CPSC 8220 – Spring 2017
Project #1: Device Driver
Due: 2 March 2017

The following driver and user code are the results of the collaborative effort of *Team eXtreme*:

x _____
Ashwin Kumar Vajantri [aashwin@g.clemson.edu]

x _____
Kunwar Deep Singh Toor [ktoor@g.clemson.edu]

x _____
Matthew Pfister [mpfiste@g.clemson.edu]

x _____
Rohith Raju [rraju@g.clemson.edu]

x _____
Saroj Kumar Dash [sdash@g.clemson.edu]

x _____
Vishnuprabhu Thirugnanasambandam [vthirug@g.clemson.edu]

```
// START OF DEVICE DRIVER CODE
// Description:
//    Device driver for the Kyouko3 PCIe graphics card
//    featuring the ability to issue commands to draw triangles
//    via the FIFO queue and DMA buffers.

#include "mymod.h"

#include <linux/cdev.h> // for struct cdev
#include <linux/delay.h> // for udelay
#include <linux/fs.h> // for struct file_operations
#include <linux/interrupt.h> // for request irq
#include <linux/irqreturn.h> // for irqreturn_t
#include <linux/mm_types.h> // for u32 and u64
#include <linux/mod_devicetable.h> // for struct pci_device_id
#include <linux/module.h> // for MODULE_LICENSE and MODULE_AUTHOR
#include <linux/pci.h> // for struct pci_driver
#include <linux/sched.h> // for schedule
#include <linux/spinlock.h> // for spinlock_t
#include <linux/spinlock_types.h> // SPIN_LOCK_UNLOCKED
#include <linux/wait.h> // for wait_event_interruptible
#include <asm/io.h> // for ioremap
#include <asm/uaccess.h> // for copy_from_user
#include <asm/current.h> // for current
#include <asm/mman.h> // for PROT_READ and PROT_WRITE

MODULE_LICENSE("Proprietary");
MODULE_AUTHOR("Team eXtreme");
DECLARE_WAIT_QUEUE_HEAD(dma_not_full); // For waiting for non-full buffer queue.
DECLARE_WAIT_QUEUE_HEAD(dma_empty);    // For waiting for empty buffer queue.

// ~~ Device Info
#define KYOUKO3_MAJOR 500
#define KYOUKO3_MINOR 127

#define PCI_VENDOR_ID_CCORSI 0x1234
#define PCI_DEVICE_ID_CCORSI_KYOUKO3 0x1113

#define CONTROL_SIZE 65536
// ~~

// ~~ Registers
#define DeviceVRAM 0x0020

#define FifoStart 0x1020
#define FifoEnd 0x1024
#define FifoHead 0x4010
#define FifoTail 0x4014

#define FrameColumns 0x8000
#define FrameRows 0x8004
#define FrameRowPitch 0x8008
#define FramePixelFormat 0x800C
#define FrameStartAddress 0x8010

#define EncoderWidth 0x9000
#define EncoderHeight 0x9004
#define EncoderOffsetX 0x9008
#define EncoderOffsetY 0x900C
#define EncoderFrame 0x9010

#define ConfigAcceleration 0x1010
#define ConfigModeSet 0x1008
#define ConfigInterrupt 0x100C
```

```
#define InterruptStatus 0x4008
#define BufferAAddress 0x2000
#define BufferAConfig 0x2008

#define DrawClearColor4fBlue 0x5100
#define DrawClearColor4fGreen 0x5104
#define DrawClearColor4fRed 0x5108
#define DrawClearColor4fAlpha 0x510C

#define RasterClear 0x3008
// ~~

// ~~ Buffer Info
#define FIFO_ENTRIES 1024
#define DMA_NUM_BUFFS 8
#define DMA_BUFF_SIZE 126976u
// ~~

// ~~ Clear Color
#define CLEAR_RED 0.3f
#define CLEAR_GREEN 0.3f
#define CLEAR_BLUE 0.3f
// ~~

// ~~ FIFO Queue
struct fifo_entry {
   u32 command;
   u32 value;
