Q&A

Q: How do I safely simulate freefall? I have an HP “portable workstation” which has about 3.5kg and a plastic case, so letting it fall to the ground is out of question …

A: The sensor is pretty sensitive, so your hands can do it. Lift it into free space, follow the fall with your hands for like 10 centimeters. That should be enough to trigger the detection.

CHAPTER

SIX

# KERNEL DRIVER MAX6875

Supported chips:

- Maxim MAX6874, MAX6875

  Prefix: 'max6875'

  Addresses scanned: None (see below)

  Datasheet: http://pdfserv.maxim-ic.com/en/ds/MAX6874-MAX6875.pdf

Author: Ben Gardner <bgardner@wabtec.com>

## 6.1 Description

The Maxim MAX6875 is an EEPROM-programmable power-supply sequencer/supervisor. It provides timed outputs that can be used as a watchdog, if properly wired. It also provides 512 bytes of user EEPROM.

At reset, the MAX6875 reads the configuration EEPROM into its configuration registers. The chip then begins to operate according to the values in the registers.

The Maxim MAX6874 is a similar, mostly compatible device, with more inputs and outputs:

| • | vin | gpi | vout |
|---|---|---|---|
| MAX6874 | 6 | 4 | 8 |
| MAX6875 | 4 | 3 | 5 |

See the datasheet for more information.

## 6.2 Sysfs entries

eeprom - 512 bytes of user-defined EEPROM space.

## 6.3 General Remarks

Valid addresses for the MAX6875 are 0x50 and 0x52.

Valid addresses for the MAX6874 are 0x50, 0x52, 0x54 and 0x56.

The driver does not probe any address, so you explicitly instantiate the devices.

Example:

```
$ modprobe max6875
$ echo max6875 0x50 > /sys/bus/i2c/devices/i2c-0/new_device
```

The MAX6874/MAX6875 ignores address bit 0, so this driver attaches to multiple addresses. For example, for address 0x50, it also reserves 0x51. The even-address instance is called 'max6875', the odd one is 'dummy'.

## 6.4 Programming the chip using i2c-dev

Use the i2c-dev interface to access and program the chips.

Reads and writes are performed differently depending on the address range.

The configuration registers are at addresses 0x00 - 0x45.

Use i2c_smbus_write_byte_data() to write a register and i2c_smbus_read_byte_data() to read a register.

The command is the register number.

Examples:

To write a 1 to register 0x45:

```
i2c_smbus_write_byte_data(fd, 0x45, 1);
```

To read register 0x45:

```
value = i2c_smbus_read_byte_data(fd, 0x45);
```

The configuration EEPROM is at addresses 0x8000 - 0x8045.

The user EEPROM is at addresses 0x8100 - 0x82ff.

Use i2c_smbus_write_word_data() to write a byte to EEPROM.

The command is the upper byte of the address: 0x80, 0x81, or 0x82. The data word is the lower part of the address or'd with data << 8:

```
cmd = address >> 8;
val = (address & 0xff) | (data << 8);
```

Example:

To write 0x5a to address 0x8003:

```
i2c_smbus_write_word_data(fd, 0x80, 0x5a03);
```

Reading data from the EEPROM is a little more complicated.

Use i2c_smbus_write_byte_data() to set the read address and then i2c_smbus_read_byte() or i2c_smbus_read_i2c_block_data() to read the data.

Example:

To read data starting at offset 0x8100, first set the address:

```
i2c_smbus_write_byte_data(fd, 0x81, 0x00);
```

And then read the data:

```
value = i2c_smbus_read_byte(fd);
```

or:

```
count = i2c_smbus_read_i2c_block_data(fd, 0x84, 16, buffer);
```

The block read should read 16 bytes.

0x84 is the block read command.

See the datasheet for more details.

CHAPTER

SEVEN

# INTEL MANY INTEGRATED CORE (MIC) ARCHITECTURE

## 7.1 Intel Many Integrated Core (MIC) architecture overview

An Intel MIC X100 device is a PCIe form factor add-in coprocessor card based on the Intel Many Integrated Core (MIC) architecture that runs a Linux OS. It is a PCIe endpoint in a platform and therefore implements the three required standard address spaces i.e. configuration, memory and I/O. The host OS loads a device driver as is typical for PCIe devices. The card itself runs a bootstrap after reset that transfers control to the card OS downloaded from the host driver. The host driver supports OSPM suspend and resume operations. It shuts down the card during suspend and reboots the card OS during resume. The card OS as shipped by Intel is a Linux kernel with modifications for the X100 devices.

Since it is a PCIe card, it does not have the ability to host hardware devices for networking, storage and console. We provide these devices on X100 coprocessors thus enabling a self-bootable equivalent environment for applications. A key benefit of our solution is that it leverages the standard virtio framework for network, disk and console devices, though in our case the virtio framework is used across a PCIe bus. A Virtio Over PCIe (VOP) driver allows creating user space backends or devices on the host which are used to probe virtio drivers for these devices on the MIC card. The existing VRINGH infrastructure in the kernel is used to access virtio rings from the host. The card VOP driver allows card virtio drivers to communicate with their user space backends on the host via a device page. Ring 3 apps on the host can add, remove and configure virtio devices. A thin MIC specific virtio_config_ops is implemented which is borrowed heavily from previous similar implementations in lguest and s390.

