

---

## 0.1 Soluzione

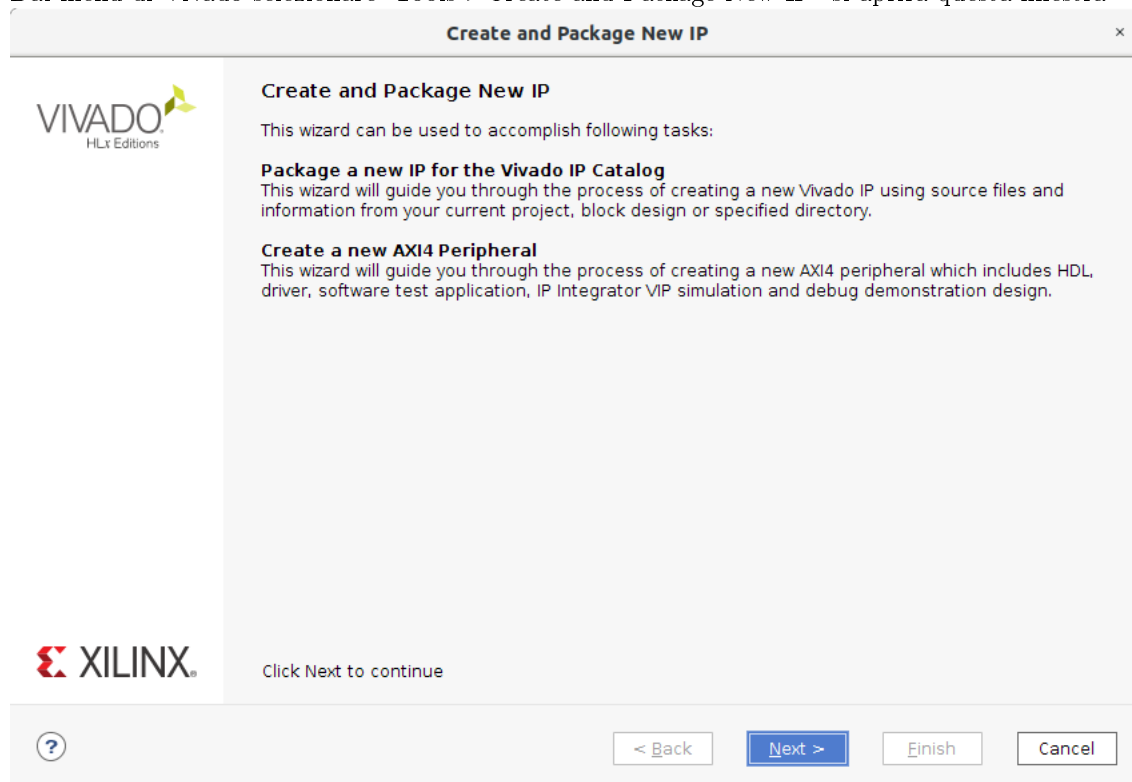
### 0.1.1 Descrizione GPIO

Il general purpose input output è un'interfaccia che permette di leggere il valore proveniente da un pin oppure di scrivere un valore.

Nel nostro design il segnale di enable decide il comportamento dell'interfaccia, nel caso questo venga settato ad uno il valore del bit write viene forzato sul pin pad, altrimenti il valore del pin al quale è collegato pad viene letto tramite il pin read.

### 0.1.2 Creazione Custom IP

Dal menù di Vivado selezionare "Tools->Create and Package New IP" si aprirà questa finestra



Cliccare su "Next", selezionare "Create a New AXI4 peripheral", di nuovo "Next"

Create and Package New IP

Create Peripheral, Package IP or Package a Block Design

Please select one of the following tasks.

Packaging Options

☐ Package your current project  
Use the project as the source for creating a new IP Definition.

☐ Package a block design from the current project  
Choose a block design as the source for creating a new IP Definition.  
Select a block design: design\_1

☐ Package a specified directory  
Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

☒ Create a new AXI4 peripheral  
Create an AXI4 IP, driver, software test application, IP Integrator AXI4 VIP simulation and debug demonstration design.

?

< Back

Next >

Finish

Cancel

Riempire i vari campi con le informazioni dell' IP, in questa finestra è importante il parametro "IP location" che permette di selezionare in quale cartella salveremo il nostro custom IP.  
Quando si avrà compilato tutti i campi, procedere cliccando su "Next"

Create and Package New IP

Peripheral Details

Specify name, version and description for the new peripheral

Name:

myip

Version:

1.0

Display name:

myip\_v1.0

Description:

My new AXI IP

IP location:

/home/saverio/ip\_repo

☐ Overwrite existing

?

< Back

Next >

Finish

Cancel

Nella finestra successiva è possibile configurare il tipo di interfaccia del nostro componente, nel nostro caso LITE, che tipo di device è, Slave dato che non gestisce le transizioni del bus AXI, la dimensione dei registri in cui andremo a scrivere ed a leggere ed il numero di registri che vogliamo utilizzare, clicchiamo su "Next" una volta settati i

parametri desiderati.

**Create and Package New IP**

**Add Interfaces**  
Add AXI4 interfaces supported by your peripheral

☒ Enable Interrupt Support

**Interfaces**  
S00\_AXI

**myi\_v1.0**  
S00\_AXI  
S\_AXI\_INTR

**Configuration for S00\_AXI:**  
Name: S00\_AXI  
Interface Type: Lite  
Interface Mode: Slave  
Data Width (Bits): 32  
Memory Size (Bytes): 64  
Number of Registers: 4 [4..512]

**Navigation:** < Back, Next >, Finish, Cancel

Selezionare “Edit IP” e successivamente cliccare su “Finish” verrà creata una nuova finestra di Vivado in cui è possibile modificare il custom IP.

**Create Peripheral**

**Peripheral Generation Summary**  
1. IP (user.org:user:myi:1.0) with 2 interface(s)  
2. Driver(v1\_00\_a) and testapp [more info](#)  
3. AXI4 VIP Simulation demonstration design [more info](#)  
4. AXI4 Debug Hardware Simulation demonstration design [more info](#)

**Peripheral created will be available in the catalog :**  
/home/saverio/ip\_repo

**Next Steps:**  
☐ Add IP to the repository  
☒ Edit IP  
☐ Verify Peripheral IP using AXI4 VIP  
☐ Verify peripheral IP using JTAG interface

**XILINX**

Click Finish to continue

**Navigation:** < Back, Next >, Finish, Cancel

Nella nuova finestra vediamo che sono stati creati due nuovi file, uno con il nome del nostro custom IP che rappresenta l’ interfaccia del nostro componente ed un altro con il “nome del nostro IP \_nome settato nella generazione del componente”.

### 0.1.3 Modifica Default IP Core

Aggiungiamo al design creato da Vivado il nostro GPIO array ed il singolo GPIO, cliccando su “Add Sources” presente sulla sinistra della interfaccia, una volta fatto ciò istanziamo il nostro componente nel file “nome del nostro IP \_nome settato nella generazione del componente”, è buona norma farlo nella sezione indicata da Vivado tra i seguenti commenti

```
1  -- Add user logic here
2  -- User logic ends
```

Il componente istanziato è il seguente

```
1  gpio_inst : GPIO_Array
2  generic map(width => width)
3  port map(    enable => slv_reg0(width-1 downto 0),
4              write => slv_reg1(width-1 downto 0),
5              read  => gpio_read(width-1 downto 0),
6              pads  => pads);
```

Il valore di enable e di write vengono gestiti dai valori presenti nei registri slv\_reg0 ed 1, invece il valore di read viene salvato nel registro gpio\_read, non può essere usato uno degli slv\_reg generato automaticamente da vivado poiché read vuole forzare dei valori essendo un pin di output, ma anche gli slv\_reg forzano dei valori essendo i loro valori forzati da un process che ne permette la scrittura dei valori, per tale motivo viene introdotto in questo caso il segnale gpio\_read che non viene scritto ma può essere solo letto modificando il process di scrittura nel seguente modo

```
1  process (slv_reg0, slv_reg1, gpio_read, slv_reg3, slv_reg4, slv_reg5, status_reg_out,
2           slv_reg7_out, axi_araddr, S_AXI_ARESETN, slv_reg_rden)
3  variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
4  begin
5      -- Address decoding for reading registers
6      loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
7      case loc_addr is
8          when b"000" =>
9              reg_data_out <= slv_reg0;
10         when b"001" =>
11             reg_data_out <= slv_reg1;
12         when b"010" =>
13             reg_data_out <= gpio_read;
14         when b"011" =>
15             reg_data_out <= slv_reg3;
16         when b"100" =>
17             reg_data_out <= slv_reg4;
18         when b"101" =>
19             reg_data_out <= slv_reg5;
20         when b"110" =>
21             reg_data_out <= status_reg_out;
22         when b"111" =>
23             reg_data_out <= slv_reg7_out;
24         when others =>
25             reg_data_out <= (others => '0');
26     end case;
27 end process;
```

In questo caso invece di leggere lo slv\_reg3 verrà letto il segnale gpio\_read, ma i valori di tale registro non vengono forzati da nessuno se non dal pin read.

