Linux Misc-devices Documentation

The kernel development community

Jun 10, 2024

CONTENTS

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

Table of contents

CHAPTER

ONE

AD525X DIGITAL POTENTIOMETERS

The ad525x_dpot driver exports a simple sysfs interface. This allows you to work with the immediate resistance settings as well as update the saved startup settings. Access to the factory programmed tolerance is also provided, but interpretation of this settings is required by the end application according to the specific part in use.

1.1 Files

Each dpot device will have a set of eeprom, rdac, and tolerance files. How many depends on the actual part you have, as will the range of allowed values.

The eeprom files are used to program the startup value of the device.

The rdac files are used to program the immediate value of the device.

The tolerance files are the read-only factory programmed tolerance settings and may vary greatly on a part-by-part basis. For exact interpretation of this field, please consult the datasheet for your part. This is presented as a hex file for easier parsing.

1.2 Example

Locate the device in your sysfs tree. This is probably easiest by going into the common i2c directory and locating the device by the i2c slave address:

```
# ls /sys/bus/i2c/devices/
0-0022  0-0027  0-002f
```

So assuming the device in question is on the first i2c bus and has the slave address of 0x2f, we descend (unrelated sysfs entries have been trimmed):

```
# ls /sys/bus/i2c/devices/0-002f/
eeprom0 rdac0 tolerance0
```

You can use simple reads/writes to access these files:

```
# cd /sys/bus/i2c/devices/0-002f/

# cat eeprom0
0
# echo 10 > eeprom0
```

```
# cat eeprom0
10

# cat rdac0
5
# echo 3 > rdac0
# cat rdac0
3
```

CHAPTER

TWO

KERNEL DRIVER APDS990X

Supported chips: Avago APDS990X

Data sheet: Not freely available

Author: Samu Onkalo <samu.p.onkalo@nokia.com>

2.1 Description

APDS990x is a combined ambient light and proximity sensor. ALS and proximity functionality are highly connected. ALS measurement path must be running while the proximity functionality is enabled.

ALS produces raw measurement values for two channels: Clear channel (infrared + visible light) and IR only. However, threshold comparisons happen using clear channel only. Lux value and the threshold level on the HW might vary quite much depending the spectrum of the light source.

Driver makes necessary conversions to both directions so that user handles only lux values. Lux value is calculated using information from the both channels. HW threshold level is calculated from the given lux value to match with current type of the lightning. Sometimes inaccuracy of the estimations lead to false interrupt, but that doesn't harm.

ALS contains 4 different gain steps. Driver automatically selects suitable gain step. After each measurement, reliability of the results is estimated and new measurement is triggered if necessary.

Platform data can provide tuned values to the conversion formulas if values are known. Otherwise plain sensor default values are used.

Proximity side is little bit simpler. There is no need for complex conversions. It produces directly usable values.

Driver controls chip operational state using pm_runtime framework. Voltage regulators are controlled based on chip operational state.

2.2 SYSFS

chip_id
RO - shows detected chip type and version

power_state
RW - enable / disable chip. Uses counting logic

1 enables the chip 0 disables the chip

lux0_input
RO - measured lux value

sysfs_notify called when threshold interrupt occurs

lux0_sensor_range
RO - lux0_input max value.

Actually never reaches since sensor tends to saturate much before that. Real max value varies depending on the light spectrum etc.

lux0_rate
RW - measurement rate in Hz

lux0_rate_avail
RO - supported measurement rates

lux0_calibscale
RW - calibration value.

Set to neutral value by default. Output results are multiplied with calibscale / calibscale_default value.

lux0_calibscale_default
RO - neutral calibration value

lux0_thresh_above_value
RW - HI level threshold value.

All results above the value trigs an interrupt. 65535 (i.e. sensor_range) disables the above interrupt.

lux0_thresh_below_value
RW - LO level threshold value.

All results below the value trigs an interrupt. 0 disables the below interrupt.

prox0_raw
RO - measured proximity value

sysfs_notify called when threshold interrupt occurs

prox0_sensor_range
RO - prox0_raw max value (1023)

prox0_raw_en
RW - enable / disable proximity - uses counting logic

- 1 enables the proximity
- 0 disables the proximity

prox0_reporting_mode
RW - trigger / periodic.

In “trigger” mode the driver tells two possible values: 0 or prox0_sensor_range value. 0 means no proximity, 1023 means proximity. This causes minimal number of interrupts. In “periodic” mode the driver reports all values above prox0_thresh_above. This causes more interrupts, but it can give _rough_ estimate about the distance.

prox0_reporting_mode_avail
RO - accepted values to prox0_reporting_mode (trigger, periodic)

prox0_thresh_above_value
RW - threshold level which trigs proximity events.