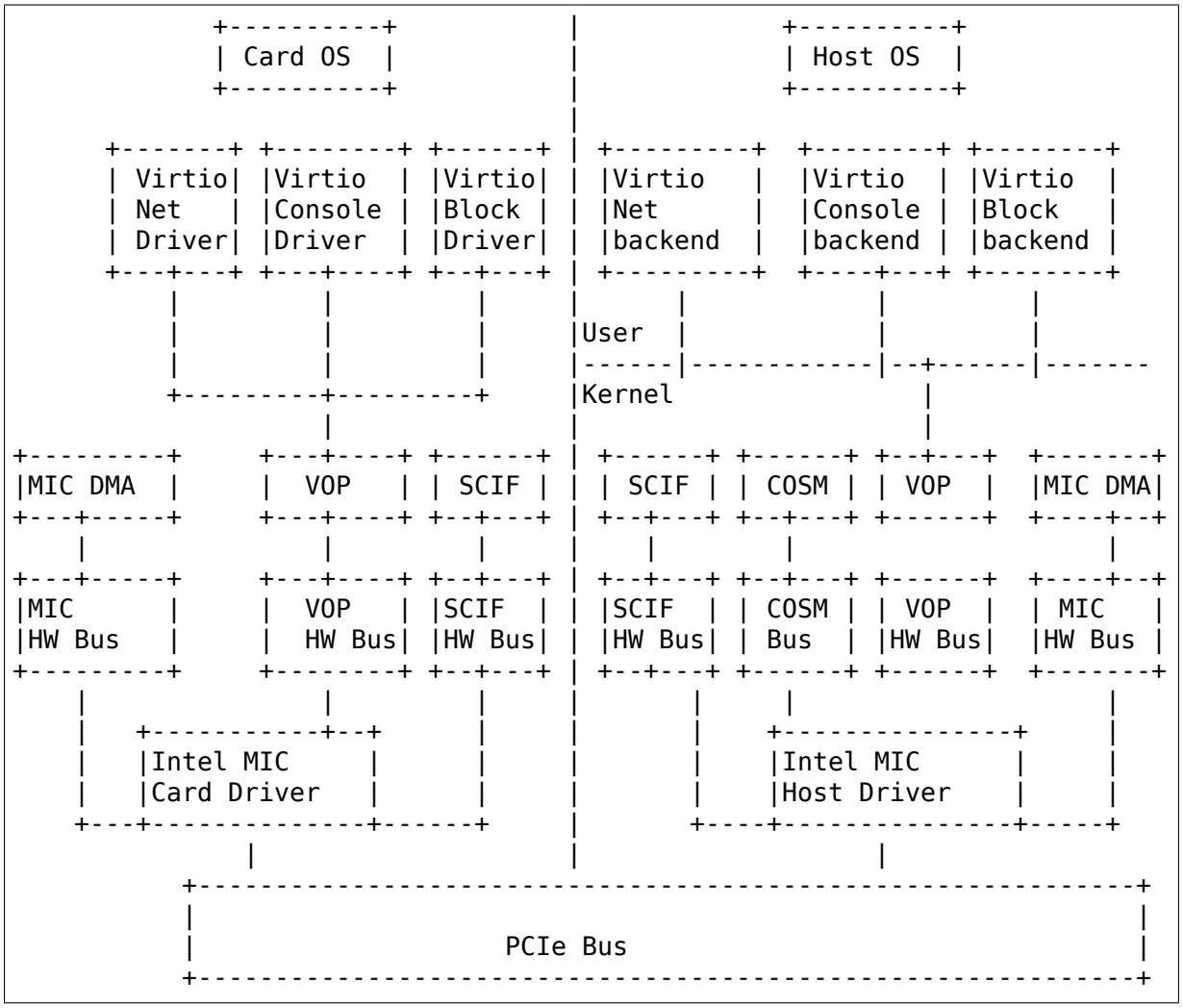
MIC PCIe card has a dma controller with 8 channels. These channels are shared between the host s/w and the card s/w. 0 to 3 are used by host and 4 to 7 by card. As the dma device doesn’ t show up as PCIe device, a virtual bus called mic bus is created and virtual dma devices are created on it by the host/card drivers. On host the channels are private and used only by the host driver to transfer data for the virtio devices.

The Symmetric Communication Interface (SCIF (pronounced as skiff)) is a low level communications API across PCIe currently implemented for MIC. More details are available at scif_overview.txt.

The Coprocessor State Management (COSM) driver on the host allows for boot, shutdown and reset of Intel MIC devices. It communicates with a COSM “client”

driver on the MIC cards over SCIF to perform these functions.

Here is a block diagram of the various components described above. The virtio backends are situated on the host rather than the card given better single threaded performance for the host compared to MIC, the ability of the host to initiate DMA's to/from the card using the MIC DMA engine and the fact that the virtio block storage backend can only be on the host:




## 7.2 Symmetric Communication Interface (SCIF)

The Symmetric Communication Interface (SCIF (pronounced as skiff)) is a low level communications API across PCIe currently implemented for MIC. Currently SCIF provides inter-node communication within a single host platform, where a node is a MIC Coprocessor or Xeon based host. SCIF abstracts the details of communicating over the PCIe bus while providing an API that is symmetric across all the nodes in the PCIe network. An important design objective for SCIF is to deliver the maximum possible performance given the communication abilities of the hardware. SCIF has been used to implement an offload compiler runtime and OFED support for MPI implementations for MIC coprocessors.

