# Handling general purpose i/o using linux kernel modules

## goal

The goal of this Lab is to build a loadable kernel module using the Yocto build environment. which provides access to one general purpose I/O (GPIO) as output and one GPIO as input, handled in interrupt mode.

## prerequisites

To follow this Lab, you need:

1. Raspberry Pi 3 board full;
2. Micro USB cable;
3. 8 GB Micro SD card;
4. USB-to-Serial debug module for Raspberry Pi 3 or USB to TTL adapter;
5. A PC provided with Ubuntu Desktop 14.04 LTS, or a virtual machine hosting Ubuntu Desktop 14.04 LTS;
6. A Micro SD card reader attached to the PC/virtual machine;
7. (**Optional**) Micro HDMI cable.

## Introduction

The Raspberry Pi 3 offers a number of I/O accessible through the connectors labeled J8 in the following picture:

|  |
| --- |
|  |

In this Lab, a loadable kernel module is implemented that programs GPIO21 as output and GPIO20 as input. Moreover, every time the value on the GPIO20 input changes, an interrupt is generated and handled to read the updated value.

The GPIO-based I/O is used as described in the lecture in module 6 and, in particular, GPIO21 and 20 will be used. This two-integer values identify univocally within Linux two hardware resources of the BCM2837 processor that powers the Raspberry Pi 3 board. The correspondence between hardware resource and Linux identifier is obtained using the following equation:

Linux ID = Bit Number

**To test the correct operations of the module, GPIO21 shall be wired to GPIO20.**

## workplace setup

Assuming you completed the previous lab, move to the directory ***raspberrypi3*** and prepare the build environment:

cd ~/raspberryPi3

source sources/poky/oe-init-build-env rpi-build

Download the following package:

sudo apt-get install gcc-arm-linux-gnueabihf

The workplace is now setup for the rest of the lab.

## Preparing the recipes

Similar to the files created during the previous lab, create and move to the directory ***raspberryPi3/sources/meta-raspberrypi/recipes-kernel/gpio-mod/files*** and make a file named ***gpio.c*** with the following content:

|  |  |
| --- | --- |
| 0:  1:  2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  35:  36:  37:  38:  39:  40:  41:  42:  43:  44:  45:  46:  47:  48:  49:  50:  51:  52:  53:  54:  55:  56:  57:  58:  59:  60:  61:  62:  63:  64:  65:  66:  67:  68:  69:  70:  71:  72:  73:  74:  75:  76:  77:  78:  79:  80:  81:  82:  83:  84:  85:  86:  87:  88:  89:  90:  91:  92:  93:  94:  95:  96:  97:  98:  99:  100:  101:  102:  103:  104:  105:  106:  107:  108:  109:  110:  111:  112:  113:  114:  115:  116:  117:  118:  119:  120:  121:  122:  123:  124:  125:  126:  127:  128:  129:  130:  131:  132:  133:  134:  135:  136:  137:  138:  139:  140:  141:  142:  143:  144:  145:  146:  147:  148:  149:  150:  151:  152:  153:  154:  155:  156:  157:  158:  159:  160:  161:  162:  163:  164:  165:  166:  167:  168:  169:  170:  171:  172:  173:  174:  175:  176: | #include <linux/module.h>  #include <linux/init.h>  #include <linux/kernel.h>  #include <linux/types.h>  #include <linux/kdev\_t.h>  #include <linux/fs.h>  #include <linux/cdev.h>  #include <asm/uaccess.h>  #include <linux/gpio.h>  #include <linux/interrupt.h>  #define GPIO\_OUT 20 // GPIO20  #define GPIO\_IN 21 // GPIO21  static dev\_t gpio\_dev;  struct cdev gpio\_cdev;  static int gpio\_lock = 0;  volatile char gpio\_in\_value = 0;  static irq\_handler\_t InterruptHandler( unsigned int irq, struct pt\_regs \*regs )  {  gpio\_in\_value = gpio\_get\_value( GPIO\_IN );  printk( KERN\_INFO "gpio\_dev: %s got GPIO\_IN with value %c\n",  \_\_func\_\_, gpio\_in\_value+'0' );  return (irq\_handler\_t) IRQ\_HANDLED;  }  int gpio\_open(struct inode \*inode, struct file \*file)  {  int ret = 0;  printk( KERN\_INFO "gpio\_dev: %s\n", \_\_func\_\_ );  if( gpio\_lock > 0 )  {  ret = -EBUSY;  }  else  gpio\_lock++;  return( ret );  }  int gpio\_close(struct inode \*inode, struct file \*file)  {  printk( KERN\_INFO "gpio\_dev: %s\n", \_\_func\_\_ );  gpio\_lock = 0;  return( 0 );  }  ssize\_t gpio\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*f\_pos)  {  char buffer[2];  printk(KERN\_INFO "gpio read (count=%d, offset=%d)\n", (int)count, (int)\*f\_pos );  buffer[0] = gpio\_in\_value;  buffer[1] = 0;  copy\_to\_user( buf, buffer, 1 );  return 1;  }  ssize\_t gpio\_write(struct file \*filp, const char \*buffer, size\_t length, loff\_t \* offset)  {  int n = 0;  while( length )  {  if( \*buffer == '0' )  {  gpio\_set\_value( GPIO\_OUT, 0 );  printk( KERN\_INFO "gpio\_dev: %s wrote %c to GPIO\_OUT\n", \_\_func\_\_, \*buffer );  }  if( \*buffer == '1' )  {  gpio\_set\_value( GPIO\_OUT, 1 );  printk( KERN\_INFO "gpio\_dev: %s wrote %c to GPIO\_OUT\n", \_\_func\_\_, \*buffer );  }  buffer++;  length--;  n++;  }  return( n );  }  struct file\_operations gpio\_fops = {  .owner = THIS\_MODULE,  .read = gpio\_read,  .write = gpio\_write,  .open = gpio\_open,  .release = gpio\_close,  };  static int \_\_init gpio\_module\_init(void)  {  char buffer[64];  int ret = 0;  printk(KERN\_INFO "Loading gpio\_module\n");  alloc\_chrdev\_region(&gpio\_dev, 0, 1, "gpio\_dev");  printk(KERN\_INFO "%s\n", format\_dev\_t(buffer, gpio\_dev));  cdev\_init(&gpio\_cdev, &gpio\_fops);  gpio\_cdev.owner = THIS\_MODULE;  cdev\_add(&gpio\_cdev, gpio\_dev, 1);  if( gpio\_request( GPIO\_OUT, "gpio\_dev" ) )  {  printk( KERN\_INFO "gpio\_dev: %s unable to get GPIO\_OUT\n", \_\_func\_\_ );  ret = -EBUSY;  goto Done;  }  if( gpio\_request( GPIO\_IN, "gpio\_dev" ) )  {  printk( KERN\_INFO "gpio\_dev: %s unable to get GPIO\_IN\n", \_\_func\_\_ );  ret = -EBUSY;  goto Done;  }  if( gpio\_direction\_output( GPIO\_OUT, 0 ) < 0 )  {  printk( KERN\_INFO "gpio\_dev: %s unable to set GPIO\_OUT as output\n", \_\_func\_\_ );  ret = -EBUSY;  goto Done;  }  if( gpio\_direction\_input( GPIO\_IN ) < 0 )  {  printk( KERN\_INFO "gpio\_dev: %s unable to set GPIO\_IN as input\n", \_\_func\_\_ );  ret = -EBUSY;  goto Done;  }  if( request\_irq( gpio\_to\_irq( GPIO\_IN ), (irq\_handler\_t) InterruptHandler,  IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING, "gpio\_dev", NULL ) < 0 )  {  printk( KERN\_INFO "gpio\_dev: %s unable to register gpio irq for GPIO\_IN\n",  \_\_func\_\_ );  ret = -EBUSY;  goto Done;  }  Done:  return ret;  }  static void \_\_exit gpio\_module\_cleanup(void)  {  printk(KERN\_INFO "Cleaning-up gpio\_dev.\n");  gpio\_free( GPIO\_OUT );  gpio\_free( GPIO\_IN );  free\_irq( gpio\_to\_irq( GPIO\_IN ), NULL );  gpio\_lock = 0;  cdev\_del(&gpio\_cdev);  unregister\_chrdev\_region(gpio\_dev, 1);  }  module\_init(gpio\_module\_init);  module\_exit(gpio\_module\_cleanup);  MODULE\_AUTHOR("Your name");  MODULE\_LICENSE("GPL"); |

The file contains the source code of the kernel module;

* Lines 0-12 include the needed header files containing the function prototypes and the data structure needed to write a kernel module;
* Lines 17-19 define the data structures to register the module as character device;
* Line 21 defines a flag used to manage the access to the device driver in mutual exclusion: only one user application is allowed to open the device file at a time;
* Line 23 defines the data structure that contains the value read from the GPIO configured as input. As this data structure is written by an interrupt service routine, it shall be defined as volatile;
* Lines 25-33 implement the interrupt service routine to read the value from the GPIO configured as input;
* Lines 36-177 implement the virtual file system interface for accessing the two GPIOs used as input/output.