Aggiungiamo i parametri del nostro componente in questo caso width tra i seguenti commenti

```
1  -- Users to add parameters here
2  width : integer := 4;
3  -- User parameters ends
```

ed i segnali che si devono esporre nella seguente sezione

```
1  -- Users to add ports here
2      pads : inout std_logic_vector(width-1 downto 0); --! se GPIO in modalità lettura
           mostra il valore letto, altrimenti forza un valore in uscita
3      interrupt : out std_logic; --! segnale di interrupt
4  -- User ports ends
```

Per catturare le variazioni del componente GPIO affinché da avviare l' interruzione è stato utilizzato il seguente process

```
1  gpio_read_sampling : process (S_AXI_ACLK, gpio_read)
2      begin
3          if (rising_edge (S_AXI_ACLK)) then
4              if ( S_AXI_ARESETN = '0' ) then
5                  last_stage <= (others => '0');
6                  current_stage <= (others => '0');
7              else
8                  last_stage <= gpio_read(width-1 downto 0);
9                  current_stage <= last_stage;
10             end if;
11         end if;
12     end process;
```

Tale frammento di codice occorre per determinare quando vi è una variazione dei segnali gpio\_read.

Essenzialmente è costituito da tre flip-flop collegati in cascata, i primi due occorrono per sincronizzare il valore di gpio\_read con il clock, poiché a noi interessa catturare cambiamenti di tale segnale solo in presenza di fronti, di salita nel nostro caso, difatti al primo colpo di clock il primo flip flop salva il valore del segnale, al secondo colpo di clock il secondo flip flop salva il valore precedentemente salvato, così da rispettare la successione degli eventi del segnale da campionare, mentre il primo flip flop campiona di nuovo il segnale in ingresso; il terzo flip flop occorre per problemi di metastabilità che possono occorrere in un circuito reale (ad esempio il valore del primo flip flop da alto passa a basso e il secondo flip flop campiona quando tale valore si trova a metà tra i due livelli, non permettendo al flip flop di salvare un valore stabile, che può essere corretto quando il terzo flip flop campiona il segnale del secondo, ovviamente tale situazione di meta stabilità può continuare anche con più flip flop, ma la percentuale con cui può capitare avendo tre flip flop è molto bassa.

Un secondo process viene utilizzato per la gestione del registro delle interruzioni pending

```
1  intr_pending : process (S_AXI_ACLK, change_detected, ack_intr)
2      begin
3          if (rising_edge (S_AXI_ACLK)) then
4              if (change_detected = '1') then
5                  pending_intr <= pending_intr_tmp or changed_bits;
6              else
7                  if (or_reduce(ack_intr)='1') then
8                      pending_intr <= pending_intr_tmp and (not ack_intr);
9                  end if;
10             end if;
11         end if;
12     end process;
```

Qui si determina quando il valore del registro pending debba essere alto o basso, osserviamo che nel caso in una variazione del segnale change\_detected il cui valore dipende da tale espressione

```
1  change_detected <= global_intr and (or_reduce(changed_bits and intr_mask and (not
           gpio_enable)));
```

cioè ci dice che vi è una variazione solamente se le interruzioni globali sono attive ed è stata rilevata un interrupt abilitato dalla maschera intr\_mask e tale pin di GPIO sia attivo, il valore del registro pending è dato da il valore di una precedente interruzione non servita oppure ad una interruzione rilevata (changed\_bits).

Nel caso arrivi un segnale di ack bisogna cambiare il bit relativo al registro pending della interruzione servita.

L' ultimo process infine occorre per la generazione del segnale di interrupt

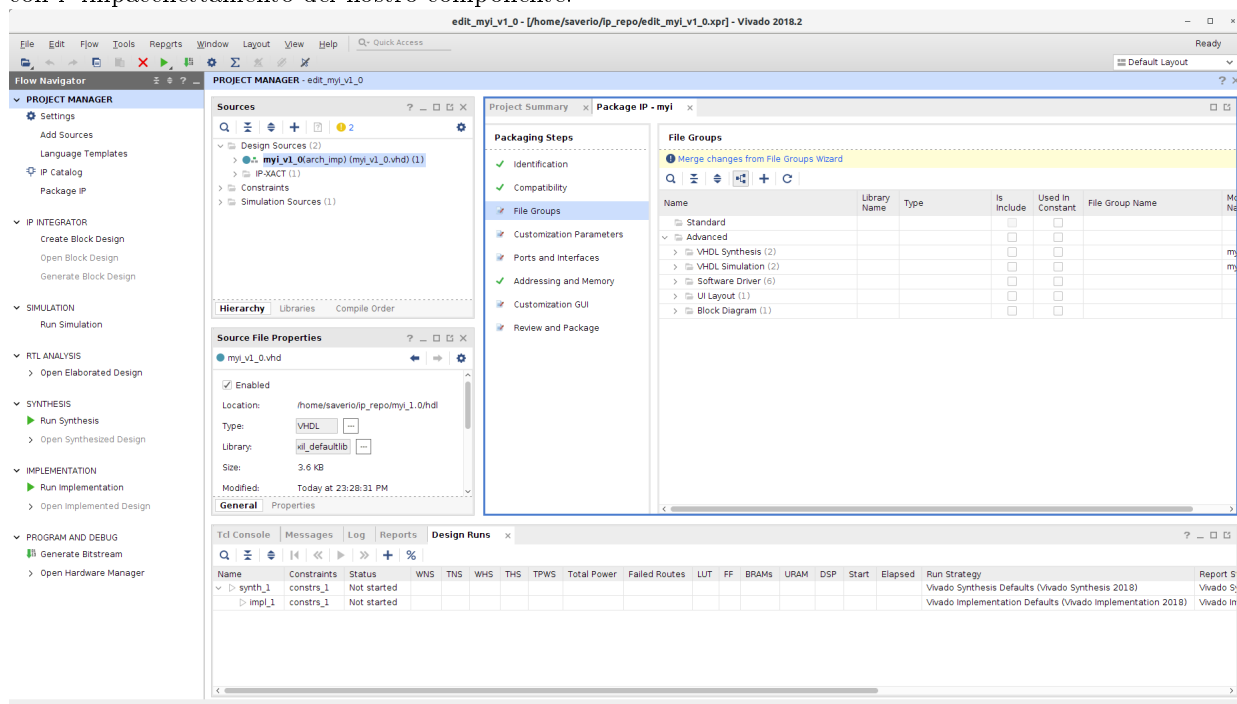
```

1 inst_irq : process(S_AXI_ACLK,pending_intr)
2     begin
3         if (rising_edge (S_AXI_ACLK)) then
4             if ( S_AXI_ARESETN = '0' ) then
5                 interrupt <= '0';
6             else
7                 if (or_reduce(pending_intr) = '1') then
8                     interrupt <= '1';
9                 else
10                    interrupt <= '0';
11                end if;
12            end if;
13        end if;
14    end process;

```

tale segnale è uno nel caso vi siano interruzioni pendenti, nel caso di reset del bus o non vi siano interruzioni è pari a 0.

Dopo aver instaziato il componente nella top level entity avente il nome del nostro custom IP possiamo procedere con l' impacchettamento del nostro componente.