### 7.2.1 SCIF API Components

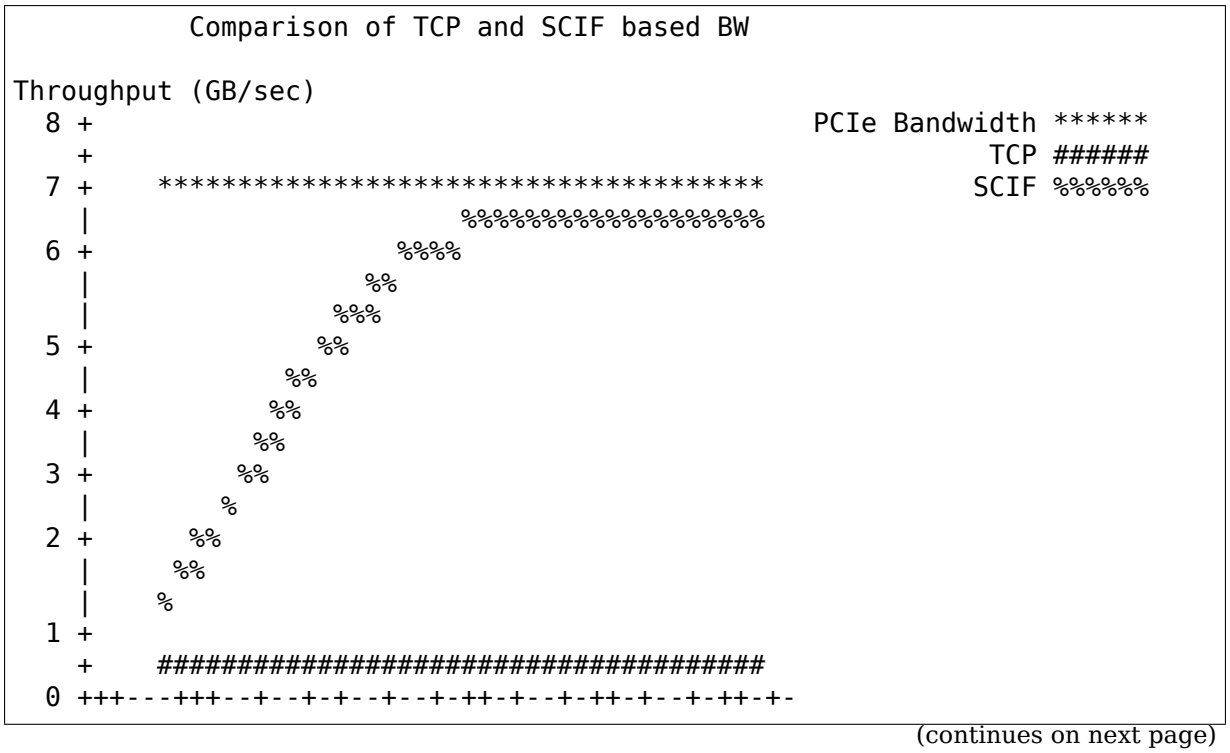
The SCIF API has the following parts:

1. Connection establishment using a client server model
2. Byte stream messaging intended for short messages
3. Node enumeration to determine online nodes
4. Poll semantics for detection of incoming connections and messages
5. Memory registration to pin down pages
6. Remote memory mapping for low latency CPU accesses via mmap
7. Remote DMA (RDMA) for high bandwidth DMA transfers
8. Fence APIs for RDMA synchronization

SCIF exposes the notion of a connection which can be used by peer processes on nodes in a SCIF PCIe “network”to share memory “windows”and to communicate. A process in a SCIF node initiates a SCIF connection to a peer process on a different node via a SCIF “endpoint”. SCIF endpoints support messaging APIs which are similar to connection oriented socket APIs. Connected SCIF endpoints can also register local memory which is followed by data transfer using either DMA, CPU copies or remote memory mapping via mmap. SCIF supports both user and kernel mode clients which are functionally equivalent.

### 7.2.2 SCIF Performance for MIC

DMA bandwidth comparison between the TCP (over ethernet over PCIe) stack versus SCIF shows the performance advantages of SCIF for HPC applications and runtimes:




(continues on next page)

(continued from previous page)

```
    1       10     100      1000   10000   100000
                Transfer Size (KBytes)
```

SCIF allows memory sharing via mmap(..) between processes on different PCIe nodes and thus provides bare-metal PCIe latency. The round trip SCIF mmap latency from the host to an x100 MIC for an 8 byte message is 0.44 usecs.

SCIF has a user space library which is a thin IOCTL wrapper providing a user space API similar to the kernel API in scif.h. The SCIF user space library is distributed @ https://software.intel.com/en-us/mic-developer

Here is some pseudo code for an example of how two applications on two PCIe nodes would typically use the SCIF API:

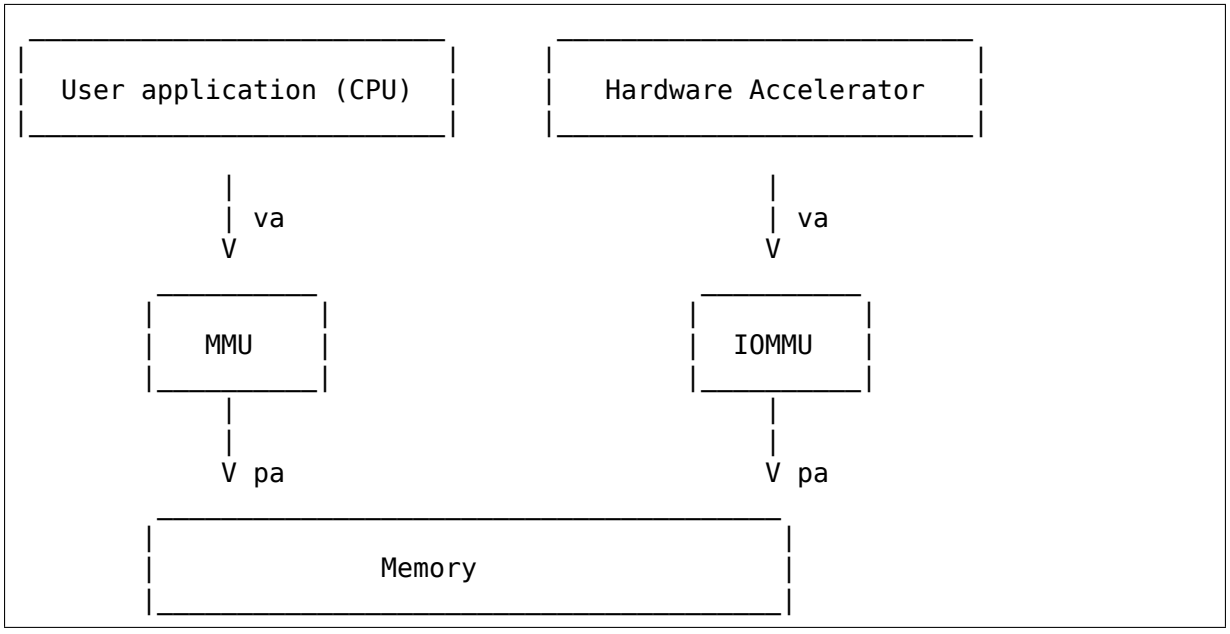
```
Process A (on node A)                   Process B (on node B)

/* get online node information */
scif_get_node_ids(..)                   scif_get_node_ids(..)
scif_open(..)                           scif_open(..)
scif_bind(..)                           scif_bind(..)
scif_listen(..)
scif_accept(..)                                 scif_connect(..)
/* SCIF connection established */

/* Send and receive short messages */
scif_send(..)/scif_recv(..)             scif_send(..)/scif_recv(..)

/* Register memory */
scif_register(..)                       scif_register(..)

/* RDMA */
scif_readfrom(..)/scif_writeto(..)      scif_readfrom(..)/scif_writeto(..)

/* Fence DMAs */
scif_fence_signal(..)                   scif_fence_signal(..)

mmap(..)                                mmap(..)

/* Access remote registered memory */

/* Close the endpoints */
scif_close(..)                                  scif_close(..)
```

