```c
/*
* switch_uio.h
*
* ECEn 427
* Clint Frandsen, Dax Eckles
* BYU 2019
*/

#include <stdint.h>


/********************************** macros **********************************/
#define SWITCH_UIO_ERROR      -1      //error return value
#define SWITCH_UIO_SUCCESS     0      //success return value
#define SWITCH_UIO_MMAP_OFFSET     0     // offset of the mmap
#define SWITCH_UIO_CHANNEL_ONE_MASK 0x1 /* enables Channel 1 interrupts from selected device */
#define SWITCH_UIO_GPIO_DATA_OFFSET 0x0 /* the data from channel one received from the inerrupts */
#define SWITCH_UIO_GPIO_FILE_PATH "/dev/uio2"

/*************************** function prototypes ****************************/
// initializes the uio driver
// devDevice: The file path to the uio dev file
// Returns: A negative error code on error, success code otherwise
int32_t switch_uio_init(char devDevice[]);

// write to a register of the UIO device
// offset : the offset of the register that we are writing to
// value : the value that we want to write to the register
void switch_uio_write(uint32_t offset, uint32_t value);

// Acknowledge interrupt(s) in the interrupt controller
// value: Bitmask of interrupt lines to acknowledge.
void switch_uio_acknowledge(uint32_t value);

// read from a register of the UIO device
// offset : the offset of the register we wich to read from
// returns : the value contained within the register we are reading from
uint32_t switch_uio_read(uint32_t offset);

// Called to exit the driver (unmap and close UIO file)
void switch_uio_exit();
```

```c
/*
* Switch Driver
*
* ECEn 427
* Clint Frandsen, Dax Eckles
* BYU 2019
*/

#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include "switch_uio.h"

/********************************** macros **********************************/
#define SWITCH_UIO_MMAP_SIZE 0x1000 /* size of memory to allocate */
#define SWITCH_UIO_GIER_REG_OFFSET 0x11C /* global interrupt register offset */
#define SWITCH_UIO_GIER_MASK (1<<31) /* top register bit (31) is set to one */
#define SWITCH_UIO_IP_IER_REG_OFFSET 0x128 /* IP IER offset value */
#define SWITCH_UIO_IP_ISR_REG_OFFSET 0x120 /* IP ISR register offset */

/********************************** globals **********************************/
static int fd; /* this is a file descriptor that describes the UIO device */
static char *va; /* virtual address of the button registers */


/*************************** functions ***************************/
// initializes the uio driver
// devDevice: The file path to the uio dev file
// Returns: A negative error code on error, success code otherwise
int32_t switch_uio_init(char devDevice[]) {
    /* open the device */
    fd = open(devDevice, O_RDWR);
    /* if there is a problem, return an error */
    if(fd == SWITCH_UIO_ERROR) {
        return SWITCH_UIO_ERROR;
    }

    va = mmap(NULL, SWITCH_UIO_MMAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, SWITCH_UIO_MMAP_OFFSET);
    /* if there is a problem, return an error */
    if(va == MAP_FAILED) {
        return SWITCH_UIO_ERROR;
    }

    /* put hardware setup here */
    /* enable channel interrupt in the IP IER */
    switch_uio_write(SWITCH_UIO_IP_IER_REG_OFFSET, SWITCH_UIO_CHANNEL_ONE_MASK);
    /* enable global interrupt by setting bit 31 of the GIR */
    switch_uio_write(SWITCH_UIO_GIER_REG_OFFSET, SWITCH_UIO_GIER_MASK);

    return SWITCH_UIO_SUCCESS;
}

// write to a register of the UIO device
// offset : the offset of the register that we are writing to
// value : the value that we want to write to the register
void switch_uio_write(uint32_t offset, uint32_t value) {
    //the address is cast as a pointer so it can be dereferenced
    *((volatile uint32_t *)(va + offset)) = value;
}

// Acknowledge interrupt(s) in the interrupt controller
// value: Bitmask of interrupt lines to acknowledge.