CHAPTER

THREE

KERNEL DRIVER BH1770GLC

Supported chips:

- ROHM BH1770GLC
- OSRAM SFH7770

Data sheet: Not freely available

Author: Samu Onkalo <samu.p.onkalo@nokia.com>

3.1 Description

BH1770GLC and SFH7770 are combined ambient light and proximity sensors. ALS and proximity parts operates on their own, but they shares common I2C interface and interrupt logic. In principle they can run on their own, but ALS side results are used to estimate reliability of the proximity sensor.

ALS produces 16 bit lux values. The chip contains interrupt logic to produce low and high threshold interrupts.

Proximity part contains IR-led driver up to 3 IR leds. The chip measures amount of reflected IR light and produces proximity result. Resolution is 8 bit. Driver supports only one channel. Driver uses ALS results to estimate reliability of the proximity results. Thus ALS is always running while proximity detection is needed.

Driver uses threshold interrupts to avoid need for polling the values. Proximity low interrupt doesn't exists in the chip. This is simulated by using a delayed work. As long as there is proximity threshold above interrupts the delayed work is pushed forward. So, when proximity level goes below the threshold value, there is no interrupt and the delayed work will finally run. This is handled as no proximity indication.

Chip state is controlled via runtime pm framework when enabled in config.

Calibscale factor is used to hide differences between the chips. By default value set to neutral state meaning factor of 1.00. To get proper values, calibrated source of light is needed as a reference. Calibscale factor is set so that measurement produces about the expected lux value.

3.2 SYSFS

chip_id
RO - shows detected chip type and version

power_state
RW - enable / disable chip

Uses counting logic

- 1 enables the chip
- 0 disables the chip

lux0_input
RO - measured lux value

sysfs_notify called when threshold interrupt occurs

lux0_sensor_range
RO - lux0_input max value

lux0_rate
RW - measurement rate in Hz

lux0_rate_avail
RO - supported measurement rates

lux0_thresh_above_value
RW - HI level threshold value

All results above the value trigs an interrupt. 65535 (i.e. sensor_range) disables the above interrupt.

lux0_thresh_below_value
RW - LO level threshold value

All results below the value trigs an interrupt. 0 disables the below interrupt.

lux0_calibscale
RW - calibration value

Set to neutral value by default. Output results are multiplied with calibscale / calibscale_default value.

lux0_calibscale_default
RO - neutral calibration value

prox0_raw
RO - measured proximity value

sysfs_notify called when threshold interrupt occurs

prox0_sensor_range
RO - prox0_raw max value

prox0_raw_en
RW - enable / disable proximity

Uses counting logic

- 1 enables the proximity

- 0 disables the proximity

prox0_thresh_above_count
RW - number of proximity interrupts needed before triggering the event

prox0_rate_above
RW - Measurement rate (in Hz) when the level is above threshold i.e. when proximity on has been reported.

prox0_rate_below
RW - Measurement rate (in Hz) when the level is below threshold i.e. when proximity off has been reported.

prox0_rate_avail
RO - Supported proximity measurement rates in Hz

prox0_thresh_above0_value
RW - threshold level which trigs proximity events.

Filtered by persistence filter (prox0_thresh_above_count)

prox0_thresh_above1_value
RW - threshold level which trigs event immediately

CHAPTER

FOUR

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.

4.1 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.

4.2 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.

4.3 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

FIVE

C2 PORT SUPPORT

5.1 Overview

This driver implements the support for Linux of Silicon Labs (Silabs) C2 Interface used for in-system programming of micro controllers.

By using this driver you can reprogram the in-system flash without EC2 or EC3 debug adapter. This solution is also useful in those systems where the micro controller is connected via special GPIOs pins.

5.2 References

The C2 Interface main references are at (https://www.silabs.com) Silicon Laboratories site], see:

- AN127: FLASH Programming via the C2 Interface at https://www.silabs.com/Support Documents/TechnicalDocs/an127.pdf
- C2 Specification at https://www.silabs.com/pages/DownloadDoc.aspx?FILEURL=Support%20Documents/TechnicalDocs/an127.pdf&src=SearchResults

however it implements a two wire serial communication protocol (bit banging) designed to enable in-system programming, debugging, and boundary-scan testing on low pin-count Silicon Labs devices. Currently this code supports only flash programming but extensions are easy to add.