CHAPTER

EIGHT

# INTRODUCTION OF UACCE

Uacce (Unified/User-space-access-intended Accelerator Framework) targets to provide Shared Virtual Addressing (SVA) between accelerators and processes. So accelerator can access any data structure of the main cpu. This differs from the data sharing between cpu and io device, which share only data content rather than address. Because of the unified address, hardware and user space of process can share the same virtual address in the communication. Uacce takes the hardware accelerator as a heterogeneous processor, while IOMMU share the same CPU page tables and as a result the same translation from va to pa.



CHAPTER

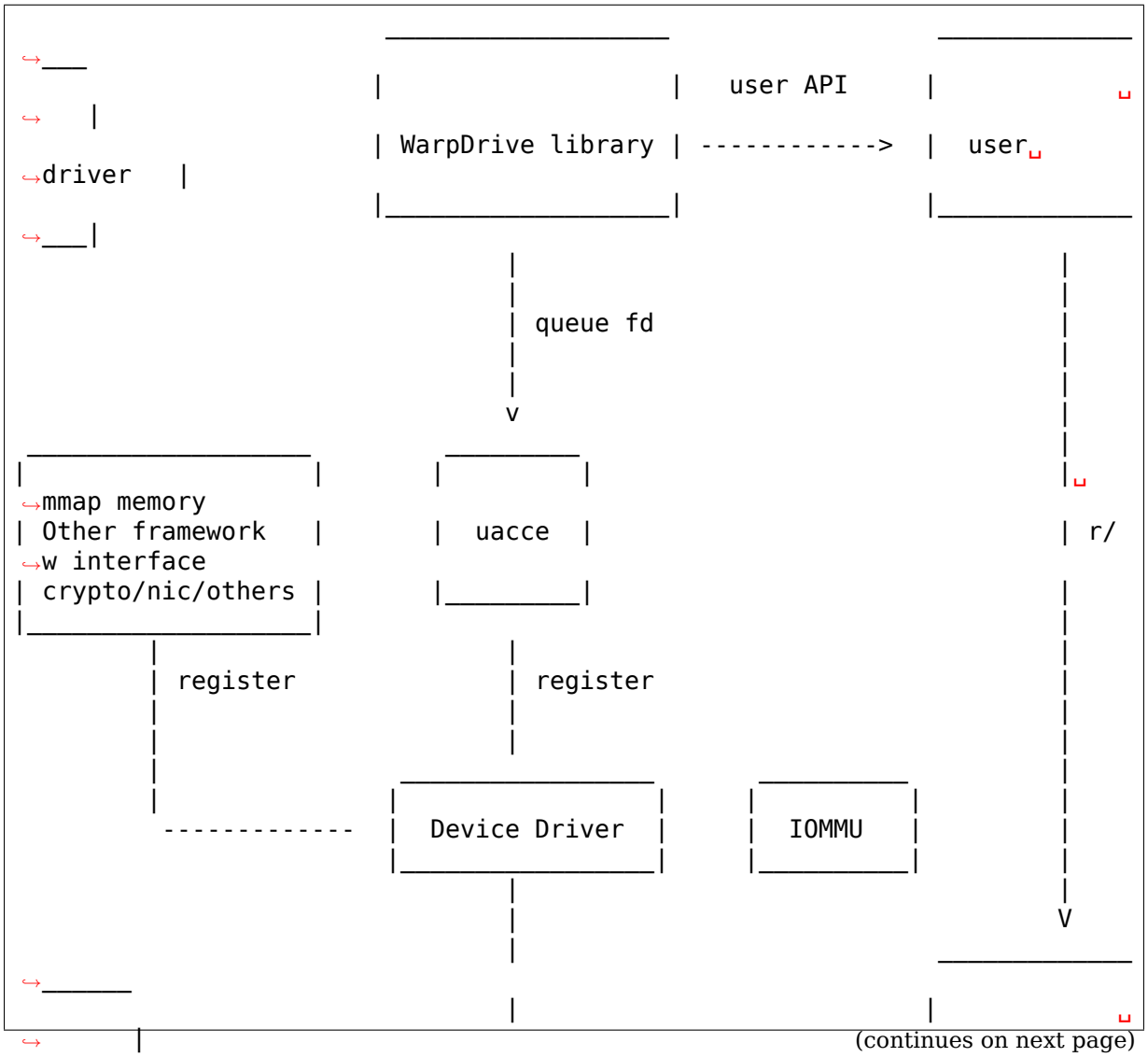
NINE

# ARCHITECTURE

Uacce is the kernel module, taking charge of iommu and address sharing. The user drivers and libraries are called WarpDrive.