Also in the ***files*** directory, create a file called "***Makefile***". Enter the following code (As with the previous labs, ensure that the lines are indented using tabs and not spaces - else it will not work):

0: obj-m := gpio.o  
1:  
2: SRC := $(shell pwd)  
3:  
4: all:  
**5**: $(MAKE) -C $(KERNEL\_SRC) M=$(SRC)  
**6**:  
**7**: modules\_install:  
**8**: $(MAKE) INSTALL\_MOD\_DIR=kernel/drivers/my\_mod -C $(KERNEL\_SRC) M=$(SRC) modules\_install  
**9**:  
**10**: clean:  
**11**: rm -f \*.o \*~core.depend .\*.cmd \*.ko \*.mod.c  
**12**: rm -f Module.markers Module.symvers modules.order  
**13**: rm -rf .tmp\_versions Modules.symvers

In ***gpio-mod***, create a file called ***gpio\_1.0.bb*** and add the following code:

0: DESCRIPTION = "gpio driver"  
1:  
2: LICENSE = "GPLv2+"  
3:  
4: LIC\_FILES\_CHKSUM = "file://${BPN}.c;endline=19;md5=02d0f4fb7e7b7125483125efd96a39dc"  
5:  
6: inherit module  
7:  
8: PR = "r0"  
9:   
10: SRC\_URI = "file://Makefile file://${BPN}.c"  
11:   
12: S = "${WORKDIR}"

Once these operations are completed, you have to tell the machine layer configuration that the new driver is needed. For this purpose, edit the file:

cd ~/raspberrypi3/rpi-build/conf/local.conf

Add the following statement as the last line of the file:

IMAGE\_INSTALL\_append += "gpio"

This line tells Yocto that when building Linux, the newly created device driver shall be built and that it shall be included in the root file system.

## Building and deploying the new system

You are now ready to build the new system as follows:

cd ~/raspberrypi3/rpi-build

bitbake -c clean rpi-basic-image

bitbake rpi-basic-image

During the build, the compiler will recognize a discrepancy in the license reference. Copy and paste the suggested reference (“the new md5 checksum is..”) into your gpio\_1.0.bb file and start the build again.

Once complete, a new Micro SD card image will be available, which you can deploy to the Micro SD as follows (assuming the Micro SD is available to the PC as /dev/sdN). Alternatively, use a program of your preference to flash the image.

First run the:

sudo fdisk -l

command to determine which device to flash to (plug in and unplug the SD card to determine which device it is). For this example, the SD card is under the name “sdc” (this may be different in your environment). Next, ensure that the device is unmounted. This can be done using the command:

sudo umount /dev/sdc\*

Once this is done, the following command can be used to copy the image across to the SD card (substitute any folder names and device names to ensure they are relevant to your specific environment).

sudo dd bs=1M if=/home/user/raspberryPi3/rpi-build/tmp-glibc/deploy/images/raspberrypi3/rpi-basic-image-raspberrypi3.rpi-sdimg of=/dev/sdc

Note that if not done properly, the image being flashed across to the SD card may cause problems when attempting to turn on the board. If this is the case, it may be worth retrying the process again and ensuring that it is done properly, or use a flash program to automate the process.

While the SD card is still connected to the development host, use the following lines to navigate to the etc folder on the SD card (assuming the device has now been mounted).

cd media/user/SD\_name/etc – Use the ls command to find the name of the SD card in the user folder.

Then, use a terminal text editor to open the shadow file.

sudo vi shadow

Or

sudo gedit shadow

Check that there are no characters between the first two colons in the first line. If there is, remove it so that the first line looks like this:

Root::17728:0:99999:7:::

Exit the text editor by entering :x or simply closing the application!

## Running the module

After booting the new Linux system, you can check whether the build process was completed successfully. After logging into the Raspberry Pi 3, you can type the following commands:

root@raspberrypi3:/# ls -la /lib/modules/4.1.21/kernel/drivers/my\_mod

drwxr-xr-x 2 root root 1024 May 15 08:47 .

drwxr-xr-x 43 root root 1024 May 10 12:09 ..

-rw-r--r-- 1 root root 9080 May 12 03:02 gpio.ko

-rw-r--r-- 1 root root 5384 May 11 08:05 hello.ko

root@raspberrypi3:/#

The directory in the root file system contains ***gpio.ko***, which is the kernel object containing the binary code for the loadable kernel module.

You can now insert the module in the kernel as follows:

root@raspberrypi3:/# insmod /lib/modules/4.1.21/kernel/drivers/my\_mod/gpio.ko

[ 1981.635413] Loading gpio\_module

[ 1981.638775] 243:0

The output messages indicate that the module has been loaded correctly, with major number 243 (This number may vary on your system - take it into account in the following steps) and minor number 0.

To test the module, you have to first create the associated device file:

root@raspberrypi3:/# mknod /dev/gpio c 243 0

You can now communicate with the module through the Linux command line.