5.3 Using the driver

Once the driver is loaded you can use sysfs support to get C2port's info or read/write in-system flash:

```
# ls /sys/class/c2port/c2port0/
access            flash_block_size  flash_erase       rev_id
dev_id            flash_blocks_num  flash_size        subsystem/
flash_access      flash_data        reset             uevent
```

Initially the C2port access is disabled since you hardware may have such lines multiplexed with other devices so, to get access to the C2port, you need the command:

```
# echo 1 > /sys/class/c2port/c2port0/access
```

after that you should read the device ID and revision ID of the connected micro controller:

```
# cat /sys/class/c2port/c2port0/dev_id
8
# cat /sys/class/c2port/c2port0/rev_id
1
```

However, for security reasons, the in-system flash access in not enabled yet, to do so you need the command:

```
# echo 1 > /sys/class/c2port/c2port0/flash_access
```

After that you can read the whole flash:

```
# cat /sys/class/c2port/c2port0/flash_data > image
```

erase it:

```
# echo 1 > /sys/class/c2port/c2port0/flash_erase
```

and write it:

```
# cat image > /sys/class/c2port/c2port0/flash_data
```

after writing you have to reset the device to execute the new code:

```
# echo 1 > /sys/class/c2port/c2port0/reset
```

CHAPTER

SIX

DRIVER FOR SYNOPSYS DESIGNWARE PCIE TRAFFIC GENERATOR (ALSO KNOWN AS XDATA)

Supported chips: Synopsys DesignWare PCIe prototype solution

Datasheet: Not freely available

Author: Gustavo Pimentel <gustavo.pimentel@synopsys.com>

6.1 Description

This driver should be used as a host-side (Root Complex) driver and Synopsys DesignWare prototype that includes this IP.

The dw-xdata-pcie driver can be used to enable/disable PCIe traffic generator in either direction (mutual exclusion) besides allowing the PCIe link performance analysis.

The interaction with this driver is done through the module parameter and can be changed in runtime. The driver outputs the requested command state information to `/var/log/kern.log` or dmesg.

6.2 Example

6.2.1 Write TLPs traffic generation - Root Complex to Endpoint direction

Generate traffic:

```
# echo 1 > /sys/class/misc/dw-xdata-pcie.0/write
```

Get link throughput in MB/s:

```
# cat /sys/class/misc/dw-xdata-pcie.0/write
204
```

Stop traffic in any direction:

```
# echo 0 > /sys/class/misc/dw-xdata-pcie.0/write
```

6.2.2 Read TLPs traffic generation - Endpoint to Root Complex direction

Generate traffic:

```
# echo 1 > /sys/class/misc/dw-xdata-pcie.0/read
```

Get link throughput in MB/s:

```
# cat /sys/class/misc/dw-xdata-pcie.0/read
199
```

Stop traffic in any direction:

```
# echo 0 > /sys/class/misc/dw-xdata-pcie.0/read
```

CHAPTER

SEVEN

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>,

7.1 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.

7.1.1 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

7.1.2 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.

7.1.3 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.

7.2 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.

7.2.1 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.

```
Management Partition            Hypervisor
                 CRQ INIT
------------------------------------------>
           CRQ INIT COMPLETE
<------------------------------------------
              CAPABILITIES
------------------------------------------>
         CAPABILITIES RESPONSE
<------------------------------------------
      ADD BUFFER (HMC IDX=0,1,..)          _
<------------------------------------------  |
          ADD BUFFER RESPONSE                | - Perform # HMCs Iterations
------------------------------------------> -
```

7.2.2 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
<----------------------------------------
```

7.2.3 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
<----------------------------------------
```

7.2.4 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
---------------------------------------->
        INTERFACE CLOSE RESPONSE
<----------------------------------------
```

7.3 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

EIGHT

KERNEL DRIVER ICS932S401

Supported chips:

- IDT ICS932S401

  Prefix: 'ics932s401'

  Addresses scanned: I2C 0x69

  Datasheet: Publicly available at the IDT website

Author: Darrick J. Wong

8.1 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.

8.2 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

NINE

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>

9.1 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.

9.2 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.

9.3 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

TEN

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>

10.1 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.

10.2 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.

10.3 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

ELEVEN

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>

11.1 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.

11.2 Sysfs entries

eeprom - 512 bytes of user-defined EEPROM space.

11.3 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'.

11.4 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

TWELVE

NOTES ON OXFORD SEMICONDUCTOR PCIE (TORNADO) 950 SERIAL PORT DEVICES

Oxford Semiconductor PCIe (Tornado) 950 serial port devices are driven by a fixed 62.5MHz clock input derived from the 100MHz PCI Express clock.

The baud rate produced by the baud generator is obtained from this input frequency by dividing it by the clock prescaler, which can be set to any value from 1 to 63.875 in increments of 0.125, and then the usual 16-bit divisor is used as with the original 8250, to divide the frequency by a value from 1 to 65535. Finally a programmable oversampling rate is used that can take any value from 4 to 16 to divide the frequency further and determine the actual baud rate used. Baud rates from 15625000bps down to 0.933bps can be obtained this way.

By default the oversampling rate is set to 16 and the clock prescaler is set to 33.875, meaning that the frequency to be used as the reference for the usual 16-bit divisor is 115313.653, which is close enough to the frequency of 115200 used by the original 8250 for the same values to be used for the divisor to obtain the requested baud rates by software that is unaware of the extra clock controls available.

The oversampling rate is programmed with the TCR register and the clock prescaler is programmed with the CPR/CPR2 register pair [OX200] [OX952] [OX954] [OX958]. To switch away from the default value of 33.875 for the prescaler the enhanced mode has to be explicitly enabled though, by setting bit 4 of the EFR. In that mode setting bit 7 in the MCR enables the prescaler or otherwise it is bypassed as if the value of 1 was used. Additionally writing any value to CPR clears CPR2 for compatibility with old software written for older conventional PCI Oxford Semiconductor devices that do not have the extra prescaler's 9th bit in CPR2, so the CPR/CPR2 register pair has to be programmed in the right order.

By using these parameters rates from 15625000bps down to 1bps can be obtained, with either exact or highly-accurate actual bit rates for standard and many non-standard rates.

Here are the figures for the standard and some non-standard baud rates (including those quoted in Oxford Semiconductor documentation), giving the requested rate (r), the actual rate yielded (a) and its deviation from the requested rate (d), and the values of the oversampling rate (tcr), the clock prescaler (cpr) and the divisor (div) produced by the new `get_divisor` handler:

```
r: 15625000, a: 15625000.00, d:  0.0000%, tcr:  4, cpr:  1.000, div:     1
r: 12500000, a: 12500000.00, d:  0.0000%, tcr:  5, cpr:  1.000, div:     1
r: 10416666, a: 10416666.67, d:  0.0000%, tcr:  6, cpr:  1.000, div:     1
r:  8928571, a:  8928571.43, d:  0.0000%, tcr:  7, cpr:  1.000, div:     1
r:  7812500, a:  7812500.00, d:  0.0000%, tcr:  8, cpr:  1.000, div:     1
r:  4000000, a:  4000000.00, d:  0.0000%, tcr:  5, cpr:  3.125, div:     1
r:  3686400, a:  3676470.59, d: -0.2694%, tcr:  8, cpr:  2.125, div:     1
r:  3500000, a:  3496503.50, d: -0.0999%, tcr: 13, cpr:  1.375, div:     1
```

```
r:  3000000, a:  2976190.48, d: -0.7937%, tcr: 14, cpr:  1.500, div:     1
r:  2500000, a:  2500000.00, d:  0.0000%, tcr: 10, cpr:  2.500, div:     1
r:  2000000, a:  2000000.00, d:  0.0000%, tcr: 10, cpr:  3.125, div:     1
r:  1843200, a:  1838235.29, d: -0.2694%, tcr: 16, cpr:  2.125, div:     1
r:  1500000, a:  1492537.31, d: -0.4975%, tcr:  5, cpr:  8.375, div:     1
r:  1152000, a:  1152073.73, d:  0.0064%, tcr: 14, cpr:  3.875, div:     1
r:   921600, a:   919117.65, d: -0.2694%, tcr: 16, cpr:  2.125, div:     2
r:   576000, a:   576036.87, d:  0.0064%, tcr: 14, cpr:  3.875, div:     2
r:   460800, a:   460829.49, d:  0.0064%, tcr:  7, cpr:  3.875, div:     5
r:   230400, a:   230414.75, d:  0.0064%, tcr: 14, cpr:  3.875, div:     5
r:   115200, a:   115207.37, d:  0.0064%, tcr: 14, cpr:  1.250, div:    31
r:    57600, a:    57603.69, d:  0.0064%, tcr:  8, cpr:  3.875, div:    35
r:    38400, a:    38402.46, d:  0.0064%, tcr: 14, cpr:  3.875, div:    30
r:    19200, a:    19201.23, d:  0.0064%, tcr:  8, cpr:  3.875, div:   105
r:     9600, a:     9600.06, d:  0.0006%, tcr:  9, cpr:  1.125, div:   643
r:     4800, a:     4799.98, d: -0.0004%, tcr:  7, cpr:  2.875, div:   647
r:     2400, a:     2400.02, d:  0.0008%, tcr:  9, cpr:  2.250, div:  1286
r:     1200, a:     1200.00, d:  0.0000%, tcr: 14, cpr:  2.875, div:  1294
r:      300, a:      300.00, d:  0.0000%, tcr: 11, cpr:  2.625, div:  7215
r:      200, a:      200.00, d:  0.0000%, tcr: 16, cpr:  1.250, div: 15625
r:      150, a:      150.00, d:  0.0000%, tcr: 13, cpr:  2.250, div: 14245
r:      134, a:      134.00, d:  0.0000%, tcr: 11, cpr:  2.625, div: 16153
r:      110, a:      110.00, d:  0.0000%, tcr: 12, cpr:  1.000, div: 47348
r:       75, a:       75.00, d:  0.0000%, tcr:  4, cpr:  5.875, div: 35461
r:       50, a:       50.00, d:  0.0000%, tcr: 16, cpr:  1.250, div: 62500
r:       25, a:       25.00, d:  0.0000%, tcr: 16, cpr:  2.500, div: 62500
r:        4, a:        4.00, d:  0.0000%, tcr: 16, cpr: 20.000, div: 48828
r:        2, a:        2.00, d:  0.0000%, tcr: 16, cpr: 40.000, div: 48828
r:        1, a:        1.00, d:  0.0000%, tcr: 16, cpr: 63.875, div: 61154
```

With the baud base set to 15625000 and the unsigned 16-bit UART_DIV_MAX limitation imposed by `serial8250_get_baud_rate` standard baud rates below 300bps become unavailable in the regular way, e.g. the rate of 200bps requires the baud base to be divided by 78125 and that is beyond the unsigned 16-bit range. The historic spd_cust feature can still be used by encoding the values for, the prescaler, the oversampling rate and the clock divisor (DLM/DLL) as follows to obtain such rates if so required:

