

Documentazione Sistemi Embedded 2018/19

Bevilacqua Michele - Mat. M63000770	Camera Andrea - Mat. M63000814
Falso Roberto - Mat. M63000788	Milo Saverio - Mat. M63000773

4 luglio 2019

Indice

1	Installare linux su zybo	3
1.1	Traccia	3
1.2	Soluzione	3
1.2.1	Prerequisiti	3
1.2.2	Compilazione u-boot	4
1.2.3	Creazione immagine di boot	5
1.2.4	Compilazione del Device Tree	8
1.2.5	Creazione della uImage	10
1.2.6	Preparazione SD card	11
2	GPIO	13
2.1	Traccia	13
2.2	Soluzione	13
2.2.1	Descrizione GPIO	13
2.2.2	Creazione Custom IP	14
2.2.3	Modifica Default IP Core	19
2.2.3.1	Istanziamento del componente GPIO_Array	19
2.2.3.2	Gestione delle interruzioni	22
2.2.4	Creazione del block design	26
2.2.5	Driver Standalone	30
2.2.6	Driver Linux	34
2.2.6.1	Kernel Mode	34
2.2.6.2	UIO	42
3	UART	49
3.1	Traccia	49
3.2	Soluzione	49
3.2.1	Dispositivo UART	49
3.2.1.1	Sezione Trasmissione	50
3.2.1.2	Sezione Ricezione	51
3.2.1.3	Modulazione del Clock	52
3.2.2	Custom AXI IP Core	53
3.2.2.1	my_uart_int_v1.0_S00_AXI	53
3.2.3	Design	56
3.2.4	Driver Standalone	56
3.2.5	Driver Linux	61

3.2.5.1	Driver Kernel Mode	61
3.2.5.2	UIO	64
4	Progetto finale	72
4.1	Traccia	72
4.2	Periferiche	73
4.2.1	UART	73
4.2.2	Spi	74
4.2.3	I2C	75
4.2.4	CAN	76
4.2.5	CRC	77
4.3	Scelte Progettuali	77
4.3.1	Architettura del sistema e topologia della rete	77
4.3.1.1	Arbitraggio dei bus	78
4.3.1.2	Modalità di Trasmissione	79
4.3.1.3	Gestione degli indirizzi	79
4.3.2	Architettura Software	81
4.3.2.1	Descrizione API	81
4.3.2.2	CAN	83
4.3.2.3	I2C	85
4.3.2.4	SPI	85
4.3.2.5	UART	86
4.3.2.6	CRC	86
4.3.3	Applicativo Utente	87
4.3.3.1	Connessioni fische broad	89

Capitolo 1

Installare linux su zybo

1.1 Traccia

Descrivere i passaggi per la creazione di una sd card per rendere possibile l' avvio di una distro linux sulla zybo.

1.2 Soluzione

1.2.1 Prerequisiti

Una distribuzione linux, dove sono presenti i seguenti pacchetti:

1. build-essential
2. u-boot-tools
3. gparted
4. git
5. libncurses-dev
6. libssl-dev
7. bison
8. flex
9. device-tree-compiler

Se non fossero presenti utilizzare il seguente comando.

```
1 sudo apt install build-essential u-boot-tools gparted git libncurses-dev  
libssl-dev bison flex device-tree-compiler
```

Clonare le seguenti repository:

1. <https://github.com/Xilinx/linux-xlnx>

2. <https://github.com/Xilinx/u-boot-xlnx>
3. <https://github.com/Xilinx/device-tree-xlnx>

Scaricare un filesystem, nel caso specifico è stato scelto Linaro:

```
1 https://releases.linaro.org/debian/images/developer-armhf/latest/
```

Prima di eseguire qualsiasi operazione, come ad esempio la cross-compilazione del kernel, è necessario configurare il terminale con cui si lavorerà. La configurazione consiste nell'esportare alcune variabili d'ambiente lanciando lo script `prepare_environment.sh` riportato di seguito:

```
1 #!/bin/bash
2 VIVADO_PATH=$HOME/Vivado
3 VIVADO_VERS=2018.3
4 SDK_PATH=$HOME/SDK
5 SDK_VERS=2018.3
6 source $VIVADO_PATH/$VIVADO_VERS/settings64.sh
7 source $SDK_PATH/$SDK_VERS/settings64.sh
8 export ARCH=arm
9 export CROSS_COMPILE=arm-linux-gnueabi-
10 export PATH=$HOME/Scrivania/UART_KERNEL_MODE/linux-xlnx/tools/:$PATH
```

1.2.2 Compilazione u-boot

Posizionarsi all'interno della cartella `u-boot-xlnx`. Prima di procedere alla compilazione bisogna modificare parte del file `zynq-common.h` presente in:

```
1 include/configs/zynq-common.h
```

La porzione di codice da modificare è la seguente

```
1 "sdboot=if mmcinfo; then " \
2 "run uenvboot;" \
3 "echo Copying Linux from SD to RAM... && " \
4 "load mmc 0 ${kernel_load_address} ${kernel_image} && " \
5 "load mmc 0 ${devicetree_load_address} ${devicetree_image} && " \
6 "load mmc 0 ${ramdisk_load_address} ${ramdisk_image} && " \
7 "bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}; "
```

e va modificata come segue

```
1 "sdboot=if mmcinfo; then " \
2 "run uenvboot;" \
3 "echo Copying Linux from SD to RAM... && " \
```

```
4 "load mmc 0 ${kernel_load_address} ${kernel_image} && " \
5 "load mmc 0 ${devicetree_load_address} ${devicetree_image} && " \
6 "bootm ${kernel_load_address} - ${devicetree_load_address}; "
```

Una volta fatto ciò è possibile procedere alla compilazione. Dal terminale in cui era stato lanciato lo script `prepare_environment`, eseguendo i seguenti comandi,

```
1 make CROSS-COMPILER=arm-linux-gnueabi- zynq_zybo_config
2 make -j 2 CROSS-COMPILER=arm-linux-gnueabi- u-boot.elf
```

verrà generato il file `u-boot.elf`

1.2.3 Creazione immagine di boot

La successiva operazione da effettuare consiste nella creazione del First Stage Boot Loader, responsabile del caricamento del bitstream che permetterà la configurazione del Processing System (PS) della Zynq all'avvio. Avviare SDK, dal menù "File -> Launch SDK" all'interno del progetto Vivado. Una volta avviato è necessario creare un nuovo progetto da "File -> New -> Application Project". Dopo aver scelto un nome al progetto, cliccando su Next apparirà la seguente schermata



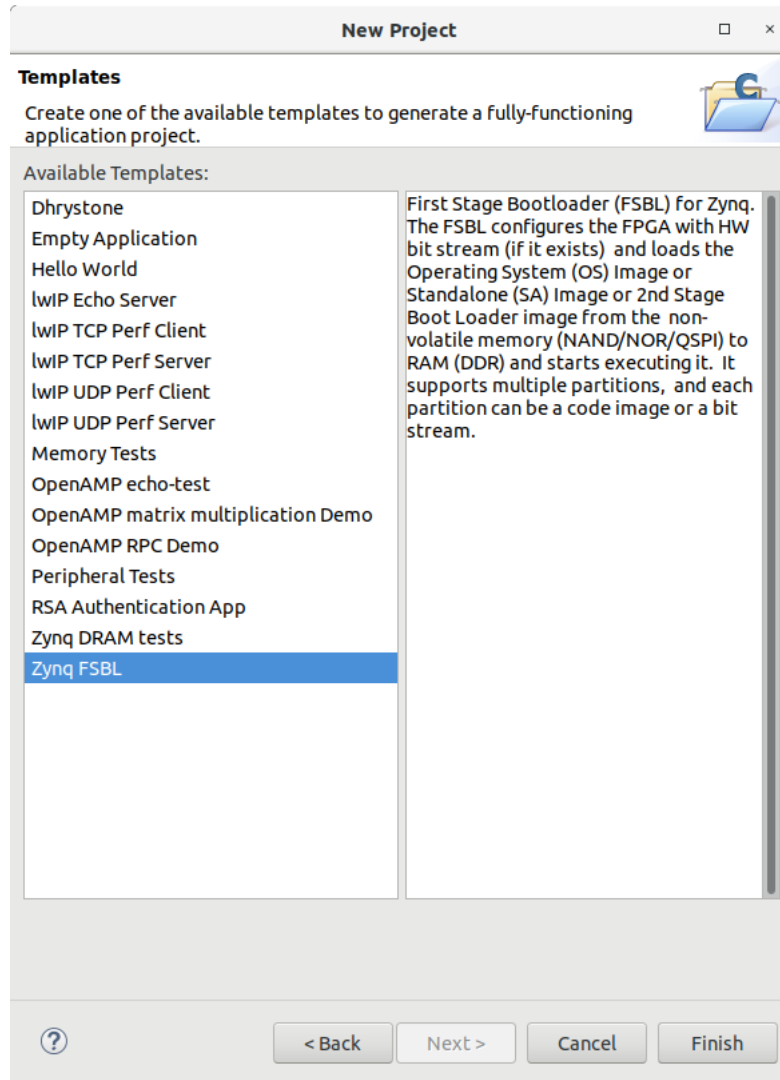


Figura 1.1: Creazione First Stage Boot Loader

Selezionare il template “Zynq FSBL” come in figura e cliccare su Finish.

Dal pannello “Project Explorer” selezionare il progetto appena creato e dal menù “Xilinx” selezionare “Create Boot Image”. Si aprirà la seguente finestra:

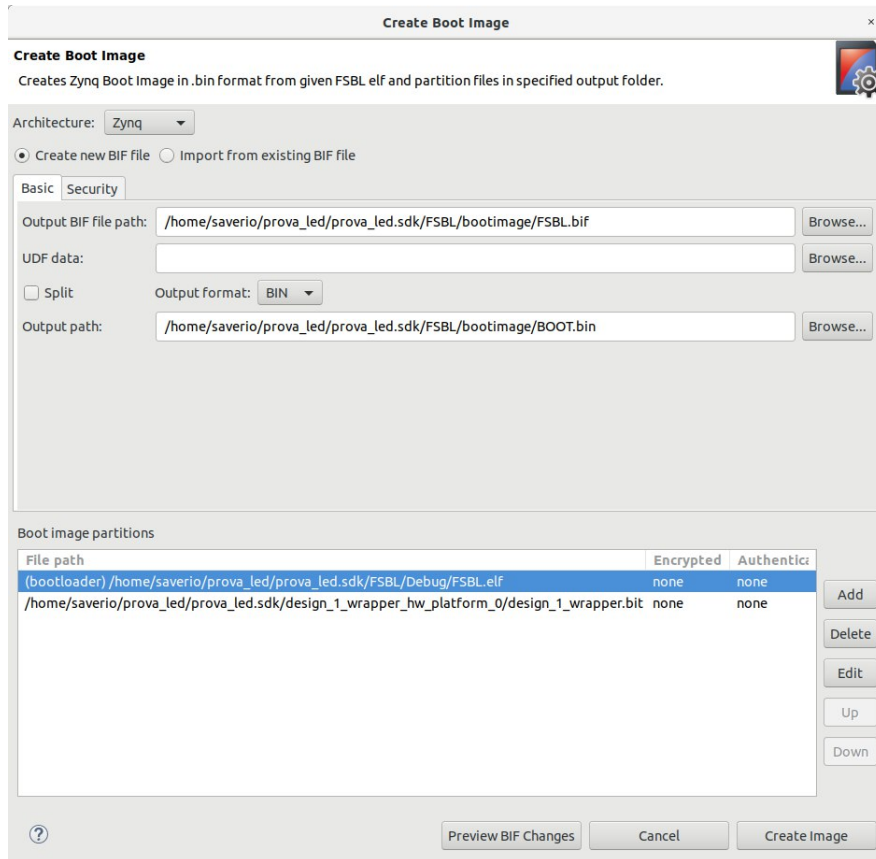


Figura 1.2: Creazione immagine di Boot

Cliccare “Add -> Browse...” e selezionare il file u-boot.elf generato dalla compilazione di u-boot.elf. Se l’inserimento è andato a buon fine si potrà visualizzare il file u-boot.elf nella lista Boot image partitions.

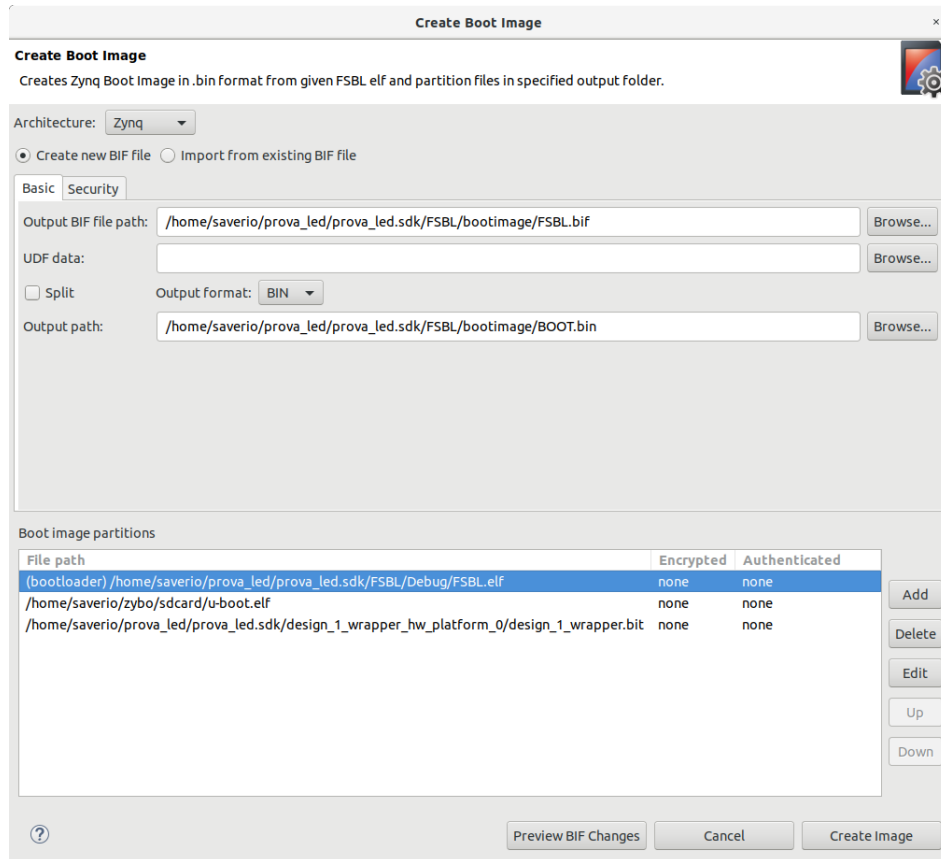
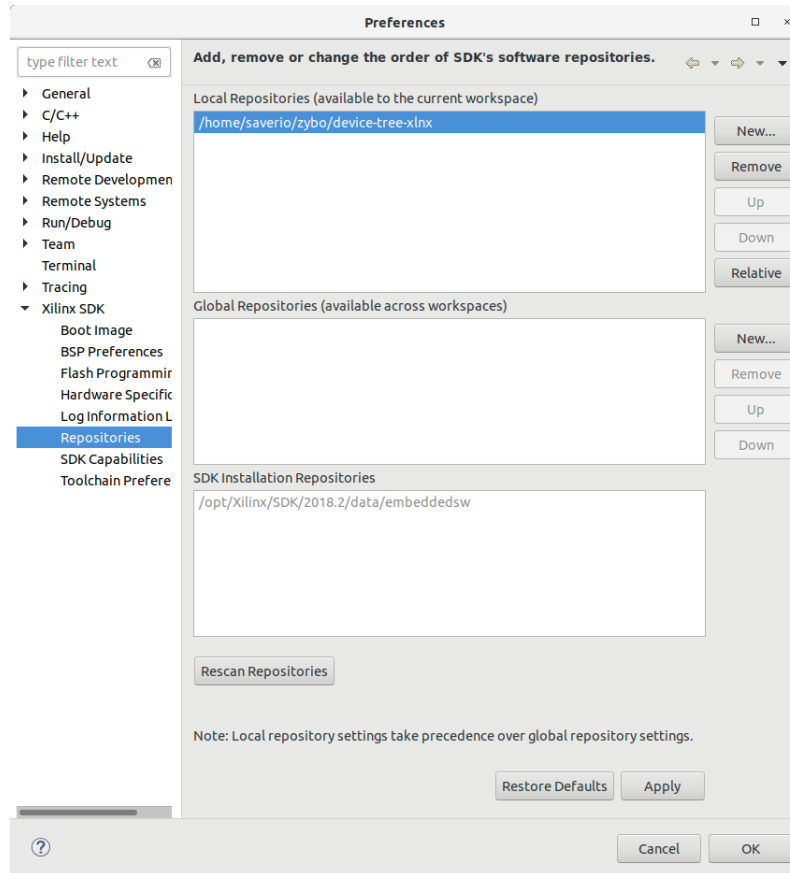


Figura 1.3: Creazione immagine di Boot

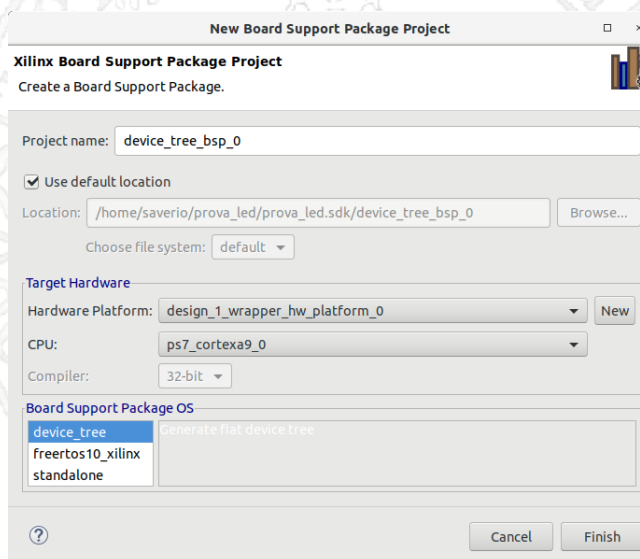
L'ultima operazione da effettuare è quella di cliccare su "Create Image". Se il processo è andato a buon fine verrà creato, nel path identificato dal percorso "Output path", il file BOOT.bin.

1.2.4 Compilazione del Device Tree

La successiva operazione è quella di creare un Device Tree Blob, il quale ha il compito di descrivere l'architettura hardware al sistema operativo. Prima di procedere alla creazione del dtb è necessario aggiungere le repositories DTS di Xilinx a quelle disponibili all'interno del SDK. Dal menù "Xilinx" selezionare "Repositories", cliccare su "New..." e selezionare la cartella device-tree-xlnx precedentemente clonata. Se l'aggiunta è andata a buon fine si visualizzerà la voce device-tree-xlnx tra le repositories locali.

**Figura 1.4:** Aggiunta repositories device-tree-xlnx

A questo punto è possibile procedere con la creazione del device-tree. Creare un “Board Support Package” dal menù “File -> New”

**Figura 1.5:** Creazione Board Support Package

Selezionare device_tree come mostrato in figura e cliccare su “Finish”. Il progetto appena creato sarà composto dai seguenti file:

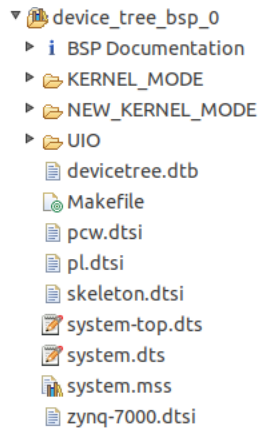


Figura 1.6: Progetto Board Support Package

Nel file system-top.dts è necessario modificare i parametri di boot per il kernel come segue:

```
1 bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk  
rootfstype=ext4 rootwait devtmpfs.mount=1 earlycon";
```

L'ultima operazione da effettuare è compilare il device-tree-source ottenendo il device-tree-blob. Aprire un terminale nella cartella del progetto relativo al Board Support Package e lanciare il seguente comando

```
1 dtc -I dts -O dtb -o devicetree.dtb system-top.dts
```

1.2.5 Creazione della uImage

A questo punto è possibile procedere con la compilazione del Kernel scaricato. Recarsi nella cartella linux-xlnx e lanciare i seguenti comandi

```
1 make xilinx_zynq_defconfig  
2 make menuconfig
```

Il secondo comando aprirà il menù di configurazione seguente:

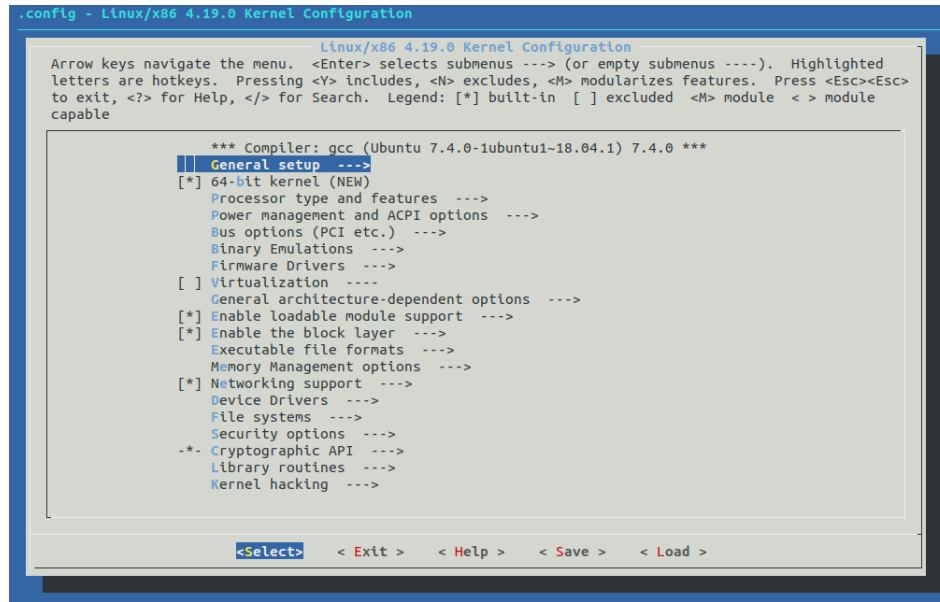


Figura 1.7: Menu di Configurazione del Kernel

Da questa schermata è possibile modificare la configurazione del kernel. Per abilitare il supporto ai driver Userspace I/O che verranno trattati nel seguente capitolo.

Rechiamoci in General Setup e verifichiamo che la spunta Userspace I/O sia asserita. Per la definitiva compilazione lanciare i seguenti comandi:

```
1 make -j 4
2 make -j 4 UIMAGE_LOADADDR=0x8000 uImage
```

Verrà creato un file uImage all'interno della directory "arch/arm/boot/"

1.2.6 Preparazione SD card

Formattare una SD card con tre partizioni divise nel seguente modo:

1. 4MB spazio non allocato (root)
2. 1GB FAT32 (BOOT)
3. lo spazio rimanente EXT4

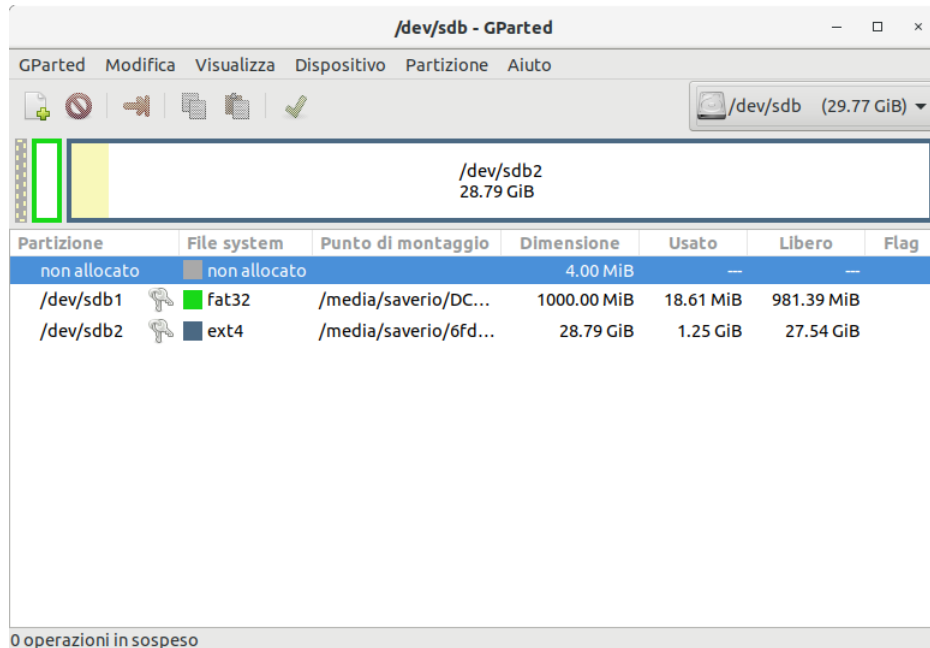


Figura 1.8: Parzionamento SD Card con GParted

Copiare infine sulla partizione di BOOT FAT32 i file Boot.bin, devicetree.dtb e la uImage. Nella partizione EXT4 va invece istanziato il filesystem di root. Per effettuare questa operazione è necessario scompattare l'archivio .tar, scaricato in precedenza dal sito di Linaro, ed impartire il seguente comando:

```
1 rsync -azv "path_cartella_scompattata_linaro"/binary/ "path mount point  
nella partizione EXT4"
```

Capitolo 2

GPIO

2.1 Traccia

Creazione di un Custom IP core per la gestione delle periferiche collegate tramite gpio e gestione dei vari segnali di interrupt.