As an example, you can issue the following command, observing the following output:

root@raspberrypi3:/# echo 1 > /dev/gpio

[ 2150.247267] gpio\_dev: gpio\_open

[ 2150.250630] gpio\_dev: InterruptHandler got GPIO\_IN with value 1

[ 2150.256641] gpio\_dev: gpio\_write wrote 1 to GPIO\_OUT

[ 2150.261938] gpio\_dev: gpio\_close

The messages displayed by the module indicate that the echo user application opens the connection with the module through the device file, then it sends the data by invoking the write function of the virtual file system, and finally, it closes the connection. If you wired output GPIO21 to input GPIO20, you will see that the interrupt is triggered and the InterruptHandler function is executed, reading the written value.

To test the read operation, we can write a simple test application called ***test\_gpio.c*** with the following code:

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main(int argc, char \*\*argv)

{

char \*app\_name = argv[0];

char \*dev\_name = "/dev/gpio";

int fd = -1;

char c;

if ((fd = open(dev\_name, O\_RDWR)) < 0)

{

fprintf(stderr, "%s: unable to open %s: %s\n", app\_name, dev\_name, strerror(errno));

return( 1 );

}

read( fd, &c, 1 );

printf( "read: %d\n", c );

close( fd );

return 0;

}

The code shall be built on the host machine, issuing the following command:

arm-linux-gnueabihf-gcc -o test\_gpio test\_gpio.c

The obtained program, ***test\_gpio***, shall be executed on the Raspberry Pi 3 target, for example, by copying it from the development host onto a USB stick that will be then plugged into the Raspberry Pi 3 USB port.

When connecting the USB to the Raspberry Pi, mount it to the media folder with the mount command:

mount /dev/sda1 media

If your USB is not called sda1, use the following command to find its name:

fdisk -l

The test program can be executed as follows (ensure /dev/gpio exists as done earlier):

root@raspberrypi3:~# ./media/test\_gpio > tmp

[ 947.552708] gpio\_dev: gpio\_open

[ 947.556149] gpio read (count=1, offset=0)

[ 947.562213] gpio\_dev: gpio\_close

The test application interacts with the driver through the virtual file system, invoking the open, read, and release functions.

The content of the tmp file contains the read value from the module. As an example:

root@raspberrypi3:~# echo 1 > /dev/gpio

[ 1068.239500] gpio\_dev: gpio\_open

[ 1068.243109] gpio\_dev: InterruptHandler got GPIO\_IN with value 1

[ 1068.250988] gpio\_dev: gpio\_write wrote 1 to GPIO\_OUT

[ 1068.256276] gpio\_dev: gpio\_close

root@raspberrypi3:~# ./media/test\_gpio > tmp

[ 1071.322513] gpio\_dev: gpio\_open

[ 1071.325956] gpio read (count=1, offset=0)

[ 1071.332148] gpio\_dev: gpio\_close

root@raspberrypi3:~# cat tmp

read: 1

We can see that the content of tmp corresponds to the value (1) sent to the module.

For removing the module, you can act as follows:

root@raspberrypi3:/# rmmod gpio

[ 240.457807] Cleaning-up gpio\_dev.

The message indicates that the module cleanup function has been executed correctly.

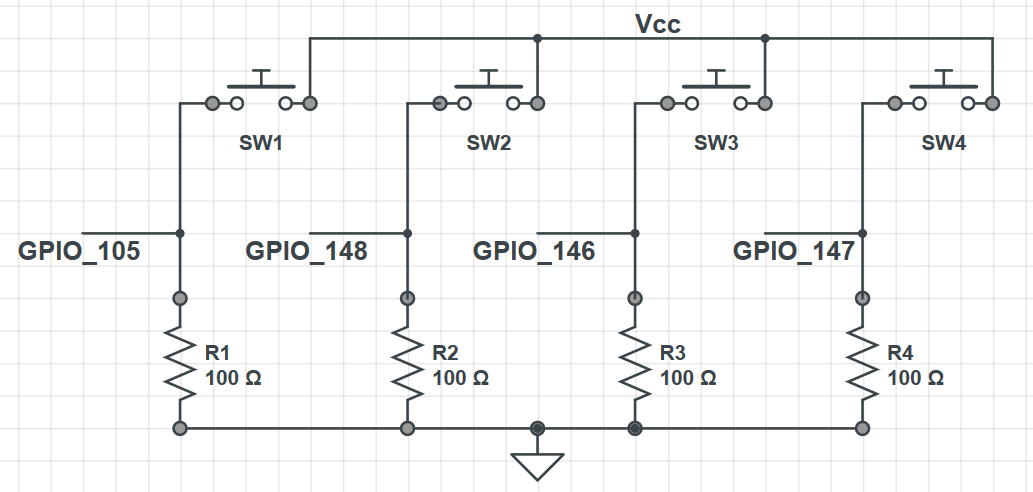
# post-lab practice

Now you have successfully loaded a game on your Raspberry Pi 3 board; however, to make it into a real embedded system, we need to enable its own I/O instead of an external keyboard to control the game. We have completed the current lab from which you learned how to write kernel module drivers to enable GPIOs; hence, this challenge session will focus on transferring the skills you just learned into real-world practice.