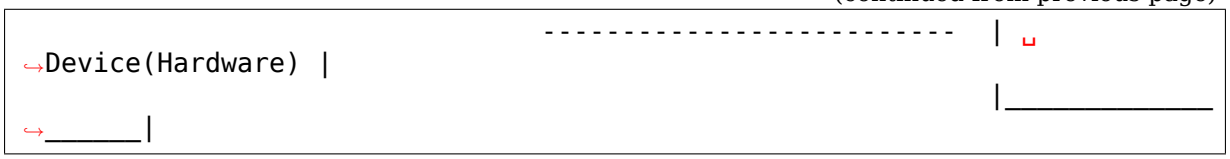
The uacce device, built around the IOMMU SVA API, can access multiple address spaces, including the one without PASID.

A virtual concept, queue, is used for the communication. It provides a FIFO-like interface. And it maintains a unified address space between the application and all involved hardware.




(continues on next page)

(continued from previous page)



CHAPTER

TEN

# HOW DOES IT WORK

Uacce uses mmap and IOMMU to play the trick.

Uacce creates a chrdev for every device registered to it. New queue is created when user application open the chrdev. The file descriptor is used as the user handle of the queue. The accelerator device present itself as an Uacce object, which exports as a chrdev to the user space. The user application communicates with the hardware by ioctl (as control path) or share memory (as data path).

The control path to the hardware is via file operation, while data path is via mmap space of the queue fd.

The queue file address space:

```
 /**
 * enum uacce_qfrt: qfrt type
 * @UACCE_QFRT_MMIO: device mmio region
 * @UACCE_QFRT_DUS: device user share region
 */
enum uacce_qfrt {
        UACCE_QFRT_MMIO = 0,
        UACCE_QFRT_DUS = 1,
};
```

All regions are optional and differ from device type to type. Each region can be mmapped only once, otherwise -EEXIST returns.

The device mmio region is mapped to the hardware mmio space. It is generally used for doorbell or other notification to the hardware. It is not fast enough as data channel.

The device user share region is used for share data buffer between user process and device.

CHAPTER

ELEVEN

# THE UACCE REGISTER API

The register API is defined in uacce.h.

```
struct uacce_interface {
  char name[UACCE_MAX_NAME_SIZE];
  unsigned int flags;
  const struct uacce_ops *ops;
};
```

According to the IOMMU capability, uacce_interface flags can be:

```
/**
 * UACCE Device flags:
 * UACCE_DEV_SVA: Shared Virtual Addresses
 *              Support PASID
 *              Support device page faults (PCI PRI or SMMU Stall)
 */
#define UACCE_DEV_SVA               BIT(0)

struct uacce_device *uacce_alloc(struct device *parent,
                                 struct uacce_interface *interface);
int uacce_register(struct uacce_device *uacce);
void uacce_remove(struct uacce_device *uacce);
```

uacce_register results can be:

a. If uacce module is not compiled, ERR_PTR(-ENODEV)

b. Succeed with the desired flags

c. Succeed with the negotiated flags, for example

   uacce_interface.flags = UACCE_DEV_SVA but uacce->flags = ~UACCE_DEV_SVA

   So user driver need check return value as well as the negotiated uacce->flags.

CHAPTER

TWELVE

# THE USER DRIVER

The queue file mmap space will need a user driver to wrap the communication protocol. Uacce provides some attributes in sysfs for the user driver to match the right accelerator accordingly. More details in Documentation/ABI/testing/sysfs-driver-uacce.

CHAPTER

THIRTEEN

# XILINX SD-FEC DRIVER

## 13.1 Overview

This driver supports SD-FEC Integrated Block for Zynq Ultrascale+™ RFSoCs.

For a full description of SD-FEC core features, see the SD-FEC Product Guide (PG256)

This driver supports the following features:

- Retrieval of the Integrated Block configuration and status information
- Configuration of LDPC codes
- Configuration of Turbo decoding
- Monitoring errors

Missing features, known issues, and limitations of the SD-FEC driver are as follows:

- Only allows a single open file handler to any instance of the driver at any time
- Reset of the SD-FEC Integrated Block is not controlled by this driver
- Does not support shared LDPC code table wraparound

The device tree entry is described in: linux-xlnx/Documentation/devicetree/bindings/misc/xlnx,sfec.txt

### 13.1.1 Modes of Operation

The driver works with the SD-FEC core in two modes of operation:

- Run-time configuration
- Programmable Logic (PL) initialization

### Run-time Configuration

For Run-time configuration the role of driver is to allow the software application to do the following:

- Load the configuration parameters for either Turbo decode or LDPC encode or decode
- Activate the SD-FEC core
- Monitor the SD-FEC core for errors
- Retrieve the status and configuration of the SD-FEC core

### Programmable Logic (PL) Initialization

For PL initialization, supporting logic loads configuration parameters for either the Turbo decode or LDPC encode or decode. The role of the driver is to allow the software application to do the following:

- Activate the SD-FEC core
- Monitor the SD-FEC core for errors
- Retrieve the status and configuration of the SD-FEC core

## 13.2 Driver Structure

The driver provides a platform device where the `probe` and `remove` operations are provided.

- probe: Updates configuration register with device-tree entries plus determines the current activate state of the core, for example, is the core bypassed or has the core been started.