void switch_uio_acknowledge(uint32_t value) {
    *((volatile uint32_t *)(va + SWITCH_UIO_IP_ISR_REG_OFFSET)) = value;
}

// read from a register of the UIO device
// offset : the offset of the register we wich to read from
// returns : the value contained within the register we are reading from
uint32_t switch_uio_read(uint32_t offset) {
    return *((volatile uint32_t *)(va + offset));
}

// Called to exit the driver (unmap and close UIO file)
void switch_uio_exit() {
    munmap(va, SWITCH_UIO_MMAP_SIZE);
    close(fd);
}
```

```c
/*
* button_uio.h
*
* ECEn 427
* Clint Frandsen, Dax Eckles
* BYU 2019
*/

#include <stdint.h>


/********************************** macros **********************************/
#define BUTTON_UIO_ERROR      -1      //error return value
#define BUTTON_UIO_SUCCESS     0      //success return value
#define BUTTON_UIO_MMAP_OFFSET     0     // offset of the mmap
#define BUTTON_UIO_CHANNEL_ONE_MASK 0x1 /* enables Channel 1 interrupts from selected device */
#define BUTTON_UIO_GPIO_DATA_OFFSET 0x0 /* the data from channel one received from the inerrupts */
#define BUTTON_UIO_GPIO_FILE_PATH "/dev/uio1"

/*************************** function prototypes ****************************/
// initializes the uio driver
// devDevice: The file path to the uio dev file
// Returns: A negative error code on error, success code otherwise
int32_t button_uio_init(char devDevice[]);

// write to a register of the UIO device
// offset : the offset of the register that we are writing to
// value : the value that we want to write to the register
void button_uio_write(uint32_t offset, uint32_t value);

// Acknowledge interrupt(s) in the interrupt controller
// value: Bitmask of interrupt lines to acknowledge.
void button_uio_acknowledge(uint32_t value);

// read from a register of the UIO device
// offset : the offset of the register we wich to read from
// returns : the value contained within the register we are reading from
uint32_t button_uio_read(uint32_t offset);

// Called to exit the driver (unmap and close UIO file)
void button_uio_exit();
```

```c
/*
* Button Driver
*
* ECEn 427
* Clint Frandsen, Dax Eckles
* BYU 2019
*/

#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include "button_uio.h"

/********************************** macros **********************************/
#define BUTTON_UIO_MMAP_SIZE 0x1000 /* size of memory to allocate */
#define BUTTON_UIO_GIER_REG_OFFSET 0x11C /* global interrupt register offset */
#define BUTTON_UIO_GIER_MASK (1<<31) /* top register bit (31) is set to one */
#define BUTTON_UIO_IP_IER_REG_OFFSET 0x128 /* IP IER offset value */
#define BUTTON_UIO_IP_ISR_REG_OFFSET 0x120 /* IP ISR register offset */

/********************************** globals **********************************/
static int fd; /* this is a file descriptor that describes the UIO device */
static char *va; /* virtual address of the button registers */


/*************************** functions ***************************/
/* initializes the uio driver */
int32_t button_uio_init(char devDevice[]) {
    /* open the device */
    fd = open(devDevice, O_RDWR);
    /* if there is a problem, return an error */
    if(fd == BUTTON_UIO_ERROR) {
        return BUTTON_UIO_ERROR;
    }

    va = mmap(NULL, BUTTON_UIO_MMAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, BUTTON_UIO_MMAP_OFFSET);
    /* if there is a problem, return an error */
    if(va == MAP_FAILED) {
        return BUTTON_UIO_ERROR;
    }

    /* put hardware setup here */
    /* enable channel interrupt in the IP IER */
    button_uio_write(BUTTON_UIO_IP_IER_REG_OFFSET, BUTTON_UIO_CHANNEL_ONE_MASK);
    /* enable global interrupt by setting bit 31 of the GIR */
    button_uio_write(BUTTON_UIO_GIER_REG_OFFSET, BUTTON_UIO_GIER_MASK);

    return BUTTON_UIO_SUCCESS;
}

// write to a register of the UIO device
// offset : the offset of the register that we are writing to
// value : the value that we want to write to the register
void button_uio_write(uint32_t offset, uint32_t value) {
    //the address is cast as a pointer so it can be dereferenced
    *((volatile uint32_t *)(va + offset)) = value;
}

// Acknowledge interrupt(s) in the interrupt controller
// value: Bitmask of interrupt lines to acknowledge.