**Q: Build a hardware circuit of 4 GPIO push buttons and write a kernel module driver to enable them.**

**Hint: consider the linux/input.h interface. For the push buttons, also consider debouncing.**

**A:** This task assumes that you have completed the main lab, in which you enabled two GPIOs for reading and writing, respectively. Now we need to build the corresponding hardware circuit for push buttons. In this session, we set the state of a push button on idle is pulled-down (i.e., grounded). A resistor is needed in the series with respect to the push button to divide the potential. The schematic is as follows:



Note that the signal to GPIOs is determined by voltage, not current; hence, the value of resistors should limit the current below a safe level. Depending on your supply voltage (Vcc), you shall adjust the resistance accordingly.

For the time being, the GPIO push buttons are enabled but yet registered to the kernel. To do this, a kernel module is required as follows:

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/errno.h>

#include <linux/gpio.h>

#include <linux/kthread.h>

#include <linux/input.h>

#include <linux/delay.h>

#include <linux/init.h>#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/errno.h>

#include <linux/gpio.h>

#include <linux/kthread.h>

#include <linux/input.h>

#include <linux/delay.h>

#include <linux/init.h>

#include <linux/types.h>

#include <linux/kdev\_t.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <asm/uaccess.h>

#include <linux/interrupt.h>

#include <linux/time.h>

#define GPIO\_IN\_0 105 // GPIO\_105

#define GPIO\_IN\_1 148 // GPIO\_148

#define GPIO\_IN\_2 146 // GPIO\_146

#define GPIO\_IN\_3 147 // GPIO\_147

#define ROW 2 // number of rows of keymap / number of states

#define COLUMN 4 // number of columns of keymap / number of GPIOs

short state=0;

short press\_counter=0;

unsigned int last\_interrupt\_time = 0;

unsigned int last\_press\_time=0;

static uint64\_t epochMilli;

struct input\_dev \*input; // create a new input device

int gpioMap[COLUMN] = {GPIO\_IN\_0, GPIO\_IN\_1, GPIO\_IN\_2, GPIO\_IN\_3}; // map of GPIOs

static unsigned short KeyMap[ROW][COLUMN] = {KEY\_A, KEY\_B, KEY\_C, KEY\_ENTER, // keys for state 0

KEY\_X, KEY\_Y, KEY\_Z, KEY\_SPACE}; // keys for state 1

//Arrow keys

/\*

{KEY\_UP, KEY\_DOWN, KEY\_SPACE, KEY\_ENTER, // keys for state 0

KEY\_LEFT, KEY\_RIGHT, KEY\_ESC, KEY\_SPACE}; // keys for state 1

\*/

// return the current operation time in millisecond

inline unsigned int millis (void)

{

struct timeval tv;

uint64\_t now;

do\_gettimeofday(&tv);

now = (uint64\_t)tv.tv\_sec \* (uint64\_t)1000 + (uint64\_t)(tv.tv\_usec / 1000);

return (uint32\_t)(now - epochMilli);

}

// send a key event to kernel

inline void sendKey (short row, short column)

{

input\_report\_key(input, KeyMap[row][column], 1);

input\_sync(input);

input\_report\_key(input, KeyMap[row][column], 0);

input\_sync(input);

}

// general interrupt service routine for key events

inline static irq\_handler\_t InterruptHandler\_general( unsigned int irq, struct pt\_regs \*regs, short column )

{

unsigned long flags;

unsigned int interrupt\_time = millis();

if (interrupt\_time - last\_interrupt\_time < 250)

// ignore the interrupt due to bouncing

return (irq\_handler\_t) IRQ\_HANDLED;

// disable hard interrupts

local\_irq\_save(flags);

sendKey (state,column);

// disable hard interrupts on the local CPU, and restore them

local\_irq\_restore(flags);

last\_interrupt\_time = interrupt\_time;

press\_counter=0;

return (irq\_handler\_t) IRQ\_HANDLED;

}

// interrupt service routine to handle GPIO\_IN\_0

static irq\_handler\_t InterruptHandler\_0( unsigned int irq, struct pt\_regs \*regs )

{

return InterruptHandler\_general (irq, regs, 0);

}

// interrupt service routine to handle GPIO\_IN\_1

static irq\_handler\_t InterruptHandler\_1( unsigned int irq, struct pt\_regs \*regs )

{

return InterruptHandler\_general (irq, regs, 1);

}

// interrupt service routine to handle GPIO\_IN\_2

static irq\_handler\_t InterruptHandler\_2( unsigned int irq, struct pt\_regs \*regs )