Nella sezione “File Groups” cliccare su “Merge changes from File Groups Wizard”, in “Customization Parameters” su “Merge changes from Customization Parameters Wizard”, selezionare “Hidden Parameters” si aprirà tale finestra

Edit IP Parameter

Use the options below to customize how the parameter will appear in the Customization GUI for users of the IP.

Name:blabla

☐ Visible in Customization GUI

☒ Show Name

Display Name:Blabla

Tooltip:

Format:long

Editable:Yes

Dependency:No

☐ Specify Range

Type:List of values

+ - ↑ ↓

Press the + button to add a value

Show As:Not Applicable

Layout:Not Applicable

Default Value:0

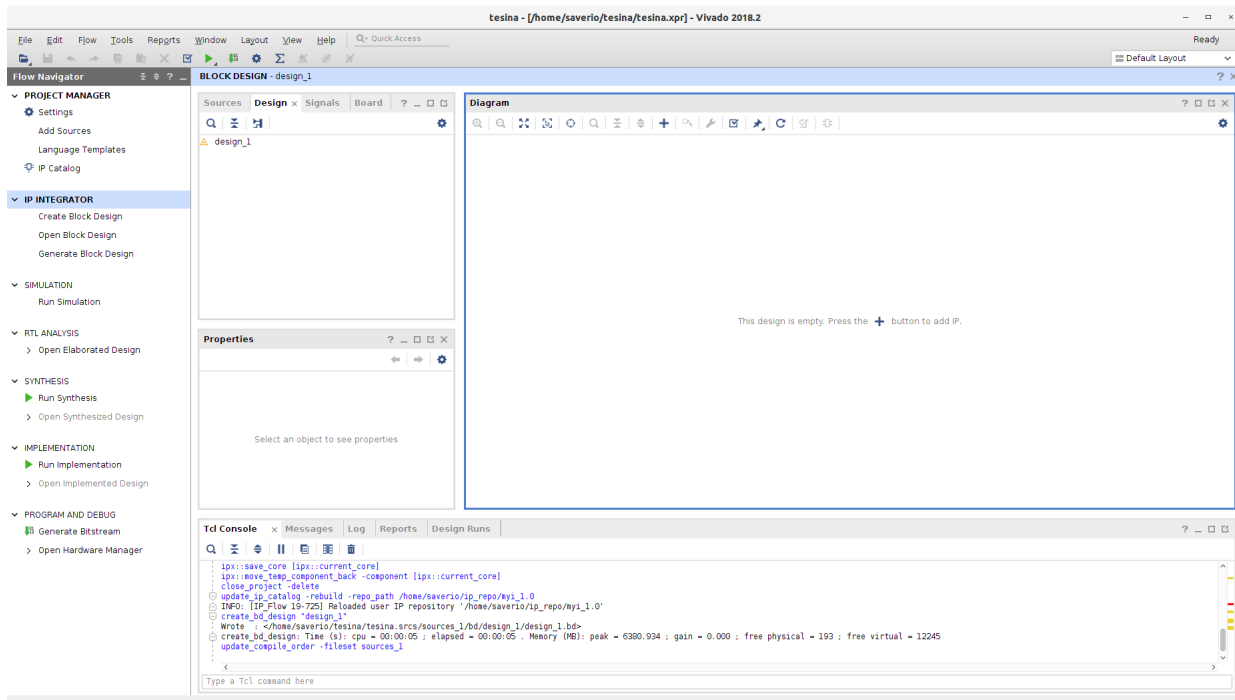
OKCancel

Da qui è possibile rendere visibile questo parametro da configurare cliccando sul box “Visible in Customization GUI” e settarne i valori che può assumere cliccando su “Specify Range”.

Recarsi infine su “Review and Package” cliccare su “Re-Package IP” per ottenere il custom ip, facendo chiudere la finestra di vivado.

### 0.1.4 Creazione del block design

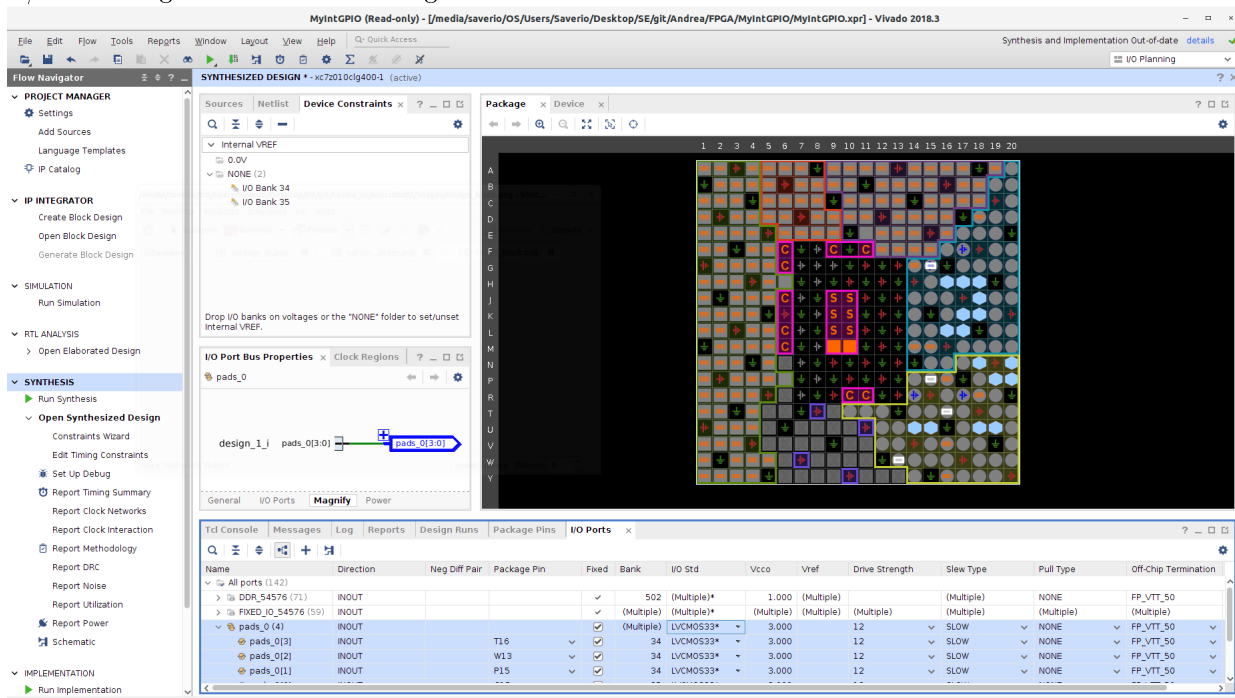
Cliccare sul menù di sinistra “Create block design” inserire i parametri desiderati e cliccare su “OK” verrà mostrato questo workspace



Cliccare sul pulsante + ed inserire il custom IP creato insieme al processore ZYNQ, appariranno due pulsanti “Run Block Automation” e “Run Connection Automation” i vari componenti si collegheranno tra di loro automaticamente, una volta fatto ciò bisogna rendere esterni i pin che si vuole pilotare dalla board cliccando con il tasto destro del mouse su un pin e selezionando “Make external” una volta fatto selezionare l’icona contrassegnata da un simbolo di spunta ed è possibile chiudere il block design.

Bisogna generare ora un wrapper HDL affinché il block design sia sintetizzabile, recandosi nella sezione “Sources”, tasto destro sul nome del block design e selezionare “Create HDL Wrapper...” cliccando “OK” ed essendo sicuri che sia selezionata l’opzione “Let Vivado manage wrapper and auto-update” fatto ciò procedere alla sintesi.

Una volta terminata, selezionare dal sottomenù “Open Synthesized Design ” e da un menù a tendina in alto a destra “I/O Planning” verrà mostrata la seguente schermata



In basso sono presenti i pin che sono stati resti esterni, bisogna selezionare il tip di “I/O Std” di solito è quello



mostrato in figura ed i “Package Pin” una volta fatto salvare i constraints cliccando sull’ icone del floppy blu e dando un nome al file di constraint, una volta fatto si può generare il bitstream. Terminato il processo possiamo esportarlo dal menù “File -> Export -> Export Hardware...” assicurarsi che sia spuntata l’opzione “Include bitstream” e selezionare “OK”, si può ora procedere alla creazione del driver linux, lanciando SDK dal menù “File->Launch SDK”.