void button_uio_acknowledge(uint32_t value) {
    *((volatile uint32_t *)(va + BUTTON_UIO_IP_ISR_REG_OFFSET)) = value;
}

// read from a register of the UIO device
// offset : the offset of the register we wich to read from
// returns : the value contained within the register we are reading from
uint32_t button_uio_read(uint32_t offset) {
    return *((volatile uint32_t *)(va + offset));
}

// Called to exit the driver (unmap and close UIO file)
void button_uio_exit() {
    munmap(va, BUTTON_UIO_MMAP_SIZE);
    close(fd);
}
```

```c
#include <stdint.h>
#include "intcFolder/intc.h"
#include <stdio.h>
#include <stdlib.h>
#include "uioFolder/button_uio.h"
#include "uioFolder/switch_uio.h"

/********************************** macros **********************************/
#define BTN_0_MASK 0x1 /* mask for button 0 */
#define BTN_1_MASK 0x2 /* mask for button 1 */
#define BTN_2_MASK 0x4 /* mask for button 2 */
#define SWTCH_0_MASK 0x1 /* mask for switch 0 */
#define SWTCH_0_MASK 0x1 /* first switch mask */
#define SECONDS_MAX 59 /* max number the seconds value can be */
#define MINUTES_MAX 59 /* max number the minutes value can be */
#define HOURS_MAX 23 /* max number the hours value can be */
#define TIME_MIN 0 /* smallest number the time values can be*/
#define SWITCH_FLAG_UP 1 /* value of the flag when it is up*/
#define SWITCH_FLAG_DOWN 0 /* value of the flag when it is down*/
#define TICKS_PER_SECOND 100 /* number of FIT interrupts taken to reach one second */
#define BOUNCE 100000 /* value of the ticks to seconds */
#define FIVE_SECONDS 1000000 /* value of the ticks to seconds */
#define DELAY_TIME 250000 /* value of the ticks to seconds for delay */

/********************************** globals **********************************/
int16_t switch_flag = SWITCH_FLAG_UP; /* this will track if the switch is flipped on or off */
int32_t seconds = TIME_MIN; /* this will track the seconds for the clock */
int32_t minutes = TIME_MIN; /* this will track the minutes for the clock */
int32_t hours = TIME_MIN; /* this will track the hours for the clock */

int32_t second_old = SECONDS_MAX; /* this will track the seconds for the clock */
int32_t minute_old = MINUTES_MAX; /* this will track the minutes for the clock */
int32_t hour_old = HOURS_MAX; /* this will track the hours for the clock */

int32_t counter = TIME_MIN; /*counter for the interrupts to increment seconds */
int32_t button_counter = TIME_MIN; /*counter for debounceing the buttons*/

int32_t multiple_buttons = 0; /* flag that states whether or not multiple buttons have been pressed */

/********************************** functions **********************************/

/* This function prints the time to the screen with every time that the time
* needs to be updated. This includes every second as the timer goes up and
* whenever a button is pressed to adjust the clock
*/
void print_time() {
  // clear screen and print new values if the time has be updated
  if(hour_old!=hours || minute_old!=minutes || second_old!=seconds){
    // Clear the screen from the previous time
    system("clear");
    printf("Time: %02zu:%02zu:%02zu\n", hours, minutes, seconds);
    // save new values into old time values
    hour_old = hours;
    minute_old = minutes;
    second_old = seconds;
  }
}

/* updates the time based on the switch possition.
* int32_t time_max : maximum amount of time (i.e. 59 secs, 59 minutes, 23 hours)
* int32_t *unit : pointer that says which unit of time must be updated
* int16_t switch_control : says whether to increment or decrement timer
*/
void update_time(int32_t time_max, int32_t *unit, int16_t switch_control) {
  int32_t temp = *unit;
  // if switch is up; increase time
  if(switch_control){
    if(temp >= time_max){ // if we have reached max time, set time back to 0
      temp = TIME_MIN;
    }
    else{ // if max time is not reached, go up one
      temp++;
    }
  }
  // if switch is down; decrease time
  else{
    if(temp <= TIME_MIN){ // if time is at 0, then reset it to the maximum time
      temp = time_max;
    }
```

```
80:     else{ // if we have not arrived at o, decrement time
81:       temp--;
82:     }
83:   }
84:   *unit = temp;
85:   print_time(); // print the time at the end to update it completely on screen
86: }
87:
88: /* a switch statement that is used to tell how the program how the time
89: * should be updated.