2.2 Soluzione

2.2.1 Descrizione GPIO

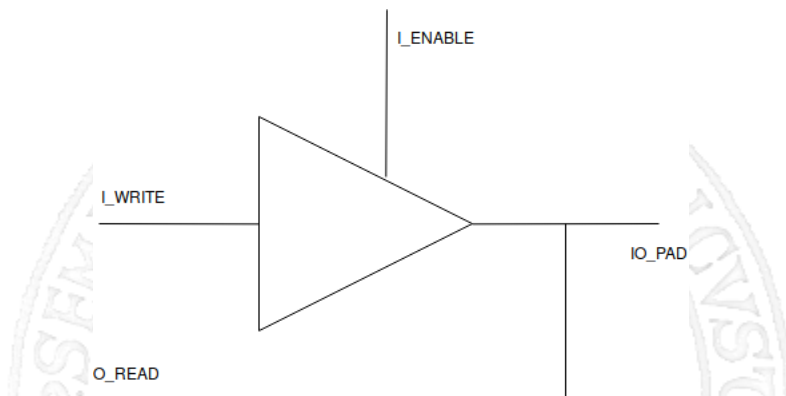


Figura 2.1: Schema singolo GPIO

Il GPIO (*general purpose input output*) permettere di leggere o scrivere un valore sul segnale *PAD* che è un segnale di *inout* per il componente.

Se il segnale di input *ENABLE* è asserito allora *WRITE* viene forzato sul segnale *PAD*, altrimenti il segnale *PAD* è di input per il componente. Il segnale *READ* mostra il valore corrente del segnale *PAD*. Si omette il codice del componente vhd1 in quanto basilare, verrà istanziato un componente GPIO_Array composto da un numero generico (*width*) di singoli GPIO.

2.2.2 Creazione Custom IP

Per utilizzare il componente è necessario che la sua interfaccia sia conforme al bus AXI. Vivado permette di adattare il nostro componente GPIO_Array al bus AXI inglobandolo all'interno di un wrapper che ne implementa l'interfaccia. Si mostra di seguito il procedimento da seguire:

1. Dal menù di Vivado selezionare **“Tools->Create and Package New IP”**. Si aprirà la seguente finestra

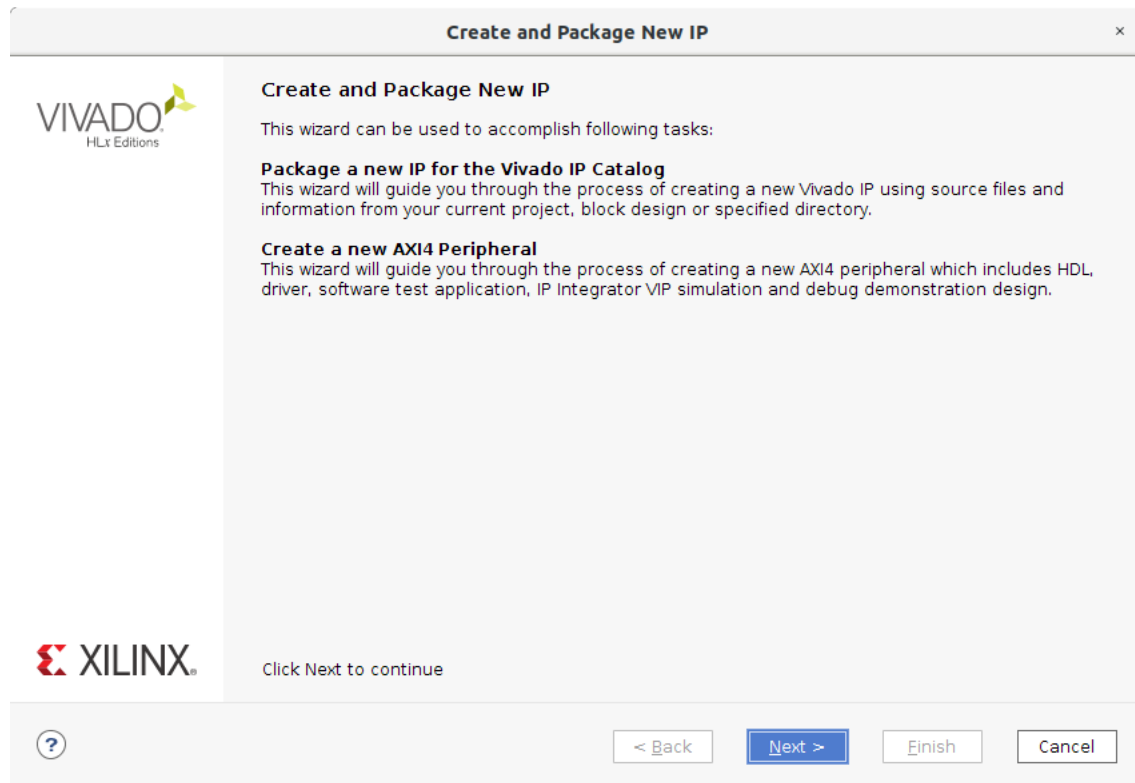


Figura 2.2: Finestra create and package new ip

2. Cliccare su “Next”, selezionare **“Create a New AXI4 peripheral”**, di nuovo “Next”

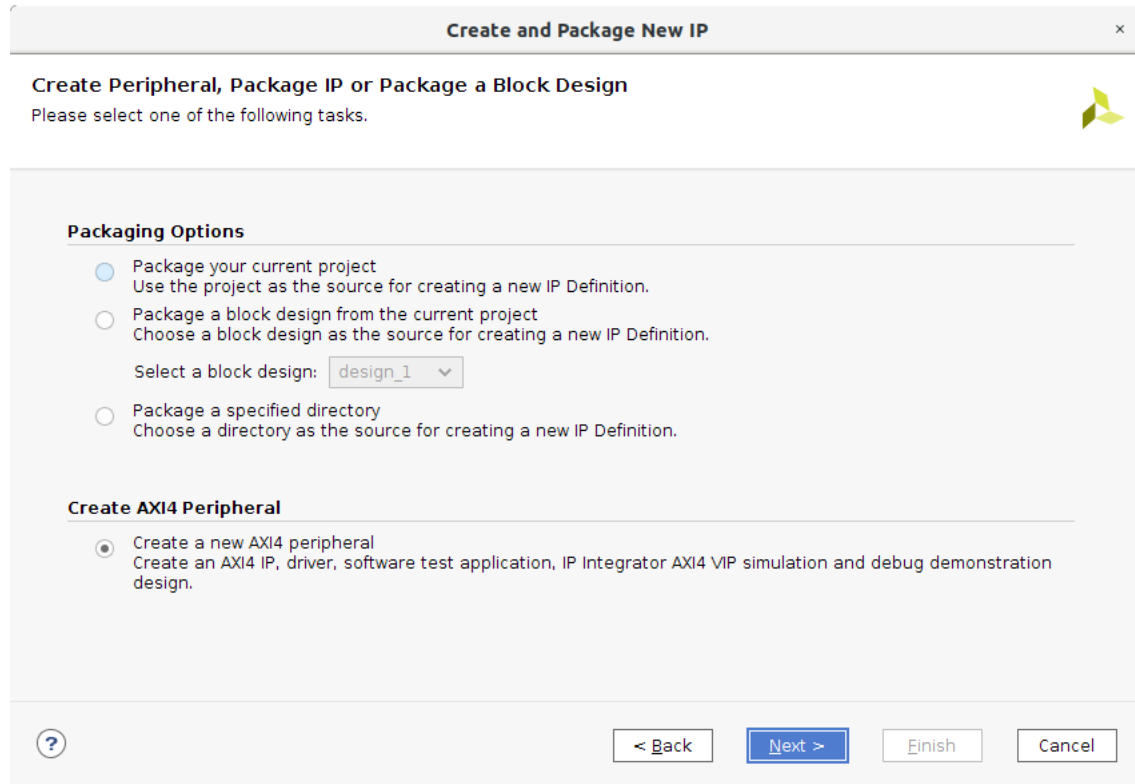
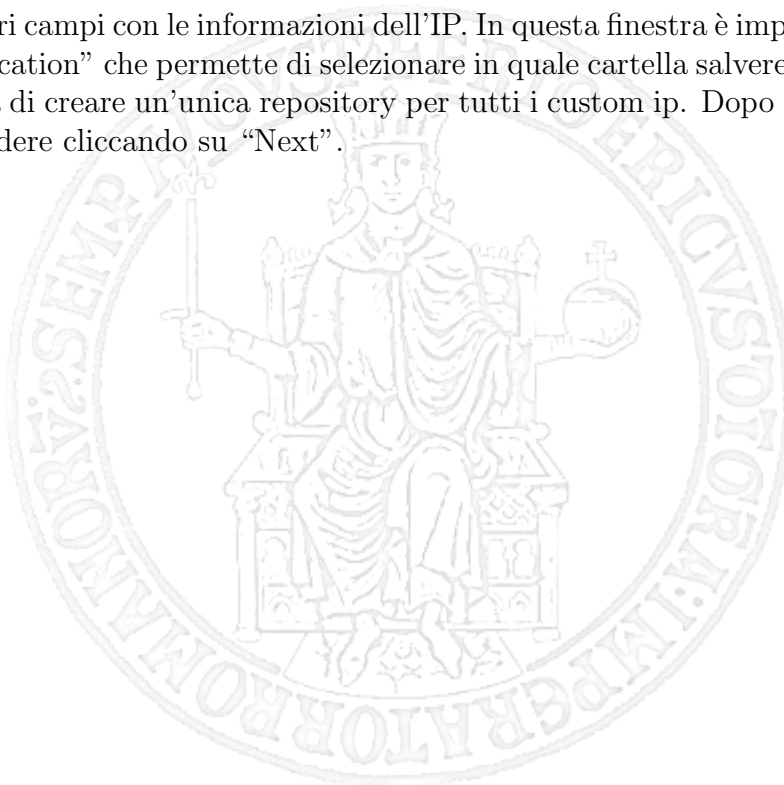
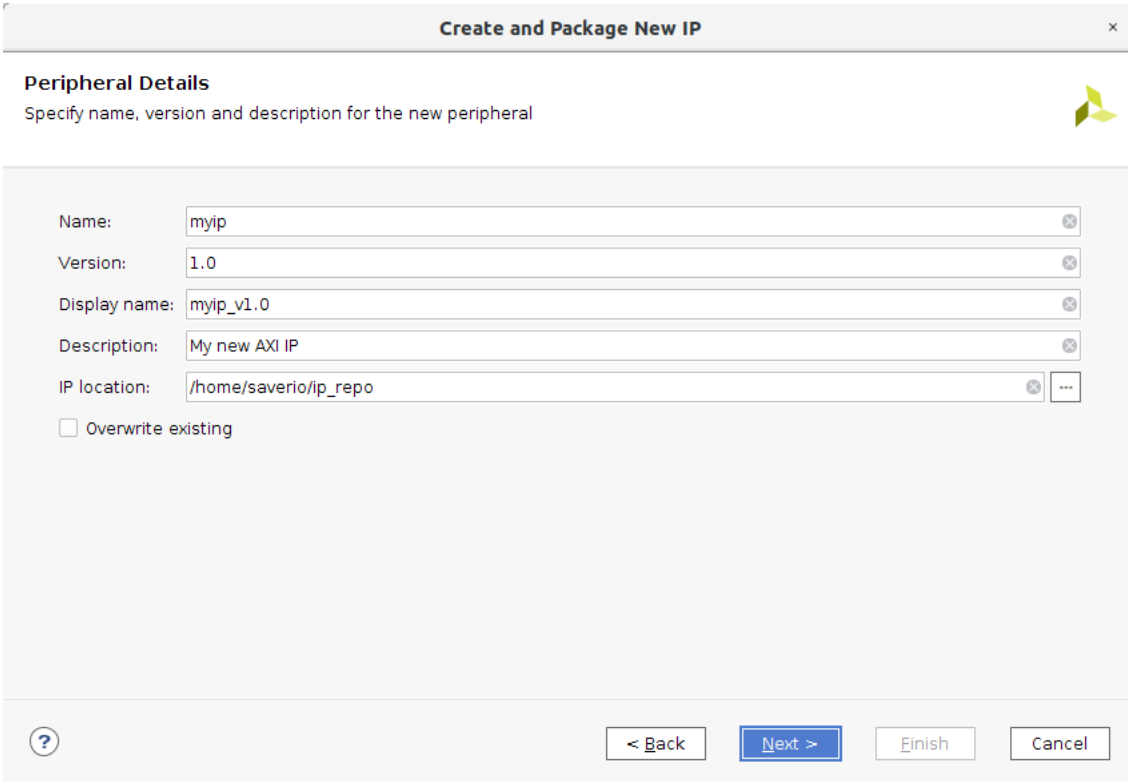


Figura 2.3: Finestra create new AXI Peripheral

3. Compilare i vari campi con le informazioni dell'IP. In questa finestra è importante il box con la dicitura "IP location" che permette di selezionare in quale cartella salveremo il nostro custom IP. Si consiglia di creare un'unica repository per tutti i custom ip. Dopo aver 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

Figura 2.4: Create and Package New IP : Passo 3

- Nella finestra successiva è possibile configurare il tipo di interfaccia del nostro componente. Nel nostro caso l'interfaccia che utilizzeremo è *AXI LITE* e il device deve essere uno Slave in quanto non gestisce le transizioni del bus AXI. Il parametro *number of register* indica il numero di registri (**SLAVE_REG**) che utilizzeremo per interfacciare il nostro componente alla CPU. Nel nostro caso i registri necessari sono tre: uno per pilotare il segnale di WRITE, uno per il segnale di ENABLE e uno per leggere il segnale READ. Si noti che il numero di registri minimo è 4 quindi uno rimarrà inutilizzato, il segnale PAD è un segnale di inout per il componente, non necessita di comunicare con il bus quindi non avrà un registro dedicato. Clicchiamo su "Next" una volta settati i parametri desiderati.

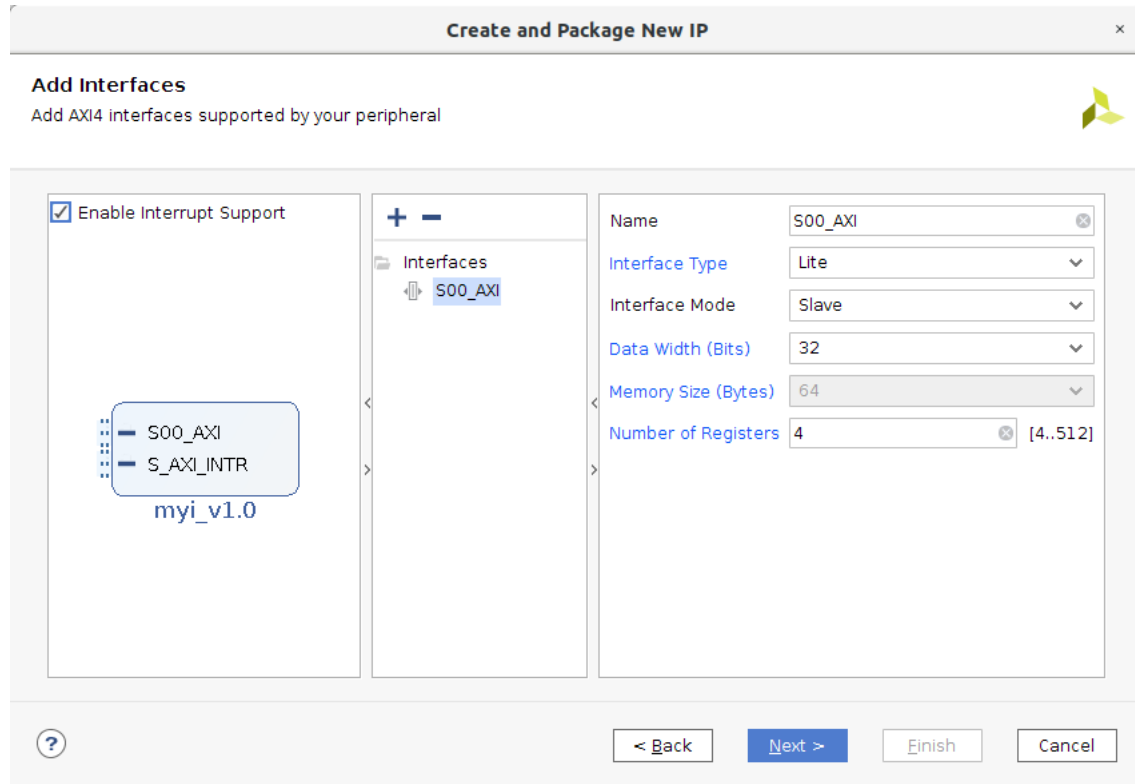
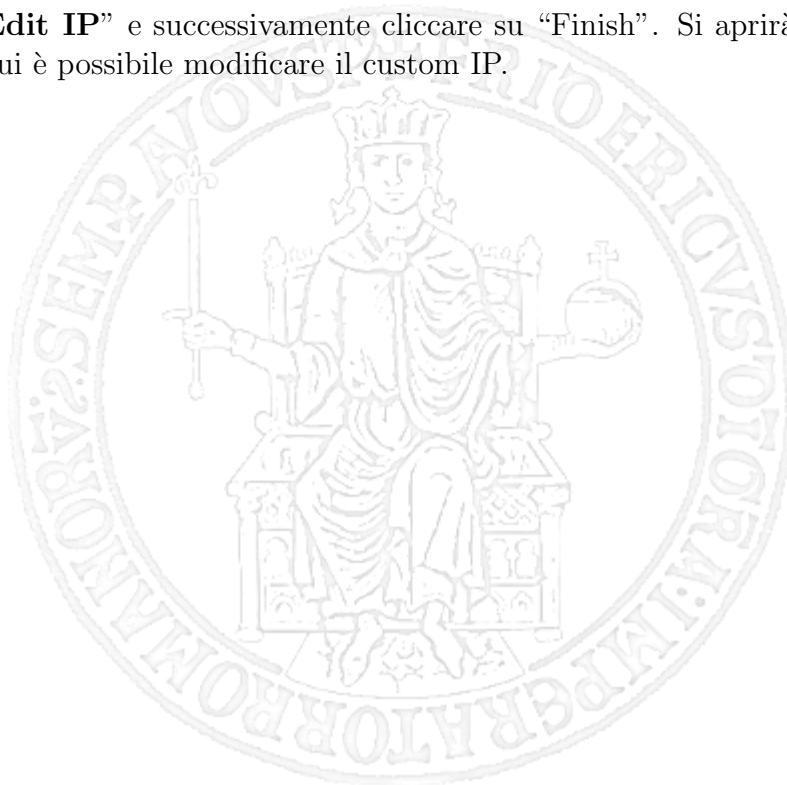


Figura 2.5: Create and Package New IP : Passo 4

5. Selezionare “**Edit IP**” e successivamente cliccare su “Finish”. Si aprirà una nuova istanza di Vivado in cui è possibile modificare il custom IP.



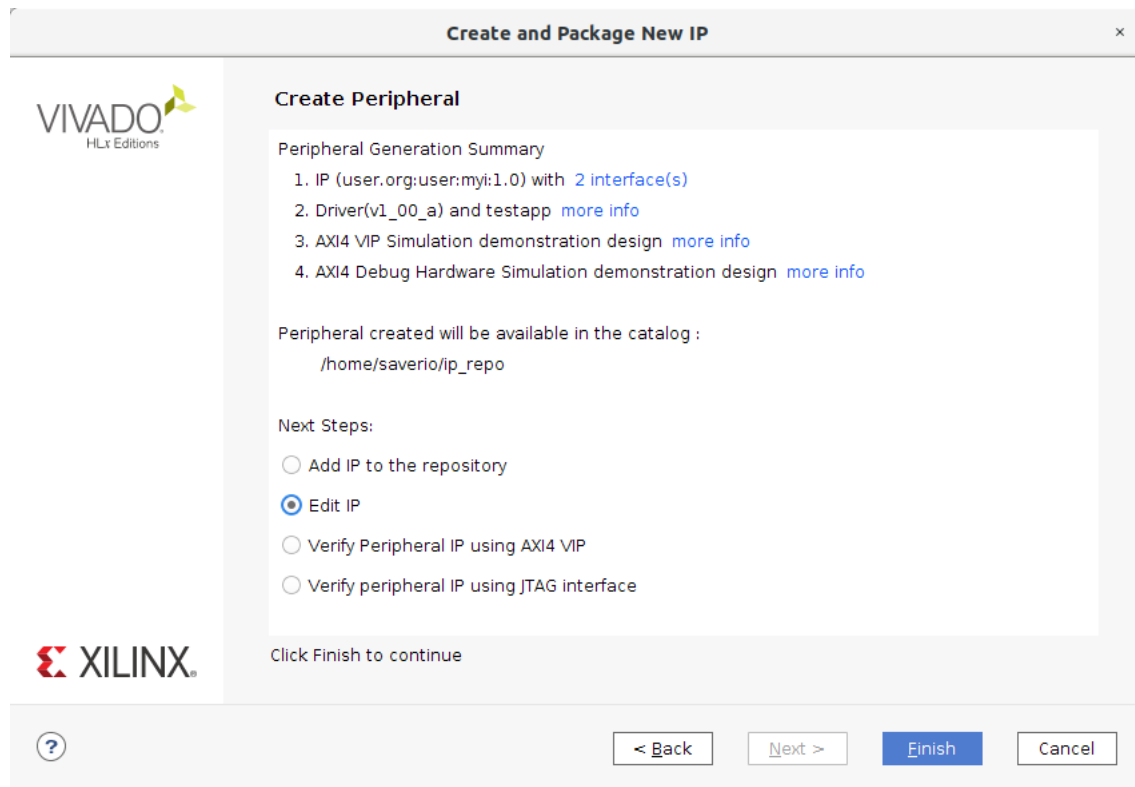
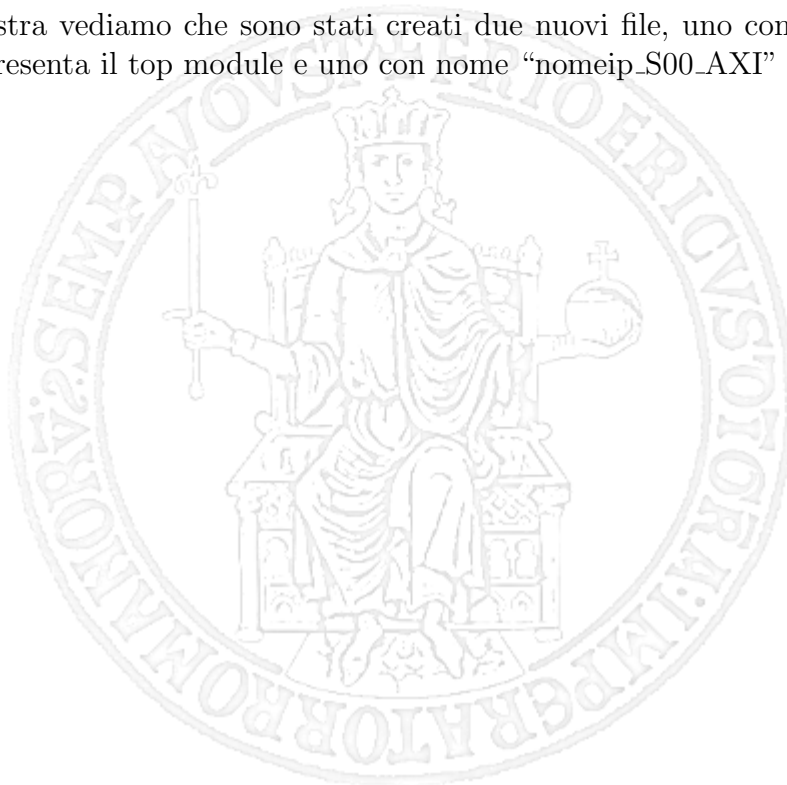


Figura 2.6: Create and Package New IP : Passo 5

Nella nuova finestra vediamo che sono stati creati due nuovi file, uno con il nome del nostro custom IP che rappresenta il top module e uno con nome “nomeip_S00_AXI”



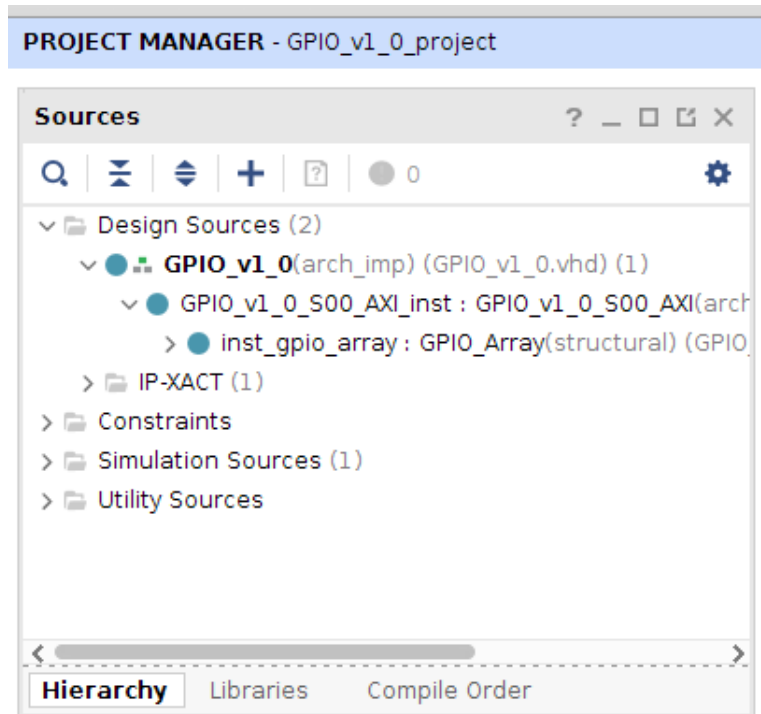
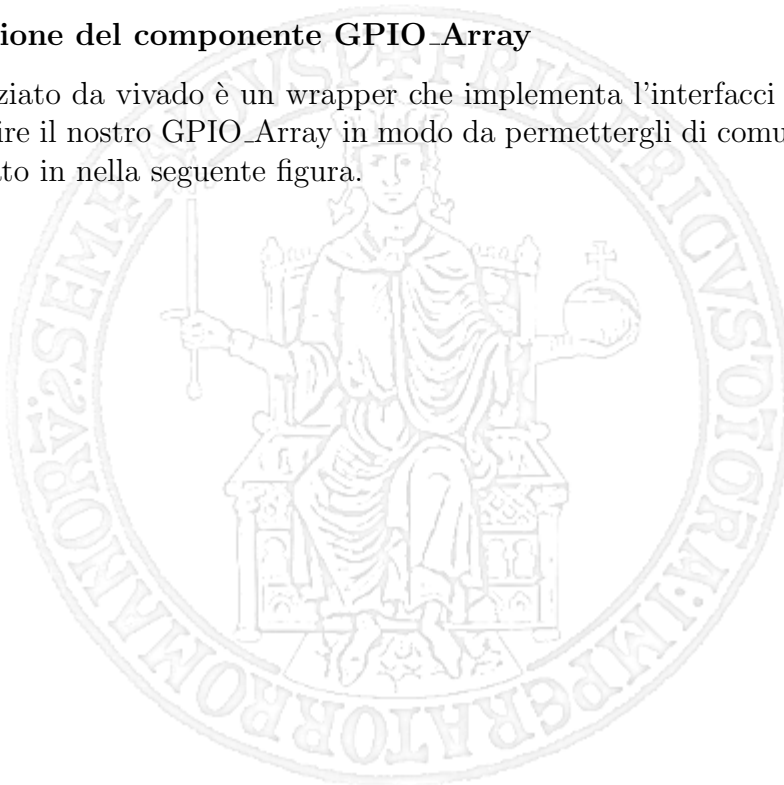


Figura 2.7: Custom IP

2.2.3 Modifica Default IP Core

2.2.3.1 Istanziamento del componente GPIO_Array

Il componente istanziato da vivado è un wrapper che implementa l'interfacci AXI, all'interno del quale dovremo inserire il nostro GPIO_Array in modo da permettergli di comunicare sul bus. Uno schematico è riportato in nella seguente figura.