### 0.1.5 Driver Standalone

Il driver ora presentato, viene eseguito direttamente dalla sezione PS della board senza il supporto di un sistema operativo linux.

Per poter procedere alla scrittura del driver, dobbiamo sapere dove i registri del nostro componente hardware sono stati mappati, tale informazione può essere reperita dal file “xparameters.h” presente nella directory “cartella\_del\_board\_support” troveremo una sezione di codice molto simile

```
1  /* Definitions for driver MYINTGPIO */
2  #define XPAR_MYINTGPIO_NUM_INSTANCES 3
3  /* Definitions for peripheral MYINTGPIO_0 */
4  #define XPAR_MYINTGPIO_0_DEVICE_ID 0
5  #define XPAR_MYINTGPIO_0_S00_AXI_BASEADDR 0x43C00000
6  #define XPAR_MYINTGPIO_0_S00_AXI_HIGHADDR 0x43C0FFFF
7  /* Definitions for peripheral MYINTGPIO_1 */
8  #define XPAR_MYINTGPIO_1_DEVICE_ID 1
9  #define XPAR_MYINTGPIO_1_S00_AXI_BASEADDR 0x43C20000
10 #define XPAR_MYINTGPIO_1_S00_AXI_HIGHADDR 0x43C2FFFF
11 /* Definitions for peripheral MYINTGPIO_2 */
12 #define XPAR_MYINTGPIO_2_DEVICE_ID 2
13 #define XPAR_MYINTGPIO_2_S00_AXI_BASEADDR 0x43C40000
14 #define XPAR_MYINTGPIO_2_S00_AXI_HIGHADDR 0x43C4FFFF
```

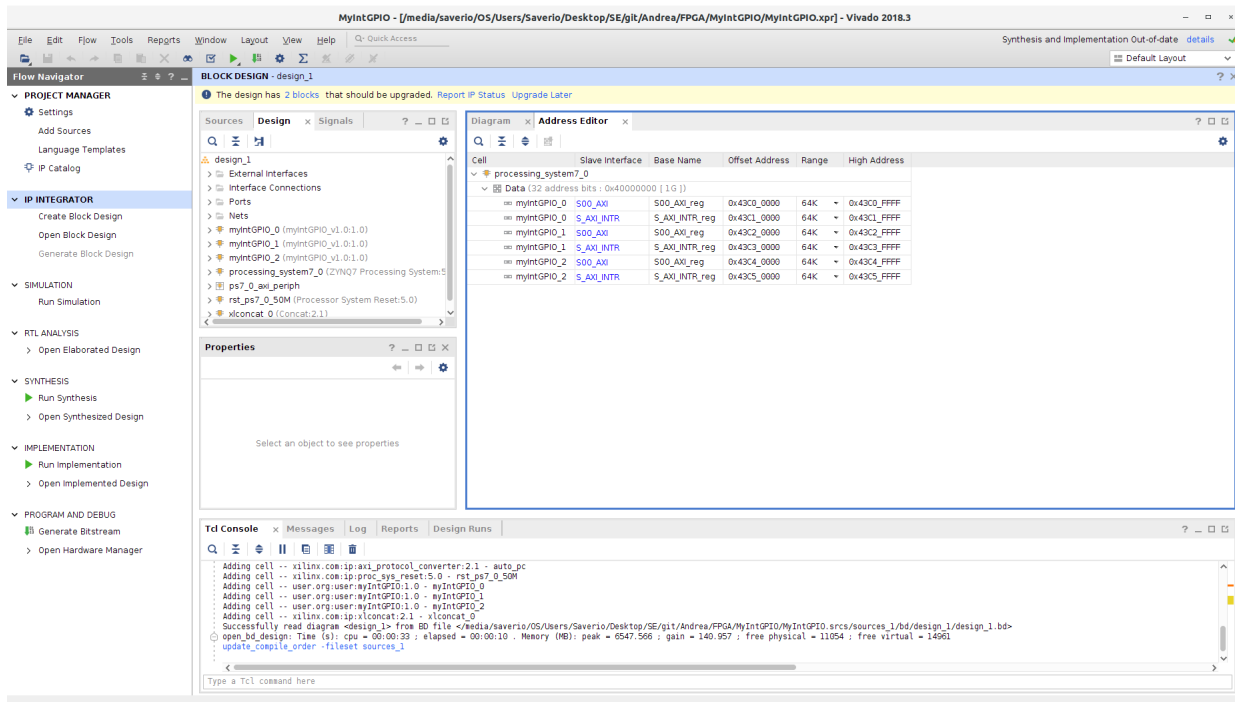
dove è possibile sapere il numero dei nostri custom IP core istanziati nel progetto HW il loro indirizzo base e quello più alto associato alla periferica.

Oltre a tale informazione dobbiamo anche conoscere di quanto sono stati spiazzati i nostri slv\_reg rispetto all’ indirizzo base, tale informazione reperibile nel file situato in “cartella\_del\_board\_support\_package\ps7\_cortexa9\_0\libsrc\’nome\_ una volta aperto sarà presente una sezione simile in alto nel file

```
1  #define MYINTGPIO_S00_AXI_SLV_REG0_OFFSET 0
2  #define MYINTGPIO_S00_AXI_SLV_REG1_OFFSET 4
3  #define MYINTGPIO_S00_AXI_SLV_REG2_OFFSET 8
4  #define MYINTGPIO_S00_AXI_SLV_REG3_OFFSET 12
```

Ottenute queste informazioni possiamo leggere e scrivere valori nei registri.

Per la gestione delle interruzioni invece abbiamo bisogno dato che il nostro componente sfrutta le interruzioni dobbiamo conoscere dove sono mappati i registri, creati da vivado, che gestiscono e ci danno informazioni sulle interruzioni che avvengono, tali indirizzi sono visibile aprendo il block design e recandoci nella sezione address editor, visibile in figura.



Per conoscere invece di quanto sono spazati i registri che controllano l'evoluzione della gestione dell'interrupt dall'indirizzo base nel file "nome del componente \_S\_AXI\_INTR" presente all'interno del nostro custom IP è presente un process simile

```

1 process (reg_global_intr_en, reg_intr_en, reg_intr_sts, reg_intr_ack, reg_intr_pending,
2   axi_araddr, S_AXI_ARESETN, intr_reg_rden)
3   variable loc_addr :std_logic_vector(2 downto 0);
4   begin
5     if S_AXI_ARESETN = '0' then
6       reg_data_out <= (others => '0');
7     else
8       -- Address decoding for reading registers
9       loc_addr := axi_araddr(4 downto 2);
10      case loc_addr is
11        when "000" =>
12          reg_data_out <= x"0000000" & "000" & reg_global_intr_en(0);
13        when "001" =>
14          reg_data_out <= reg_intr_en;
15        when "010" =>
16          reg_data_out <= reg_intr_sts;
17        when "011" =>
18          reg_data_out <= reg_intr_ack;
19        when "100" =>
20          reg_data_out <= reg_intr_pending;
21        when others =>
22          reg_data_out <= (others => '0');
23      end case;
24    end if;
25  end process;

```

Vediamo che sono presenti cinque registri di dimensione pari a trentadue bit aventi il seguente compito:

1. il primo abilita il componente a poter generare interrupt;
2. il secondo dice quali segnali di interrupt vogliamo abilitare;
3. il terzo ci dice quali interruzioni sono avvenute;

4. il quarto quando viene scritto si asserisce che l' interruzione è stata gestita;
5. il quinto le interruzioni pendenti;

La differenza del registro di stato è quello pendenti sta che, una volta abilitate tutte le interruzioni quello di stato mostra tutte le interruzioni che avvengono nel componente, invece quello pendenti mostra solamente le interruzioni avvenute e che sono abilitate.

Possiamo procedere alla creazione del driver, instanziamo un nuovo progetto vuoto dal menù "File -> New -> Application Project" e creare un progetto vuoto, sul progetto appena creato andremo ad inserire la logica del nostro driver, di seguito verrà fornita un' idea di come funziona il nostro driver, per dettagli sulla implementazione riferirsi al codice.