90: * uint32_t buttonPressed : tells us which of the 3 buttons were pressed
91: */
92: void increment_time(uint32_t buttonPressed){
93:   switch(buttonPressed) { // takes care of updating time depending on the btn pressed
94:     case BTN_0_MASK:
95:       // button 0 pressed
96:       update_time(SECONDS_MAX, &seconds, switch_flag);
97:       break;
98:     case BTN_1_MASK:
99:       // button 1 pressed
100:      update_time(MINUTES_MAX, &minutes, switch_flag);
101:      break;
102:    case BTN_2_MASK:
103:      // button 2 pressed
104:      update_time(HOURS_MAX, &hours, switch_flag);
105:      break;
106:    default:
107:      // multiple buttons or button 4 was pressed
108:      multiple_buttons = 1;
109:      break;
110:  }
111: }
112:
113: // This is invoked in response to a timer interrupt.
114: // It does 2 things: 1) debounce switches, and 2) advances the time.
115: void isr_fit() {
116:   intc_ack_interrupt(INTC_FIT_MASK); // acknowledges the received FIT interrupt
117:   // This will count up to 100 ticks before updating the time to one second
118:   if(counter >= TICKS_PER_SECOND){
119:     int32_t temp_seconds = seconds;
120:     int32_t temp_minutes = minutes;
121:
122:     update_time(SECONDS_MAX, &seconds, SWITCH_FLAG_UP);
123:     // If the maximum amount of seconds has been reached, then we count up 1 min
124:     if(temp_seconds>=SECONDS_MAX){
125:       update_time(MINUTES_MAX, &minutes, SWITCH_FLAG_UP);
126:     }
127:     // If the max number of mins is reached, we count up 1 hr
128:     if(temp_seconds>=SECONDS_MAX && temp_minutes >= MINUTES_MAX){
129:       update_time(HOURS_MAX, &hours, SWITCH_FLAG_UP);
130:     }
131:     // reset the tick counter so we can increment back to the ticks per second
132:     counter = TIME_MIN;
133:   }
134:   else{
135:     // if we haven't yet reached the ticks per sec parameter, increment the
136:     // counter to keep counting until the number is reached
137:     counter++;
138:   }
139: }
140:
141: // This function is used to read, react, and acknowledge interrupts received from the switches
142: void isr_switches(){
143:   uint32_t switchState = switch_uio_read(SWITCH_UIO_GPIO_DATA_OFFSET); // read data from switches
144:   // if the switch is in the on position, set the switch flag to on (increment)
145:   if(switchState & SWTCH_0_MASK) {
146:     switch_flag = SWITCH_FLAG_UP;
147:   }
148:   else if(~(switchState & SWTCH_0_MASK)) {
149:     // if the switch is off, set the flag to off (decrement)
150:     switch_flag = SWITCH_FLAG_DOWN;
151:   }
152:   switch_uio_acknowledge(SWITCH_UIO_CHANNEL_ONE_MASK); /* acknowledges an interrupt from the GPIO */
153:   intc_ack_interrupt(INTC_SWITCHES_MASK); /* acknowledges an interrupt from the interrupt controller */
154: }
155:
156: // This is invoked each time there is a change in the button state (result of a push or a bounce).