The driver defines the following driver file operations to provide user application interfaces:

- open: Implements restriction that only a single file descriptor can be open per SD-FEC instance at any time
- release: Allows another file descriptor to be open, that is after current file descriptor is closed
- poll: Provides a method to monitor for SD-FEC Error events
- unlocked_ioctl: Provides the the following ioctl commands that allows the application configure the SD-FEC core:
  - `XSDFEC_START_DEV`
  - `XSDFEC_STOP_DEV`
  - `XSDFEC_GET_STATUS`
  - `XSDFEC_SET_IRQ`
  - `XSDFEC_SET_TURBO`

- XSDFEC_ADD_LDPC_CODE_PARAMS
- XSDFEC_GET_CONFIG
- XSDFEC_SET_ORDER
- XSDFEC_SET_BYPASS
- XSDFEC_IS_ACTIVE
- XSDFEC_CLEAR_STATS
- XSDFEC_SET_DEFAULT_CONFIG

## 13.3 Driver Usage

### 13.3.1 Overview

After opening the driver, the user should find out what operations need to be performed to configure and activate the SD-FEC core and determine the configuration of the driver. The following outlines the flow the user should perform:

- Determine Configuration
- Set the order, if not already configured as desired
- Set Turbo decode, LPDC encode or decode parameters, depending on how the SD-FEC core is configured plus if the SD-FEC has not been configured for PL initialization
- Enable interrupts, if not already enabled
- Bypass the SD-FEC core, if required
- Start the SD-FEC core if not already started
- Get the SD-FEC core status
- Monitor for interrupts
- Stop the SD-FEC core

Note: When monitoring for interrupts if a critical error is detected where a reset is required, the driver will be required to load the default configuration.

### 13.3.2 Determine Configuration

Determine the configuration of the SD-FEC core by using the ioctl `XSDFEC_GET_CONFIG`.

### 13.3.3 Set the Order

Setting the order determines how the order of Blocks can change from input to output.

Setting the order is done by using the ioctl `XSDFEC_SET_ORDER`

Setting the order can only be done if the following restrictions are met:

- The `state` member of struct `xsdfec_status` filled by the ioctl `XSDFEC_GET_STATUS` indicates the SD-FEC core has not STARTED

### 13.3.4 Add LDPC Codes

The following steps indicate how to add LDPC codes to the SD-FEC core:

- Use the auto-generated parameters to fill the `struct xsdfec_ldpc_params` for the desired LDPC code.
- Set the SC, QA, and LA table offsets for the LPDC parameters and the parameters in the structure `struct xsdfec_ldpc_params`
- Set the desired Code Id value in the structure `struct xsdfec_ldpc_params`
- Add the LPDC Code Parameters using the ioctl `XSDFEC_ADD_LDPC_CODE_PARAMS`
- For the applied LPDC Code Parameter use the function `xsdfec_calculate_shared_ldpc_table_entry_size()` to calculate the size of shared LPDC code tables. This allows the user to determine the shared table usage so when selecting the table offsets for the next LDPC code parameters unused table areas can be selected.
- Repeat for each LDPC code parameter.

Adding LDPC codes can only be done if the following restrictions are met:

- The `code` member of `struct xsdfec_config` filled by the ioctl `XSDFEC_GET_CONFIG` indicates the SD-FEC core is configured as LDPC
- The `code_wr_protect` of `struct xsdfec_config` filled by the ioctl `XSDFEC_GET_CONFIG` indicates that write protection is not enabled
- The `state` member of struct `xsdfec_status` filled by the ioctl `XSDFEC_GET_STATUS` indicates the SD-FEC core has not started

### 13.3.5 Set Turbo Decode

Configuring the Turbo decode parameters is done by using the ioctl `XSDFEC_SET_TURBO` using auto-generated parameters to fill the `struct xsdfec_turbo` for the desired Turbo code.

Adding Turbo decode can only be done if the following restrictions are met:

- The `code` member of `struct xsdfec_config` filled by the ioctl `XSDFEC_GET_CONFIG` indicates the SD-FEC core is configured as TURBO

- The `state` member of struct `xsdfec_status` filled by the ioctl `XSDFEC_GET_STATUS` indicates the SD-FEC core has not STARTED

### 13.3.6 Enable Interrupts

Enabling or disabling interrupts is done by using the ioctl `XSDFEC_SET_IRQ`. The members of the parameter passed, `struct xsdfec_irq`, to the ioctl are used to set and clear different categories of interrupts. The category of interrupt is controlled as following:

- `enable_isr` controls the `tlast` interrupts
- `enable_ecc_isr` controls the ECC interrupts

If the `code` member of `struct xsdfec_config` filled by the ioctl `XSDFEC_GET_CONFIG` indicates the SD-FEC core is configured as TURBO then the enabling ECC errors is not required.

### 13.3.7 Bypass the SD-FEC

Bypassing the SD-FEC is done by using the ioctl `XSDFEC_SET_BYPASS`

Bypassing the SD-FEC can only be done if the following restrictions are met:

- The `state` member of `struct xsdfec_status` filled by the ioctl `XSDFEC_GET_STATUS` indicates the SD-FEC core has not STARTED

### 13.3.8 Start the SD-FEC core

Start the SD-FEC core by using the ioctl `XSDFEC_START_DEV`

### 13.3.9 Get SD-FEC Status

Get the SD-FEC status of the device by using the ioctl `XSDFEC_GET_STATUS`, which will fill the `struct xsdfec_status`

### 13.3.10 Monitor for Interrupts

- Use the poll system call to monitor for an interrupt. The poll system call waits for an interrupt to wake it up or times out if no interrupt occurs.
- **On return Poll `revents` will indicate whether stats and/or state have been updated**
  - `POLLPRI` indicates a critical error and the user should use `XSDFEC_GET_STATUS` and `XSDFEC_GET_STATS` to confirm
  - `POLLRDNORM` indicates a non-critical error has occurred and the user should use `XSDFEC_GET_STATS` to confirm
- **Get stats by using the ioctl `XSDFEC_GET_STATS`**