Dobbiamo abilitare lo scugic per essere sensibili alle interruzioni, chiamando prima la funzione

```
1 XScuGic_LookupConfig("id scugic");
```

per conoscere la configurazione attuale dello scugic e successivamente

```
1 XScuGic_CfgInitialize("variabile di tipo XScuGic", "variabile configurazione", "indirizzo base  
della CPU");
```

Ora possiamo abilitare le interruzioni del nostro componente programmabile:

1. Abilitiamo l' interruzione scrivendo un uno nel registro reg\_intr\_en;
2. Abilitiamo le interruzioni globali del componente un uno nel registro reg\_global\_intr\_en;
3. Settiamo la routine che gestisce le interruzioni;
4. Connettiamo l' handler fornito da Xilinx con la logica di handling fornita dal processore con la primitiva

```
1 Xil_ExceptionRegisterHandler
```

5. Assegnamo alla ricezione di un determinato interrupt la routine da noi creata per la sua gestione con la primitiva

```
1 XScuGic_Connect
```

così che il gic sappia quale routine lanciare alla ricezione di quella interruzione

6. Abilitiamo le eccezioni nel processore ARM con

```
1 Xil_ExceptionEnable
```

7. Settiamo la maschera del nostro componente detector per dire quali pin del GPIO vogliamo campionare
8. Infine abilitiamo lo scugic con

```
1 XScuGic_Enable
```

Nella routine di gestione dell' interruzione :

1. Disabilitiamo l' interruzione scrivendo uno zero nel registro reg\_intr\_en;
2. Scriviamo un uno nel registro reg\_intr\_ack per dire al componente che l' interruzione è stata catturata;
3. Riabilitiamo le interruzioni sul componente;

### 0.1.6 Driver Linux

La seguente sezione mostra come scrivere un driver con il supporto di un sistema operativo Linux. In questo ambiente è possibile scrivere un driver come un modulo kernel oppure utilizzando l'Userspace I/O (UIO). In entrambi i casi la prima operazione da effettuare è quella di creare un First Stage Boot Loader e un Device-Tree come mostrato nel primo capitolo.

### 0.1.6.1 Kernel Mode

Un driver può essere scritto sottoforma di modulo kernel e poi caricato dinamicamente. Questa pratica fornisce più flessibilità rispetto al “build statico” di un modulo all’interno del kernel, in quanto potrebbe risultare non necessario inserire moduli che poi non verranno utilizzati. Per l’astrazione del nostro device GPIO si è realizzata una struct, definita nel file GPIO.h, che contiene tutte le informazioni necessarie per la gestione del dispositivo.

```
1  /**
2   * @brief Struttura che astrae un device GPIO in kernel-mode.
3   * Contiene ciò che è necessario al funzionamento del driver.
4   */
5  typedef struct {
6   /** Major e minor number associati al device (M: identifica il driver associato al device; m
7    : utilizzato dal driver per discriminare il singolo device tra quelli a lui associati)*/
8   dev_t Mm;
9   /** Puntatore a struttura platform_device cui l’oggetto GPIO si riferisce */
10   struct platform_device *pdev;
11   /** Struttura per l’astrazione di un device a caratteri */
12   struct cdev cdev;
13   /** Puntatore alla struttura che rappresenta l’istanza del device*/
14   struct device* dev;
15   /** Puntatore a struttura che rappresenta una vista alto livello del device*/
16   struct class* class;
17   /** Interrupt-number a cui il device è connesso*/
18   uint32_t irqNumber;
19   /** Puntatore alla regione di memoria cui il device è mappato*/
20   struct resource *mreg;
21   /** Device Resource Structure*/
22   struct resource res;
23   /** Maschera delle interruzioni interne attive per il device*/
24   uint32_t irq_mask;
25   /** res.end - res.start; numero di indirizzi associati alla periferica.*/
26   uint32_t res_size;
27   /** Indirizzo base virtuale della periferica*/
28   void __iomem *vrtl_addr;
29   /** wait queue per la sys-call read() */
30   wait_queue_head_t read_queue;
31   /** wait queue per la sys-call poll()*/
32   wait_queue_head_t poll_queue;
33   /** Flag che indica, quando asserito, la possibilità di effettuare una chiamata a read*/
34   uint32_t can_read;
35   /** Spinlock usato per garantire l’accesso in mutua esclusione alla variabile can_read*/
36   spinlock_t slock_int;
37 } GPIO;
```

Per le funzioni necessarie all’interfacciamento con il device si rimanda alla documentazione interna. Il device è stato gestito come un **character device**.

All’inserimento del modulo viene chiamata la funzione **Probe**, la quale si occupa dell’inizializzazione del driver chiamando la funzione GPIO\_Init() del device da registrare. Questa deve occuparsi dunque di tutte le operazioni necessarie alla registrazione e all’inserimento di un dispositivo a caratteri all’interno del sistema:

```
1  /**
2   * @brief Inizializza una struttura GPIO per il corrispondente device
3   *
4   * @param GPIO_device puntatore a struttura GPIO, corrispondente al device su cui operare
5   * @param owner puntatore a struttura struct module, proprietario del device (THIS_MODULE)
6   * @param pdev puntatore a struct platform_device
7   * @param driver_name nome del driver
8   * @param device_name nome del device
9   * @param serial numero seriale del device
```

```

10  * @param f_ops puntatore a struttura struct file_operations, specifica le funzioni che
    * agiscono sul device
11  * @param irq_handler puntatore irq_handler_t alla funzione che gestisce gli interrupt
    * generati dal device
12  * @param irq_mask maschera delle interruzioni attive del device
13  *
14  * @retval "0" se non si è verificato nessun errore
15  *
16  * @details
17  */
18  int GPIO_Init(    GPIO* GPIO_device,
19                  struct module *owner,
20                  struct platform_device *pdev,
21                  struct class* class,
22                  const char* driver_name,
23                  const char* device_name,
24                  uint32_t serial,
25                  struct file_operations *f_ops,
26                  irq_handler_t irq_handler,
27                  uint32_t irq_mask) {
28      int error = 0;
29      struct device *dev = NULL;
30
31      char *file_name = kmalloc(strlen(driver_name) + 5, GFP_KERNEL);
32      sprintf(file_name, device_name, serial);
33
34      GPIO_device->pdev = pdev;
35      GPIO_device->class = class;
36
37      /** Alloca un range di Mj e min numbers per il device a caratteri */
38
39      if ((error = alloc_chrdev_region(&GPIO_device->Mm, 0 , 1, file_name)) != 0) {
40          printk(KERN_ERR "%s: alloc_chrdev_region() ha restituito %d\n", __func__, error);
41          return error;
42      }
43
44      /** Inizializza la struttura cdev specificando la struttura file operations associata al
    * device a caratteri */
45
46      cdev_init (&GPIO_device->cdev, f_ops);
47      GPIO_device->cdev.owner = owner;
48
49      /** Crea il device all'interno del filesystem assegnandogli i numbers richiesti in
    * precedenza e ne restituisce il puntatore. */
50
51      if ((GPIO_device->dev = device_create(class, NULL, GPIO_device->Mm, NULL, file_name)) ==
    NULL) {
52          printk(KERN_ERR "%s: device_create() ha restituito NULL\n", __func__);
53          error = -ENOMEM;
54          goto device_create_error;
55      }
56
57      /** Aggiunge il device a caratteri al sistema. Se l'operazione va a buon fine sarà possibile
    * vedere il device sotto /dev */
58
59
60      if ((error = cdev_add(&GPIO_device->cdev, GPIO_device->Mm, 1)) != 0) {
61          printk(KERN_ERR "%s: cdev_add() ha restituito %d\n", __func__, error);
62          goto cdev_add_error;
63      }