```
 31 29 28             20 19   16 15                             0
+-----+-----------------+-------+---------------------------------+
|0 0 0|    CPR2:CPR     |  TCR  |            DLM:DLL              |
+-----+-----------------+-------+---------------------------------+
```

Use a value such encoded for the `custom_divisor` field along with the ASYNC_SPD_CUST flag set in the `flags` field in `struct serial_struct` passed with the TIOCSSERIAL ioctl(2), such as with the setserial(8) utility and its `divisor` and `spd_cust` parameters, and then select the baud rate of 38400bps. Note that the value of 0 in TCR sets the oversampling rate to 16 and prescaler values below 1 in CPR2/CPR are clamped by the driver to 1.

For example the value of 0x1f4004e2 will set CPR2/CPR, TCR and DLM/DLL respectively to 0x1f4, 0x0 and 0x04e2, choosing the prescaler value, the oversampling rate and the clock divisor of 62.500, 16 and 1250 respectively. These parameters will set the baud rate for the serial

port to 62500000 / 62.500 / 1250 / 16 = 50bps.

Maciej W. Rozycki <macro@orcam.me.uk>

CHAPTER

THIRTEEN

DRIVER FOR PCI ENDPOINT TEST FUNCTION

This driver should be used as a host side driver if the root complex is connected to a configurable PCI endpoint running `pci_epf_test` function driver configured according to[1].

The “pci_endpoint_test” driver can be used to perform the following tests.

The PCI driver for the test device performs the following tests:

1) verifying addresses programmed in BAR
2) raise legacy IRQ
3) raise MSI IRQ
4) raise MSI-X IRQ
5) read data
6) write data
7) copy data

This misc driver creates /dev/pci-endpoint-test.<num> for every `pci_epf_test` function connected to the root complex and “ioctls” should be used to perform the above tests.

13.1 ioctl

PCITEST_BAR:
Tests the BAR. The number of the BAR to be tested should be passed as argument.

PCITEST_LEGACY_IRQ:
Tests legacy IRQ

PCITEST_MSI:
Tests message signalled interrupts. The MSI number to be tested should be passed as argument.

PCITEST_MSIX:
Tests message signalled interrupts. The MSI-X number to be tested should be passed as argument.

PCITEST_SET_IRQTYPE:
Changes driver IRQ type configuration. The IRQ type should be passed as argument (0: Legacy, 1:MSI, 2:MSI-X).

[1] Documentation/PCI/endpoint/function/binding/pci-test.rst

PCITEST_GET_IRQTYPE:
Gets driver IRQ type configuration.

PCITEST_WRITE:
Perform write tests. The size of the buffer should be passed as argument.

PCITEST_READ:
Perform read tests. The size of the buffer should be passed as argument.

PCITEST_COPY:
Perform read tests. The size of the buffer should be passed as argument.

CHAPTER

FOURTEEN

SPEAR PCIE GADGET DRIVER

14.1 Author

Pratyush Anand (pratyush.anand@gmail.com)

14.2 Location

driver/misc/spear13xx_pcie_gadget.c

14.3 Supported Chip:

SPEAr1300 SPEAr1310

14.4 Menuconfig option:

Device Drivers

Misc devices
PCIe gadget support for SPEAr13XX platform

14.5 purpose

This driver has several nodes which can be read/written by configfs interface. Its main purpose is to configure selected dual mode PCIe controller as device and then program its various registers to configure it as a particular device type. This driver can be used to show spear's PCIe device capability.

14.6 Description of different nodes:

14.6.1 read behavior of nodes:

link	gives ltssm status.
int_type	type of supported interrupt
no_of_msi	zero if MSI is not enabled by host. A positive value is the number of MSI vector granted.
vendor_id	returns programmed vendor id (hex)
device_id	returns programmed device id(hex)
bar0_size:	returns size of bar0 in hex.
bar0_address	returns address of bar0 mapped area in hex.
bar0_rw_offset	returns offset of bar0 for which bar0_data will return value.
bar0_data	returns data at bar0_rw_offset.

14.6.2 write behavior of nodes:

link	write UP to enable ltsmm DOWN to disable
int_type	write interrupt type to be configured and (int_type could be INTA, MSI or NO_INT). Select MSI only when you have programmed no_of_msi node.
no_of_msi	number of MSI vector needed.
inta	write 1 to assert INTA and 0 to de-assert.
send_msi	write MSI vector to be sent.
vendor_id	write vendor id(hex) to be programmed.
device_id	write device id(hex) to be programmed.
bar0_size	write size of bar0 in hex. default bar0 size is 1000 (hex) bytes.
bar0_address	write address of bar0 mapped area in hex. (default mapping of bar0 is SYSRAM1(E0800000). Always program bar size before bar address. Kernel might modify bar size and address for alignment, so read back bar size and address after writing to cross check.
bar0_rw_offset	write offset of bar0 for which bar0_data will write value.
bar0_data	write data to be written at bar0_rw_offset.

14.7 Node programming example

Program all PCIe registers in such a way that when this device is connected to the PCIe host, then host sees this device as 1MB RAM.