  - For critical error the `isr_err_count` or `uecc_count` member of `struct xsdfec_stats` is non-zero
  - For non-critical errors the `cecc_count` member of `struct xsdfec_stats` is non-zero

- **Get state by using the ioctl `XSDFEC_GET_STATUS`**
  - For a critical error the `state` of `xsdfec_status` will indicate a Reset Is Required
- Clear stats by using the ioctl `XSDFEC_CLEAR_STATS`

If a critical error is detected where a reset is required. The application is required to call the ioctl `XSDFEC_SET_DEFAULT_CONFIG`, after the reset and it is not required to call the ioctl `XSDFEC_STOP_DEV`

Note: Using poll system call prevents busy looping using `XSDFEC_GET_STATS` and `XSDFEC_GET_STATUS`

### 13.3.11 Stop the SD-FEC Core

Stop the device by using the ioctl `XSDFEC_STOP_DEV`

### 13.3.12 Set the Default Configuration

Load default configuration by using the ioctl `XSDFEC_SET_DEFAULT_CONFIG` to restore the driver.

### 13.3.13 Limitations

Users should not duplicate SD-FEC device file handlers, for example fork() or dup() a process that has a created an SD-FEC file handler.

## 13.4 Driver IOCTLs

**`XSDFEC_START_DEV`**

**Description**

ioctl to start SD-FEC core

This fails if the XSDFEC_SET_ORDER ioctl has not been previously called

**`XSDFEC_STOP_DEV`**

**Description**

ioctl to stop the SD-FEC core

**`XSDFEC_GET_STATUS`**

**Description**

ioctl that returns status of SD-FEC core

**XSDFEC_SET_IRQ**

**Parameters**

**struct xsdfec_irq** * Pointer to the `struct xsdfec_irq` that contains the interrupt settings for the SD-FEC core

**Description**

ioctl to enable or disable irq

**XSDFEC_SET_TURBO**

**Parameters**

**struct xsdfec_turbo** * Pointer to the `struct xsdfec_turbo` that contains the Turbo decode settings for the SD-FEC core

**Description**

ioctl that sets the SD-FEC Turbo parameter values

This can only be used when the driver is in the XSDFEC_STOPPED state

**XSDFEC_ADD_LDPC_CODE_PARAMS**

**Parameters**

**struct xsdfec_ldpc_params** * Pointer to the `struct xsdfec_ldpc_params` that contains the LDPC code parameters to be added to the SD-FEC Block

**Description** ioctl to add an LDPC code to the SD-FEC LDPC codes

This can only be used when:

- Driver is in the XSDFEC_STOPPED state
- SD-FEC core is configured as LPDC
- SD-FEC Code Write Protection is disabled

**XSDFEC_GET_CONFIG**

**Parameters**

**struct xsdfec_config** * Pointer to the `struct xsdfec_config` that contains the current configuration settings of the SD-FEC Block

**Description**

ioctl that returns SD-FEC core configuration

**XSDFEC_SET_ORDER**

**Parameters**

**struct unsigned long** * Pointer to the unsigned long that contains a value from the **enum** xsdfec_order

**Description**

ioctl that sets order, if order of blocks can change from input to output

This can only be used when the driver is in the XSDFEC_STOPPED state

**XSDFEC_SET_BYPASS**

**Parameters**

**struct bool *** Pointer to bool that sets the bypass value, where false results in normal operation and false results in the SD-FEC performing the configured operations (same number of cycles) but output data matches the input data

**Description**

ioctl that sets bypass.

This can only be used when the driver is in the XSDFEC_STOPPED state

**XSDFEC_IS_ACTIVE**

**Parameters**

**struct bool *** Pointer to bool that returns true if the SD-FEC is processing data

**Description**

ioctl that determines if SD-FEC is processing data

**XSDFEC_CLEAR_STATS**

**Description**

ioctl that clears error stats collected during interrupts

**XSDFEC_GET_STATS**

**Parameters**

**struct xsdfec_stats *** Pointer to the `struct xsdfec_stats` that will contain the updated stats values

**Description**

ioctl that returns SD-FEC core stats

This can only be used when the driver is in the XSDFEC_STOPPED state

**XSDFEC_SET_DEFAULT_CONFIG**

**Description**

ioctl that returns SD-FEC core to default config, use after a reset

This can only be used when the driver is in the XSDFEC_STOPPED state

## 13.5 Driver Type Definitions

enum **xsdfec_code**
Code Type.

**Constants**

**XSDFEC_TURBO_CODE** Driver is configured for Turbo mode.

**XSDFEC_LDPC_CODE** Driver is configured for LDPC mode.

**Description**

This enum is used to indicate the mode of the driver. The mode is determined by checking which codes are set in the driver. Note that the mode cannot be changed by the driver.

enum **xsdfec_order**
Order

**Constants**

**XSDFEC_MAINTAIN_ORDER** Maintain order execution of blocks.

**XSDFEC_OUT_OF_ORDER** Out-of-order execution of blocks.

**Description**

This enum is used to indicate whether the order of blocks can change from input to output.

enum **xsdfec_turbo_alg**
Turbo Algorithm Type.

**Constants**

**XSDFEC_MAX_SCALE** Max Log-Map algorithm with extrinsic scaling. When scaling is set to this is equivalent to the Max Log-Map algorithm.

**XSDFEC_MAX_STAR** Log-Map algorithm.

**XSDFEC_TURBO_ALG_MAX** Used to indicate out of bound Turbo algorithms.

**Description**

This enum specifies which Turbo Decode algorithm is in use.

enum **xsdfec_state**
State.