{

return InterruptHandler\_general (irq, regs, 2);

}

// interrupt service routine to handle GPIO\_IN\_3

static irq\_handler\_t InterruptHandler\_3( unsigned int irq, struct pt\_regs \*regs )

{

unsigned long flags;

unsigned int interrupt\_time = millis();

if (interrupt\_time - last\_interrupt\_time < 80)

return (irq\_handler\_t) IRQ\_HANDLED;

// disable hard interrupts on the local CPU, and restore them

local\_irq\_save(flags);

// examine the switch

if (interrupt\_time-last\_press\_time < 350)

{

// The state toggles when pressed 5 times in a row

if (press\_counter > 6)

{

state^=1; // toggle the current key state

press\_counter=0;

last\_press\_time=interrupt\_time;

local\_irq\_restore(flags);

return (irq\_handler\_t) IRQ\_HANDLED;

}

press\_counter++;

}

else

{

sendKey(state,3);

press\_counter=0; // clear the press\_counter

}

// restore hard interrupts

local\_irq\_restore(flags);

last\_press\_time=interrupt\_time;

return (irq\_handler\_t) IRQ\_HANDLED;

}

int keyboard\_init(void)

{

int ret = 0;

int err, i, j;

// Initialise the GPIO

for(i=0; i<COLUMN; i++)

{

//sprintf(string, "in\_%d", i);

if (gpio\_request(gpioMap[i], "dummy\_dev"))

{

printk( KERN\_INFO "dummy\_dev: %s unable to get GPIO\_OUT\n", \_\_func\_\_ );

ret = -EBUSY;

goto DONE;

}

if (gpio\_direction\_input(gpioMap[i]) < 0 )

{

printk( KERN\_INFO "dummy\_dev: %s unable to set GPIO\_IN as input\n", \_\_func\_\_ );

ret = -EBUSY;

goto DONE;

}

}

// Allocate a new input device structure

input = input\_allocate\_device();

if (!input)

{

err = -ENOMEM;

}

input->name = "GPIO keyboard";

input->keycode = KeyMap;

input->keycodesize = sizeof(unsigned short);

input->keycodemax = ROW\*COLUMN;

input->evbit[0] = BIT(EV\_KEY);

// register all the keybits one by one

for (i = 0; i < ROW; i++)

for (j=0; j < COLUMN; j++)

\_\_set\_bit(KeyMap[i][j], input->keybit);

err = input\_register\_device(input);

if (err)

{

printk(KERN\_CRIT "input\_register\_device failed");

goto DONE;

}

// make an array of declared interrupt handler pointers

irq\_handler\_t\* InterruptHandlerMap [COLUMN] = {InterruptHandler\_0, InterruptHandler\_1, InterruptHandler\_2, InterruptHandler\_3};

// request interrupts for GPIOs

for (i=0; i<COLUMN; i++)

{

if( request\_irq( gpio\_to\_irq(gpioMap[i]), (irq\_handler\_t) InterruptHandlerMap[i],

IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING, "dummy\_dev", NULL ) < 0 )

{

printk( KERN\_INFO "dummy\_dev: %s unable to register gpio irq for GPIO\_IN\n", \_\_func\_\_ );

ret = -EBUSY;

goto DONE;

}

}

printk(KERN\_NOTICE "Keyboard driver initialised\n");

DONE:

return ret;

}

void keyboard\_exit(void)

{

short i;

input\_unregister\_device(input);

input\_free\_device(input);

// free GPIOs and interrupts

for (i=0; i<COLUMN; i++)

{

free\_irq(gpio\_to\_irq(gpioMap[i]), NULL);

gpio\_free(gpioMap[i]);

}

printk(KERN\_NOTICE "Keyboard driver exit\n");

return;

}

module\_init(keyboard\_init);

module\_exit(keyboard\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("ARM University Program");

#include <linux/types.h>

#include <linux/kdev\_t.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <asm/uaccess.h>

#include <linux/interrupt.h>

#include <linux/time.h>

#define GPIO\_IN\_0 105 // GPIO\_105

#define GPIO\_IN\_1 148 // GPIO\_148

#define GPIO\_IN\_2 146 // GPIO\_146

#define GPIO\_IN\_3 147 // GPIO\_147

#define ROW 2 // number of rows of keymap / number of states

#define COLUMN 4 // number of columns of keymap / number of GPIOs

short state=0;

short press\_counter=0;

unsigned int last\_interrupt\_time = 0;

unsigned int last\_press\_time=0;

static uint64\_t epochMilli;

struct input\_dev \*input; // create a new input device

int gpioMap[COLUMN] = {GPIO\_IN\_0, GPIO\_IN\_1, GPIO\_IN\_2, GPIO\_IN\_3}; // map of GPIOs

static unsigned short KeyMap[ROW][COLUMN] = {KEY\_A, KEY\_B, KEY\_C, KEY\_ENTER, // keys for state 0