};

struct fifo {
   u64 p_base;
   struct fifo_entry *k_base;
   u32 head;
   u32 tail_cache;
};
//~~

// ~~ DMA Buffer
struct dma_buffer {
   u64 p_base;           // physical address on the card
   unsigned int* k_base; // kernel virtual address
   unsigned long u_base; // user virtual address
   int stored_count;     // number of bytes in buffer
};

// ~~ God Structure
struct kyouko3 {
   struct cdev cdev;
   struct pci_dev* pci_dev;
   unsigned long p_control_base;
   unsigned long p_ram_card_base;
   unsigned int* k_control_base;
   unsigned int* k_ram_card_base;
   struct fifo fifo;
   _Bool graphics_on;
   _Bool dma_mapped;
   struct dma_buffer dma_buffers[DMA_NUM_BUFFS];
   int fill;
   int drain;
   int queued_count;
} kyouko3;
// ~~
```

```
 // ~~ Helper methods
unsigned int K_READ_REGISTER(unsigned int reg) {
   udelay(1);
   rmb();
   return *(kyouko3.k_control_base + (reg >> 2));
}

void K_WRITE_REGISTER(unsigned int reg, unsigned int value) {
   udelay(1);
   *(kyouko3.k_control_base + (reg >> 2)) = value;
}

void FIFO_WRITE(unsigned int reg, unsigned int value) {
   kyouko3.fifo.k_base[kyouko3.fifo.head].command = reg;
   kyouko3.fifo.k_base[kyouko3.fifo.head].value = value;
   ++kyouko3.fifo.head;
   if(kyouko3.fifo.head >= FIFO_ENTRIES) kyouko3.fifo.head = 0;
}
//~~

// ~~ Interrupt Handler
irqreturn_t dma_intr(int irq, void *dev_id, struct pt_regs *regs) {
   unsigned int iflags;

   iflags = K_READ_REGISTER(InterruptStatus);
   K_WRITE_REGISTER(InterruptStatus, iflags & 0xf);
   if((iflags & 0x02) == 0) return (IRQ_NONE);

   --kyouko3.queued_count;
   kyouko3.drain = (kyouko3.drain + 1) % DMA_NUM_BUFFS;

   if(kyouko3.queued_count + 1 == DMA_NUM_BUFFS)
      wake_up_interruptible(&dma_not_full);

   if(kyouko3.queued_count != 0) {
      FIFO_WRITE(BufferAAddress,kyouko3.dma_buffers[kyouko3.drain].p_base);
      FIFO_WRITE(BufferAConfig,kyouko3.dma_buffers[kyouko3.drain].stored_count);
      K_WRITE_REGISTER(FifoHead,kyouko3.fifo.head);
   }
   else  wake_up_interruptible(&dma_empty);

   return (IRQ_HANDLED);
}
// ~~

// ~~ DMA Commands
unsigned long bind_dma(struct file* fp) {
   int i, result;

   if(kyouko3.dma_mapped) {
      printk(KERN_ALERT "User tried to bind DMA multiple times.\n");
      return 0;
   }

   result = pci_enable_msi(kyouko3.pci_dev);
   if(result) {
      printk(KERN_ALERT "Failed to enable message signal interrupts.\n");
      return 0;
   }

   result = request_irq(
      kyouko3.pci_dev->irq,
      (irq_handler_t)dma_intr,
      IRQF_SHARED, "dma_intr", &kyouko3);
```

```
   if(result) {
      pci_disable_msi(kyouko3.pci_dev);
      printk(KERN_ALERT "Failed to request irq.\n");
      return 0;
   }

   for(i = 0; i < DMA_NUM_BUFFS; ++i) {
      kyouko3.dma_buffers[i].k_base =
         pci_alloc_consistent(
            kyouko3.pci_dev,
            DMA_BUFF_SIZE,
            &kyouko3.dma_buffers[i].p_base);
      kyouko3.dma_buffers[i].u_base =
         vm_mmap(
            fp, 0, DMA_BUFF_SIZE,
            PROT_READ|PROT_WRITE,