```
#mount -t configfs none /Config
```

For nth PCIe Device Controller:

```
# cd /config/pcie_gadget.n/
```

Now you have all the nodes in this directory. program vendor id as 0x104a:

```
# echo 104A >> vendor_id
```

program device id as 0xCD80:

```
# echo CD80 >> device_id
```

program BAR0 size as 1MB:

```
# echo 100000 >> bar0_size
```

check for programmed bar0 size:

```
# cat bar0_size
```

Program BAR0 Address as DDR (0x2100000). This is the physical address of memory, which is to be made visible to PCIe host. Similarly any other peripheral can also be made visible to PCIe host. E.g., if you program base address of UART as BAR0 address then when this device will be connected to a host, it will be visible as UART.

```
# echo 2100000 >> bar0_address
```

program interrupt type : INTA:

```
# echo INTA >> int_type
```

go for link up now:

```
# echo UP >> link
```

It will have to be insured that, once link up is done on gadget, then only host is initialized and start to search PCIe devices on its port.

```
/*wait till link is up*/
# cat link
```

Wait till it returns UP.

To assert INTA:

```
# echo 1 >> inta
```

To de-assert INTA:

```
# echo 0 >> inta
```

if MSI is to be used as interrupt, program no of msi vector needed (say4):

```
# echo 4 >> no_of_msi
```

select MSI as interrupt type:

```
# echo MSI >> int_type
```

go for link up now:

```
# echo UP >> link
```

wait till link is up:

```
# cat link
```

An application can repetitively read this node till link is found UP. It can sleep between two read.

wait till msi is enabled:

```
# cat no_of_msi
```

Should return 4 (number of requested MSI vector)

to send msi vector 2:

```
# echo 2 >> send_msi
# cd -
```

CHAPTER

FIFTEEN

TEXAS INSTRUMENTS TPS6594 PFSM DRIVER

Author: Julien Panis (jpanis@baylibre.com)

15.1 Overview

Strictly speaking, PFSM (Pre-configurable Finite State Machine) is not hardware. It is a piece of code.

The TPS6594 PMIC (Power Management IC) integrates a state machine which manages operational modes. Depending on the current operational mode, some voltage domains remain energized while others can be off.

The PFSM driver can be used to trigger transitions between configured states. It also provides R/W access to the device registers.

15.1.1 Supported chips

- tps6594-q1
- tps6593-q1
- lp8764-q1

15.2 Driver location

drivers/misc/tps6594-pfsm.c

15.3 Driver type definitions

include/uapi/linux/tps6594_pfsm.h

15.4 Driver IOCTLs

:c:macro::*PMIC_GOTO_STANDBY* All device resources are powered down. The processor is off, and no voltage domains are energized.

:c:macro::*PMIC_GOTO_LP_STANDBY* The digital and analog functions of the PMIC, which are not required to be always-on, are turned off (low-power).

:c:macro::*PMIC_UPDATE_PGM* Triggers a firmware update.

:c:macro::*PMIC_SET_ACTIVE_STATE* One of the operational modes. The PMICs are fully functional and supply power to all PDN loads. All voltage domains are energized in both MCU and Main processor sections.

:c:macro::*PMIC_SET_MCU_ONLY_STATE* One of the operational modes. Only the power resources assigned to the MCU Safety Island are on.

:c:macro::*PMIC_SET_RETENTION_STATE* One of the operational modes. Depending on the triggers set, some DDR/GPIO voltage domains can remain energized, while all other domains are off to minimize total system power.

15.5 Driver usage

See available PFSMs:

```
# ls /dev/pfsm*
```

Dump the registers of pages 0 and 1:

```
# hexdump -C /dev/pfsm-0-0x48
```

See PFSM events:

```
# cat /proc/interrupts
```

15.5.1 Userspace code example

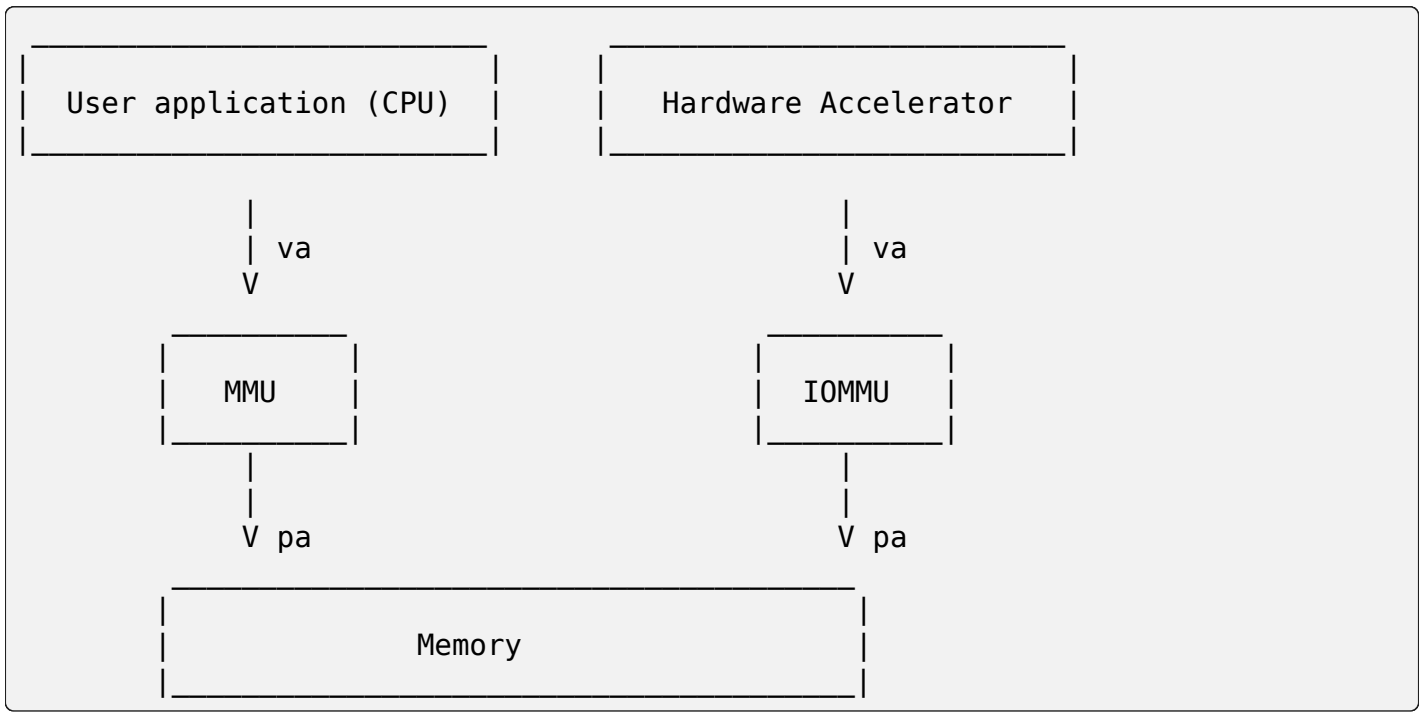
samples/pfsm/pfsm-wakeup.c

CHAPTER

SIXTEEN

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

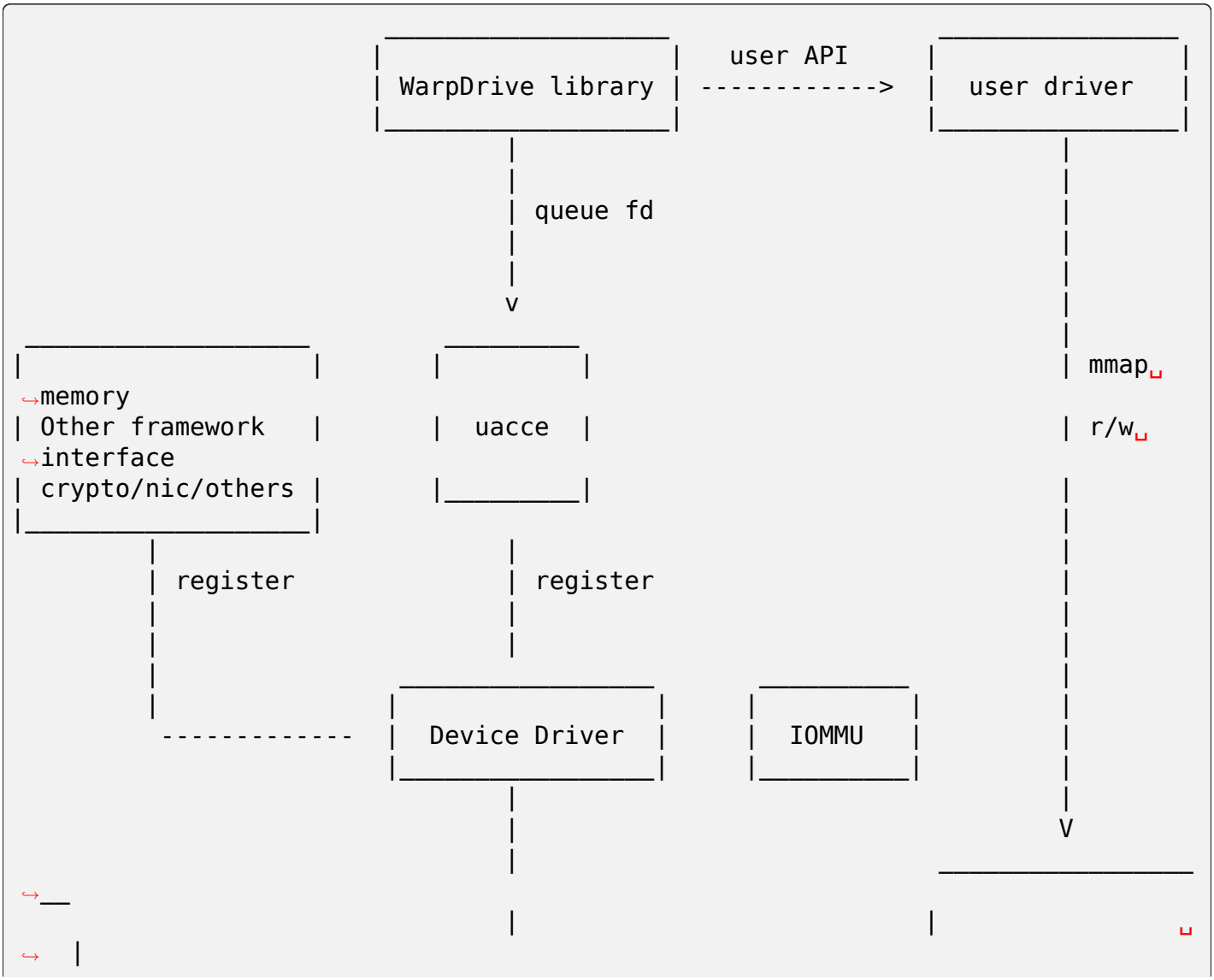
SEVENTEEN

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.