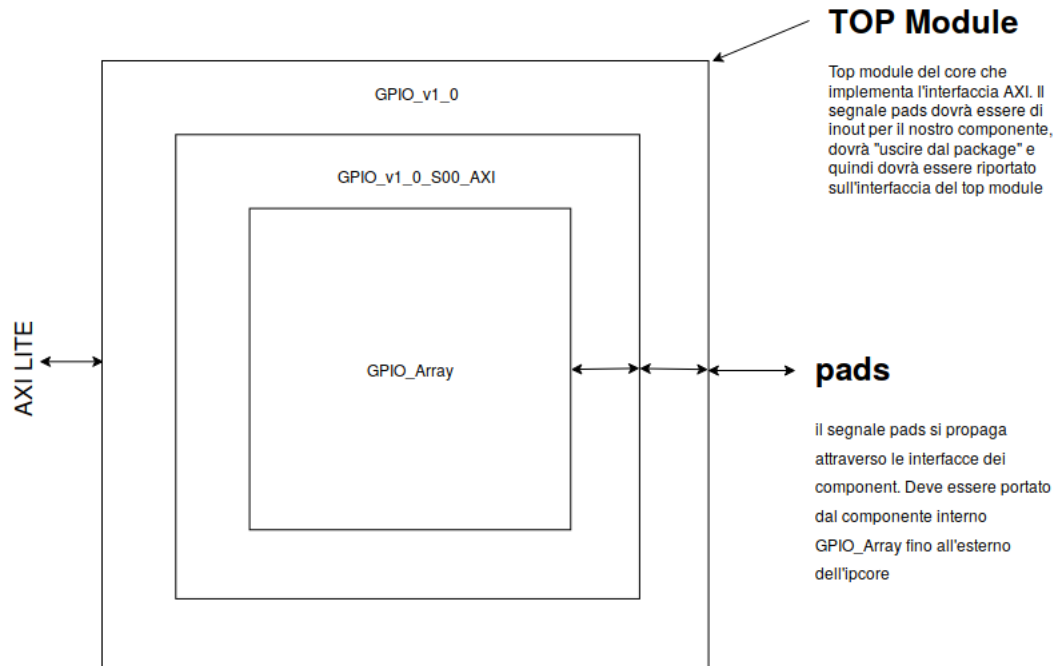


Figura 2.8: Schema IP

Aggiungiamo al design creato da Vivado quello appena creato da noi, i sorgenti vhdl di componenti GPIO_Array ed il singolo GPIO. Clicchiamo su “Add Sources” presente sulla sinistra della interfaccia. Una volta fatto ciò istanziamo il nostro componente (GPIO_Array) nel modulo “GPIO_v1_0_S00_AXI”. E’ necessario farlo nella sezione indicata da Vivado tra i seguenti commenti

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

Prima di tutto è necessario riportare fra le interfacce dei due componenti i segnali che necessitano di comunicare con l'esterno dell'ip e non con il bus axi. Il codice che va aggiunto nelle interfacce dei due componenti è il seguente:

```
1  -- Users to add ports here
2      pads : inout std_logic_vector(width-1 downto 0);
3      interrupt : out std_logic; --! segnale di interrupt
4  -- User ports ends
```

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
```

Si noti che sarà necessario aggiungere i segnali e i parametri sopra riportati anche nell'istanziamento del componente GPIO_v1_0_S00_AXI nel top module.

Superata la prima fase è necessario interfacciare correttamente i segnali del nostro GPIO_Array con il bus e il processore. Per fare ciò il componente utilizza gli *slv_reg*, dei registri con i quali sarà possibile scrivere\leggere i valori dei segnali di ENABLE, WRITE e READ. Collegheremo dunque i segnali di READ e WRITE opportunamente nell'istanziamento del componente come segue:

```

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);

```

I segnali di ENABLE e di WRITE vengono pilotati dai registri *slv_reg0* ed *slv_reg1*. Il valore di READ viene salvato nel registro *gpio_read* perchè per leggerlo, non può essere usato uno degli *slv_reg* generati automaticamente da vivado in quanto READ, essendo un pin di output per il GPIO, forzerebbe dei valori sullo slave reg. Anche il bus axi, tramite un process detto “di scrittura”, forza dei valori sugli slave reg che quindi si ritroverebbero pilotati da due segnali contemporaneamente. Per eliminare questo conflitto viene introdotto un nuovo segnale *gpio_read* che verrà pilotato esclusivamente dal read del nostro GPIO_Array (Figura 2.9).

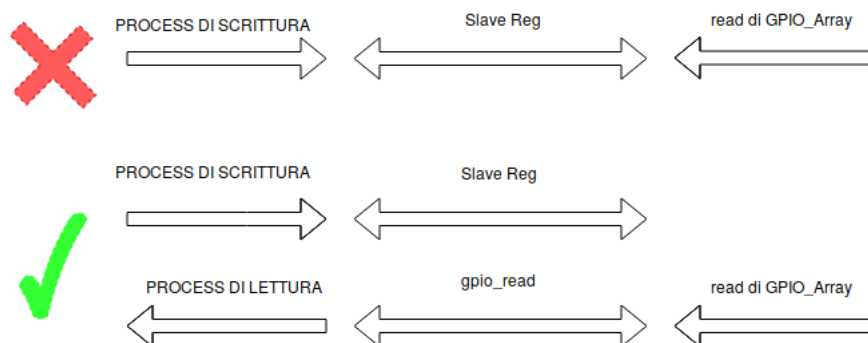


Figura 2.9: Corretta gestione dei segnali di output del componente GPIO_Array

Segue il codice del process “di lettura” modificato opportunamente per leggere il valore di READ:

```

1 process (slv_reg0, slv_reg1, gpio_read, slv_reg3, slv_reg4, slv_reg5,
2          status_reg_out, 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" =>

```

```

12         reg_data_out <= gpio_read;
13     when b"011" =>
14         reg_data_out <= slv_reg3;
15     when b"100" =>
16         reg_data_out <= slv_reg4;
17     when b"101" =>
18         reg_data_out <= slv_reg5;
19     when b"110" =>
20         reg_data_out <= status_reg_out;
21     when b"111" =>
22         reg_data_out <= slv_reg7_out;
23     when others =>
24         reg_data_out <= (others => '0');
25 end case;
26 end process;

```

In questo caso invece di leggere lo *slv_reg2* verrà letto il segnale *gpio_read*. Si noti che questo procedimento va applicato per tutti i segnali di out del componente istanziato.

2.2.3.2 Gestione delle interruzioni

Segue la logica creata per permettere all'ip di lavorare sotto interruzioni. Siamo interessati a generare un evento di interruzione ogni qual volta vi sia una variazione del segnale di READ del GPIO_Array. La variazione deve asserire il segnale di interrupt se e solo se:

- Le interruzioni globali del componente sono abilitate
- La singola linea interna del GPIO_Array è abilitata (*mascherata*) a generare l'interruzione
- Il segnale di READ è pilotato da PADS e non da WRITE

Per catturare le variazioni del segnale READ 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;
13
14 --!  indica quale bit, se mascherato e pilotato da pads, è cambiato
15 changed_bits <= (last_stage xor current_stage) and intr_mask and (not
    gpio_enable);

```



```

16
17  --! il segnale assume valore 1 se e solo se le interruzioni globali sono
18  abilitate e c'è un cambiamento del segnale READ
change_detected <= global_intr and or_reduce(changed_bits));

```

Si noti che siamo interessati a qualunque variazione del segnale read con le relazioni sopra elencate.

Il process viene sintetizzato da Vivado come due Flip-Flop collegati in cascata, necessari per campionare il valore di gpio_read con il clock. Al primo colpo di clock il primo flip flop salva il valore del segnale, al secondo colpo di clock il secondo flip flop conterrà il valore precedentemente salvato nel primo. I due segnali sono posti in xor per vedere se ci sono variazioni. Segue uno schematico della soluzione realizzata (non considerare Q negato):

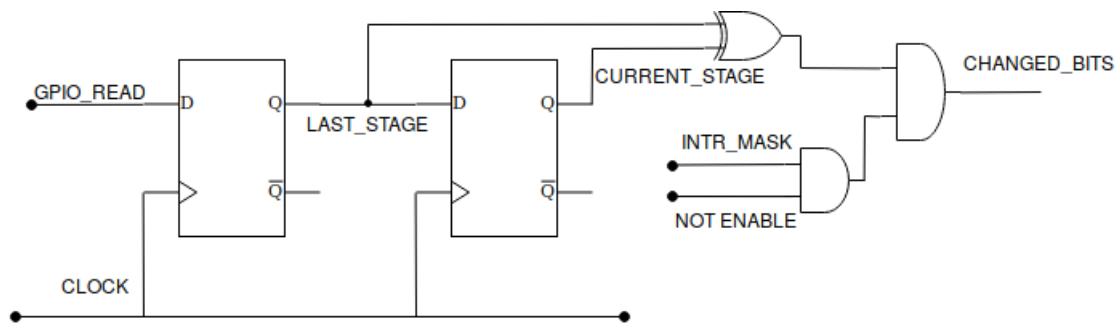


Figura 2.10: Schema rilevamento fronti

Un secondo process viene utilizzato per la gestione del registro delle interruzioni pendenti (*pending_intr*):

```

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

```

Il valore del registro pending è dato da il valore di una precedente interruzione non servita oppure ad una interruzione rilevata (changed_bits). Il registro contiene un 1 nella posizione relativa al GPIO che chiede di generare un interrupt.

Nello stesso process è gestito anche il meccanismo di ack utilizzato per pulire il relativo bit nel registro pending. Il segnale ack_intr deve contenere un 1 nella posizione relativa al gpio al quale si vuol dare l'ack.

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 che va portato all'esterno dell'ip:

```

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' and global_intr = '1')
8           then
9             interrupt <= '1';
10          else
11            interrupt <= '0';
12          end if;
13        end if;
14      end process;

```

Tale segnale è alto nel caso vi siano interruzioni pendenti e le interruzioni globali siano abilitate, invece nel caso di reset del bus, non vi siano interruzioni o queste siano disabilitate 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 device.



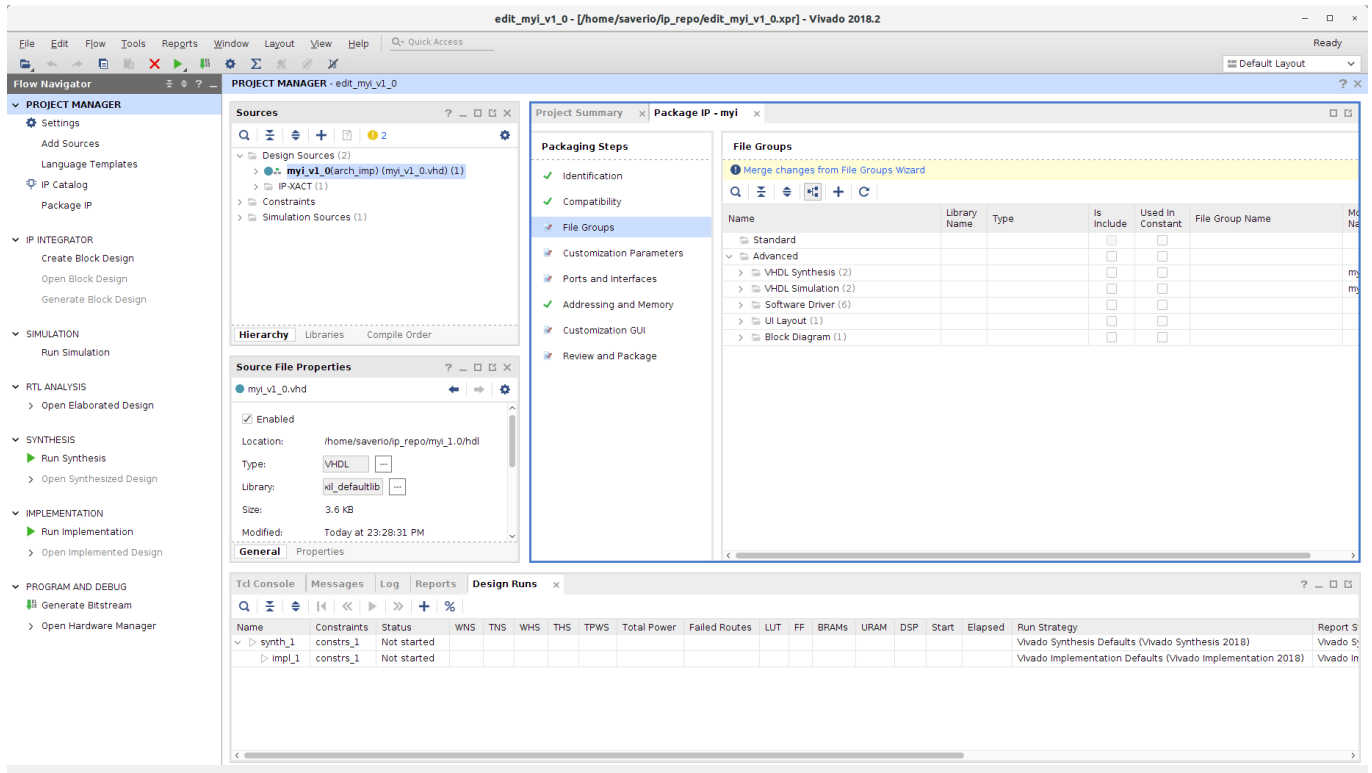


Figura 2.11: Package IP

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:

☐ Visible in Customization GUI

☒ Show Name

Display Name:

Tooltip:

Format:

Editable:

Dependency:

☐ Specify Range

Type:

Press the + button to add a value

Show As:

Layout:

Default Value:

OK Cancel

Figura 2.12

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.

2.2.4 Creazione del block design

Istanziamo 3 IP ognuno con dimensione 4. Serviranno a controllare rispettivamente switch, led e button. Cliccare sul menù di sinistra “**Create block design**” inserire i parametri desiderati e cliccare su “OK” verrà mostrato questo workspace

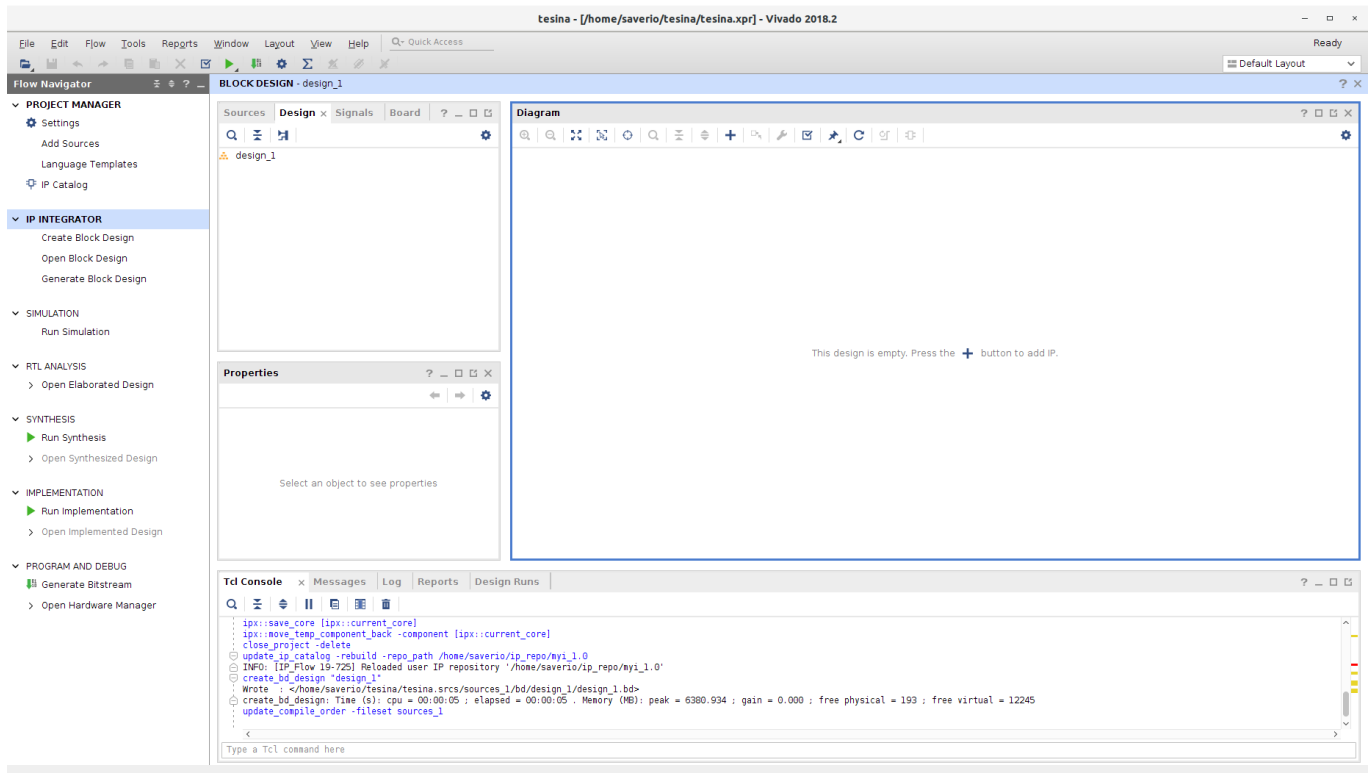


Figura 2.13

Cliccare sul pulsante **+** ed inserire il custom IP creato insieme al processore ZYNQ, appariranno due pulsanti **“Run Block Automation”** e **“Run Connection Automation”** una volta cliccati fanno sì che i vari componenti si collegheranno tra di loro automaticamente. Successivamente è necessario rendere esterni i pin che si vuole pilotare dalla board. Cliccare con il tasto destro del mouse su un pin e selezionare **“Make external”**, successivamente selezionare l’icona contrassegnata da un simbolo di spunta. Il risultato è il seguente:

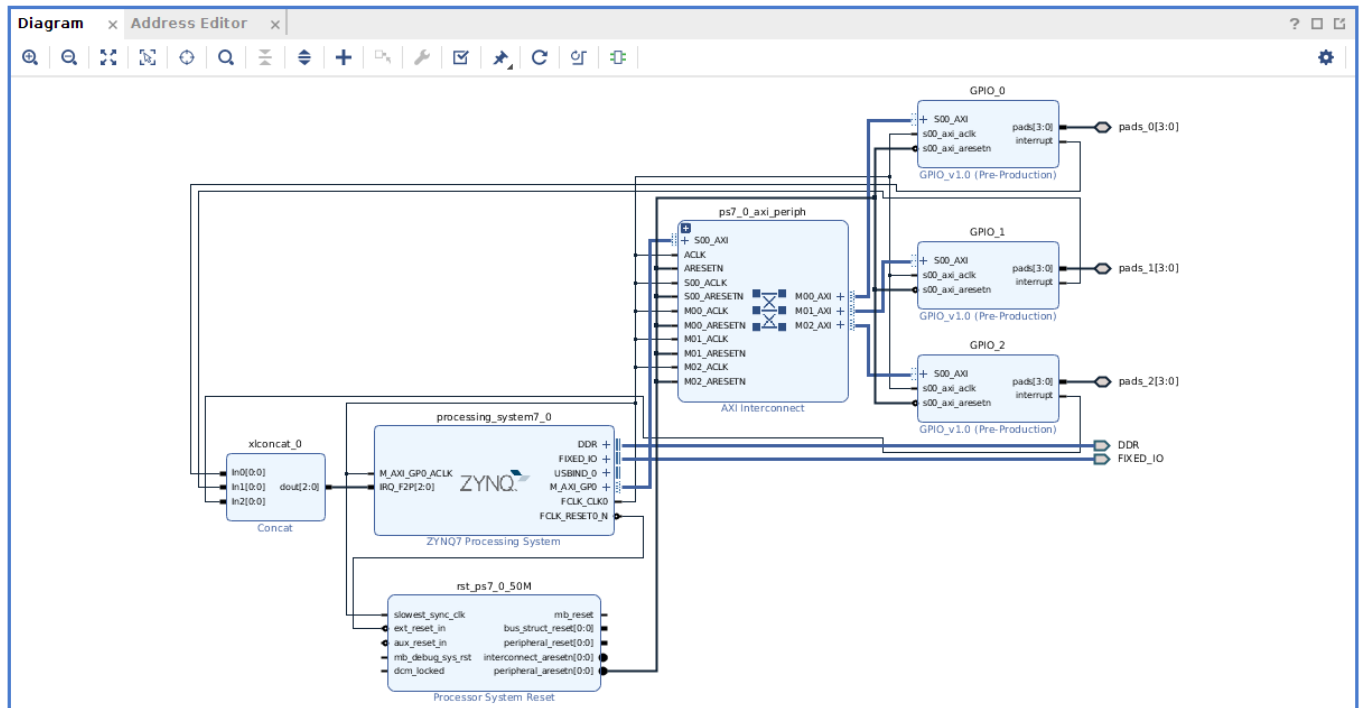
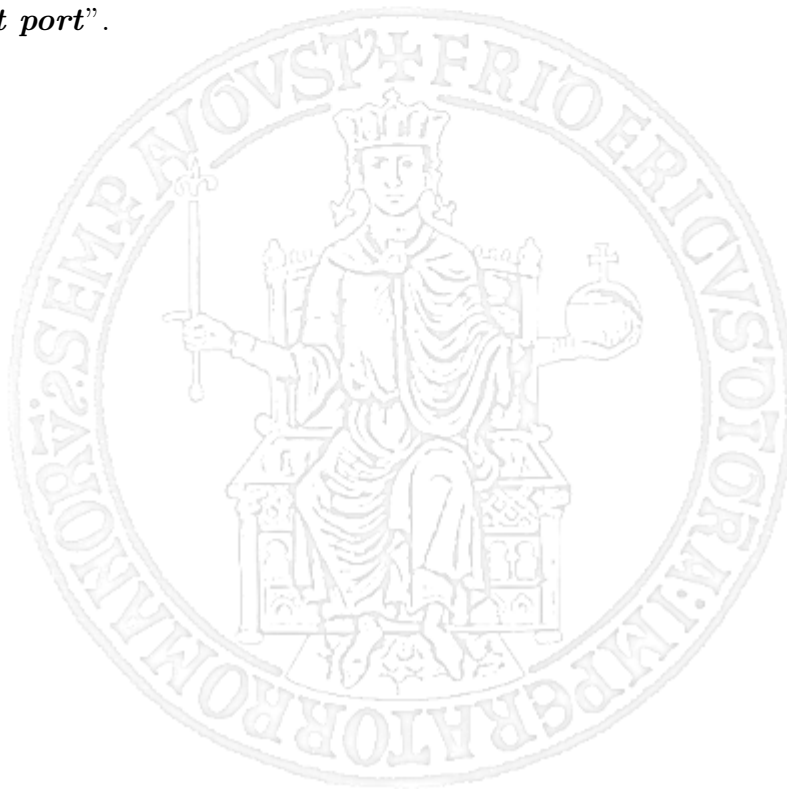


Figura 2.14: Block design completo

Per permettere alla PL di interrompere la PS è necessario abilitare le interruzioni dello zynq processing system. Cliccare due volte sul componente e recarsi nella sezione Interrupt e abilitare “*PL-PS interrupt port*”.