            MAP_SHARED, (i + 1) << PAGE_SHIFT) ;
   }

   kyouko3.dma_mapped = 1;
   kyouko3.fill = 0;
   kyouko3.drain = 0;
   kyouko3.queued_count = 0;
   K_WRITE_REGISTER(ConfigInterrupt, 0x02);

   return kyouko3.dma_buffers[kyouko3.fill].u_base;
}

void late_unbind_dma(void) {
   int i;

   if(kyouko3.queued_count != 0)
      wait_event_interruptible(dma_empty, kyouko3.queued_count == 0);

   for(i = 0; i < DMA_NUM_BUFFS; ++i)
      pci_free_consistent(
         kyouko3.pci_dev,
         DMA_BUFF_SIZE,
         kyouko3.dma_buffers[i].k_base,
         kyouko3.dma_buffers[i].p_base);

   K_WRITE_REGISTER(ConfigInterrupt, 0x0);
   free_irq(kyouko3.pci_dev->irq, &kyouko3);
   pci_disable_msi(kyouko3.pci_dev);
   kyouko3.dma_mapped = 0;
}

unsigned long on_time_unbind_dma(void) {
   int i;

   if(! kyouko3.dma_mapped) {
      printk(KERN_ALERT "User tried to unbind before a call to bind.\n");
      return 1;
   }

   for(i = 0; i < DMA_NUM_BUFFS; ++i)
      vm_munmap(kyouko3.dma_buffers[i].u_base, DMA_BUFF_SIZE);

   late_unbind_dma();
   return 0;
}
```

```
unsigned long start_dma(int count) {
   unsigned long flags;
   DEFINE_SPINLOCK(spinlock);
   _Bool suspend = 0;

   if(count == 0) {
      printk(KERN_ALERT "User called start_dma with a count of zero.\n");
      return 0;
   }

   if(count > DMA_BUFF_SIZE) {
      printk(
         KERN_ALERT "User called start_dma with too large of a byte size.\n");
      return 0;
   }

   spin_lock_irqsave(&spinlock, flags);
   ++kyouko3.queued_count;

   if(kyouko3.fill == kyouko3.drain) {
      spin_unlock_irqrestore(&spinlock, flags);
      kyouko3.fill = (kyouko3.fill + 1) % DMA_NUM_BUFFS;
      FIFO_WRITE(BufferAAddress, kyouko3.dma_buffers[kyouko3.drain].p_base);
      FIFO_WRITE(BufferAConfig, count);
      K_WRITE_REGISTER(FifoHead, kyouko3.fifo.head);
      return kyouko3.dma_buffers[kyouko3.fill].u_base;
   }

   kyouko3.dma_buffers[kyouko3.fill].stored_count = count;
   kyouko3.fill = (kyouko3.fill + 1) % DMA_NUM_BUFFS;

   if(kyouko3.fill == kyouko3.drain) suspend = 1;
   spin_unlock_irqrestore(&spinlock, flags);
   if(suspend)
      wait_event_interruptible(
         dma_not_full,
         kyouko3.queued_count != DMA_NUM_BUFFS);

   return kyouko3.dma_buffers[kyouko3.fill].u_base;
}
// ~~

// ~~ FIFO Commands
long fifo_queue(unsigned int cmd, unsigned long arg) {
   long ret;
   struct fifo_entry fifo_entry;
   ret =
      copy_from_user(&fifo_entry,
         (struct fifo_entry*)arg,
          sizeof(struct fifo_entry));
   FIFO_WRITE(fifo_entry.command, fifo_entry.value);
   return ret;
}

void fifo_flush(void) {
   K_WRITE_REGISTER(FifoHead, kyouko3.fifo.head);
   while(kyouko3.fifo.tail_cache != kyouko3.fifo.head) {
      kyouko3.fifo.tail_cache = K_READ_REGISTER(FifoTail);
      schedule();
   }
}
// ~~
```

```
// ~~ Graphics Commands
void graphics_on(void) {
   float clear_red = CLEAR_RED,
         clear_green = CLEAR_GREEN,
         clear_blue = CLEAR_BLUE,
         clear_alpha = 0.0f;

   K_WRITE_REGISTER(FrameColumns, 1024);