KEY\_X, KEY\_Y, KEY\_Z, KEY\_SPACE}; // keys for state 1

//Arrow keys

/\*

{KEY\_UP, KEY\_DOWN, KEY\_SPACE, KEY\_ENTER, // keys for state 0

KEY\_LEFT, KEY\_RIGHT, KEY\_ESC, KEY\_SPACE}; // keys for state 1

\*/

// return the current operation time in millisecond

inline unsigned int millis (void)

{

struct timeval tv;

uint64\_t now;

do\_gettimeofday(&tv);

now = (uint64\_t)tv.tv\_sec \* (uint64\_t)1000 + (uint64\_t)(tv.tv\_usec / 1000);

return (uint32\_t)(now - epochMilli);

}

// send a key event to kernel

inline void sendKey (short row, short column)

{

input\_report\_key(input, KeyMap[row][column], 1);

input\_sync(input);

input\_report\_key(input, KeyMap[row][column], 0);

input\_sync(input);

}

// general interrupt service routine for key events

inline static irq\_handler\_t InterruptHandler\_general( unsigned int irq, struct pt\_regs \*regs, short column )

{

unsigned long flags;

unsigned int interrupt\_time = millis();

if (interrupt\_time - last\_interrupt\_time < 250)

// ignore the interrupt due to bouncing

return (irq\_handler\_t) IRQ\_HANDLED;

// disable hard interrupts

local\_irq\_save(flags);

sendKey (state,column);

// disable hard interrupts on the local CPU, and restore them

local\_irq\_restore(flags);

last\_interrupt\_time = interrupt\_time;

press\_counter=0;

return (irq\_handler\_t) IRQ\_HANDLED;

}

// interrupt service routine to handle GPIO\_IN\_0

static irq\_handler\_t InterruptHandler\_0( unsigned int irq, struct pt\_regs \*regs )

{

return InterruptHandler\_general (irq, regs, 0);

}

// interrupt service routine to handle GPIO\_IN\_1

static irq\_handler\_t InterruptHandler\_1( unsigned int irq, struct pt\_regs \*regs )

{

return InterruptHandler\_general (irq, regs, 1);

}

// interrupt service routine to handle GPIO\_IN\_2

static irq\_handler\_t InterruptHandler\_2( unsigned int irq, struct pt\_regs \*regs )

{

return InterruptHandler\_general (irq, regs, 2);

}

// interrupt service routine to handle GPIO\_IN\_3

static irq\_handler\_t InterruptHandler\_3( unsigned int irq, struct pt\_regs \*regs )

{

unsigned long flags;

unsigned int interrupt\_time = millis();

if (interrupt\_time - last\_interrupt\_time < 80)

return (irq\_handler\_t) IRQ\_HANDLED;

// disable hard interrupts on the local CPU, and restore them

local\_irq\_save(flags);

// examine the switch

if (interrupt\_time-last\_press\_time < 350)

{

// The state toggles when pressed 5 times in a row

if (press\_counter > 6)

{

state^=1; // toggle the current key state

press\_counter=0;

last\_press\_time=interrupt\_time;

local\_irq\_restore(flags);

return (irq\_handler\_t) IRQ\_HANDLED;

}

press\_counter++;

}

else

{

sendKey(state,3);

press\_counter=0; // clear the press\_counter

}

// restore hard interrupts

local\_irq\_restore(flags);

last\_press\_time=interrupt\_time;

return (irq\_handler\_t) IRQ\_HANDLED;

}

int keyboard\_init(void)

{

int ret = 0;

int err, i, j;

// Initialise the GPIO

for(i=0; i<COLUMN; i++)

{

//sprintf(string, "in\_%d", i);

if (gpio\_request(gpioMap[i], "dummy\_dev"))

{

printk( KERN\_INFO "dummy\_dev: %s unable to get GPIO\_OUT\n", \_\_func\_\_ );

ret = -EBUSY;

goto DONE;

}

if (gpio\_direction\_input(gpioMap[i]) < 0 )

{

printk( KERN\_INFO "dummy\_dev: %s unable to set GPIO\_IN as input\n", \_\_func\_\_ );

ret = -EBUSY;

goto DONE;

}

}

// Allocate a new input device structure

input = input\_allocate\_device();

if (!input)

{

err = -ENOMEM;

}

input->name = "GPIO keyboard";

input->keycode = KeyMap;

input->keycodesize = sizeof(unsigned short);

input->keycodemax = ROW\*COLUMN;

input->evbit[0] = BIT(EV\_KEY);

// register all the keybits one by one

for (i = 0; i < ROW; i++)

for (j=0; j < COLUMN; j++)

\_\_set\_bit(KeyMap[i][j], input->keybit);

err = input\_register\_device(input);

if (err)