**Constants**

**XSDFEC_INIT** Driver is initialized.

**XSDFEC_STARTED** Driver is started.

**XSDFEC_STOPPED** Driver is stopped.

**XSDFEC_NEEDS_RESET** Driver needs to be reset.

**XSDFEC_PL_RECONFIGURE** Programmable Logic needs to be recofigured.

**Description**

This enum is used to indicate the state of the driver.

enum **xsdfec_axis_width**
AXIS_WIDTH.DIN Setting for 128-bit width.

**Constants**

**XSDFEC_1x128b** DIN data input stream consists of a 128-bit lane

**XSDFEC_2x128b** DIN data input stream consists of two 128-bit lanes

**XSDFEC_4x128b** DIN data input stream consists of four 128-bit lanes

### Description

This enum is used to indicate the AXIS_WIDTH.DIN setting for 128-bit width. The number of lanes of the DIN data input stream depends upon the AXIS_WIDTH.DIN parameter.

enum **xsdfec_axis_word_include**
Words Configuration.

### Constants

**XSDFEC_FIXED_VALUE** Fixed, the DIN_WORDS AXI4-Stream interface is removed from the IP instance and is driven with the specified number of words.

**XSDFEC_IN_BLOCK** In Block, configures the IP instance to expect a single DIN_WORDS value per input code block. The DIN_WORDS interface is present.

**XSDFEC_PER_AXI_TRANSACTION** Per Transaction, configures the IP instance to expect one DIN_WORDS value per input transaction on the DIN interface. The DIN_WORDS interface is present.

**XSDFEC_AXIS_WORDS_INCLUDE_MAX** Used to indicate out of bound Words Configurations.

### Description

This enum is used to specify the DIN_WORDS configuration.

struct **xsdfec_turbo**
User data for Turbo codes.

### Definition

```
struct xsdfec_turbo {
  __u32 alg;
  __u8 scale;
};
```

### Members

**alg** Specifies which Turbo decode algorithm to use

**scale** Specifies the extrinsic scaling to apply when the Max Scale algorithm has been selected

### Description

Turbo code structure to communicate parameters to XSDFEC driver.

struct **xsdfec_ldpc_params**
User data for LDPC codes.

### Definition

```
struct xsdfec_ldpc_params {
  __u32 n;
  __u32 k;
  __u32 psize;
  __u32 nlayers;
```

(continues on next page)

(continued from previous page)

```
  __u32 nqc;
  __u32 nmqc;
  __u32 nm;
  __u32 norm_type;
  __u32 no_packing;
  __u32 special_qc;
  __u32 no_final_parity;
  __u32 max_schedule;
  __u32 sc_off;
  __u32 la_off;
  __u32 qc_off;
  __u32 *sc_table;
  __u32 *la_table;
  __u32 *qc_table;
  __u16 code_id;
};
```

**Members**

**n** Number of code word bits

**k** Number of information bits

**psize** Size of sub-matrix

**nlayers** Number of layers in code

**nqc** Quasi Cyclic Number

**nmqc** Number of M-sized QC operations in parity check matrix

**nm** Number of M-size vectors in N

**norm_type** Normalization required or not

**no_packing** Determines if multiple QC ops should be performed

**special_qc** Sub-Matrix property for Circulant weight > 0

**no_final_parity** Decide if final parity check needs to be performed

**max_schedule** Experimental code word scheduling limit

**sc_off** SC offset

**la_off** LA offset

**qc_off** QC offset

**sc_table** Pointer to SC Table which must be page aligned

**la_table** Pointer to LA Table which must be page aligned

**qc_table** Pointer to QC Table which must be page aligned

**code_id** LDPC Code

**Description**

This structure describes the LDPC code that is passed to the driver by the application.

struct **xsdfec_status**
Status of SD-FEC core.

**Definition**

```
struct xsdfec_status {
  __u32 state;
  __s8 activity;
};
```

**Members**

**state** State of the SD-FEC core

**activity** Describes if the SD-FEC instance is Active

struct **xsdfec_irq**
Enabling or Disabling Interrupts.

**Definition**

```
struct xsdfec_irq {
  __s8 enable_isr;
  __s8 enable_ecc_isr;
};
```

**Members**

**enable_isr** If true enables the ISR

**enable_ecc_isr** If true enables the ECC ISR

struct **xsdfec_config**
Configuration of SD-FEC core.

**Definition**

```
struct xsdfec_config {
  __u32 code;
  __u32 order;
  __u32 din_width;
  __u32 din_word_include;
  __u32 dout_width;
  __u32 dout_word_include;
  struct xsdfec_irq irq;
  __s8 bypass;
  __s8 code_wr_protect;
};
```

**Members**

**code** The codes being used by the SD-FEC instance

**order** Order of Operation

**din_width** Width of the DIN AXI4-Stream

**din_word_include** How DIN_WORDS are inputted

**dout_width** Width of the DOUT AXI4-Stream

**dout_word_include** HOW DOUT_WORDS are outputted

**irq** Enabling or disabling interrupts

**bypass** Is the core being bypassed

**code_wr_protect** Is write protection of LDPC codes enabled

struct **xsdfec_stats**
Stats retrived by ioctl XSDFEC_GET_STATS. Used to buffer atomic_t variables from struct xsdfec_dev. Counts are accumulated until the user clears them.

**Definition**

```
struct xsdfec_stats {
  __u32 isr_err_count;
  __u32 cecc_count;
  __u32 uecc_count;
};
```

**Members**

**isr_err_count** Count of ISR errors

**cecc_count** Count of Correctable ECC errors (SBE)

**uecc_count** Count of Uncorrectable ECC errors (MBE)

struct **xsdfec_ldpc_param_table_sizes**
Used to store sizes of SD-FEC table entries for an individual LPDC code parameter.

**Definition**

```
struct xsdfec_ldpc_param_table_sizes {
  __u32 sc_size;
  __u32 la_size;
  __u32 qc_size;
};
```

**Members**

**sc_size** Size of SC table used

**la_size** Size of LA table used

**qc_size** Size of QC table used