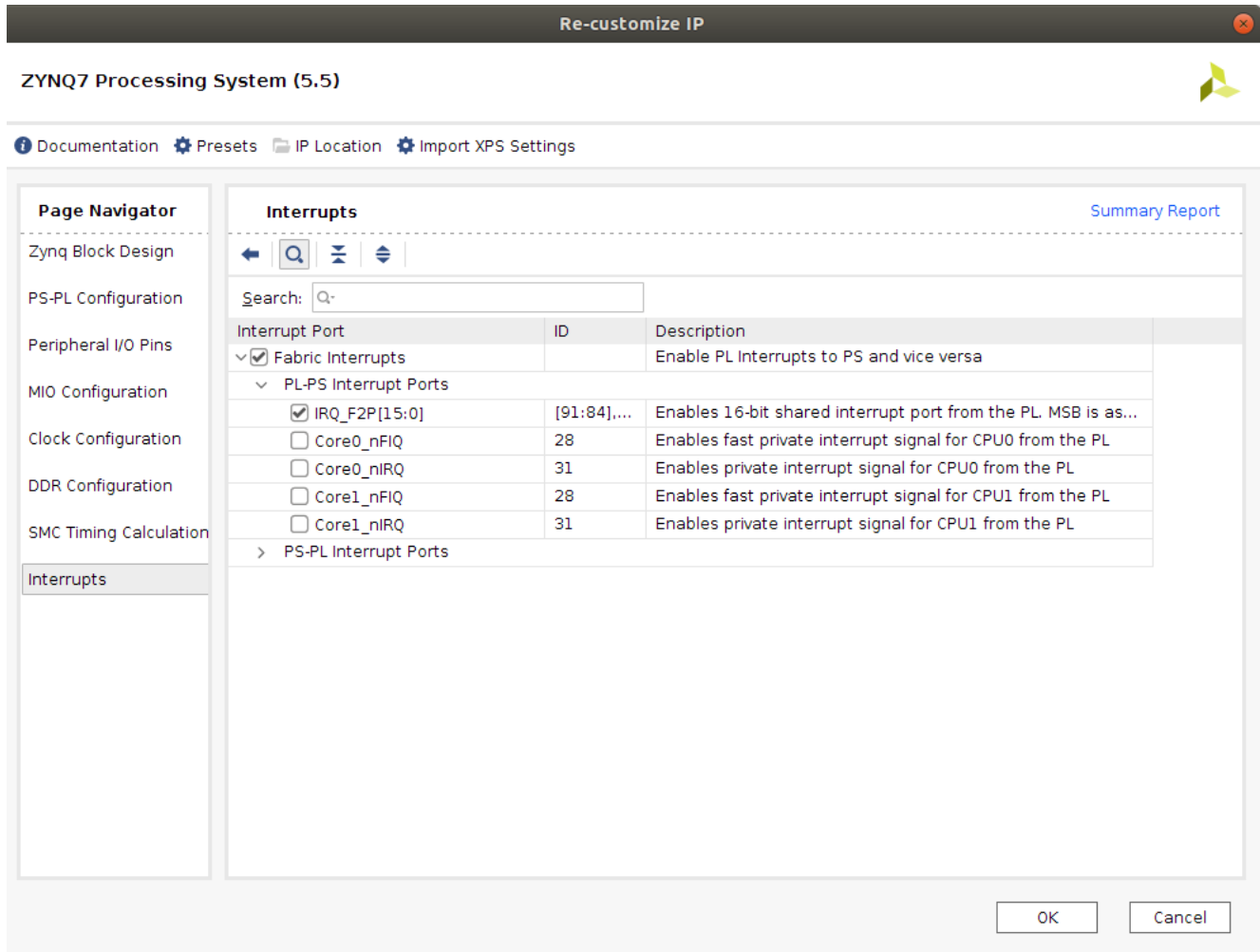


Figura 2.15: Abilitazione interrupt Zynq Processing System

La linea di interrupt che apparirà sul processing system è unica, quindi è necessario utilizzare il componente Concat per collegare i 3 GPIO_Array. Il size verrà automaticamente aggiornato.

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

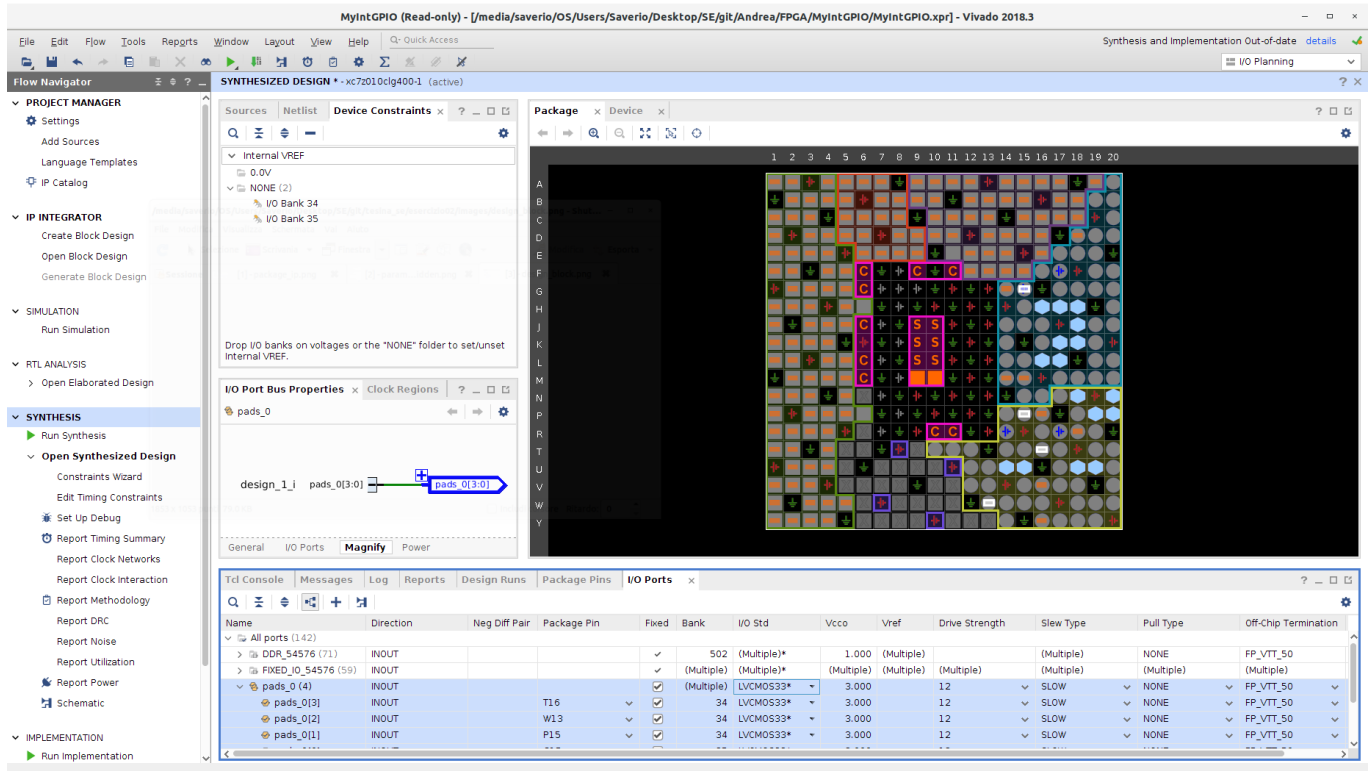


Figura 2.16: Pin dopo la sintesi

In basso sono presenti i pin che sono stati resi esterni, bisogna selezionare il tipo di “I/O Std” usualmente è quello mostrato in figura ed i “Package Pin”, una volta fatto salvare i constraints cliccando sull’ icona del floppy blu e dando un nome al file di constraint, ora 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”.

2.2.5 Driver Standalone

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

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_package\ps7_cortexa9_0\include*”, troveremo una sezione di codice simile alla seguente

```

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.

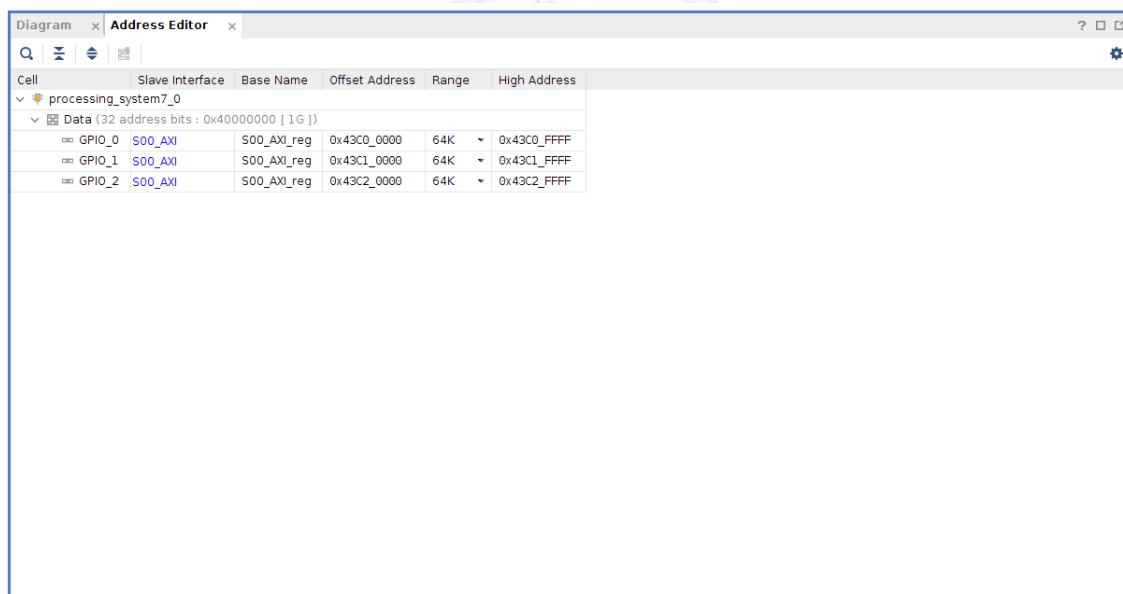
Oltre a tale informazione dobbiamo anche conoscere di quanto sono stati spiazzati i nostri `slv_reg` rispetto all'indirizzo base, tale informazione reperibile nel file situato in “cartella_del_board_support_package\ps7_cortexa9_0\libsrc\nome_del_ip_core_HW\src\nome_del_ip_core_HW.h”, nel quale, una volta aperto, sarà presente una sezione simile:

```

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 gli indirizzi base degli ip si consulti l'Address Editor di Vivado.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
GPIO_0	S00_AXI	S00_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF
GPIO_1	S00_AXI	S00_AXI_reg	0x43C1_0000	64K	0x43C1_FFFF
GPIO_2	S00_AXI	S00_AXI_reg	0x43C2_0000	64K	0x43C2_FFFF

Figura 2.17: Adress delle periferiche

Possiamo procedere alla creazione del driver, istanziamo un nuovo progetto vuoto dal menù “**File -> New -> Application Project**” e creare un progetto vuoto. La PS della Zynq 7000 è composta da un *Cortex-A9* e un *GIC pl390 interrupt controller*. Si mostano per completezza

alcune nozioni sul GIC, il quale dovrà essere gestito da noi non essendoci un sistema operativo di supporto. Si mostra uno schema generico del GIC.

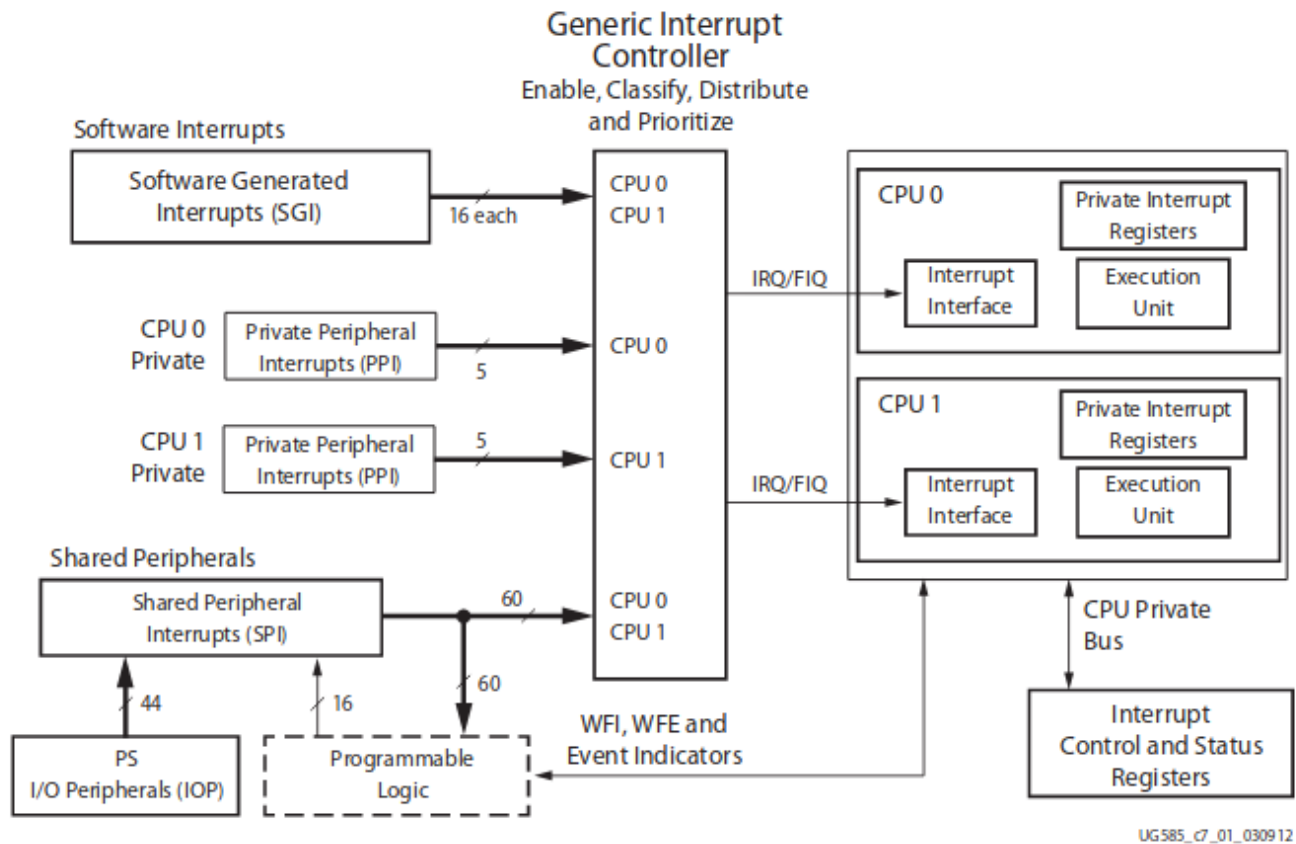


Figura 2.18: Schema logico GIC

L'interruzione che abbiamo bisogno di gestire sono interruzioni provenienti dalla PL e quindi visti dal GIC come interruzioni SPI. Sarebbe possibile indicare, per ogni interruzione, quale dei due Cortex deve gestirla, ma lasciamo questa decisione al GIC. Ogni linea di interrupt è identificata da un ID unico. Per il funzionamento interno riferirsi a **ug585-Zynq-7000-TRM**.

Noi ci interfaceremo solo con il modello di programmazione del GIC utilizzando i driver offerti da Xilinx nella libreria **scugic**.

Il workflow da eseguire per configurare il device è il seguente:

1. Configurare il GIC.

```

1  int Status;
2  /* Istanza del Gic */
3  XScuGic InterruptController;
4
5  /*Istanza della configurazione del Gic*/
6  XScuGic_Config *GicConfig;
7
8  GicConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);

```

```

9
10     Status = XScuGic_CfgInitialize(&InterruptController, GicConfig,
11     GicConfig->CpuBaseAddress);
12     if ( Status != XST_SUCCESS) return XST_FAILURE;

```

Questa fase di inizializzazione è fissa.

2. Abilitare la gestione delle eccezioni relative al GIC (opzionale ma utile)

```

1     Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
2     (Xil_ExceptionHandler)XScuGic_InterruptHandler, &
3     InterruptController);
4     Xil_ExceptionEnable();

```

Questa fase di inizializzazione è fissa.

3. Associare alle 3 le linee di interruzione i relativi handler

```

1     Status = XScuGic_Connect(&InterruptController,
2     XPAR_FABRIC_GPIO_0_INTERRUPT_INTR,
3     (Xil_ExceptionHandler)SwitchISR, (void *)&InterruptController);
4     if ( Status != XST_SUCCESS) return XST_FAILURE;
5
6     Status = XScuGic_Connect(&InterruptController,
7     XPAR_FABRIC_GPIO_1_INTERRUPT_INTR,
8     (Xil_ExceptionHandler)ButtonISR, (void *)&InterruptController);
9     if ( Status != XST_SUCCESS) return XST_FAILURE;
10
11    Status = XScuGic_Connect(&InterruptController,
12    XPAR_FABRIC_GPIO_2_INTERRUPT_INTR,
13    (Xil_ExceptionHandler)LedISR, (void *)&InterruptController);
14    if ( Status != XST_SUCCESS) return XST_FAILURE;

```

La define **XPAR_FABRIC_GPIO_0_INTERRUPT_INTR** indica la ID della linea di interruzione al quale è collegato l'ip GPIO_0. Possiamo trovarla nel file "*xparameters.h*". Il parametro SwitchISR è il nome dell'handler.

4. Enable della linea di interruzione

```

1     XScuGic_Enable(&InterruptController, XPAR_FABRIC_GPIO_0_INTERRUPT_INTR
2     );
3     XScuGic_Enable(&InterruptController, XPAR_FABRIC_GPIO_1_INTERRUPT_INTR
4     );
5     XScuGic_Enable(&InterruptController, XPAR_FABRIC_GPIO_2_INTERRUPT_INTR
6     );

```

Si mostra di seguito una routine d' esempio di gestione dell' interruzione che:

1. Disabilita le interruzioni globali dell'ip.
2. Verifica quale delle linee chiedono di essere servite;
3. Da l'ack alle linee pendenti
4. Riabilita le interruzioni globali del componente;

```

1  void SwitchISR() {
2
3  XGPIO_GlobalDisableInterrupt (&GPIO_Switch, 0x01);
4  InterruptProcessed = TRUE;
5  print ("\n\n*****ISR SWITCH*****\n\n");
6  uint8_t pendingReg = XGPIO_GetPending (&GPIO_Switch);
7  XGPIO_ACK (&GPIO_Switch, pendingReg);
8  XGPIO_GlobalEnableInterrupt (&GPIO_Switch, 0x01);
9  }

```

2.2.6 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.

2.2.6.1 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
7    associato al device; m: utilizzato dal driver per discriminare il
8    singolo device tra quelli a lui associati)*/
9   dev_t Mm;
10  /** Puntatore a struttura platform_device cui l'oggetto GPIO si riferisce */
11  struct platform_device *pdev;
12  /** Struttura per l'astrazione di un device a caratteri */
13  struct cdev cdev;

```

```

12 /** Puntatore alla struttura che rappresenta l'istanza del device*/
13     struct device* dev;
14 /** Puntatore a struttura che rappresenta una vista alto livello del device
15     */
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 effettuale una
34     chiamata a read*/
35     uint32_t can_read;
36 /** Spinlock usato per garantire l'accesso in mutua esclusione alla
37     variabile can_read*/
38     spinlock_t slock_int;
39 } 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
5  *          su cui operare
6  * @param   owner puntatore a struttura struct module, proprietario del
7  *          device (THIS_MODULE)
8  * @param   pdev puntatore a struct platform_device
9  * @param   driver_name nome del driver
10  * @param   device_name nome del device
11  * @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)) !=
40          0) {
41          printk(KERN_ERR "%s: alloc_chrdev_region() ha restituito %d\n", __func__
42              , error);
43          return error;
44      }
45
46      /** Inizializza la struttura cdev specificando la struttura file operations
47          associata al device a caratteri */
48
49      cdev_init (&GPIO_device->cdev, f_ops);
50      GPIO_device->cdev.owner = owner;
51
52      /** Crea il device all'interno del filesystem assegnandogli i numbers
53          richiesti in precedenza e ne restituisce il puntatore. */
54
55      if ((GPIO_device->dev = device_create(class, NULL, GPIO_device->Mm, NULL,
56          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
58     fine sarà possibile vedere il device sotto /dev */
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
66     corrispondente al device */
67
68 dev = &pdev->dev;
69 if ((error = of_address_to_resource(dev->of_node, 0, &GPIO_device->res))
70     != 0) {
71     printk(KERN_ERR "%s: address_to_resource() ha restituito %d\n", __func__,
72         error);
73     goto of_address_to_resource_error;
74 }
75 GPIO_device->res_size = GPIO_device->res.end - GPIO_device->res.start + 1;
76
77 /** Alloca una quantita res_size di memoria fisica per il dispositivo IO a
78     partire dall'indirizzo res.start e ne restituisce l'indirizzo */
79
80 if ((GPIO_device->mreg = request_mem_region(GPIO_device->res.start,
81     GPIO_device->res_size, file_name)) == NULL) {
82     printk(KERN_ERR "%s: request_mem_region() ha restituito NULL\n",
83         __func__);
84     error = -ENOMEM;
85     goto request_mem_region_error;
86 }
87
88 /** Mappa la memoria fisica allocata e restituisce l'indirizzo virtuale */
89
90 if ((GPIO_device->virtl_addr = ioremap(GPIO_device->res.start, GPIO_device
91     ->res_size)) == NULL) {
92     printk(KERN_ERR "%s: ioremap() ha restituito NULL\n", __func__);
93     error = -ENOMEM;
94     goto ioremap_error;
95 }
96
97 /** Cerca le specifiche dell'interrupt nel device tree e restituisce il suo
98     numero identificativo */

```



```

92 GPIO_device->irqNumber = irq_of_parse_and_map(dev->of_node, 0);
93 if ((error = request_irq(GPIO_device->irqNumber, irq_handler, 0,
94     file_name, NULL)) != 0) {
95     printk(KERN_ERR "%s: request_irq() ha restituito %d\n", __func__, error)
96     ;
97     goto irq_of_parse_and_map_error;
98 }
99 GPIO_device->irq_mask = irq_mask;
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"
132         , (unsigned long) 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 ed 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:/#

```

Figura 2.19: Log del kernel dopo *insmod*

Per mostrare la correttezza 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 rmmod my_kernel_GPIO.ko
```

```
root@linaro-developer:/# rmmod 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
```

Figura 2.20: Log kernel dopo rimozioni moduli

2.2.6.2 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
9              * update the entries */
10            clock-names = "s00_axi_aclk";
11            clocks = <&clk 15>;
12            compatible = "generic-uio";
13            interrupt-names = "interrupt";
14            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
21        update the entries */
22     clock-names = "s00_axi_aclk";
23     clocks = <&clkc 15>;
24     compatible = "generic-uio";
25     interrupt-names = "interrupt";
26     interrupt-parent = <&intc>;
27     interrupts = <0 30 4>;
28     reg = <0x43c10000 0x10000>;
29     xlnx,s00-axi-addr-width = <0x5>;
30     xlnx,s00-axi-data-width = <0x20>;
31 };
32 GPIO_2: GPIO@43c20000 {
33     /* This is a place holder node for a custom IP, user may need to
34        update the entries */
35     clock-names = "s00_axi_aclk";
36     clocks = <&clkc 15>;
37     compatible = "generic-uio";
38     interrupt-names = "interrupt";
39     interrupt-parent = <&intc>;
40     interrupts = <0 31 4>;
41     reg = <0x43c20000 0x10000>;
42     xlnx,s00-axi-addr-width = <0x5>;
43     xlnx,s00-axi-data-width = <0x20>;
44 };
45 };
46 };

```

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.