157: void isr_buttons() {
158:   uint32_t buttonPressed = button_uio_read(BUTTON_UIO_GPIO_DATA_OFFSET); // reads data from buttons
```

```
159:   // debounces the buttons
160:   while(button_counter < BOUNCE) {
161:       button_counter++;
162:   }
163:
164:   // read the button again to check to see if the button is still pressed
165:   if(buttonPressed == button_uio_read(BUTTON_UIO_GPIO_DATA_OFFSET)){
166:     isr_switches(); // check state of the switches
167:     increment_time(buttonPressed); // update time accordingly
168:   }
169:   uint32_t auto_counter = 0; // auto_counter counts how long we should delay before auto increment begins
170:   uint32_t delay_counter = 0; // delay_counter allows for time until we need to count seconds
171:   // while we are holding down the button, do auto increment
172:   while (buttonPressed == button_uio_read(BUTTON_UIO_GPIO_DATA_OFFSET) && buttonPressed != 0){
173:       auto_counter++; // increment auto_counter
174:       if(auto_counter > FIVE_SECONDS) { // auto_counter must reach five seconds before we begin incrementing
175:        delay_counter++; // increment delay_counter
176:        if(delay_counter > DELAY_TIME) { // delay_counter must reach threshold before changing time
177:          isr_switches(); // check status of switches
178:          increment_time(buttonPressed); // update time accordingly
179:          delay_counter = 0; // reset the delay_counter
180:        }
181:       }
182:       // if we are pressing multiple buttons at the same time, do nothing
183:       if(multiple_buttons){
184:         break;
185:       }
186:   }
187:   // reset all counters
188:   auto_counter = 0;
189:   multiple_buttons = 0;
190:   button_counter = 0;
191:   button_uio_acknowledge(BUTTON_UIO_CHANNEL_ONE_MASK); /* acknowledges an interrupt from the GPIO */
192:   intc_ack_interrupt(INTC_BTNS_MASK); /* acknowledges an interrupt from the interrupt controller */
193: }
194:
195: // intitializes the drivers, runs the clock, and passes interrupts to their
196: // appropriate handlers
197: int main() {
198:   // Initialize interrupt controller driver
199:   intc_init(INTC_GPIO_FILE_PATH);
200:   // Initialize buttons
201:   button_uio_init(BUTTON_UIO_GPIO_FILE_PATH);
202:   // Initialize switches
203:   switch_uio_init(SWITCH_UIO_GPIO_FILE_PATH);
204:   // Enable button and FIT interrupt lines on interrupt controller
205:
206:   // Main body of the code, runs in a loop forever to keep time
207:   while(1) {
208:     /* need to run this each time that we block, because this function will unblock */
209:     intc_enable_uio_interrupts(); /* enables Linux interrupts */
210:     // Call interrupt controller function to wait for interrupt
211:     uint32_t interrupts = intc_wait_for_interrupt();
212:
213:     // printf("%zu  \r\n", interrupts);
214:
215:     // Check which interrupt lines are high and call the appropriate ISR functions
216:     if(interrupts & INTC_FIT_MASK) {
217:       isr_fit();
218:     }
219:     if(interrupts & INTC_BTNS_MASK) {
220:       isr_buttons();
221:     }
222:     if(interrupts & INTC_SWITCHES_MASK) {
223:       isr_switches();
224:     }
225:     print_time(); // prints the time to the screen
226:   }
227: }
```

```c
/*
* intc.h
*
* ECEn 427
* Clint Frandsen, Dax Eckles
* BYU 2019
*/

#include <stdint.h>

/********************************** macros **********************************/
#define INTC_MMAP_OFFSET   0
#define INTC_SUCCESS  0
#define INTC_ERROR   -1      //error return value
#define INTC_GPIO_FILE_PATH "/dev/uio4"
#define INTC_FIT_MASK 0x1 /* interrupt handler's first bit: corresponds to fit */
#define INTC_BTNS_MASK 0x2 /* interrupt handler's second bit: corresponds to buttons */
#define INTC_SWITCHES_MASK 0x4 /* interrupt handler's third bit: corresponds to switches */

/*************************** function prototypes ****************************/
// Initializes the driver (opens UIO file and calls mmap)
// devDevice: The file path to the uio dev file
// Returns: A negative error code on error, INTC_SUCCESS otherwise
// This must be called before calling any other intc_* functions
int32_t intc_init(char devDevice[]);

// Called to exit the driver (unmap and close UIO file)
void intc_exit();

// This function will block until an interrupt occurrs
// Returns: Bitmask of activated interrupts
uint32_t intc_wait_for_interrupt();

// Acknowledge interrupt(s) in the interrupt controller
// irq_mask: Bitmask of interrupt lines to acknowledge.