{

printk(KERN\_CRIT "input\_register\_device failed");

goto DONE;

}

// make an array of declared interrupt handler pointers

irq\_handler\_t\* InterruptHandlerMap [COLUMN] = {InterruptHandler\_0, InterruptHandler\_1, InterruptHandler\_2, InterruptHandler\_3};

// request interrupts for GPIOs

for (i=0; i<COLUMN; i++)

{

if( request\_irq( gpio\_to\_irq(gpioMap[i]), (irq\_handler\_t) InterruptHandlerMap[i],

IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING, "dummy\_dev", NULL ) < 0 )

{

printk( KERN\_INFO "dummy\_dev: %s unable to register gpio irq for GPIO\_IN\n", \_\_func\_\_ );

ret = -EBUSY;

goto DONE;

}

}

printk(KERN\_NOTICE "Keyboard driver initialised\n");

DONE:

return ret;

}

void keyboard\_exit(void)

{

short i;

input\_unregister\_device(input);

input\_free\_device(input);

// free GPIOs and interrupts

for (i=0; i<COLUMN; i++)

{

free\_irq(gpio\_to\_irq(gpioMap[i]), NULL);

gpio\_free(gpioMap[i]);

}

printk(KERN\_NOTICE "Keyboard driver exit\n");

return;

}

module\_init(keyboard\_init);

module\_exit(keyboard\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("ARM University Program");

* Lines 0-14 include the needed header files containing the function prototypes and the data structure needed to write a kernel module;
* Lines 16-21 define the macros that will be used for the program. You can refer to the comment for specific meanings;
* Line 23 defines a state variable since we want to map multiple groups of keys to the buttons. The state variable stores the current keymap state (you can imagine an equivalence to input sources);
* Line 24 defines a press counter variable that will be used later for switching keymaps;
* Lines 33-41 define a function to return current operation time in millisecond;
* Lines 43-50 define a function to send key events to the kernel. As a most important part of this driver, the kernel provides a handy API called <linux/input.h> for us;
* Lines 52-72 define a common interrupt handler with button-debouncing capability. This will be used for different GPIOs and will send different key events according to its arguments;
* Lines 74-90 declare 3 different ISRs for the first 3 GPIOs.
* Lines 93-131 declare the last ISR for GPIO\_IN\_3. This ISR provides the ability of switching keymaps by quickly pressing this button for 5 times in a row. In addition, the button can also send key events as normal when pressed.
* Lines 134-198 specify the initialization function including request of GPIOs, configuration of GPIO direction, request of the virtual input device, allocation of an input device structure, and request of GPIO interrupts.
* Lines 205-221 indicate the exit function including free of interrupts, GPIOs, and the virtual input device.

You can easily modify different key events you want to send by modifying the KeyMap. Now copy the driver to the ***hello.c*** file in ***/recipes-kernel/hello-mod/files*** as you did in the Lab and build it:

bitbake -c clean core-image-sato

bitbake core-image-sato

Once complete, a new Micro SD card image will be available, which you can deploy to the Micro SD as follows (assuming the Micro SD is available to the PC as /dev/sdN). Alternatively, use a program of your preference to flash the image.

First run the:

sudo fdisk -l

command to determine which device to flash to (plug in and unplug the SD card to determine which device it is). For this example, the SD card is under the name “sdc” (this may be different in your environment). Next, ensure that the device is unmounted. This can be done using the command:

sudo umount /dev/sdc\*

Once this is done, the following command can be used to copy the image across to the SD card (substitute any folder names and device names to ensure they are relevant to your specific environment).

sudo dd bs=1M if=/home/user/raspberryPi3/rpi-build/tmp-glibc/deploy/images/raspberrypi3/core-image-sato-raspberrypi3.rpi-sdimg of=/dev/sdc

Note that if not done properly, the image being flashed across to the SD card may cause problems when attempting to turn on the board. If this is the case, it may be worth retrying the process again and ensuring that it is done properly, or use a flash program to automate the process.

While the SD card is still connected to the development host, use the following lines to navigate to the etc folder on the SD card (assuming the device has now been mounted).

If, upon booting, the board requires a password - simply carry out the same steps as done in previous labs.

If you have finished the previous post-lab session, the kernel module should be loaded automatically. If not, you can always load it with:

insmod /lib/modules/4.1.21/kernel/drivers/my\_mod/hello.ko

And remove it with:

rmmod /lib/modules/4.1.21/kernel/drivers/my\_mod/hello.ko

Now press the buttons and observe key events. You can customize the keymap according to your needs and play the atanks game.

**Note:** Serial communication (minicom) cannot detect key strokes from the button press/sensor output, and hence the Raspberry Pi 3 board must be connected to a display via Micro HDMI to successfully observe the key events.