```

root@linaro-developer:~# ls /dev/
block          memory_bandwidth ram6   tty19  tty39  tty59  vcsa1
char           mmcbk0          ram7   tty2   tty4   tty6   vcsa2
console        mmcbk0p1        ram8   tty20  tty40  tty60  vcsa3
cpu_dma_latency mmcbk0p2        ram9   tty21  tty41  tty61  vcsa4
disk           network_latency random tty22  tty42  tty62  vcsa5
fd             network_throughput shm    tty23  tty43  tty63  vcsa6
full           null            snd     tty24  tty44  tty7   vcsu
gpiochip0      port            stderr  tty25  tty45  tty8   vcsu1
iio:device0    ptmx            stdin   tty26  tty46  tty9   vcsu2
initctl        pts            stdout  tty27  tty47  ttyPS0 vcsu3
kmsg           ram0            tty     tty28  tty48  uio0   vcsu4
log            ram1            tty0    tty29  tty49  uio1   vcsu5
loop-control   ram10           tty1    tty3   tty5   uio2   vcsu6
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  vcsa

```

Figura 2.21

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.

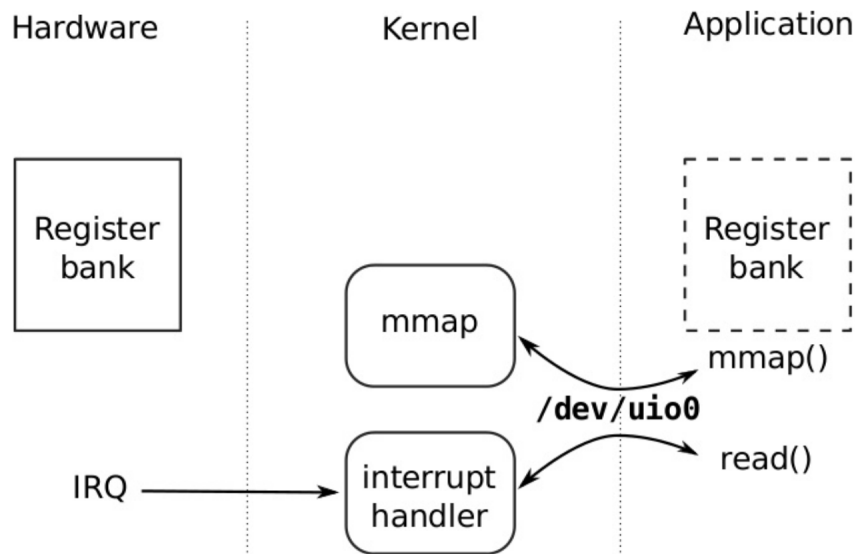


Figura 2.22

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,
124         fd_gpio_0, 0);
125
126     write_reg(gpio_0_ptr, GLOBAL_INTR_EN, 1); // abilitazione interruzioni
127         globali
128     write_reg(gpio_0_ptr, INTR_EN, INTR_MASK); // abilitazione interruzioni
129
130     //-----MAPPING GPIO_1-----//
131
132     int fd_gpio_1 = open("/dev/uio1", O_RDWR);
133     if (fd_gpio_1 < 1){
134         printf("Errore nell'accesso al device UIO1.\n");
135         return -1;
136     }
137
138     gpio_1_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED,
139         fd_gpio_1, 0);
140
141     write_reg(gpio_1_ptr, GLOBAL_INTR_EN, 1); // abilitazione interruzioni
142         globali
143     write_reg(gpio_1_ptr, INTR_EN, INTR_MASK); // abilitazione interruzioni
144
145     //-----MAPPING GPIO_2-----//
146
147     int fd_gpio_2 = open("/dev/uio2", O_RDWR);
148     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,
150     fd_gpio_2, 0);
151
152 write_reg(gpio_2_ptr, GLOBAL_INTR_EN, 1); // abilitazione interruzioni
153     globali
154 write_reg(gpio_2_ptr, INTR_EN, INTR_MASK); // abilitazione interruzioni
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,
159         gpio_1_ptr, gpio_2_ptr);
160 }
161
162 // unmap the gpio device
163 munmap(gpio_0_ptr, dimensione_pag);
164 munmap(gpio_1_ptr, dimensione_pag);
165 munmap(gpio_2_ptr, dimensione_pag);
166
167 return 0;
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, per il quale il processo deve attendere notifiche di eventi dal device espresso in millisecondi.

Capitolo 3

UART

3.1 Traccia

Sviluppo e test di un ipcore che implementa il protocollo UART e comunica su bus AXI. Sintesi del dispositivo su Zynq 7000 e sviluppo dei driver (bare metal, userspace, kernelspace) per utilizzare la periferica in ambiente LINUX.

3.2 Soluzione

3.2.1 Dispositivo UART

L'implementazione del componente UART è stata realizzata seguendo lo schema di un generico dispositivo commerciale, dividendo dunque la logica nei seguenti blocchi (Figura 3.1):

- sezione ricevitore: implementa la logica di ricezione. Quando un byte è ricevuto viene copiato nell'Holding Register e vi rimane fino alla completa ricezione del successivo byte. Al completamento della ricezione il segnale **RDA** viene asserito fino all'inizio di una successiva ricezione.
- sezione trasmettitore: implementa la logica di trasmissione. Il trasferimento viene abilitato asserendo il segnale **TX_ENABLE**. All'inizio dell'trasferimento il segnale tx_busy diviene attivo e vi resta fino alla fine del trasferimento. Per questione di temporizzazione è necessario che il segnale di enable del trasferimento sia un pulse in modo che ritorni automaticamente al valore basso, evitando un nuovo ciclo di trasferimento involontario. Dunque viene utilizzato il componente Level to Pulse che prende in ingresso il segnale di enable esterno e sul rising edge di quest'ultimo produce in uscita un pulse.
- modulazione del clock: componente che prende in ingresso il clock esterno e adegua i clock dei componenti interni per rispettare le velocità imposte dal protocollo.

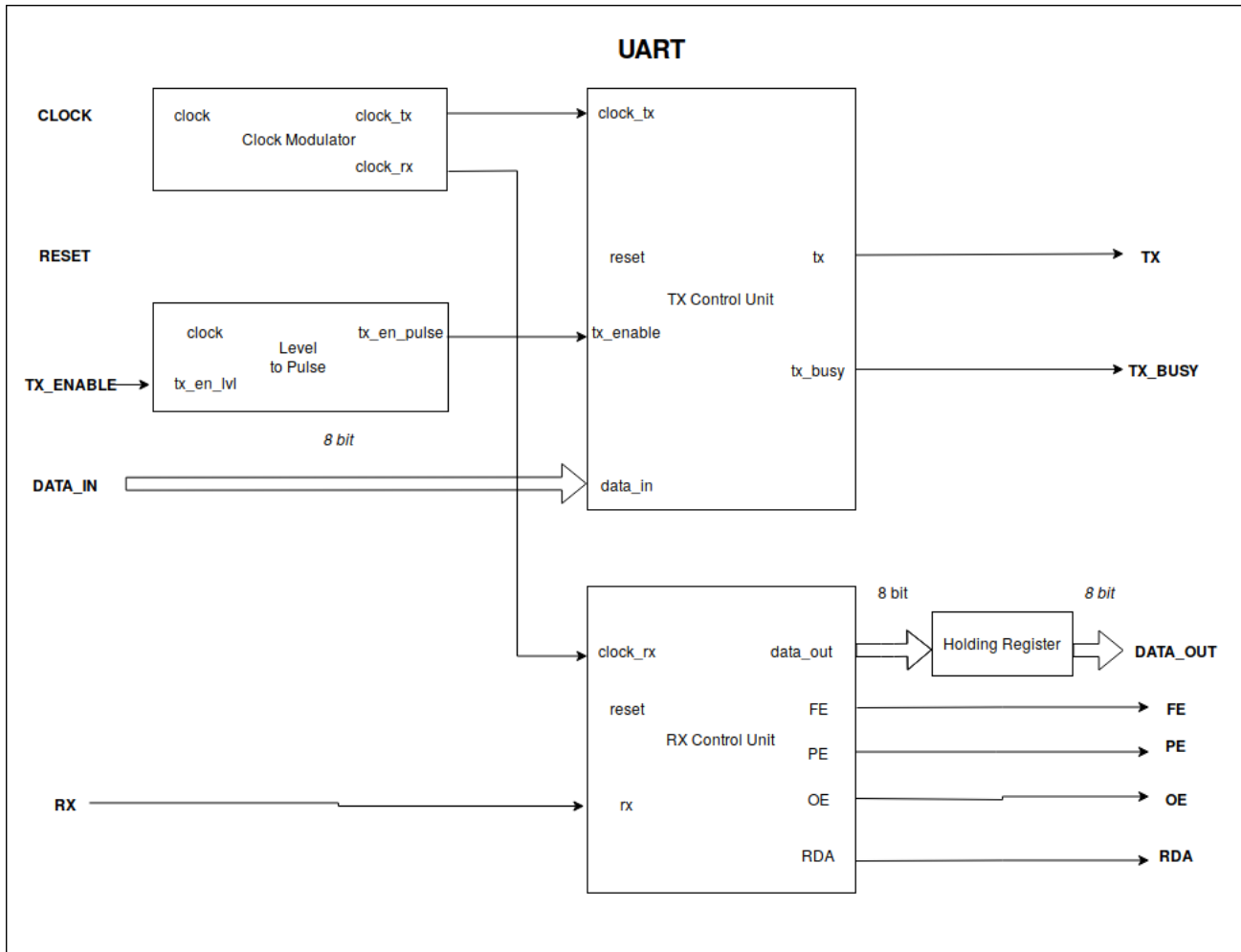


Figura 3.1: Schema a blocchi componente UART

Con riferimento alla Figura 3.1, si stabilisca la seguente convenzione: i segnali in grassetto rappresentano l'interfaccia del componente, quelli sulla sinistra indicano segnali di ingresso, quelli sulla destra di uscita. Alcuni segnali (**clock**, **reset**), avendo multiple destinazioni, non sono collegati per pura questione di comprensione.

3.2.1.1 Sezione Trasmissione

- Shift Register con scorrimento a destra, caricamento parallelo del dato da trasmettere ed uscita seriale per la trasmissione sul canale.
- Contatore Mod 11 incrementato ad ogni bit trasmesso il cui segnale di uscita **counter_done** viene utilizzato dalla control unit per verificare la fine della trasmissione.
- Macchina a stati finiti che implementa la logica di trasmissione del protocollo. Segue un grafo degli stati per descriverne il funzionamento:

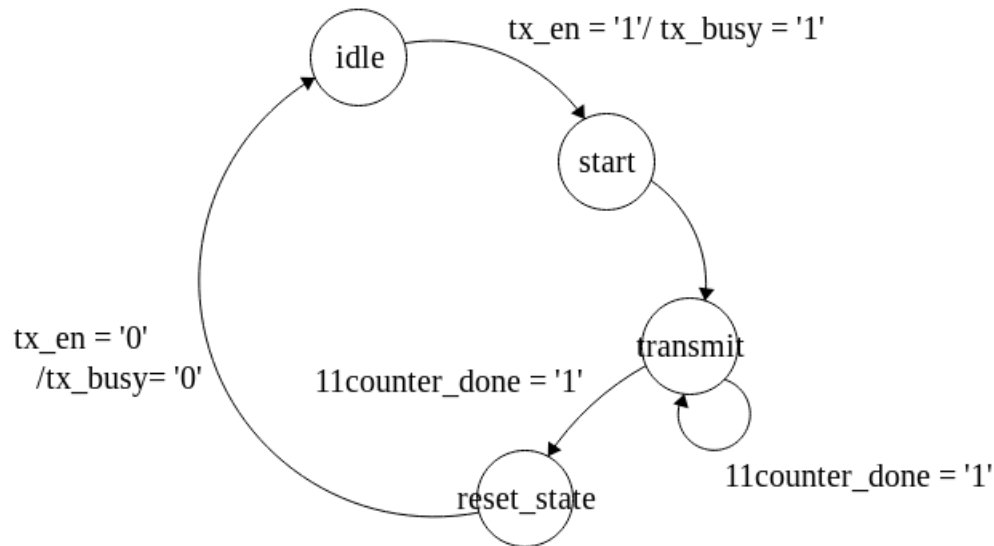


Figura 3.2: Diagramma Stati FSM trasferimento

Il segnale di reset forza la macchina nello stato di idle.

3.2.1.2 Sezione Ricezione

- Shift Register con scorrimento a destra, ingresso seriale del bit ricevuto ed uscita parallela. Si è scelto di connettere l'uscita direttamente ad un Holding Register esterno, il quale viene abilitato dalla FSM solo quando la ricezione è completata e successivamente viene disabilitato per preservare il dato fino all'arrivo del successivo.
- Porte XOR per il calcolo del bit di parità e verifica integrità del frame.
- Contatore Mod 8 utilizzato all'inizio della ricezione per lo sfasamento necessario per effettuare il campionamento della linea di ingresso al centro del bit.
- Contatore Mod 16 utilizzato per il campionamento dei bit.
- Contatore Mod 10 utilizzato per tenere traccia del numero dei bit già ricevuti. Dimensionato a 10 in quanto non viene memorizzato il bit di start.
- Macchina a stati finiti che implementa la logica di ricezione del protocollo. Segue un grafo degli stati per descriverne il funzionamento:

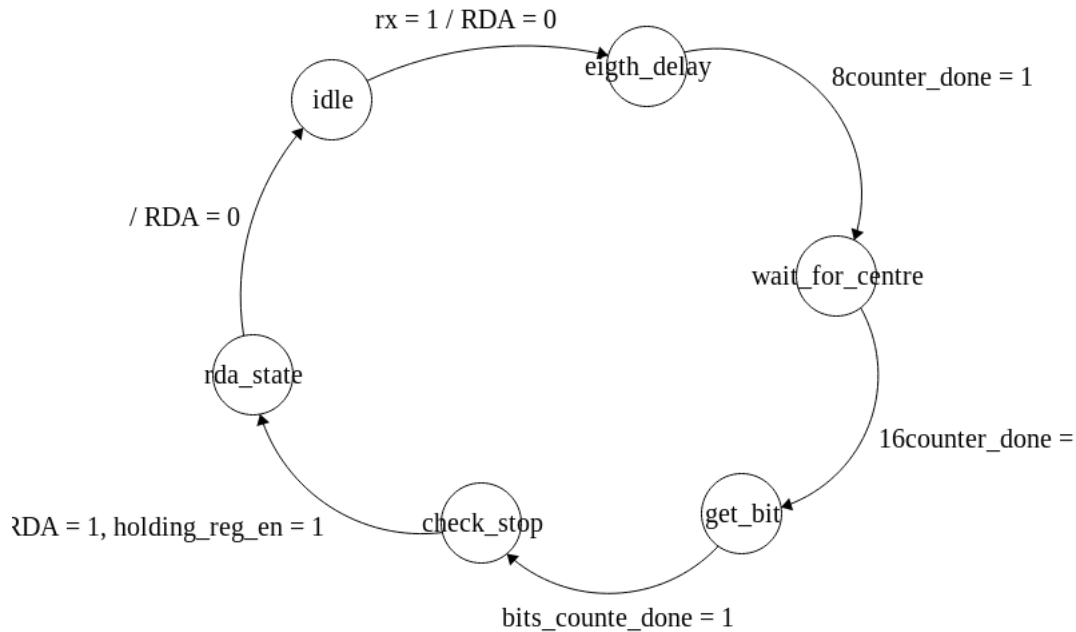


Figura 3.3: FSM ricezione

Si è scelto, per questioni di temporizzazione e per fornire un modello di programmazione adeguato, aggiungendo uno stato `rda_state`. Questo perchè quando il carattere è stato ricevuto completamente si passa dallo stato `get_bit` a quello `check_stop` dove viene dato l'enable dell'holding register nel quale verrà copiato il dato. Dunque ci si riserva un ciclo di clock per effettuare questa operazione e alzare RDA nello stato successivo quando il dato è già stato copiato nell'holding register.

Naturalmente, il segnale di reset forza la macchina a tornare nello stato `idle`.

3.2.1.3 Modulazione del Clock

Il componente UART prende in ingresso due parametri generic che indicano il Baud Rate e la frequenza del clock che fa da base dei tempi. È stato realizzato un componente Clock Mod che rallenta la frequenza di ingresso del doppio del parametro dato in ingresso dal Generic. Il seguente componente, con parametri differenti è stato utilizzato sia per il Baud Rate sia per gestire la diversa tempificazione di ricevitore e trasmettitore a partire dalla stessa base dei tempi.

- Baud Generator: prende in ingresso la costante `BaudDivide`, calcolata come $BaudDivide = \frac{freq_{in}}{BaudRate * 16 * 2}$
- Tx clock Mod: prende in ingresso 8 per far sì che la frequenza del clock del trasmettitore sia 16 volte più lenta di quella del ricevitore.

3.2.2 Custom AXI IP Core

Si procede dunque alla creazione di un custom IP Core come mostrato nel precedente capitolo. Verranno istanziati due componenti:

1. **my_uart_intv1**: top module dell'IP. Il segnale TX è in out all'ip mentre quello di RX in ingresso.
2. **my_uart_int_v1_0_S00_AXI**: si occupa dell'interfacciamento del componente **UART** con il bus per la logica di trasmissione da e verso il processore. Intefaccia i segnali TX e RX dal componente UART al top module. Gestisce la logica di interruzione della periferica.

3.2.2.1 my_uart_int_v1_0_S00_AXI

Segue una tabella degli indirizzi dei registri utilizzati dal componente

Nome	Offset	Map bit->segnali	DIR
TX_DATA	0	TX_DATA[7..0]	W
TX_ENABLE	4	TX_EN[0]	W
UART_STATUS_REG	8	TX_BUSY[4] RDA[3] PE[2] FE[1] OE[0]	R
RX_DATA	12	RX_DATA[7..0]	R
GLOBAL_INT_ENALBE	16	GBL_INT_EN[0]	W
INT_ENABLE_MASK	20	INT_MASK[1..0]	W
PENDING_INT/ACK	28	INTR_PEND[1..0]/ACK[1..0]	R/W

Tabella 3.1: Mapping indirizzi

Si omette la connessione del componente UART in quanto basilare e riassumibile tramite la Tabella 3.1.

Il componente genera il segnale di interrupt se è stato completato il trasferimento di un carattere (*falling edge di TX_BUSY*) oppure se ne è stata completata la ricezione (*rising edge di RDA*).

Si mostra la porzione di codice VHDL che consente la rilevazione di una delle due condizioni.

```

1  --! process utilizzato per captare variazione dei segnali RDA(bit 3) e
    tx_busy(bit 4)
2  --! la sintesi da due FF in cascata
3  status_reg_sampling : process (S_AXI_ACLK, uart_status_reg)
4  begin
5  if (rising_edge (S_AXI_ACLK)) then
6      if ( S_AXI_ARESETN = '0' ) then
7          last_stage <= (others => '0');
8          current_stage <= (others => '0');
9      else
10         last_stage <= uart_status_reg(4 downto 3);
11         current_stage <= last_stage;
12     end if;

```

```

13     end if;
14     end process;
15
16
17     tx_busy_falling_detect <= not last_stage(1) and current_stage(1);
18     --! detect falling edge tx_busy
19     rx_rising_detect <= not current_stage(0) and last_stage(0);
20     --! detect rising edge RDA
21
22     changed_bits <= (rx_rising_detect & tx_busy_falling_detect)
23         and intr_mask;
24     --! and con la intr_mask perchè sono interessato a vedere l'edge del
25         segnale
26     --! solo se la relativa interruzione è abilitata
27
28     change_detected <= global_intr or_reduce(changed_bits);
29     --! Segnale che indica se è stato rilevata una variazione di tx_busy o
        RDA
        --! alla quale si è interessati

```

Siamo interessati a rilevare uno dei due edge solo se le due linee di interruzione sono abilitate (INTR_MASK) e se sono abilitate le interruzioni globali del componente. Segue il process di gestione delle interruzioni pendenti e dell'ack. Se viene rilevata una nuova richiesta di interruzione su una delle due linee essa viene aggiunta alle precedenti interruzioni pendenti. Se viene dato un ack per la specifica interruzione essa viene rimossa da quelle pendenti.

```

1
2     pending_intr_tmp <= pending_intr;
3
4     --! process per la gestione della logica di interruzione pendente
5     --! e meccanismo di ack per rimuovere l'interruzione pendente
6     intr_pending : process (S_AXI_ACLK, change_detected, ack_intr)
7     begin
8         if (rising_edge (S_AXI_ACLK)) then
9             if (change_detected = '1') then
10                 --! se c'è richiesta di interruzione su una delle due linee
11                 pending_intr <= pending_intr_tmp or changed_bits;
12                 --! aggiungi la richiesta alle interruzioni pendenti
13             elsif (or_reduce(ack_intr)='1') then
14                 --! se viene dato un ack
15                 pending_intr <= pending_intr_tmp and (not ack_intr);
16                 --! rimuovi la richiesta pendente relativa
17             else
18                 pending_intr <= pending_intr_tmp;
19                 --! altrimenti il segnale resta al suo valore corrente
20             end if;
21         end if;
22     end process;

```

Segue il codice per la gestione dell'unico segnale di interrupt uscente dall'IP Core. Per capire quale delle due linee interne ha generato la richiesta di interruzione alla CPU dovrà essere verificato tramite software