```

```

64
65 /** Inizializza la struct resource con il valori recuperati dal device tree corrispondente
    al device */
66
67 dev = &pdev->dev;
68 if ((error = of_address_to_resource(dev->of_node, 0, &GPIO_device->res)) != 0) {
69     printk(KERN_ERR "%s: address_to_resource() ha restituito %d\n", __func__, error);
70     goto of_address_to_resource_error;
71 }
72 GPIO_device->res_size = GPIO_device->res.end - GPIO_device->res.start + 1;
73
74 /** Alloca una quantita res_size di memoria fisica per il dispositivo IO a partire dall'
    indirizzo res.start e ne restituisce l'indirizzo */
75
76 if ((GPIO_device->mreg = request_mem_region(GPIO_device->res.start, GPIO_device->res_size,
    file_name)) == NULL) {
77     printk(KERN_ERR "%s: request_mem_region() ha restituito NULL\n", __func__);
78     error = -ENOMEM;
79     goto request_mem_region_error;
80 }
81
82 /** Mappa la memoria fisica allocata e restituisce l'indirizzo virtuale */
83
84 if ((GPIO_device->vrtl_addr = ioremap(GPIO_device->res.start, GPIO_device->res_size)) ==
    NULL) {
85     printk(KERN_ERR "%s: ioremap() ha restituito NULL\n", __func__);
86     error = -ENOMEM;
87     goto ioremap_error;
88 }
89
90 /** Cerca le specifiche dell'interrupt nel device tree e restituisce il suo numero
    identificativo */
91
92 GPIO_device->irqNumber = irq_of_parse_and_map(dev->of_node, 0);
93 if ((error = request_irq(GPIO_device->irqNumber, irq_handler, 0, file_name, NULL)) != 0)
94 {
95     printk(KERN_ERR "%s: request_irq() ha restituito %d\n", __func__, error);
96     goto irq_of_parse_and_map_error;
97 }
98 GPIO_device->irq_mask = irq_mask;
99
100
101 /** Inizializzazione della wait-queue per la system-call read() e poll() */
102
103 init_waitqueue_head(&GPIO_device->read_queue);
104 init_waitqueue_head(&GPIO_device->poll_queue);
105
106 /** Inizializzazione degli spinlock */
107
108 spin_lock_init(&GPIO_device->slock_int);
109 GPIO_device->can_read = 0;
110 /** Abilitazione degli interrupt del device */
111
112 GPIO_GlobalInterruptEnable(GPIO_device);
113 GPIO_PinInterruptEnable(GPIO_device, GPIO_device->irq_mask);
114 goto no_error;
115
116 irq_of_parse_and_map_error:
117 iounmap(GPIO_device->vrtl_addr);

```

```

118 ioremap_error:
119     release_mem_region(GPIO_device->res.start, GPIO_device->res_size);
120 request_mem_region_error:
121 of_address_to_resource_error:
122 cdev_add_error:
123     device_destroy(GPIO_device->class, GPIO_device->Mm);
124 device_create_error:
125     cdev_del(&GPIO_device->cdev);
126     unregister_chrdev_region(GPIO_device->Mm, 1);
127
128 no_error:
129
130     printk(KERN_INFO " IRQ registered as %d\n", GPIO_device->irqNumber);
131     printk(KERN_INFO " Driver succesfully probed at Virtual Address 0x%08lx\n", (unsigned long
132         ) GPIO_device->vrtl_addr);
133
134     return error;
135 }

```

Quando invece il modulo viene rimosso viene chiamata la funzione **Remove**, la quale effettua le operazioni inverse chiamando la funzione `GPIO_Destroy`. Sia la Probe che la Remove devono essere ridefinite all'interno del modulo e si utilizza la struttura `platform_driver` per effettuare il matching.

```

1 /**
2  * @brief Definisce le funzioni probe() e remove() da chiamare al caricamento del driver.
3  */
4 static struct platform_driver GPIO_driver = {
5     .driver = {
6         .name = DRIVER_NAME,
7         .owner = THIS_MODULE,
8         .of_match_table = of_match_ptr(__test_int_driver_id),
9     },
10    .probe = GPIO_probe,
11    .remove = GPIO_remove
12 };

```

La funzione `of_match_ptr(__test_int_driver_id)` si occupa di effettuare il matching con i device contenuti all'interno del device-tree. Per ogni device contenente un campo `compatible` uguale a quello specificato mediante la struttura `of_device_id` verrà chiamata la funzione di Probe per far sì che il driver possa gestire quel device.

```

1 /**
2  * @brief Identifica il device all'interno del device tree
3  *
4  */
5 static const struct of_device_id __test_int_driver_id[]={
6     {.compatible = "GPIO"},
7     {}
8 };

```

Dato che un driver può gestire più di un singolo device GPIO è stato implementato un meccanismo di gestione tramite lista. La Probe dunque inizializza il corrispondente device GPIO e lo inserisce all'interno della lista, se questa non contiene già il numero massimo consentito di dispositivi. Il device all'interno del sistema operativo Linux è visto come un file, per cui il device driver deve implementare tutte le system-call per l'interfacciamento con un file. La corrispondenza tra queste e la relativa funzione fornita dal driver viene stabilita attraverso la struttura `file_operations`.

```

1 /**
2  * @brief Struttura che specifica le funzioni che agiscono sul device
3  *
4  */
5 static struct file_operations GPIO_fops = {

```

```

6  .owner      = THIS_MODULE,
7  .llseek    = GPIO_llseek,
8  .read      = GPIO_read,
9  .write     = GPIO_write,
10 .poll      = GPIO_poll,
11 .open      = GPIO_open,
12 .release   = GPIO_rele
13 };

```

- owner: rappresenta puntatore al modulo che è il possessore della struttura. Ha lo scopo di evitare che il modulo venga rimosso quando uno delle funzionalità fornite è in uso. Inizializzato mediante la macro THIS\_MODULE
- GPIO\_llseek: sposta l'offset di lettura/scrittura sul file.
- GPIO\_read: utilizzata per leggere dal device. La chiamata a GPIO\_read potrebbe avvenire quando il device non ha dati disponibili, in questo caso il processo chiamante deve essere messo in una coda di processi sleeping in modo tale da mascherare all'esterno le dinamiche interne del device. Per far ciò viene utilizzata una variabile "can\_read". La funzione read effettua un controllo sullo stato di quest'ultima e se rileva che non è possibile effettuare una lettura mette il processo in sleep. L'ISR avrà il compito di settare la variabile per poter rendere possibile la lettura e risvegliare i processi dalla coda. Per realizzare questo meccanismo sono stati utilizzati spinlock e wait\_queue fornite dal kernel.
- GPIO\_write: utilizzata per inviare dati al device.
- GPIO\_poll: utilizzata per verificare se un'operazione di lettura sul device risulti bloccante. Verifica lo stato della variabile can\_read e in caso sia possibile effettuare una lettura ritorna un'opportuna maschera.
- GPIO\_open: chiamata all'apertura del file descriptor associato al device. Se alla chiamata viene specificato il flag O\_NONBLOCK tutte le operazioni di lettura sul file descriptor aperto non risulteranno essere bloccanti.
- GPIO\_release: chiamata alla chiusura del file descriptor associato al device.

Il codice allegato è diviso in:

- GPIO.h/GPIO.c : definizione e implementazione di una struttura che astrae il nostro device GPIO in kernel mode. Contiene ciò che è necessario al funzionamento del driver, compreso lo spinlock per l'accesso in mutua esclusione alla variabile can\_read e le wait\_queue.
- GPIO\_list.h/GPIO\_list.c : definizione e implementazione di una lista di oggetti GPIO. Fornisce tutte le funzioni necessarie per l'interfacciamento quali inizializzazione, cancellazione, aggiunta oggetto, ricerca.
- GPIO\_kernel\_main.c: rappresenta il vero e proprio modulo kernel che reimplementa le tutte funzioni necessarie all'interfacciamento.