   K_WRITE_REGISTER(FrameRows, 768);
   K_WRITE_REGISTER(FrameRowPitch, 1024*4);
   K_WRITE_REGISTER(FramePixelFormat, 0xf888);
   K_WRITE_REGISTER(FrameStartAddress, 0);

   K_WRITE_REGISTER(ConfigAcceleration, 0x40000000);

   K_WRITE_REGISTER(EncoderWidth, 1024);
   K_WRITE_REGISTER(EncoderHeight, 768);
   K_WRITE_REGISTER(EncoderOffsetX, 0);
   K_WRITE_REGISTER(EncoderOffsetY, 0);
   K_WRITE_REGISTER(EncoderFrame, 0);

   K_WRITE_REGISTER(ConfigModeSet, 0);

   msleep(50);

   FIFO_WRITE(DrawClearColor4fRed, *(unsigned int*)(&clear_red));
   FIFO_WRITE(DrawClearColor4fGreen, *(unsigned int*)(&clear_green));
   FIFO_WRITE(DrawClearColor4fBlue, *(unsigned int*)(&clear_blue));
   FIFO_WRITE(DrawClearColor4fAlpha, *(unsigned int*)(&clear_alpha));

   FIFO_WRITE(RasterClear, 0x03);
   FIFO_WRITE(RasterFlush, 0x0);

   fifo_flush();

   kyouko3.graphics_on = 1;
}

void graphics_off(void) {
   FIFO_WRITE(RasterClear, 0x03);
   fifo_flush();
   K_WRITE_REGISTER(ConfigAcceleration, 0x80000000);
   K_WRITE_REGISTER(ConfigModeSet, 0);
   msleep(10);
   kyouko3.graphics_on = 0;
}
// ~~

// ~~ File Operations
int kyouko3_open(struct inode *inode, struct file *fp) {
   unsigned int ram_size;

   kyouko3.k_control_base = ioremap(kyouko3.p_control_base, CONTROL_SIZE);
   ram_size = K_READ_REGISTER(DeviceVRAM);
   ram_size *= 1024 * 1024;
   kyouko3.k_ram_card_base = ioremap(kyouko3.p_ram_card_base, ram_size);

   kyouko3.fifo.k_base =
      pci_alloc_consistent(kyouko3.pci_dev, 8192u, &kyouko3.fifo.p_base);
   K_WRITE_REGISTER(FifoStart, kyouko3.fifo.p_base);
   K_WRITE_REGISTER(FifoEnd, kyouko3.fifo.p_base + 8192u);
   kyouko3.fifo.head = 0;
   kyouko3.fifo.tail_cache = 0;
```

```
   kyouko3.graphics_on = 0;
   kyouko3.dma_mapped = 0;

   printk(KERN_ALERT "The device has been opened.\n");
   return 0;
}

int kyouko3_release(struct inode *indoe, struct file *fp) {
   if(kyouko3.dma_mapped) late_unbind_dma();
   if(kyouko3.graphics_on) graphics_off();

   pci_free_consistent(
      kyouko3.pci_dev,
      8192u,
      kyouko3.fifo.k_base,
      kyouko3.fifo.p_base);
   iounmap(kyouko3.k_control_base);
   iounmap(kyouko3.k_ram_card_base);

   printk(KERN_ALERT "BUUH BYE.\n");
   return 0;
}

long kyouko3_ioctl(struct file *fp, unsigned int cmd, unsigned long arg) {
   long ret = 0;
   unsigned long next_dma_buffer;
   int byte_count;

   switch(cmd) {
      case FIFO_QUEUE:
         ret = fifo_queue(cmd, arg);
         break;
      case FIFO_FLUSH:
         fifo_flush();
         break;
      case VMODE:
         if( ((int)arg) == GRAPHICS_OFF) graphics_off();
         else graphics_on();
         break;
      case BIND_DMA:
         next_dma_buffer = bind_dma(fp);
         if(next_dma_buffer == 0) ret = 1;
         else ret =
            copy_to_user(
               (unsigned long*)arg,
               &next_dma_buffer,
               sizeof(unsigned long));
         break;
      case UNBIND_DMA:
         ret = on_time_unbind_dma();
         break;
      case START_DMA:
         ret = copy_from_user(&byte_count, (int*)arg, sizeof(int));
         next_dma_buffer = start_dma(byte_count);
         if(next_dma_buffer == 0) ret = 1;
         else ret +=
            copy_to_user(
               (unsigned long*)arg,
               &next_dma_buffer,
               sizeof(unsigned long));
         break;
   }
   return ret;
}
```

```
int kyouko3_mmap(struct file *fp, struct vm_area_struct *vma) {
   int ret;
   int offset;

   if(current->cred->uid.val != 0) {
      printk(KERN_ALERT "Non-root user attempted to call mmap!\n");
      return 0;
   }

   offset = vma->vm_pgoff << PAGE_SHIFT;
   switch(offset) {
      case 0:
         ret = io_remap_pfn_range(
            vma,
            vma->vm_start,
            kyouko3.p_control_base >> PAGE_SHIFT,
            vma->vm_end - vma->vm_start,
            vma->vm_page_prot);
         break;
      case 0x80000000:
         ret = io_remap_pfn_range(
            vma,
            vma->vm_start,
            kyouko3.p_ram_card_base >> PAGE_SHIFT,
            vma->vm_end - vma->vm_start,
            vma->vm_page_prot);
         break;
      default:
         ret = io_remap_pfn_range(
            vma,
            vma->vm_start,
            kyouko3.dma_buffers[(offset >> PAGE_SHIFT)-1].p_base >> PAGE_SHIFT,
            vma->vm_end - vma->vm_start,
            vma->vm_page_prot);
         break;
   }
   return ret;
}

struct file_operations kyouko3_fops = {
   .open = kyouko3_open,
   .release = kyouko3_release,
   .unlocked_ioctl = kyouko3_ioctl,
   .mmap = kyouko3_mmap,
   .owner = THIS_MODULE
};
// ~~

// ~~ Prepare struct pci_driver
int kyouko3_probe(struct pci_dev *pci_dev, const struct pci_device_id *pci_id) {
   int ret;

   kyouko3.p_control_base = pci_resource_start(pci_dev, 1);
   kyouko3.p_ram_card_base = pci_resource_start(pci_dev, 2);

   ret = pci_enable_device(pci_dev);
   pci_set_master(pci_dev);

   kyouko3.pci_dev = pci_dev;

   return ret;
}
```

```
void kyouko3_remove(struct pci_dev *pci_dev) {
   pci_disable_device(pci_dev);
}

struct pci_device_id kyouko3_dev_ids[] = {
   { PCI_DEVICE(PCI_VENDOR_ID_CCORSI, PCI_DEVICE_ID_CCORSI_KYOUKO3) },
   { 0 }
};

struct pci_driver kyouko3_pci_drv = {
   .name = "whatever",
   .id_table = kyouko3_dev_ids,
   .probe = kyouko3_probe,
   .remove = kyouko3_remove
};
// ~~

// ~~ module_init and module_exit
int kyouko3_init(void) {
   int ret;

   cdev_init(&kyouko3.cdev, &kyouko3_fops);
   kyouko3.cdev.owner = THIS_MODULE;
   cdev_add(&kyouko3.cdev, MKDEV(KYOUKO3_MAJOR, KYOUKO3_MINOR), 1);
   ret = pci_register_driver(&kyouko3_pci_drv);

   printk(KERN_ALERT "The device has been initialized.\n");

   return ret;
}

void kyouko3_exit(void) {
   pci_unregister_driver(&kyouko3_pci_drv);
   cdev_del(&kyouko3.cdev);
   printk(KERN_ALERT "The device has been exitted.\n");
}

module_init(kyouko3_init);
module_exit(kyouko3_exit);
// ~~
// END OF DEVICE DRIVER CODE
```

```
// START OF SHARED HEADER
// Description: Shared header for the Kyouko3 PCIe graphics card driver.