```

1
2  --! process per gestire l'unica linea di interruzione
3  --! in uscita dal componente
4  inst_irq : process(S_AXI_ACLK,pending_intr, global_intr)
5  begin
6      if (rising_edge (S_AXI_ACLK)) then
7          if ( S_AXI_ARESETN = '0' ) then
8              interrupt <= '0';
9          else
10             if (or_reduce(pending_intr) = '1' and global_intr = '1')
11                 then
12                 --! Se c'è almeno un interruzione pendente e globali sono abilitate
13                     interrupt <= '1';
14                 --! interrupt = '1'
15             else
16                 interrupt <= '0';
17                 --! altrimenti 0
18             end if;
19         end if;
20     end if;
21 end process;

```

Si mostra di seguito l'andamento dei segnali che gestiscono il processo di interruzione del componente. Si consideri i segnali TX e RX rispettivamente il bit 0 ed il bit 1 degli omonimi segnali.

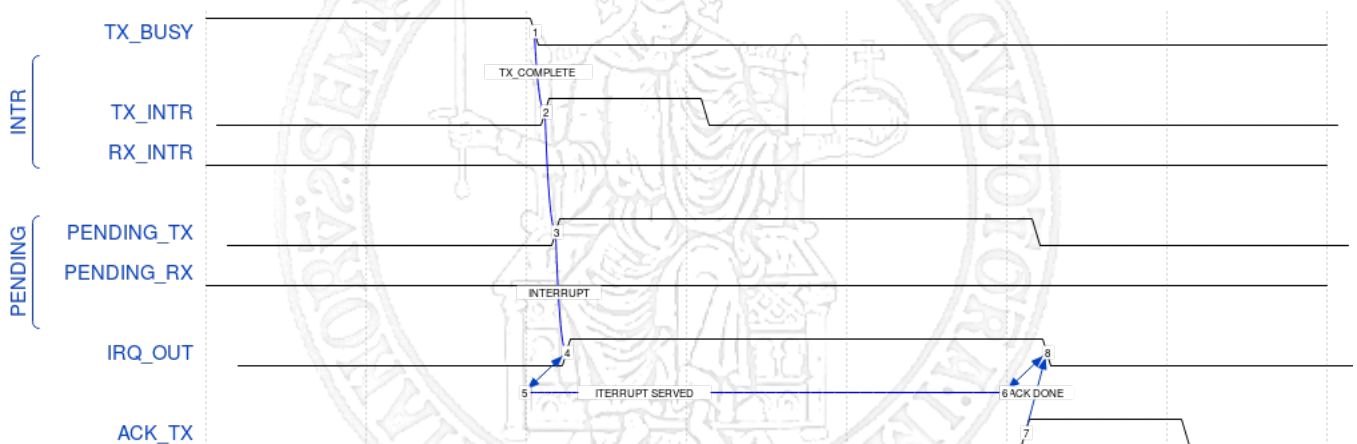


Figura 3.4: Gestione interruzioni trasmissione

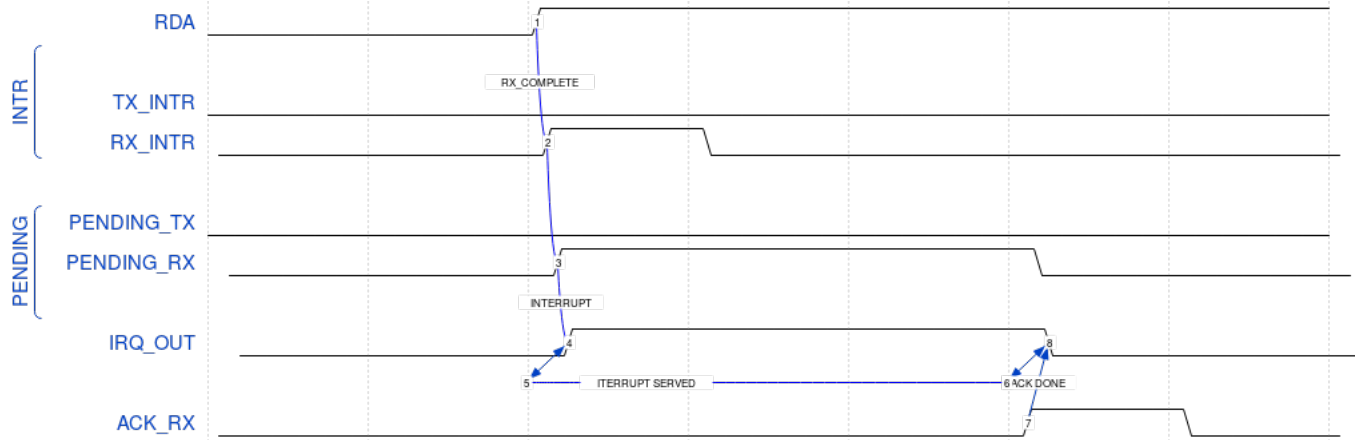


Figura 3.5: Gestione interruzioni ricezione

3.2.3 Design

Per testare il componente si è scelto di inserire due dispositivi identici UART nel Block Design. UART_1 sarà utilizzato esclusivamente da trasmettitore, UART_0 da ricevitore. Per collegare i due segnali di interrupt è stato utilizzato il componente Concat.

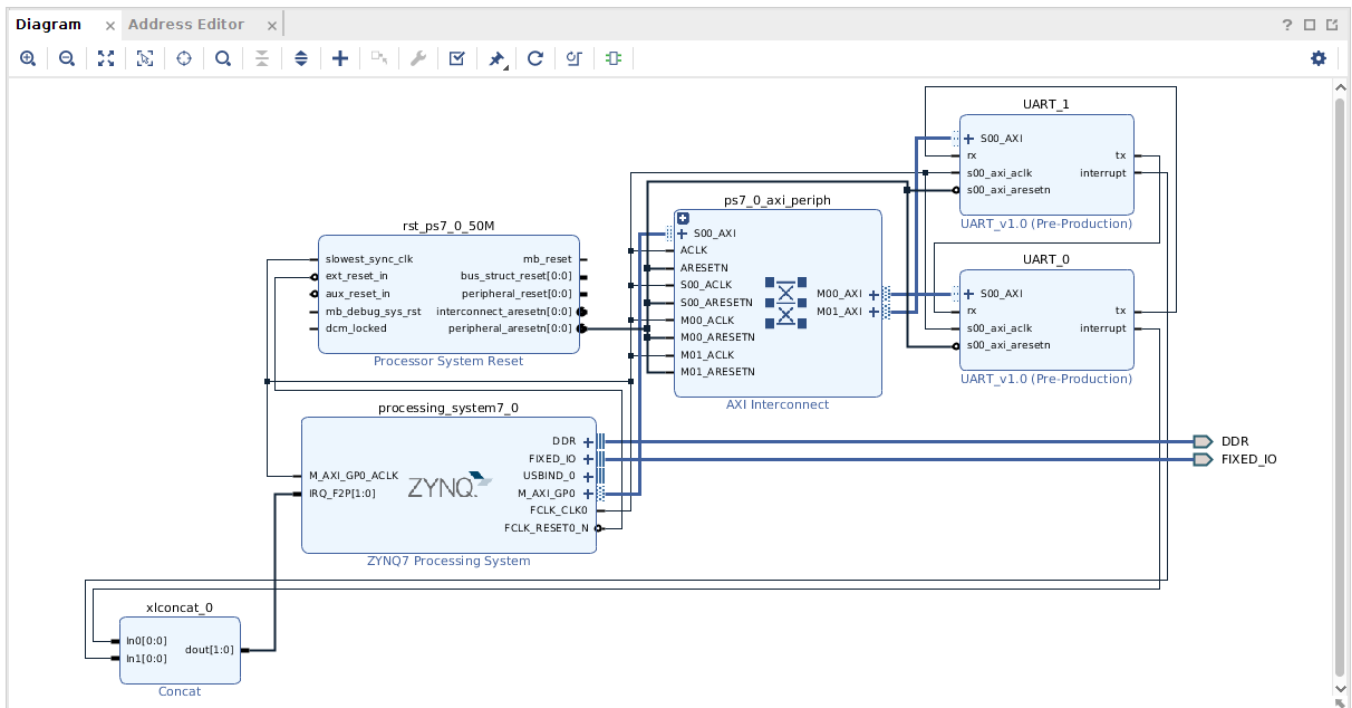


Figure 3.6: Block Design

3.2.4 Driver Standalone

Il driver Standalone è stato realizzato per verificare il comportamento basilare del dispositivo. Come applicativo di test si è scelto di realizzare un main che esegue cinque invii dello stesso

carattere. Il corretto funzionamento può essere verificato controllando le variabili “count” di debug e i registri della periferica. Per la stesura delle librerie di gestione delle periferiche si rimanda alla documentazione del codice. Il primo carattere viene inviato nel main del programma, i successivi nell’ISR del trasmettitore. Si noti che, avendo due linee di interruzione con la stessa priorità attribuita dal software, il GIC per risolvere eventuali conflitti nel caso in cui le due interrupt si verificano insieme, *assegna priorità maggiore alla linea con ID più basso* (parametro `XPAR_FABRIC_UART_X_INTERRUPT_INTR`). In questo caso il ricevitore (UART_0), collegato alla linea 61 del GIC, avrà maggiore priorità. Si riporta di seguito il main del programma.

```

1  #include "myuart.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "xil_io.h"
5  #include "xil_exception.h"
6  #include "xparameters.h"
7  #include "xil_cache.h"
8  #include "xil_printf.h"
9  #include "xil_types.h"
10 #include "xscugic.h"
11 #include "xil_cache_l.h"
12
13 /***** Constant Definitions
14  *****/
15 #define INTC_DEVICE_ID      XPAR_SCUGIC_0_DEVICE_ID
16
17 #define UART0_BASE_ADDR    0x43C00000
18 #define UART1_BASE_ADDR    0x43C10000
19
20 #define INT_MASK_TX        0x1
21 #define INT_MASK_RX        0x2
22 /***** Type Definitions
23  *****/
24 void DeviceDriverHandler0();
25 void DeviceDriverHandler1();
26 void ISR_TX(UART UARTInstance);
27 void ISR_RX(UART UARTInstance);
28
29 UART UARTInstance0, UARTInstance1;
30
31 u32 pendingReg, dataReg;
32 volatile static int count0, count1, tx_count, rx_count;
33 XScuGic InterruptController;
34 XScuGic_Config *GicConfig;
35 int i=1;
36
37

```

```

38  /**
39   *
40   * @brief Effettua la configurazione e l'abilitazione del GIC.
41   *
42   * @return XST_SUCCESS se la configurazione è avvenuta correttamente
43   *         XST_FAILURE altrimenti
44   *
45   * @note
46   *
47   */
48  int SetupInterrupt() {
49
50      int Status;
51
52      //inizializzazione driver xscugic per la gestione del gic
53      GicConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
54      Status = XScuGic_CfgInitialize(&InterruptController, GicConfig, GicConfig
55      ->CpuBaseAddress);
56
57      if ( Status != XST_SUCCESS) return XST_FAILURE;
58
59      //abilita la gestione delle eccezioni relative alla linea di interruzione
60      //in ingresso.
61      Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
62      (Xil_ExceptionHandler)XScuGic_InterruptHandler, &InterruptController)
63      ;
64      Xil_ExceptionEnable();
65
66      //Associa l'handler definito dall'utente alla linea di interruzione in
67      //ingresso al gic
68      //relativa al componente.
69      Status = XScuGic_Connect(&InterruptController,
70      XPAR_FABRIC_UART_0_INTERRUPT_INTR,
71      (Xil_ExceptionHandler)DeviceDriverHandler0, (void *)&
72      InterruptController);
73      if ( Status != XST_SUCCESS) return XST_FAILURE;
74
75      Status = XScuGic_Connect(&InterruptController,
76      XPAR_FABRIC_UART_1_INTERRUPT_INTR,
77      (Xil_ExceptionHandler)DeviceDriverHandler1, (void *)&
78      InterruptController);
79      if ( Status != XST_SUCCESS) return XST_FAILURE;
80
81      //Abilita la linea di interruzione del gic relativa al componente
82      //mappato
83      XScuGic_Enable(&InterruptController, XPAR_FABRIC_UART_0_INTERRUPT_INTR);
84      XScuGic_Enable(&InterruptController, XPAR_FABRIC_UART_1_INTERRUPT_INTR);
85
86      return Status;
87  }

```

```

78
79
80 }
81
82
83 int main(void) {
84
85     //Configurazione ed enable del Gic
86     if(SetupInterrupt() != XST_SUCCESS)
87         return XST_FAILURE;
88
89     //Init della struttura dati che astrae il componente UART
90     UART_Init(&UARTInstance0, UART0_BASE_ADDR);
91     UART_Init(&UARTInstance1, UART1_BASE_ADDR);
92
93
94     //Abilitazione delle interruzioni globali del componente e delle singole
        linee interne di interrupt
95     UART_GlobalEnableInterrupt(&UARTInstance0, 0x1);
96     UART_EnableInterrupt(&UARTInstance0, INT_MASK_RX);
97
98     UART_GlobalEnableInterrupt(&UARTInstance1, 0x1);
99     UART_EnableInterrupt(&UARTInstance1, INT_MASK_TX);
100
101
102     //contatore utilizzato per verificare il corretto funzionamento del
        dispositivo.
103     //indica il numero di volte che l'handler è stato chiamato.
104     count0=0;
105     count1=0;
106
107     //invio di prova
108     UART_SetData(&UARTInstance1, 0xF);
109     UART_Start(&UARTInstance1);
110
111     while(i>0) {
112         i++;
113     }
114
115     return 0;
116 }
117
118 void DeviceDriverHandler0()
119 {
120
121     count0++;
122     //vengono disabilitate le interruzioni globali del componente
123     UART_GlobalDisableInterrupt(&UARTInstance0, 0x1);
124     pendingReg = UART_GetPending(&UARTInstance0);

```

```

125 printf("PENDING REG :%08x \n\n",pendingReg);
126
127 /* avendo una sola linea di interruzione diretta verso il processore
128    è necessario identificare quale delle due linee interne ha
129    attivato la linea IRQ. Nel fare questo è necessario esplicitare uno
130    schema di priorità interno di gestione delle interruzioni.
131    In questo caso viene gestita prima l'interruzione relativa alla linea RX
132    */
133
134 if((pendingReg & 0x00000002) == 0x00000002)
135     ISR_RX(UARTInstance0);
136 else if((pendingReg & 0x00000001) == 0x00000001){
137     ISR_TX(UARTInstance0);
138 }
139
140
141 //abilitazione interruzioni globali del componente
142 UART_GlobalEnableInterrupt(&UARTInstance0,0x1);
143 }
144
145
146 void DeviceDriverHandler1()
147 {
148
149     count1++;
150     //vengono disabilitate le interruzioni globali del componente
151     UART_GlobalDisableInterrupt(&UARTInstance1,0x1);
152     pendingReg = UART_GetPending(&UARTInstance1);
153     printf("PENDING REG :%08x \n\n",pendingReg);
154
155     /* avendo una sola linea di interruzione diretta verso il processore
156        è necessario identificare quale delle due linee interne ha
157        attivato la linea IRQ. Nel fare questo è necessario esplicitare uno
158        schema di priorità interno di gestione delle interruzioni.
159        In questo caso viene gestita prima l'interruzione relativa alla linea RX
160        */
161
162     if((pendingReg & 0x00000002) == 0x00000002)
163         ISR_RX(UARTInstance1);
164     else if((pendingReg & 0x00000001) == 0x00000001){
165         ISR_TX(UARTInstance1);
166         if(tx_count<5){
167             UART_SetData(&UARTInstance1,0xF);
168             UART_Start(&UARTInstance1);}
169     }
170     //abilitazione interruzioni globali del componente
171     UART_GlobalEnableInterrupt(&UARTInstance1,0x1);
172 }
173

```

```

174
175 void ISR_TX(UART UARTInstance){
176     printf("***** ISR TX*****\n\n");
177     tx_count++;
178     //ACK interruzione relativa a TX
179     UART_ACK(&UARTInstance,0x1);
180 }
181
182 void ISR_RX(UART UARTInstance){
183     printf("***** ISR RX*****\n\n");
184     rx_count++;
185     //ACK interruzione relativa a RX
186     UART_ACK(&UARTInstance,0x2);
187 }

```

3.2.5 Driver Linux

3.2.5.1 Driver Kernel Mode

Per una spiegazione più dettagliata della scrittura del driver sottoforma di modulo kernel si rimanda alla sezione corrispondente del precedente capitolo. Per l'astrazione del nostro device UART si è realizzata una struct, definita nel file UART.h, che contiene tutte le informazioni necessarie per la gestione del dispositivo.

```

1  /**
2   * @brief Struttura che astrae un device UART 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
7       * associato al device; m: utilizzato dal driver per discriminare il
8       * singolo device tra quelli a lui associati) */
9      dev_t Mm;
10     /** Puntatore a struttura platform_device cui l'oggetto UART si riferisce */
11     struct platform_device *pdev;
12     /** Struttura per l'astrazione di un device a caratteri */
13     struct cdev cdev;
14     /** Puntatore alla struttura che rappresenta l'istanza del device */
15     struct device* dev;
16     /** Puntatore a struttura che rappresenta una vista alto livello del device
17      */
18     struct class* class;
19     /** Interrupt-number a cui il device è connesso */
20     uint32_t irqNumber;
21     /** Puntatore alla regione di memoria cui il device è mappato */
22     struct resource *mreg;
23     /** Device Resource Structure */
24     struct resource res;

```

```

22 /** res.end - res.start; numero di indirizzi associati alla periferica. */
23     uint32_t res_size;
24 /** Indirizzo base virtuale della periferica */
25     void __iomem *vrtl_addr;
26 /** wait queue per la sys-call read() */
27     wait_queue_head_t read_queue;
28 /** wait queue per la sys-call poll() */
29     wait_queue_head_t poll_queue;
30 /** wait queue per la sys-call write() */
31     wait_queue_head_t write_queue;
32 /** Flag che indica, quando asserito, la possibilità di effettuare una
33     chiamata a read */
34     uint32_t can_read;
35 /** Flag che indica, quando asserito, la possibilità di effettuare una
36     chiamata a write */
37     uint32_t can_write;
38 /** Spinlock usato per garantire l'accesso in mutua esclusione alla
39     variabile can_read */
40     spinlock_t slock_int;
41 /** Spinlock usato per garantire l'accesso in mutua esclusione alla
42     variabile can_write */
43     spinlock_t write_lock;
44 /** Buffer utilizzato per contenere i caratteri da trasmettere */
45     uint8_t * buffer_tx;
46 /** Buffer utilizzato per contenere i caratteri da ricevere */
47     uint8_t * buffer_rx;
48 } UART;

```

Per le funzioni necessarie all'interfacciamento con il device si rimanda alla documentazione interna. Il device è stato gestito come un **character device**. Vengono analizzate nel seguito le funzionalità delle system-call offerte dal modulo:

```

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     = UART_llseek,
8     .read       = UART_read,
9     .write      = UART_write,
10    .poll        = UART_poll,
11    .open        = UART_open,
12    .release     = UART_release,
13 };

```

- owner: rappresenta puntatore al modulo che è il possessore della struttura. Ha lo scopo di evitare che il modulo venga rimosso quando una delle funzionalità fornite è in uso. Inizializzato

mediante la macro `THIS_MODULE`.

- `UART_llseek`: sposta l'offset di lettura/scrittura sul file.
- `UART_read`: utilizzata per leggere dal device. La chiamata a `UART_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, quando il trasferimento è completo, 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.
- `UART_write`: utilizzata per inviare dati al device. La chiamata a `UART_write` potrebbe avvenire quando il device è impegnato a gestire un trasferimento ancora non terminato, 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 è stato realizzato un meccanismo analogo a quello per la lettura, ovvero utilizzando una variabile "**can_write**". La funzione `write` effettua un controllo sullo stato di quest'ultima e se rileva che non è possibile effettuare una scrittura mette il processo in sleep. L'ISR, quando il trasferimento è completo, avrà il compito di settare la variabile per poter rendere possibile la scrittura e risvegliare i processi dalla coda. Per realizzare questo meccanismo sono stati utilizzati `spinlock` e `wait_queue` fornite dal kernel.
- `UART_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.
- `UART_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 risulteranno essere non bloccanti.
- `UART_release`: chiamata alla chiusura del file descriptor associato al device.

Il codice allegato è diviso in:

- `UART.h/UART.c`: definizione e implementazione di una struttura che astrae il nostro device UART in kernel mode. Contiene ciò che è necessario al funzionamento del driver, compreso lo `spinlock` per l'accesso in mutua esclusione alle variabili `can_read`, `can_write` e le `wait_queue`.
- `UART_list.h/UART_list.c`: definizione e implementazione di una lista di oggetti UART. Fornisce tutte le funzioni necessarie per l'interfacciamento quali inizializzazione, cancellazione, aggiunta oggetto, ricerca.
- `UART_kernel_main.c`: rappresenta il vero e proprio modulo kernel che reimplementa tutte le funzioni necessarie all'interfacciamento.

Per compilare il modulo è sufficiente lanciare lo script "*prepare_environment.sh*" (vedi capitolo 1) prima di dare il comando `make`. Segue il Makefile utilizzato per la compilazione:


```

1 obj-m += my_kernel_UART.o
2 my_kernel_UART-objs :=UART_kernel_main.o UART.o UART_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_UART.ko

```

Per mostrare il corretto funzionamento di tutte le funzionalità implementate sono state create due user application:

1. user_app.c: l'utente indica da riga di comando la stringa che vuole trasmettere tramite il device UART. Si è scelto di demandare l'onere di gestire l'invio di più caratteri all'utente che scrive l'applicazione. Vengono aperti dunque i descrittori del file associati ai due device e vengono invocate un numero di write e read pari al numero di caratteri che compongono la stringa.
2. poll_user_app.c: l'utente indica da riga di comando la stringa che vuole trasmettere tramite il device UART. Vengono aperti dunque i descrittori del file associati ai due device e viene effettuata una chiamata a poll per verificare se sia possibile o meno effettuare una lettura che non risulti bloccante. Dato che non sono state ancora effettuate trasmissioni il buffer di ricezione è ancora vuoto e la variabile can_read indica che non è possibile effettuare una lettura. La poll dunque restituirà, dopo un timeout specificato, una maschera pari a 0 e la chiamata a read non sarà effettuata. Dopo un numero prefissato di chiamate a poll verrà effettuata una write, per cui alla successiva chiamata la maschera restituita indicherà la possibilità di effettuare una lettura non bloccante e verrà effettuata una read.

Per rimuovere il modulo impartire il comando:

```

1 rmmod my_kernel_UART.ko

```

3.2.5.2 UIO

Per una spiegazione più dettagliata della scrittura del driver userspace I/O si rimanda alla sezione corrispondente del precedente capitolo. Dopo aver aggiunto ai bootargs nel file system-top.dts il parametro "uio_pdrv_genirq.of_id=generic-uio" si imposta nel file pl.dtsi il campo compatible dei device UART a "generic-uio" come segue:

```

1 / {
2     amba_pl: amba_pl {

```

```

3  #address-cells = <1>;
4  #size-cells = <1>;
5  compatible = "simple-bus";
6  ranges ;
7  UART_0: UART@43c00000 {
8      /* This is a place holder node for a custom IP, user may need to
9         update the entries */
10     clock-names = "s00_axi_aclk";
11     clocks = <&clkc 15>;
12     compatible = "xlnx,UART-1.0";
13     interrupt-names = "interrupt";
14     interrupt-parent = <&intc>;
15     interrupts = <0 29 4>;
16     reg = <0x43c00000 0x10000>;
17     xlnx,s00-axi-addr-width = <0x5>;
18     xlnx,s00-axi-data-width = <0x20>;
19 };
20 UART_1: UART@43c10000 {
21     /* This is a place holder node for a custom IP, user may need to
22        update the entries */
23     clock-names = "s00_axi_aclk";
24     clocks = <&clkc 15>;
25     compatible = "xlnx,UART-1.0";
26     interrupt-names = "interrupt";
27     interrupt-parent = <&intc>;
28     interrupts = <0 30 4>;
29     reg = <0x43c10000 0x10000>;
30     xlnx,s00-axi-addr-width = <0x5>;
31     xlnx,s00-axi-data-width = <0x20>;
32 };

```

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 device uio0 e uio1 corrispondenti ai nostri device UART. Il driver userspace effettuerà il mapping dei device per poi mettersi in attesa di notifica di interrupt tramite chiamata a read. 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 <string.h>

```

```

10 #include <poll.h>
11 #include "UART_interrupt_uio.h"
12 /**
13  * @file UART_interrupt_uio.c
14  * @brief permette la gestione della periferica UART utilizzando un driver
15  *       di tipo UIO
16  */
17 #define DATA_IN      0 // DATA TO SEND
18 #define TX_EN         4 // TRANSFER ENABLE (0)
19 #define STATUS_REG    8 // OE(0) FE(1) DE(2) RDA(3) TX_BUSY(4)
20 #define RX_REG        12 // DATA RECEIVED
21 #define GLOBAL_INTR_EN 16 // GLOBAL INTERRUPT ENABLE
22 #define INTR_EN        20 // LOCAL INTERRUPT ENABLE
23 #define INTR_ACK_PEND  28 // PENDING/ACK REGISTER
24
25 #define TX             1
26 #define RX             2
27
28 #define TIMEOUT        5000
29
30 /**
31  * @file UART_interrupt_uio.c
32  * @page driver_UART_UIO
33  * @brief funzioni per gestire la trasmissione e la ricezione dei
34  *       dati utilizzando il protocollo UART
35  */
36 int tx_count, rx_count = 0;
37 int buffer_size = 0;
38 char * buffer_tx;
39 char * buffer_rx;
40 struct pollfd poll_fds [2];
41
42 /**
43  *
44  * @brief Utilizzata per scrivere un valore all'interno di un registro
45  *       della periferica, specificando l'indirizzo base virtuale e
46  *       l'offset del registro in cui scrivere
47  *
48  * @param addr indirizzo virtuale della periferica
49  * @param offset offset del registro a cui scrivere
50  * @param valore da scrivere
51  *
52  */
53
54 void write_reg(void *addr, unsigned int offset, unsigned int value)
55 {
56     *((unsigned*) (addr + offset)) = value;

```

```

58 }
59 /**
60  *
61  * @brief Utilizzata per leggere un valore da un registro
62  *       della periferica, specificando l'indirizzo base virtuale e
63  *       l'offset del registro da cui leggere
64  *
65  * @param addr indirizzo virtuale della periferica
66  * @param offset offset del registro a cui leggere
67  *
68  * @return valore presente all'interno del registro
69  *
70  *
71  */
72 unsigned int read_reg(void *addr, unsigned int offset)
73 {
74     return *((unsigned*)(addr + offset));
75 }
76
77 /**
78  *
79  * @brief Attende l' arrivo di un interrupt utilizzando la read
80  *       su un device UIO
81  *
82  * @param poll_fds struct contenente i due descrittori del file per
83  *       i due device UART
84  * @param uart_rx_ptr indirizzo virtuale della periferica UART utilizzata
85  *       in ricezione
86  * @param  uart_tx_ptr indirizzo virtuale della periferica UART utilizzata
87  *       in trasmissione
88  *
89  */
90 void wait_for_interrupt(struct pollfd * poll_fds, void *uart_rx_ptr, void *
    uart_tx_ptr)
91 {
92     int pending = 0;
93     int reenable = 1;
94     u_int32_t pending_reg = 0;
95     u_int32_t reg_sent_data = 0;
96     u_int32_t reg_received_data = 0;
97
98     int ret = poll(poll_fds, 2, TIMEOUT);
99     if (ret > 0) {
100
101     /** Se vi è un'interruzione sul device UIO0 associato all'UART per la
        ricezione */
102         if(poll_fds[0].revents && POLLIN) {
103
104             read(poll_fds[0].fd, (void *)&pending, sizeof(int));

```

```

105
106 /* Disabilita le interruzioni */
107     write_reg(uart_rx_ptr, GLOBAL_INTR_EN, 0);
108
109     pending_reg = read_reg(uart_rx_ptr, INTR_ACK_PEND);
110
111     if((pending_reg & RX) == RX){
112
113         printf("ISR RX detected!\n");
114
115         if(rx_count <= buffer_size){
116             rx_count++;
117             reg_received_data = read_reg(uart_rx_ptr, RX_REG);
118             printf("ISR RX - value received: %c\n", reg_received_data);
119             buffer_rx[rx_count] = reg_received_data;
120         }
121     /* ACK ricezione */
122         write_reg(uart_rx_ptr, INTR_ACK_PEND, RX);
123         write_reg(uart_rx_ptr, INTR_ACK_PEND, 0);
124     /* Riabilitazione interruzioni*/
125         write_reg(uart_rx_ptr, GLOBAL_INTR_EN, 1);
126     }
127 /* Riabilita l'interrupt nell'interrupt controller attraverso il
    sottosistema UIO */
128     write(poll_fds[0].fd, (void *)&reenable, sizeof(int));
129 }
130
131 /** Se vi è un'interruzione sul device UIO0 associato all'UART per la
    trasmissione */
132     if(poll_fds[1].revents && POLLIN){
133
134         read(poll_fds[1].fd, (void *)&pending, sizeof(int));
135
136     /* Disasserisce il transfer enable e disabilita le interruzioni */
137         write_reg(uart_tx_ptr, TX_EN, 0);
138         write_reg(uart_tx_ptr, GLOBAL_INTR_EN, 0);
139
140         pending_reg = read_reg(uart_tx_ptr, INTR_ACK_PEND);
141
142         if((pending_reg & TX) == TX){
143
144             printf("ISR TX Detected\n");
145             tx_count++;
146
147             if(tx_count <= buffer_size){
148
149                 reg_sent_data = read_reg(uart_tx_ptr, DATA_IN);
150                 printf("ISR TX - value sent: %c\n", reg_sent_data);
151