Per compilare il modulo è sufficiente lanciare lo script "prepare\_environment.sh" prima di dare il comando make. Segue il Makefile utilizzato per la compilazione:

```

1  obj-m += my_kernel_GPIO.o
2  my_kernel_GPIO-objs :=GPIO_kernel_main.o GPIO.o GPIO_list.o
3
4  all:
5      make -C linux-xlnx/ M=$(PWD) modules
6
7  clean:
8      make -C linux-xlnx/ M=$(PWD) clean

```

Una volta ottenuto il kernel object (.ko) l'ultima operazione da effettuare è quella di inserirlo mediante il comando:

```

1  insmod my_kernel_GPIO.ko

```



```
root@linaro-developer:/# insmod my_kernel_GPIO.ko
```

Se l'operazione è andata a buon fine si visualizzeranno i seguenti messaggi stampando il log del kernel tramite il comando dmesg:

```
[ 35.366499] my_kernel_GPIO: loading out-of-tree module taints kernel.
[ 35.367646] Chiamata GPIO_probe
[ 35.368084] IRQ registered as 46
[ 35.368093] Driver succesfully probed at Virtual Address 0xe0b20000
[ 35.368100] 43c00000.GPIO => GPIO0
[ 35.368270] Chiamata GPIO_probe
[ 35.371049] IRQ registered as 47
[ 35.371059] Driver succesfully probed at Virtual Address 0xe0b40000
[ 35.371066] 43c10000.GPIO => GPIO1
[ 35.371245] Chiamata GPIO_probe
[ 35.371540] IRQ registered as 48
[ 35.371550] Driver succesfully probed at Virtual Address 0xe0b60000
[ 35.371556] 43c20000.GPIO => GPIO2
root@linaro-developer:/#
```

Per mostrare il corretto funzionamento di tutte le funzionalità implementate sono state create due user application: read\_block\_user\_app.c, read\_NON\_block\_user\_app.c e poll\_user\_app.c che sono allegate.

- read\_block\_user\_app.c : l'utente specifica tramite linea di comando quale GPIO vuole utilizzare (-s GPIO0 switches; -b GPIO1 buttons; -l GPIO2 leds). Effettua in un loop infinito la chiamata a read per controllare se siano presenti nuovi valori da leggere sul dispositivo selezionato.
- read\_NON\_block\_user\_app.c : l'utente specifica tramite linea di comando quale GPIO vuole utilizzare (-s GPIO0 switches; -b GPIO1 buttons; -l GPIO2 leds). Effettua in un loop infinito la chiamata a read (distanziata l'una dall'altra di un tempo specificato tramite il parametro TIMEOUT per rendere verificabile il funzionamento) per controllare se siano presenti nuovi valori da leggere sul dispositivo. L'apertura del device è effettuata specificando il flag O\_NONBLOCK per cui le chiamate a read non saranno mai bloccanti.
- poll\_user\_app.c : l'utente specifica tramite linea di comando quale GPIO vuole utilizzare (-s GPIO0 switches; -b GPIO1 buttons; -l GPIO2 leds). Effettua una chiamata a poll con un TIMEOUT specificato: se prima della scadenza di questo vengono rilevati nuovi valori da leggere la funzione ritorna la maschera degli eventi rilevati e viene effettuata una chiamata a read che non sarà bloccante; altrimenti la funzione ritorna il valore 0 e non verrà effettuata la chiamata a read in quanto bloccante.

Per rimuovere il modulo impartire il comando:

```
1 rmmmod my_kernel_GPIO.ko
```

```
root@linaro-developer:/# rmmmod my_kernel_GPIO.ko
[ 21.903441] Chiamata GPIO_remove
ptr: de585e00
name: 43c20000.GPIO
id: 4294967295
[ 21.904022] Chiamata GPIO_remove
ptr: de57e000
name: 43c10000.GPIO
id: 4294967295
[ 21.904514] Chiamata GPIO_remove
ptr: de57e200
name: 43c00000.GPIO
id: 4294967295
```

### 0.1.6.2 UIO

L'Userspace I/O (UIO) è un framework che permette di gestire i driver direttamente nell'userspace e fornisce meccanismi di **gestione delle interruzioni a livello utente**. La prima operazione da effettuare prima di scrivere un driver UIO è recarsi all'interno del progetto del device-tree e aggiungere ai bootargs nel file system-top.dts il parametro "uio\_pdrv\_genirq.of\_id=generic-uio" e all'interno del file pl.dtsi impostare il campo compatible dei device GPIO a "generic-uio". Segue il file pl.dtsi:

```
1 / {
2     amba_pl: amba_pl {
```

```

3  #address-cells = <1>;
4  #size-cells = <1>;
5  compatible = "simple-bus";
6  ranges ;
7  GPIO_0: GPIO@43c00000 {
8      /* This is a place holder node for a custom IP, user may need to update the entries */
9      clock-names = "s00_axi_aclk";
10     clocks = <&clkc 15>;
11     compatible = "generic-uio";
12     interrupt-names = "interrupt";
13     interrupt-parent = <&intc>;
14     interrupts = <0 29 4>;
15     reg = <0x43c00000 0x10000>;
16     xlnx,s00-axi-addr-width = <0x5>;
17     xlnx,s00-axi-data-width = <0x20>;
18 };
19 GPIO_1: GPIO@43c10000 {
20     /* This is a place holder node for a custom IP, user may need to update the entries */
21     clock-names = "s00_axi_aclk";
22     clocks = <&clkc 15>;
23     compatible = "generic-uio";
24     interrupt-names = "interrupt";
25     interrupt-parent = <&intc>;
26     interrupts = <0 30 4>;
27     reg = <0x43c10000 0x10000>;
28     xlnx,s00-axi-addr-width = <0x5>;
29     xlnx,s00-axi-data-width = <0x20>;
30 };
31 GPIO_2: GPIO@43c20000 {
32     /* This is a place holder node for a custom IP, user may need to update the entries */
33     clock-names = "s00_axi_aclk";
34     clocks = <&clkc 15>;
35     compatible = "generic-uio";
36     interrupt-names = "interrupt";
37     interrupt-parent = <&intc>;
38     interrupts = <0 31 4>;
39     reg = <0x43c20000 0x10000>;
40     xlnx,s00-axi-addr-width = <0x5>;
41     xlnx,s00-axi-data-width = <0x20>;
42 };
43 };
44 };

```

A questo punto si ricompila il device-tree generando il file .dtb e lo si sposta nella partizione di BOOT della SD Card. All'avvio del sistema operativo si potranno osservare sotto /dev i tre device. Il driver userspace effettuerà il mapping dei device per poi mettersi in attesa di notifica di interrupt tramite chiamata a read. Segue uno schema generale.

```

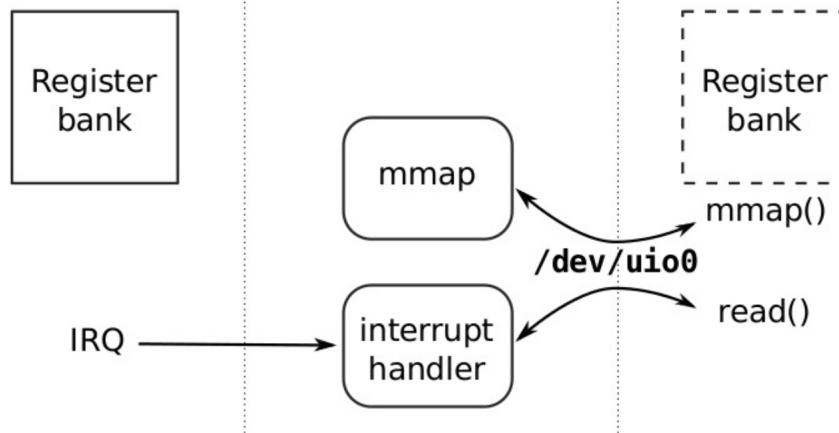
root@linaro-developer:~# ls /dev/
block      memory_bandwidth  ram6   tty19  tty39  tty59  vcsc1
char       mmcblk0           ram7   tty2   tty4   tty6   vcsc2
console    mmcblkop1        ram8   tty20  tty40  tty60  vcsc3
cpu_dma_latency mmcblkop2      ram9   tty21  tty41  tty61  vcsc4
disk       network_latency   random tty22  tty42  tty62  vcsc5
fd         network_throughput shm     tty23  tty43  tty63  vcsc6
full       null             snd     tty24  tty44  tty7   vcscu
gpiochip0  port            stderr  tty25  tty45  tty8   vcscu1
iio:device0 ptmx           stdin   tty26  tty46  tty9   vcscu2
initctl    pts           stdout  tty27  tty47  ttyPS0 vcscu3
kmsg       ram0           tty     tty28  tty48  uio0   vcscu4
log        ram1           tty0    tty29  tty49  uio1   vcscu5
loop-control ram10         tty1    tty3   tty5   uio2   vcscu6
loop0      ram11         tty10   tty30  tty50  urandom vga_arbiter
loop1      ram12         tty11   tty31  tty51  vcs     watchdog
loop2      ram13         tty12   tty32  tty52  vcs1    watchdog0
loop3      ram14         tty13   tty33  tty53  vcs2    xconsole
loop4      ram15         tty14   tty34  tty54  vcs3    zero
loop5      ram2          tty15   tty35  tty55  vcs4
loop6      ram3          tty16   tty36  tty56  vcs5
loop7      ram4          tty17   tty37  tty57  vcs6
mem        ram5          tty18   tty38  tty58  vcsc