```
                                  ---------------------------  | ␣
↪Device(Hardware) |
                                                               |_________________
↪__|
```

CHAPTER

EIGHTEEN

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

NINETEEN

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

TWENTY

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

TWENTYONE

XILINX SD-FEC DRIVER

21.1 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,sd-fec.txt

21.1.1 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

21.2 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 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*

21.3 Driver Usage

21.3.1 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.

21.3.2 Determine Configuration

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

21.3.3 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

21.3.4 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

21.3.5 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

21.3.6 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.

21.3.7 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

21.3.8 Start the SD-FEC core

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

21.3.9 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*

21.3.10 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*

21.3.11 Stop the SD-FEC Core

Stop the device by using the ioctl *XSDFEC_STOP_DEV*

21.3.12 Set the Default Configuration

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

21.3.13 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.

21.4 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

21.5 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;
    __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

BIBLIOGRAPHY

[OX200] “OXPCIe200 PCI Express Multi-Port Bridge”, Oxford Semiconductor, Inc., DS-0045, 10 Nov 2008, Section “950 Mode”, pp. 64-65

[OX952] “OXPCIe952 PCI Express Bridge to Dual Serial & Parallel Port”, Oxford Semiconductor, Inc., DS-0046, Mar 06 08, Section “950 Mode”, p. 20

[OX954] “OXPCIe954 PCI Express Bridge to Quad Serial Port”, Oxford Semiconductor, Inc., DS-0047, Feb 08, Section “950 Mode”, p. 20

[OX958] “OXPCIe958 PCI Express Bridge to Octal Serial Port”, Oxford Semiconductor, Inc., DS-0048, Feb 08, Section “950 Mode”, p. 20

INDEX

X