```

```

152 /* ACK trasmissione*/
153     write_reg(uart_tx_ptr, INTR_ACK_PEND, TX);
154     write_reg(uart_tx_ptr, INTR_ACK_PEND, 0);
155
156 /* Riabilitazione interruzioni*/
157     write_reg(uart_tx_ptr, GLOBAL_INTR_EN, 1);
158
159 /* Abilitazione del trasferimento nuovo carattere */
160     if(tx_count != buffer_size){
161         printf("ISR TX - start sending next value: %c\n", buffer_tx[
            tx_count]);
162         write_reg(uart_tx_ptr, DATA_IN, buffer_tx[tx_count]);
163         write_reg(uart_tx_ptr, TX_EN, 1);
164     }
165 }
166
167 }
168 /* Riabilita l'interrupt nell'interrupt controller attraverso il
    sottosistema UIO */
169     write(poll_fds[1].fd, (void *)&reenable, sizeof(int));
170 }
171 }
172 }
173 int main(int argc, char *argv[]){
174
175     int j,i;
176     void * uart_rx_ptr;
177     void * uart_tx_ptr;
178
179     int DIM = strlen(argv[1]);
180     buffer_size = DIM;
181     buffer_tx = malloc(sizeof(char)*DIM);
182     buffer_rx = malloc(sizeof(char)*DIM);
183
184     buffer_tx = argv[1];
185
186     int rx_file_descr = open("/dev/ui0", O_RDWR);
187     if (rx_file_descr < 1){
188         printf("Errore nell'accesso al device UIO.\n");
189         return -1;
190     }
191
192     unsigned dimensione_pag = sysconf(_SC_PAGESIZE);
193     uart_rx_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED,
        rx_file_descr, 0);
194
195     int tx_file_descr = open("/dev/ui1", O_RDWR);
196     if (tx_file_descr < 1){
197         printf("Errore nell'accesso al device UIO.\n");

```

```

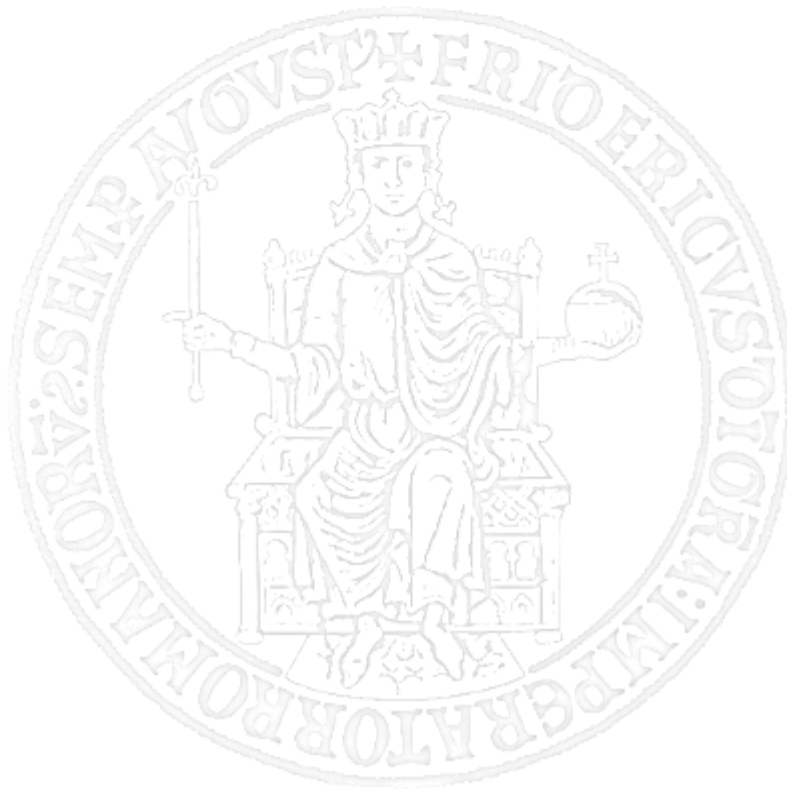
198     return -1;
199 }
200
201 uart_tx_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED,
    tx_file_descr, 0);
202
203 printf("L'utente ha chiesto di mandare la stringa: %s, di %d caratteri.\n"
    , buffer_tx, DIM);
204
205 /* Abilitazione interruzioni globali */
206 write_reg(uart_rx_ptr, GLOBAL_INTR_EN, 1);
207 /* Abilitazione interruzioni */
208 write_reg(uart_rx_ptr, INTR_EN, RX);
209
210 /* Abilitazione interruzioni globali */
211 write_reg(uart_tx_ptr, GLOBAL_INTR_EN, 1);
212 /* Abilitazione interruzioni */
213 write_reg(uart_tx_ptr, INTR_EN, TX);
214
215 /* Settaggio del primo carattere da mandare */
216 write_reg(uart_tx_ptr, DATA_IN, buffer_tx[0]);
217
218 /* Abilitazione del trasferimento */
219 write_reg(uart_tx_ptr, TX_EN, 1);
220
221 poll_fds[0].fd = rx_file_descr;
222 poll_fds[0].events = POLLIN;
223
224 poll_fds[1].fd = tx_file_descr;
225 poll_fds[1].events = POLLIN;
226
227 while(tx_count < buffer_size){
228     printf("Waiting for interrupts.....");
229     //sleep(1);
230     wait_for_interrupt(poll_fds, uart_rx_ptr, uart_tx_ptr);
231 }
232
233 printf("Trasmissione/Ricezione completata, valore ricevuto: ");
234 for(i=0; i<=rx_count; i++)
235     printf("%c",buffer_rx[i]);
236
237 printf("\n");
238 /* Fa l'unmap dei device UART */
239 munmap(uart_tx_ptr, dimensione_pag);
240 munmap(uart_rx_ptr, dimensione_pag);
241
242 close(tx_file_descr);
243 close(rx_file_descr);
244

```



```
245 | free(buffer_rx);  
246 | return 0;  
247 |  
248 | }
```

La prima operazione del driver è quella di aprire il file descriptor relativo ai due device uio0 e uio1. Successivamente viene calcolata la dimensione della pagina e viene effettuato il mapping dell'indirizzo virtuale tramite chiamata a `mmap()`. Dopo aver rispettivamente abilitato le interruzioni di ricezione e trasmissione per i due dispositivi UART, viene invocata una `write_reg` sul registro `DATA_IN` per inserire il primo carattere da inviare nel registro di trasmissione del dispositivo UART uio1 e successivamente viene dato il segnale di `TX.EN` per abilitare il trasferimento. Iterativamente, finchè non sono stati trasmessi un numero di caratteri pari alla dimensione della stringa da trasmettere, viene effettuata una `poll` per mettersi in attesa di eventi interrompenti dai device. Una volta risvegliato dalla chiamata il processo si occupa della gestione dell'interruzione. La prima operazione da effettuare è controllare se l'interruzione rilevata sia relativa ad una avvenuta ricezione o trasmissione. Nel primo caso la funzione si occupa di prelevare il dato dal registro di ricezione `RX.REG` e di incrementare il contatore dei caratteri ricevuti. Se sono stati ricevuti un numero di caratteri pari alla dimensione della stringa viene stampato il buffer di ricezione, dato il segnale di `ACK` e riabilitate le interruzione. Nel caso in cui invece l'interruzione rilevata sia legata al completamento della trasmissione di un carattere la funzione incrementa il contatore dei caratteri trasmessi e, se questo non è pari alla dimensione della stringa, setta il prossimo carattere da trasmettere nel registro `DATA_IN` e asserisce il `TX.EN`. dopo aver dato il segnale di `ACK` e riabilitato le interruzioni.



Capitolo 4

Progetto finale

4.1 Traccia

Progetto corso Embedded aa 2018-19

Descrizione del sistema da realizzare

Occorre progettare e realizzare un sottosistema S_{com} adibito alla gestione dello scambio di messaggi M fra due o più board di un sistema embedded operante in regime critico e ad alta affidabilità. Lo scambio dei messaggi deve essere conforme con lo standard ferroviario “Protocollo Vitale Standard (PVS) che si allega.

Le board da utilizzare montano o SOC della serie STM32F3 su board Discovery o processori ibridi della famiglia ARM Zynq su board della Digilent (Zybo/ZedBoard).

La connettività fra le varie board (solitamente una board ha funzioni di master e le altre di slave) deve poter essere realizzata con connessioni seriali punto-punto (con protocollo UART) e con connessioni con bus seriali di tipo I2C, SPI e CAN.

Si ricorda che la trasmissione con UART e quelle su bus I2C e CAN utilizzano una connessione a tre fili (Rx, Tx e massa). Per l’UART non si devono prendere in considerazione eventuali coppie di segnali di handshaking previsti dal protocollo asincrono). Quella con SPI avviene, invece, su 3 fili (SCLK, MISO, MOSI) più i k fili di selezione fisica che vanno dalla board master alle k slave (un filo di selezione per ciascuna delle k unità). I protocolli I2C e CAN selezionano l’entità con cui interoperare in base al valore di un “campo indirizzi” presente nei frame trasmessi.

Architettura Software

L’architettura software da realizzare è riportata in fig.xx. Essa è strutturata in unità funzionali organizzate su due livelli:

- un primo livello che si interfaccia con le funzionalità di uno strato HAL (librerie fornite con le schede, in particolare, quelle dell’ambiente Cube di St e quelle di ARM-Digilent per la Zynq). Per ciascuno dei dispositivi associati ai quattro protocolli di comunicazione utilizzati, devono essere sviluppati i relativi driver che dovranno operare con uso di interruzioni.

- un secondo livello verso lo strato applicativo caratterizzato da API per lo scambio messaggi (primitive di send e receive che parametrizzano il particolare dispositivo di comunicazione da utilizzare). Tale strato provvede a incapsulare i driver dei quattro dispositivi mediante gusci software e a virtualizzarne e parametrizzarne le funzionalità in un'unica funzione avente come parametri il messaggio M da trasmettere, il canale C (uno o anche più canali) da utilizzare per il trasferimento e lo stato dell'operazione. Per ciascuno dei messaggi da trasferire, tale funzione calcola anche due CRC (CRC1 e CRC2, secondo il formato PVS) e genera un frame standard che li include.

Applicativo di prova

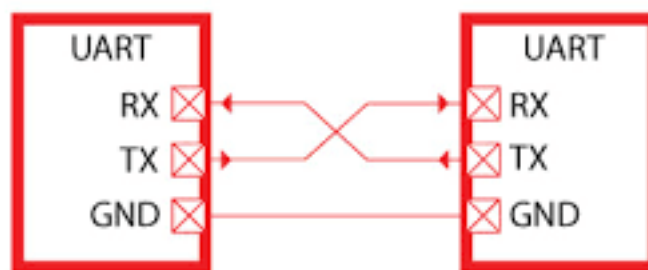
Il linguaggio con cui sviluppare i due livelli è C. Il sistema realizzato deve essere testato mediante uno o più applicativi in grado di verificarne tutte le funzionalità. Il software sviluppato deve essere verificato nei tempi di esecuzione delle varie funzioni.

4.2 Periferiche

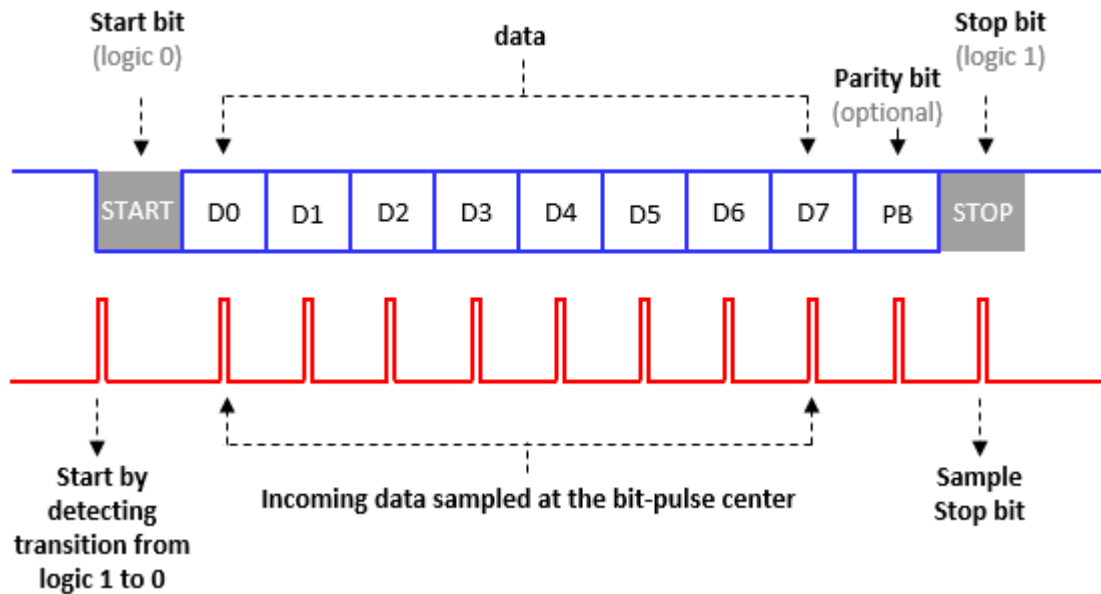
Nel seguito vengono presentate le principali caratteristiche delle connessioni utilizzate all'interno del progetto.

4.2.1 UART

UART è un protocollo asincrono in cui la comunicazione è solamente punto punto, ha bisogno solamente due fili per la comunicazione, a cui bisogna aggiungere il cavo per portarle allo stesso potenziale affinché sia possibile capire se è stato inviato un valore 1 oppure 0 come mostrato in figura.



Un esempio di comunicazione è il seguente:



La comunicazione inizia con un bit di start, che consiste in una transizione da un valore logico 1 ad uno che valga 0, dopodiché vengono inviati i successivi otto bit di informazione, un bit di parità ed un bit di stop che indica la fine della trasmissione.

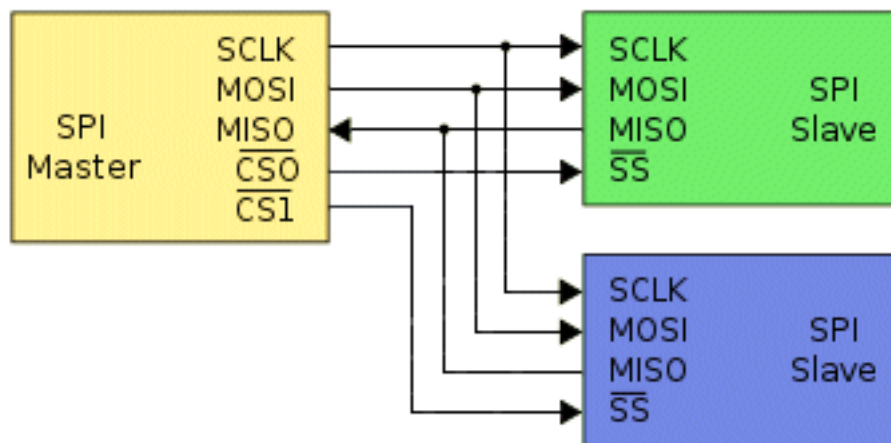
Viene effettuata la lettura di un valore dal ricevitore di solito a metà periodo di trasmissione di un bit, ciò può avvenire perché la velocità di comunicazione è nota tra i due dispositivi ed è misurata in baud.

Gli errori nella comunicazione possono essere di tre tipi:

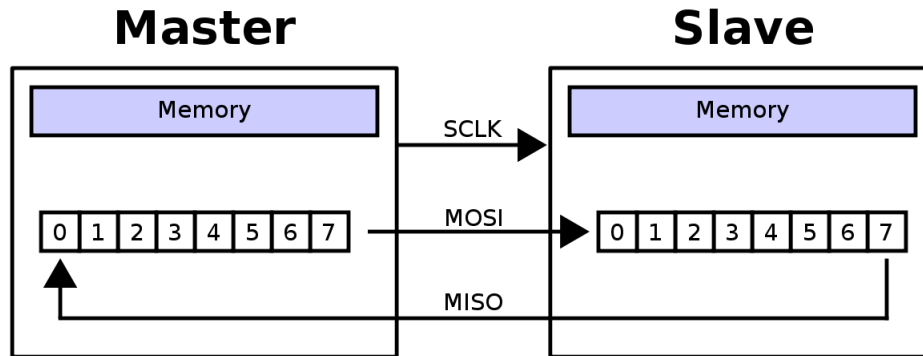
1. Parity, se il bit di parità non è corretto;
2. Overrun, quando arriva il successivo bit senza che si sia campionato il precedente;
3. Frame error, non viene rilevato lo stop quando dovrebbe arrivare;

4.2.2 Spi

Spi è un protocollo singolo master e multi slave che sono collegabili utilizzando il seguente schema



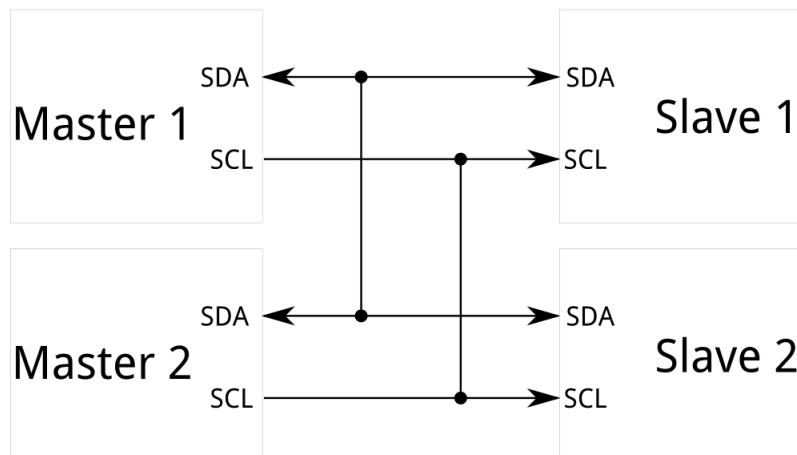
Il device master si collega agli slave, con i pin SCLK che corrisponde al clock che tempifica la comunicazione, MOSI (master output slave input) l' interfaccia che invia i dati dal master allo slave, MISO (master input slave output) l' interfaccia che invia i dati dallo slave al master e SS (slave select), il cui scopo consiste nel selezionare lo slave con cui comunicare, asserendo di solito uno zero logico.



Dopo aver selezionato la periferica asserendo lo slave select, ad ogni colpo di clock il bit più significativo dello shift register del master va ad occupare il bit meno significativo dello shift register dello slave, la stessa cosa avviene per il bit più significativo contenuto dallo slave.

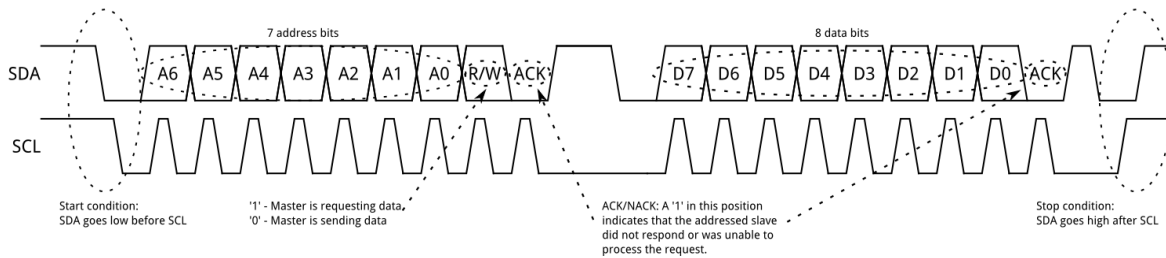
4.2.3 I2C

I2C è un protocollo sincrono, multimaster e multislave a differenza di SPI utilizza un numero di connessione minore per inviare dati



SCL è l' interfaccia utilizzata dal master per temporizzare la comunicazione mentre invece SDA viene utilizzata per inviare i dati.

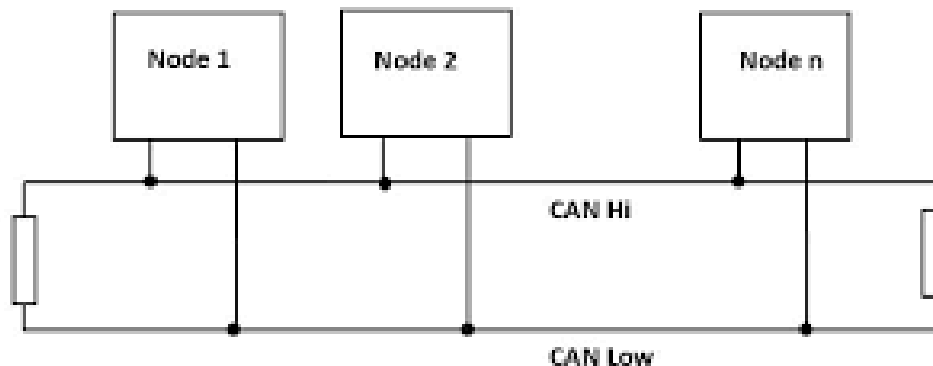
Una comunicazione I2C di solito consta di questa sequenza



Lo start bit è dato da uno zero logico presente sull' interfaccia dato ed un successivo abbassamento della linea di clock, dopodiché ad ogni oscillazione del clock viene inviato un bit sul pin SDA, il primo dato inviato consiste nell' indirizzo della periferica con cui parlare seguito dal tipo di operazione se di lettura o di scrittura, se la comunicazione è andata a buon fine si riceve un ack, sull' interfaccia SDA dopo l' invio dell' ultimo bit si trova uno 0, dopodiché è possibile inviare un dato allo stesso in cui si invia un indirizzo, la comunicazione termina quando il clock viene settato ad un 1 logico e il pin SDA si pone allo stesso stato.

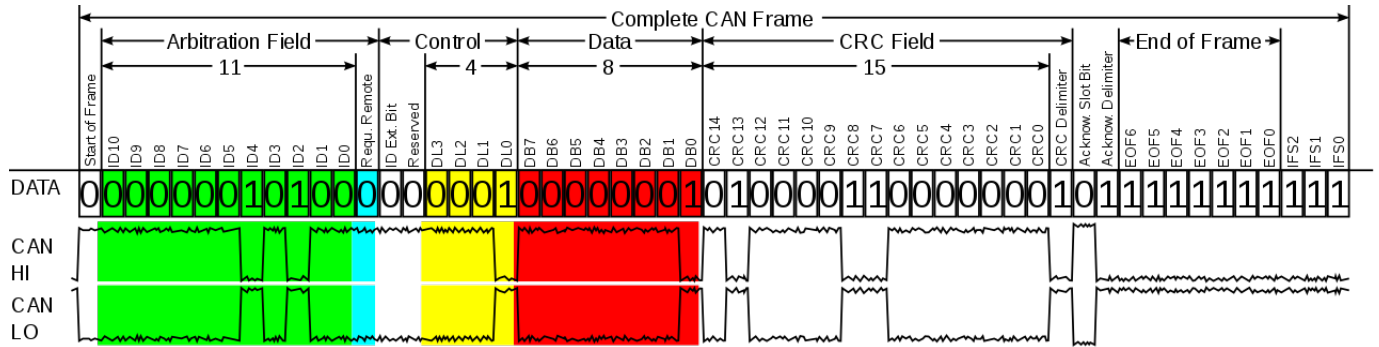
4.2.4 CAN

Bus utilizzato solitamente nel campo automotive è di tipo asincrono, multimaster e multislave.



I dispositivi si collegano tra di loro utilizzando solamente due collegamenti CANL e CANH, tali valori sono pilotati al fine di generare uno 0 o 1 fisico sulla linea di comunicazione, poiché si asserisce uno zero quando la differenza tra CANH e CANL è maggiore di una determinata soglia, mentre invece si asserisce uno quando tale differenza è zero.

Il valore fisico 0 viene detto bit dominante, invece un 1 viene detto bit recessivo, possiamo capire il perché di questi nomi osservando un esempio di comunicazione:



I frame inviati sul bus CAN hanno la struttura mostrata in figura, le parti di maggiore importanza consistono nel bit di inizio frame, l'indirizzo del nodo da voler contattare (o del gruppo poiché possiamo assegnare lo stesso indirizzo ad un insieme di dispositivi) il dato, il campo del CRC per controllare che la trasmissione sia andata a buon fine è l'end of frame.

Il bit 0 viene detto dominante poiché se nello stesso momento sul bus un altro dispositivo invia un 1, su quest'ultimo verrà forzato uno 0, avendo poi i dispositivi la possibilità di leggere il valore sul bus oltre che quello di imporre un valore su di esso, se questi leggono un valore diverso da quello che hanno settato il dispositivo non continua la trasmissione poiché nel protocollo CAN l'indirizzo avente il valore più basso ha la priorità maggiore (un indirizzo di soli 0 ha priorità massima).

Di CAN ne esiste una versione successiva a questa in cui l'indirizzo è formato da 29 bit invece di 11.

4.2.5 CRC

Il CRC è una tecnica utilizzata per la rilevazione di errori che possono occorrere durante la trasmissione di una sequenza di bit.

L'algoritmo consiste nel dividere la sequenza di bit che vogliamo inviare, per un polinomio divisore che deve essere conosciuto dal trasmettitore e dal ricevitore affinché questo possa rilevare l'errore.