#include <linux/ioctl.h>

// ~~ Only Kernel/User Shared Definitions Appear Here
// ioctl commands
#define VMODE      _IOW (0xCC, 0, unsigned long)
#define BIND_DMA   _IOW (0xCC, 1, unsigned long)
#define START_DMA  _IOWR(0xCC, 2, unsigned long)
#define FIFO_QUEUE _IOWR(0xCC, 3, unsigned long)
#define FIFO_FLUSH _IO  (0xCC, 4)
#define UNBIND_DMA _IOW (0xCC, 5, unsigned long)

// graphics modes
#define GRAPHICS_OFF 0
#define GRAPHICS_ON 1

// fifo commands
#define VertexColor 0x5010
#define VertexCoordinate 0x5000
#define CommandPrimitive 0x3000
#define VertexEmit 0x3004
#define RasterFlush 0x3FFC
// ~~
// END OF SHARED HEADER
```

```
// START OF USER CODE
// Description:
//    User code for issuing commands to the Kyouko3 PCIe graphics card
//    in order to draw triangles.

// Usage:
//    To draw a triangle with commands issued through the FIFO,
//    run the program with a command-line argument of 0 (or no argument).
//
//    To draw 50,000 triangles with commands issued through DMA buffers,
//    run the program with a command-line argument of 1. The commands are
//    separated across 50 DMA buffer requests, each buffer holding the commands
//    to draw 1,000 triangles.

#include <fcntl.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include "mymod.h"

#define U_NUM_TRIANGLES 1000 // # of triangle written to each DMA buffer
#define U_NUM_BUFFS 50       // # of DMA buffers written to

void die_with_message(char *message) {
   fprintf(stderr, "%s\n", message);
   exit(1);
}

void u_fifo_queue(int fd, unsigned int command, unsigned int value) {
   unsigned int fifo_entry[] = { command, value };
   if(ioctl(fd, FIFO_QUEUE, &fifo_entry))
      die_with_message("FIFO_QUEUE call to ioctl failed.");
}

void fifo_commands(int fd) {
   int i, j;
   static float coordinates[3][4] = {
      -0.7f, -0.7f,  0.0f,  1.0f, // upper-left
       0.7f, -0.7f,  0.0f,  1.0f, // upper-right
       0.7f,  0.7f,  0.0f,  1.0f  // lower-right
   };
   static float colors[3][4] = {
       0.0f,  0.0f,  1.0f,  0.0f, // red
       0.0f,  1.0f,  0.0f,  0.0f, // green
       1.0f,  0.0f,  0.0f,  0.0f  // blue
   };

   // Fill the queue with vertex information
   u_fifo_queue(fd, CommandPrimitive, 1);
   for(i = 0; i < 3; ++i) {
      for(j = 0; j < 4; ++j)
         u_fifo_queue(fd, VertexColor + j * 4, *(unsigned int*)&colors[i][j]);

      for(j = 0; j < 4; ++j)
         u_fifo_queue(
            fd, VertexCoordinate + j * 4, *(unsigned int*)&coordinates[i][j]);
      u_fifo_queue(fd, VertexEmit, 0);
   }
   u_fifo_queue(fd, CommandPrimitive, 0);

   u_fifo_queue(fd, RasterFlush, 0);
```

```
   ioctl(fd, FIFO_FLUSH, 0);
}

void generate_coordinates(float *coords) {
   int i;