```

Hardware

Kernel

Application



Segue il codice relativo al driver UIO:

```

1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <limits.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <sys/mman.h>
9  #include <poll.h>
10 #include "GPIO_interrupt_uio_poll.h"
11
12 #define DIR_OFF      0  // DIRECTION
13 #define WRITE_OFF    4  // WRITE
14 #define READ_OFF     8  // READ
15 #define GLOBAL_INTR_EN 12 // GLOBAL INTERRUPT ENABLE
16 #define INTR_EN      16 // LOCAL INTERRUPT ENABLE
17 #define INTR_ACK_PEND 28 // PENDING/ACK REGISTER
18
19 #define INTR_MASK 15

```

```

20
21 #define TIMEOUT 2000
22
23 typedef u_int8_t u8;
24 typedef u_int32_t u32;
25
26 void write_reg(void *addr, unsigned int offset, unsigned int value)
27 {
28     *((unsigned*)(addr + offset)) = value;
29 }
30
31 unsigned int read_reg(void *addr, unsigned int offset)
32 {
33     return *((unsigned*)(addr + offset));
34 }
35
36
37 void wait_for_interrupt(int fd0, int fd1, int fd2, void *addr_0, void *addr_1, void *addr_2)
38 {
39
40     int pending = 0;
41     int reenable = 1;
42     u32 read_value;
43     struct pollfd poll_fds [3];
44     int ret;
45
46     printf("Waiting for interrupts....\n");
47
48     poll_fds[0].fd = fd0;
49     poll_fds[0].events = POLLIN; //The field events is an input parameter, a bit mask
        specifying the
50         //events the application is interested in for the file descriptor fd.
51         //Means that we are interested at the event: there is data to read.
52     poll_fds[1].fd = fd1;
53     poll_fds[1].events = POLLIN;
54
55     poll_fds[2].fd = fd2;
56     poll_fds[2].events = POLLIN;
57
58     // non blocking wait for an interrupt on file descriptors specified in the pollfd
        structure*/
59     ret = poll(poll_fds, 3, TIMEOUT); //timeout of TIMEOUT ms
60     if (ret > 0){
61         if(poll_fds[0].revents && POLLIN){
62
63             read(fd0, (void *)&pending, sizeof(int));
64             write_reg(addr_0, GLOBAL_INTR_EN, 0); //disabilito interruzioni
65             printf("*****ISR SWITCH*****\n");
66             read_value = read_reg(addr_0, READ_OFF);
67             printf("Read value: %08x\n", read_value);
68             write_reg(addr_0, INTR_ACK_PEND, INTR_MASK); //ACK
69             sleep(1);
70             write_reg(addr_0, INTR_ACK_PEND, 0); //ACK
71             write_reg(addr_0, GLOBAL_INTR_EN, 1); //abiito interruzioni
72             write(fd0, (void *)&reenable, sizeof(int));
73
74         }
75         if(poll_fds[1].revents && POLLIN){
76
77             read(fd1, (void *)&pending, sizeof(int));

```

```

78     write_reg(addr_1, GLOBAL_INTR_EN, 0); //disabilito interruzioni
79     printf("*****ISR BUTTON*****\n");
80     read_value = read_reg(addr_1, READ_OFF);
81     printf("Read value: %08x\n", read_value);
82     write_reg(addr_1, INTR_ACK_PEND, INTR_MASK); //ACK
83     sleep(1);
84     write_reg(addr_1, INTR_ACK_PEND, 0); //ACK
85     write_reg(addr_1, GLOBAL_INTR_EN, 1); //abiito interruzioni
86     write(fd1, (void *)&reenable, sizeof(int));
87
88 }
89 if(poll_fds[2].revents && POLLIN){
90
91     read(fd2, (void *)&pending, sizeof(int));
92     write_reg(addr_2, GLOBAL_INTR_EN, 0); //disabilito interruzioni
93     printf("*****ISR LED*****\n");
94     read_value = read_reg(addr_2, READ_OFF);
95     printf("Read value: %08x\n", read_value);
96     write_reg(addr_2, INTR_ACK_PEND, INTR_MASK); //ACK
97     sleep(1);
98     write_reg(addr_2, INTR_ACK_PEND, 0); //ACK
99     write_reg(addr_2, GLOBAL_INTR_EN, 1); //abiito interruzioni
100    write(fd2, (void *)&reenable, sizeof(int));
101
102 }
103 }
104
105 }
106
107 int main(int argc, char *argv[]){
108
109     void *gpio_0_ptr;
110     void *gpio_1_ptr;
111     void *gpio_2_ptr;
112
113     //-----MAPPING GPIO_0-----//
114
115     int fd_gpio_0 = open("/dev/uio0", O_RDWR);
116     if (fd_gpio_0 < 1){
117         printf("Errore nell'accesso al device UIO0.\n");
118         return -1;
119     }
120
121     unsigned dimensione_pag = sysconf(_SC_PAGESIZE);
122
123     gpio_0_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED, fd_gpio_0, 0);
124
125     write_reg(gpio_0_ptr, GLOBAL_INTR_EN, 1); // abilitazione interruzioni globali
126     write_reg(gpio_0_ptr, INTR_EN, INTR_MASK); // abilitazione interruzioni
127
128     //-----MAPPING GPIO_1-----//
129
130     int fd_gpio_1 = open("/dev/uio1", O_RDWR);
131     if (fd_gpio_1 < 1){
132         printf("Errore nell'accesso al device UIO1.\n");
133         return -1;
134     }
135
136     gpio_1_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED, fd_gpio_1, 0);
137

```

```

138 write_reg(gpio_1_ptr, GLOBAL_INTR_EN, 1); // abilitazione interruzioni globali
139 write_reg(gpio_1_ptr, INTR_EN, INTR_MASK); // abilitazione interruzioni
140
141 //-----MAPPING GPIO_2-----//
142
143 int fd_gpio_2 = open("/dev/uio2", O_RDWR);
144 if (fd_gpio_2 < 1){
145     printf("Errore nell'accesso al device UIO2.\n");
146     return -1;
147 }
148
149 gpio_2_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED, fd_gpio_2, 0);
150
151 write_reg(gpio_2_ptr, GLOBAL_INTR_EN, 1); // abilitazione interruzioni globali
152 write_reg(gpio_2_ptr, INTR_EN, INTR_MASK); // abilitazione interruzioni
153
154
155
156 while (1) {
157     printf("Calling function wait_for_interrupt: ");
158     wait_for_interrupt(fd_gpio_0, fd_gpio_1, fd_gpio_2, gpio_0_ptr, gpio_1_ptr, gpio_2_ptr);
159 }
160
161 // unmap the gpio device
162 munmap(gpio_0_ptr, dimensione_pag);
163 munmap(gpio_1_ptr, dimensione_pag);
164 munmap(gpio_2_ptr, dimensione_pag);
165
166 return 0;
167
168 }

```

La prima operazione del driver, come introdotto all'inizio della sezione, è quella di aprire tre file descriptor sui tre device uio corrispondenti ai tre GPIO. Successivamente calcola la dimensione della pagina e effettua il mapping tramite chiamata a `mmap()`. Si è scelto di non effettuare chiamate a `read()` bloccanti ma di utilizzare la system call `poll()` per verificare se sono disponibili nuovi dati prima di effettuare una lettura. La funzione prende in ingresso un array di strutture `pollfd` composte da tre campi:

1. file descriptor: descrittore del file associato al device.
2. events: maschera di bit che indica gli eventi, relativi al file descriptor, ai quali l'applicazione è interessata.
3. revents: maschera riempita dal kernel contenente gli eventi rilevati.

La chiamata `poll()` prende in ingresso la suddetta struttura, un intero che indica quanti oggetti sono presenti in quest'ultima e un parametro che indica il tempo che il processo deve attendere notifiche di eventi dal device espresso in millisecondi.