void intc_ack_interrupt(uint32_t irq_mask);

// Instruct the UIO to enable interrupts for this device in Linux
// (see the UIO documentation for how to do this)
void intc_enable_uio_interrupts();

// Enables global interrupts in the GPIO
void intc_enable_global_interrupts();

// Enable interrupt line(s)
// irq_mask: Bitmask of lines to enable
// This function only enables interrupt lines, ie, a 0 bit in irq_mask
//  will not disable the interrupt line
void intc_irq_enable(uint32_t irq_mask);

// Same as intc_irq_enable, except this disables interrupt lines
void intc_irq_disable(uint32_t irq_mask);
```

```c
/*
* Interrupt Driver
* Initializes interrupts
*
* ECEn 427
* Clint Frandsen, Dax Eckles
* BYU 2019
*/

#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "intc.h"

/********************************** macros **********************************/
#define INTC_MMAP_SIZE 0x1000 /* size of memory to allocate */
#define FOUR_BYTES_SIZE 4 /* 32 bits to write to fd */
#define SIE_REG_OFFSET 0x10 /* sets the register bits in the IER */
#define CIE_REG_OFFSET 0x14 /* clears the resgiter bits in the IER */
#define IAR_REG_OFFSET 0xC /* acknowledges interrupts */
#define MER_REG_OFFSET 0x1C /* Master Enable Register */
#define ISR_REG_OFFSET 0x0 /* ISR offset */
#define GPIO_BITS 0x7 /* turns on all GPIO interrupts */
#define MER_BITS 0x3 /* need to turn on lower two bits to enable interrupts */


/********************************** globals **********************************/
static int fd; /* this is a file descriptor that describes the UIO device */
static char *va; /* virtual address of the interrupt handler registers */
static int enable = 1; /* enable code for interrupt */
static int32_t int_buffer = 0; /* buffer for the interrupt */

/*************************** functions ***************************/
// Initializes the driver (opens UIO file and calls mmap)
// devDevice: The file path to the uio dev file
// Returns: A negative error code on error, INTC_SUCCESS otherwise
// This must be called before calling any other intc_* functions
int32_t intc_init(char devDevice[]){
    /* open the device */
    fd = open(devDevice, O_RDWR);
    /* if there is a problem, return an error */
    if(fd == INTC_ERROR) {
        return INTC_ERROR;
    }

    /* map the virtual address to the appropriate location on the pynq */
    va = mmap(NULL, INTC_MMAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, INTC_MMAP_OFFSET);
    /* if there's a problem, return an error */
    if(va == MAP_FAILED) {
        return INTC_ERROR;
    }

    intc_irq_enable(GPIO_BITS); /* enables all the GPIO interrupts */
    /* turns on Master IRQ enable & Hardware interrupt enable */
    *((volatile uint32_t *)(va + MER_REG_OFFSET)) = MER_BITS;

    return INTC_SUCCESS;
}

// Called to exit the driver (unmap and close UIO file)
void intc_exit() {
    munmap(va, INTC_MMAP_SIZE);
    close(fd);
}

// This function will block until an interrupt occurrs
// Returns: Bitmask of activated interrupts
uint32_t intc_wait_for_interrupt() {
    read(fd, &int_buffer, FOUR_BYTES_SIZE); /* blocks interrupts */
    return *((volatile uint32_t *)(va + ISR_REG_OFFSET));
}

// Acknowledge interrupt(s) in the interrupt controller
// irq_mask: Bitmask of interrupt lines to acknowledge.
void intc_ack_interrupt(uint32_t irq_mask) {
    *((volatile uint32_t *)(va + IAR_REG_OFFSET)) = irq_mask;
}
```

```
80:
81: // Instruct the UIO to enable interrupts for this device in Linux
82: // (see the UIO documentation for how to do this)
83: void intc_enable_uio_interrupts() {
84:     write(fd, &enable, FOUR_BYTES_SIZE); /* enable linux interrupts from the GPIO */
85: }
86:
87: // Enable interrupt line(s)
88: // irq_mask: Bitmask of lines to enable
89: // This function only enables interrupt lines, ie, a 0 bit in irq_mask
90: //  will not disable the interrupt line
91: void intc_irq_enable(uint32_t irq_mask) {
92:     //the address is cast as a pointer so it can be dereferenced
93:     *((volatile uint32_t *)(va + SIE_REG_OFFSET)) = irq_mask;
94: }
95:
96: // Same as intc_irq_enable, except this disables interrupt lines
97: void intc_irq_disable(uint32_t irq_mask) {
98:     *((volatile uint32_t *)(va + CIE_REG_OFFSET)) = irq_mask;
99: }
```