   for(i = 0; i < 3; ++i) {
      coords[i*3 + 0] = 2.0f * drand48() - 1.0f;
      coords[i*3 + 1] = 2.0f * drand48() - 1.0f;
      coords[i*3 + 2] = 0.0f;
   }
}

void generate_colors(float *colors) {
   int i;
   for(i = 0; i < 3; ++i) {
      colors[i*3 + 0] = drand48();
      colors[i*3 + 1] = drand48();
      colors[i*3 + 2] = drand48();
   }
}

struct u_dma_header {
   uint32_t address: 14;
   uint32_t count: 10;
   uint32_t opcode: 8;
};

void dma_commands(int fd) {
   unsigned int i, j, k, *u_buff_base, *u_buff_curr;
   unsigned long byte_count;
   float colors[9], coords[9];
   struct u_dma_header dma_header = {
      .address = 0x1045,
      .count = 0,
      .opcode = 0x14
   };

   srand(time(NULL));
   if(ioctl(fd, BIND_DMA, &u_buff_base))
      die_with_message("BIND_DMA call to ioctl failed.");

   for(i = 0; i < U_NUM_BUFFS; ++i) {
      u_buff_curr = u_buff_base + 1;
      dma_header.count = byte_count = 0;

      for(j = 0; j < U_NUM_TRIANGLES; ++j) {
         generate_colors(colors);
         generate_coordinates(coords);

         for(k = 0; k < 3; ++k) {
            *(u_buff_curr++) = *(unsigned int*)&colors[k*3 + 0];
            *(u_buff_curr++) = *(unsigned int*)&colors[k*3 + 1];
            *(u_buff_curr++) = *(unsigned int*)&colors[k*3 + 2];
         }


         for(k = 0; k < 3; ++k) {
            *(u_buff_curr++) = *(unsigned int*)&coords[k*3 + 0];
            *(u_buff_curr++) = *(unsigned int*)&coords[k*3 + 1];
            *(u_buff_curr++) = *(unsigned int*)&coords[k*3 + 2];
         }

         dma_header.count += 3;
         byte_count += 72;
```

```
      }

      *u_buff_base = *(unsigned int *)&dma_header; // Prepend the header.
      u_buff_base = (unsigned int*)byte_count;     // Pull double-duty
                                                   // as the buffer byte size.

      u_fifo_queue(fd, RasterFlush, 0);
      ioctl(fd, FIFO_FLUSH, 0);
      if(ioctl(fd, START_DMA, &u_buff_base))
         die_with_message("START_DMA call to ioctl failed.");
   }
   u_fifo_queue(fd, RasterFlush, 0);
   ioctl(fd, FIFO_FLUSH, 0);
   ioctl(fd, UNBIND_DMA, 0);
}

int main(int argc, char **argv) {
   int fd, mode = 0;

   // Parse potential command-line argument.
   if(argc > 2) die_with_message("Invalid number of command-line arguments.");
   if(argc == 2) mode = atoi(argv[1]);

   // Open the character device.
   fd = open("/dev/kyouko3", O_RDWR);
   if(fd < 0) die_with_message("Character device was unable to be opened.");

   // Run graphics commands.
   ioctl(fd, VMODE, GRAPHICS_ON);
   if(mode == 0) fifo_commands(fd);
   else          dma_commands(fd);
   sleep(3);
   ioctl(fd, VMODE, GRAPHICS_OFF);

   close(fd);
}
// END OF USER CODE
```