Di algoritmi per il calcolo del CRC ne esistono tanti poiché differiscono dal polinomio utilizzato o in alcuni casi vi sono procedure pre-CRC che fanno restituire un resto dalla operazione di divisione con peculiarità che sono utili per determinare gli errori riscontrati.

4.3 Scelte Progettuali

4.3.1 Architettura del sistema e topologia della rete

Si è sviluppato il software supponendo che il sistema sia composto da un numero n di nodi che differiscono esclusivamente dal fatto di poter assumere il ruolo di Master o Slave nelle comunicazioni sui bus. Ogni nodo è realizzato utilizzando una board *STM32F3 Discovery*. Nel sistema sono "idealmente" presenti un bus *I2C*, un bus *SPI* e un bus *CAN* ai quali tutti i nodi sono connessi. Ogni nodo, inoltre, è connesso "punto punto" ad un altro nodo del sistema utilizzando una periferica UART.

4.3.1.1 Arbitraggio dei bus

I bus utilizzati offrono diverse possibilità ai nodi per “interfacciarsi” agli stessi:

- **I2C**: è un bus che permette nativamente un arbitraggio sia *SingleMaster-Multislave* che *Multimaster*. Nel primo caso un solo nodo (master) può prendere possesso del bus e iniziare trasmissioni\ricezioni verso\da altri nodi, nel secondo caso più nodi possono gestire il bus e prendere iniziativa di trasferimento\ricezione. Nella soluzione realizzata si è scelto di realizzare lo schema **Multimaster**. Specificamente nella gestione delle comunicazioni fra i nodi si è scelto di far sì che il nodo si comporti da master quando effettua una trasmissione e da slave quando effettua una ricezione. Questo implica che nella trasmissione il nodo è conscio del nodo al quale sta trasmettendo, mentre non lo è nella ricezione.
- **SPI**: è un bus nativamente *SingleMaster-Multislave*. Le board STM32 permettono di realizzare una soluzione *multimaster* che tuttavia non risulta utile per il numero di fili necessari se il numero di nodi è superiore a 2. Si è scelto di realizzare dunque la soluzione **multislave** con un solo nodo che funge da master e gestisce k slave con k segnali di Slave Select che permettono di effettuare l'arbitraggio. Il master seleziona lo slave con il quale comunicare sia in caso di trasmissione che ricezione.
- **CAN**: bus nativamente ed esclusivamente **multimaster**. Viene utilizzato naturalmente in questa configurazione.

Si noti che la scelta sulla comunicazione di I2C differisce solo a livello software, è possibile trasmettere e ricevere in modalità master. Si è scelta questa soluzione per rendere uniforme il comportamento delle periferiche dei nodi nelle trasmissioni.

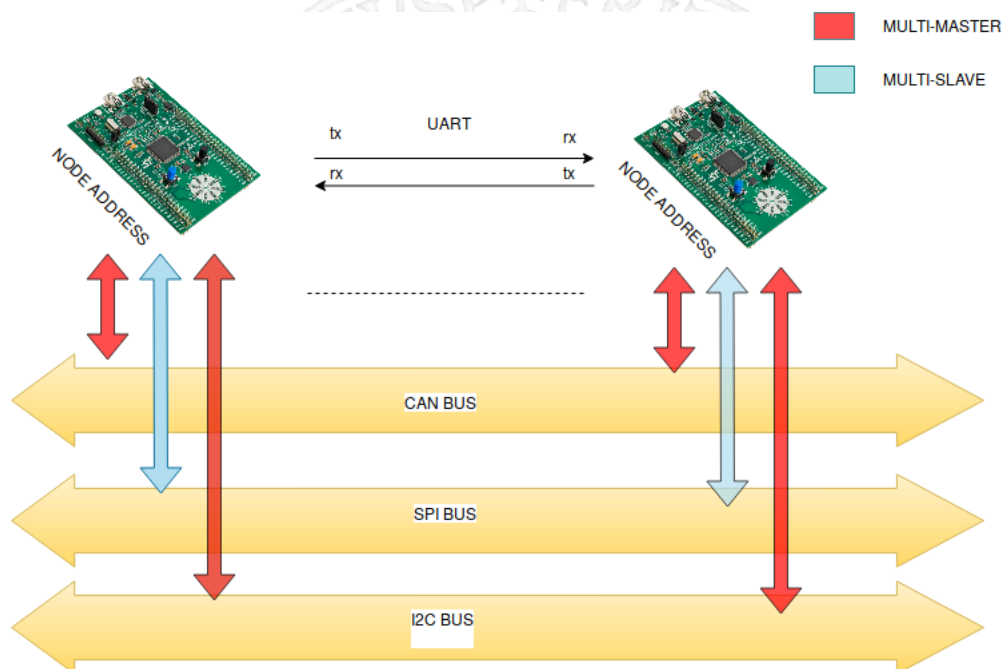


Figura 4.1: Connessione dei nodi

4.3.1.2 Modalità di Trasmissione

La trasmissione può essere effettuata con tre modalità differenti: *unicast*, *multicast* e *broadcast*.

- Unicast: il nodo manda il messaggio ad un solo nodo destinatario. E' possibile realizzare questa modalità di trasmissione su tutti e tre i bus.
- Multicast: il nodo manda il messaggio ad un insieme di bord associata ad un "gruppo". La trasmissione in questa modalità è disponibile solo sul bus CAN. Se si volesse realizzare sugli altri bus sarebbe necessario implementare una politica di trasmissione multi-unicast ma non si è ritenuto necessario data la disponibilità di CAN.
- Broadcast: il nodo manda il messaggio a tutti i nodi collegati al bus. La trasmissione in questa modalità è disponibile solo sul bus CAN.

4.3.1.3 Gestione degli indirizzi

Ogni periferica è dotata di una gestione degli indirizzi differente e può essere configurata indipendentemente da come sono settate le altre, ma ogni periferica risponde ad "spazio degli indirizzi unico". Per le trasmissioni sul bus si è scelto che ad ogni nodo siano associati due indirizzi:

1. **NODE ADDRESS**: indirizzo univoco del nodo. Utilizzato per effettuare comunicazioni unicast.
2. **GROUP ADDRESS**: indirizzo non univoco e condiviso fra tutti i nodi appartenente ad un determinato gruppo. Permette di realizzare le comunicazioni multicast con CAN.



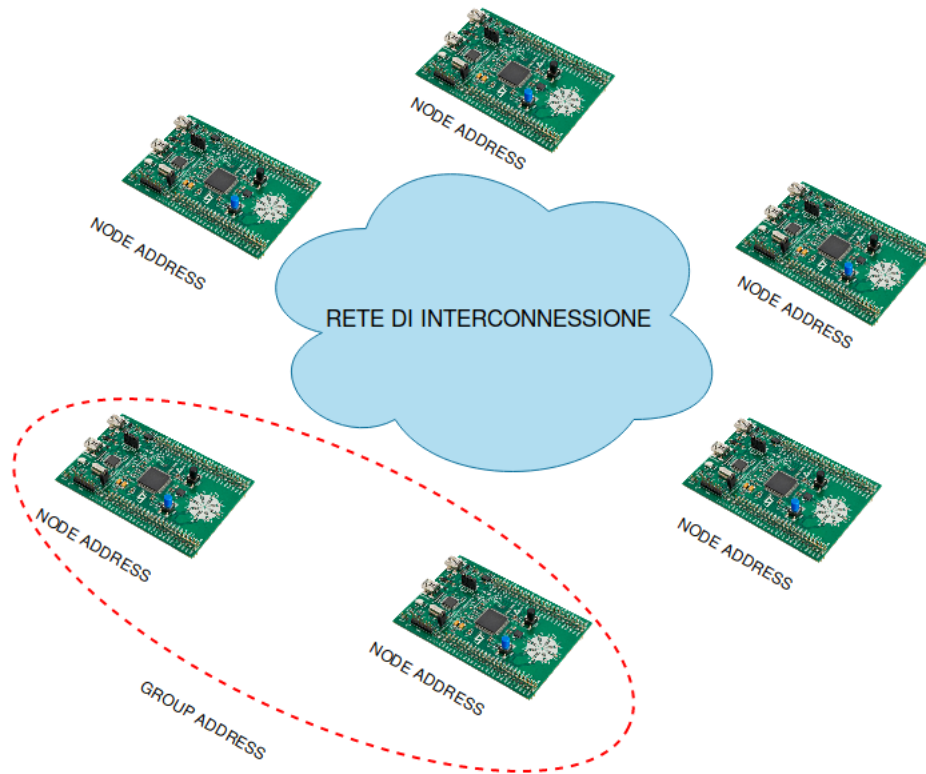


Figura 4.2: Gestione degli indirizzi

Consideriamo che:

- **I2C** permette di associare alla periferica un indirizzo di *lunghezza massima 10 bit*. Per effettuare la ricezione del messaggio è necessario che tutti i bit del campo address dello stesso (o una parte di essi non mascherati se si utilizzano maschere) corrispondano all'indirizzo della periferica.
- **SPI** non utilizza indirizzi per identificare i nodi ma collegamenti fisici.
- **CAN** non associa un indirizzo univoco al nodo poichè il messaggio è inviato sempre in broadcast e spetta al singolo nodo stabilire se è interessato o meno al messaggio mediante dei filtri. Questi filtri possono operare in due modalità: *mask* o *IDList*. Nel primo caso, al filtro viene dato un ID ed una maschera da utilizzare per confrontare i bit del campo ID del messaggio con il campo ID del filtro (i bit mascherati con 0 non vengono confrontati ma viene automaticamente dato matching), nel secondo caso invece ogni filtro corrisponde ad uno o più ID (1 da 32bit o 2 da 16 bit) che devono “essere accettati”.

Si utilizza questo meccanismo (*IDList*) per assegnare l'indirizzo del nodo sul bus CAN, prestando accortezza che nella fase di inizializzazione ogni periferica abbia un filtro con indirizzo univoco pari a quello associato ad I2C. Si è scelto di realizzare i gruppi con la modalità *IDList* e non *mask* per praticità e per poter avere un maggiore controllo sugli indirizzi da assegnare e totale libertà nel decidere quale nodo appartenga a quale gruppo. Dunque un nodo, per appartenere ad un

gruppo, dovrà specificare il GROUP_ADDRESS nel campo id di un filtro. Un filtro, in questa modalità, può rappresentare un singolo ID da 32 bit oppure due ID diversi da 16. Dato che il nostro spazio degli indirizzi è limitato superiormente dall'indirizzo di I2C a 10 bit, utilizzeremo un unico filtro contenente due ID (NODE_ADDRESS e GROUP_ADDRESS) da 16 bit (i bit[15 10] saranno sempre nulli).

Essendo lo spazio degli indirizzi su 10, ed essendo condiviso fra nodi e gruppi, potremmo avere teoricamente:

$\text{numero_nodi} + \text{numero_gruppi} = 2^{10}$. Fra questi indirizzi non possiamo però considerare:

- 0x0000000000: indirizzo generical call su I2C che potrebbe essere utilizzato per effettuare broadcast. Si è scelto di non utilizzare questo indirizzo e di non effettuare broadcast su I2C poichè non è presente ACK da parte degli slave che ricevono il messaggio. Ciò non lo differenzerebbe da CAN.
- 0x0000000001: indirizzo scelto per realizzare una comunicazione broadcast su CAN facendo sì che ogni nodo inserisca nella propria coda RX i messaggi con questo ID. Il meccanismo è stato realizzato aggiungendo ad ogni nodo un ulteriore filtro con questo ID.

4.3.2 Architettura Software

4.3.2.1 Descrizione API

L'architettura software, come richiesto, è composta da due livelli. Il livello più alto astrae le seguenti primitive che vengono offerte all'applicativo, ovvero:

```

1 void CRC_Check(uint32_t * ReceivedFrame);
2 uint8_t Receive_CRC(uint32_t * ReceivedData, uint8_t channel, uint16_t
   address);
3 uint8_t Send_CRC(uint32_t * MSG, uint16_t address, uint8_t channel, uint8_t
   mode);
4 void Configure_Peripheral(uint8_t peripheral, uint16_t nodeAddress, uint16_t
   groupAddress);

```

- **CRC_Check**: prende in ingresso un puntatore al buffer contenente il messaggio ricevuto. Quest'ultimo è composto dal payload e dai due CRC di 32 bit calcolati dal nodo che ha trasmesso il messaggio. La funzione calcola i due CRC relativi al payload ricevuto e li confronta con quelli ricevuti, se sono uguali accende i led 10 e 3 di colore verde. Dopo aver calcolato i CRC ed effettuato il confronto sosstituisce i CRC presenti nel frame con quelli appena calcolati.
- **Send_CRC**: prende in ingresso un puntatore al buffer contenente il messaggio da trasmettere, l'indirizzo del nodo destinazione (utilizzato solo da I2C, SPI e CAN), una maschera che indica su quali canali effettuare la trasmissione e la modalità di trasmissione. Nel precedente paragrafo è stato evidenziato che l'unico canale su cui è possibile effettuare una comunicazione in modalità multicast e broadcast risulti essere CAN. Se l'utente dovesse scegliere una modalità non compatibile ai canali selezionati la funzione ritornerà il valore -1, negando dunque la trasmissione, altrimenti ritorna una maschera indicante le periferiche sulle quali si è

effettuata la trasmissione. Se la maschera dei canali in ingresso indica che l'invio debba essere effettuato su più canali questo avviene in maniera prettamente sequenziale. La trasmissione su una periferica non inizia se non è stato rilevato il completamento di quella precedente.

- **Receive_CRC**: prende in ingresso un puntatore al buffer che verrà utilizzato per salvare il messaggio ricevuto, una maschera che indica i canali sui quali effettuare la ricezione e il parametro address, utilizzato nel caso la trasmissione sia richiesta su SPI per effettuare il matching con il corrispondente segnale di Slave Select, la funzione restituisce la maschera indicante le periferiche sulle quali si è effettuata la ricezione. Tale operazione ha lo stesso comportamento adottato nella trasmissione. Avrebbe potuto aver senso, dato l'utilizzo degli interrupt, abilitare tutte le periferiche in ricezione e poi controllarne ciclicamente lo stato in Round Robin, ma non è stata seguita tale scelta perchè si suppone che tutte le periferiche ricevano dallo stesso nodo mentre si trovano all'interno della funzione di receive, quindi il comportamento ottenuto sarebbe stato il medesimo. Nel caso di utilizzo di più periferiche per la ricezione il messaggio viene sovrascritto a ogni nuova ricezione, ma il calcolo del CRC viene effettuato ad ogni singola intercettazione di un messaggio.

	UART	I2C	SPI	CAN
UNICAST	✓	✓	✓	✓
MULTICAST	✗	✗	✗	✓
BROADCAST	✗	✗	✗	✓

Tabella 4.1: Modalità di trasmissione

-
- **Configure_Peripheral**: prende in ingresso una maschera la quale indica le periferiche da configurare, l'indirizzo del nodo e del gruppo a cui appartiene il nodo, per poter inizializzare le periferiche CAN e I2C. Nel caso il nodo sia master allora la periferica SPI verrà configurata come tale, altrimenti come slave.

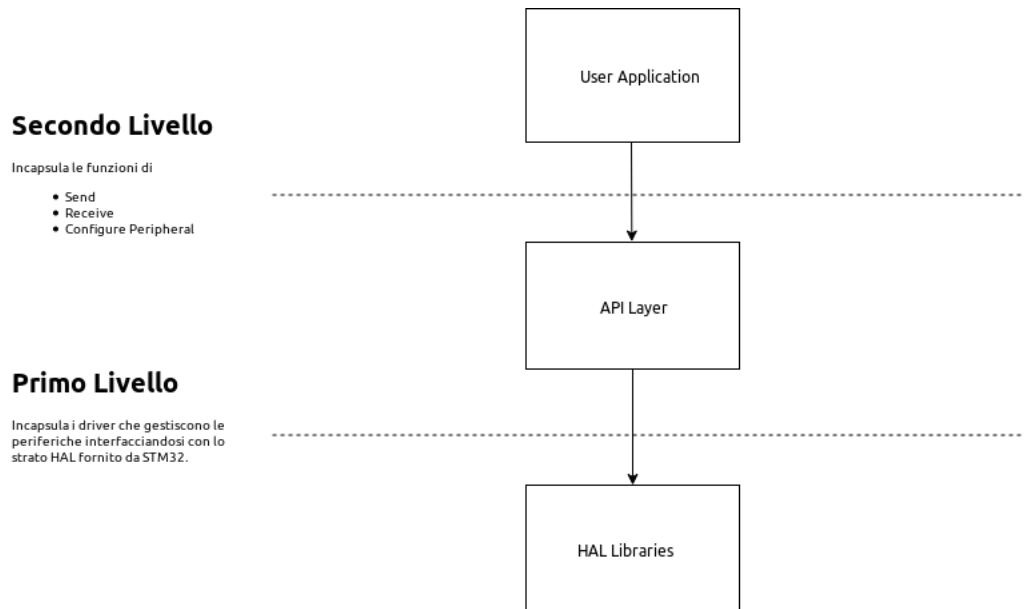


Figura 4.3: Architettura Software

Il secondo livello è composto da tutti i driver implementati per l'inizializzazione e l'interfacciamento con le varie periferiche utilizzate.

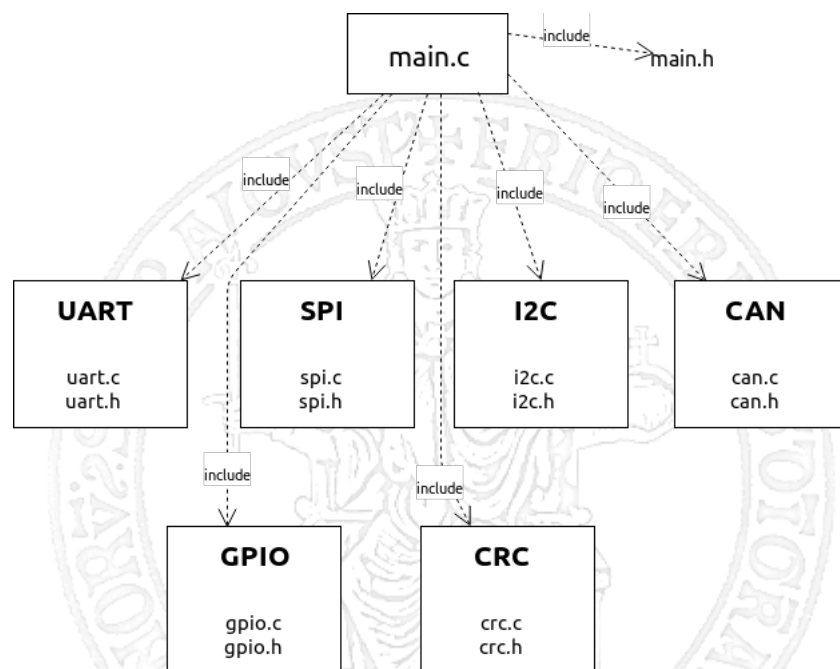


Figura 4.4: Module view

4.3.2.2 CAN

Come già detto in precedenza, i filtri CAN sono stati impostati in modalità ID_LIST con scala a 16 bit. Viene assegnato dunque l'indirizzo del nodo ad IdHigh e l'indirizzo del gruppo ad IdLow. Segue il codice relativo a tale configurazione:

```

1 sFilterConfig.FilterMode = CAN_FILTERMODE_IDLIST;
2 sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT;
3 sFilterConfig.FilterIdHigh = nodeAddress<<5;
4 sFilterConfig.FilterIdLow = groupAddress<<5;

```

Viene mostrato in seguito come viene costruito un Frame CAN A (si noti che è supportato anche il protocollo CAN B). E' necessario costruire l'header del messaggio specificando:

1. Standard CAN Identifier: identificativo del messaggio codificato su 11 bit secondo il protocollo CAN Standard.
2. Extended CAN Identifier: identificativo del messaggio codificato su 29 bit secondo il protocollo CAN Extended.
3. Tipo di messaggio da trasmettere.
4. Tipo di identificativo per il messaggio da trasmettere: *Standard* o *Extended*.
5. Lunghezza in byte del messaggio da trasmettere. Può assumere un valore da 0 ad 8.
6. Timestamp acquisito all'avvio della trasmissione del Frame.

```

1 TxHeader.StdId = address;
2 TxHeader.ExtId = 0x00;
3 TxHeader.RTR = CAN_RTR_DATA;
4 TxHeader.IDE = CAN_ID_STD;
5 TxHeader.DLC = 8;
6 TxHeader.TransmitGlobalTime = DISABLE;

```

```

1 HAL_CAN_AddTxMessage(&CanHandle, &TxHeader, CanTx_Frame, &TxMailbox);

```

Creiamo tanti frame affinché possiamo trasmettere tutto il contenuto di aTxBuffer ed aggiungiamo tale messaggio alla Mailbox di invio.

```

1 void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan) {
2     HAL_CAN_GetRxMessage(&CanHandle, CAN_RX_FIFO0, &RxHeader, CanRx_Frame);
3     if((RxHeader.RTR != CAN_RTR_DATA) || (RxHeader.IDE != CAN_ID_STD) || (
4         RxHeader.DLC != 8)){
5         Error_Handler(); }
6     else{
7         BSP_LED_On(LED5);
8         for(int k=0; k<8; k++){
9             CAN_RxBuffer[rx_callback_count*8+k] = CanRx_Frame[k];
10        }
11        rx_callback_count++;
12        if(rx_callback_count == CAN_CALLBACK_COUNT){

```

```

13     uint32_t ReceivedData[FRAME_SIZE];
14     Frame8to32(CAN_RxBuffer, ReceivedData);
15     CRC_Check(ReceivedData); }
16 }

```

Quando un messaggio CAN arriva al dispositivo viene attivata questa Callback che come prima operazione preleva il frame dalla mailbox e controlla con i campi del frame che il messaggio sia corretto, oltre a verificare l' integrità del messaggio con il CRC.

4.3.2.3 I2C

Per quando riguarda I2C la parte importante riguarda l' assegnazione dell' indirizzo da dare alla periferica, come mostrato nel codice

```

1 hi2c2.Init.OwnAddress1 = nodeAddress;
2 hi2c2.Init.AddressingMode = I2C_ADDRESSINGMODE_10BIT;

```

Per la trasmissione semplicemente basta indicare chi contattare, cosa inviare e quanto grande sia questo dato.

```

1 HAL_I2C_Master_Transmit_IT(&hi2c2, (uint16_t) address, (uint8_t*) aTxBuffer,
    BUFFER_SIZE)

```

Come possiamo notare la board quando invia è master del bus.

Per quanto riguarda la ricezione, dato che essa avviene da slave, non va indicato alcun indirizzo.

```

1 HAL_I2C_Slave_Receive_IT(&hi2c2, (uint8_t *)I2C_RxBuffer, BUFFER_SIZE)

```

4.3.2.4 SPI

Per SPI riguardo alla configurazione non ci sono particolari riflessioni da fare. E' interessante notare il meccanismo necessario per indirizzare la comunicazione. La funzione Send_CRC prende come parametro di ingresso un indirizzo per indentificare il destinatario, mentre SPI necessita di abilitare un pin per dare il segnale di SS. La funzione getSSPin dovrebbe dunque conoscere tutta la configurazione della rete ed effettuare la corrispondenza. Nel nostro caso, ai fini dell'applicativo di prova si è scelto di implementare uno STUB.

```

1 uint16_t getSSPin(uint16_t address){
2     return SPI_EN_OUTPUT_Pin;
3 }

```

Si noti che SPI prevede che il segnale di SS allo slave venga fornito mediante il PIN NSS della periferica. Quando NSS diviene low vuol dire che va abilitata la transazione con il master. Per evitare funzionamenti oscillanti del pin NSS nativo dell'STM, seguendo i consiglio della community ST e dopo vari test, si è scelto di ricreare il medesimo meccanismo utilizzando un GPIO opportunamente configurato in sostituzione del pin NSS. Dunque la gestione sarà software e non più

hardware. Se il nodo è master sul bus allora, tramite un suo GPIO (necessario per realizzare il multislave) dovrà abbassare il segnale prima di iniziare la transazione, se il nodo è uno slave dovrà aspettare che il segnale SS (in lettura su un pin GPIO deciso da noi) diventi basso.

```

1 #ifdef MASTER_BOARD
2 uint16_t slaveSelectPin = getSSPin(address);
3 HAL_GPIO_WritePin(SPI_EN_OUTPUT_GPIO_Port, slaveSelectPin , GPIO_PIN_RESET);
4 #else
5 while (HAL_GPIO_ReadPin(SPI_EN_OUTPUT_GPIO_Port, SPI_EN_INPUT_Pin) !=
        GPIO_PIN_RESET) {      }
6 #endif /* MASTER_BOARD */

```

4.3.2.5 UART

```

1 huart2.Init.BaudRate = Baudrate;
2 huart2.Init.WordLength = UART_WORDLENGTH_8B;
3 huart2.Init.StopBits = UART_STOPBITS_1;
4 huart2.Init.Parity = UART_PARITY_NONE;

```

Nella configurazione di UART è necessario indicare la velocità di comunicazione, la struttura del frame da inviare: il numero di bit per il dato, quanti bit di stop e se utilizzare il bit di parità.

Per la trasmissione non ci sono ulteriori note, mentre invece per la ricezione prima di effettuare la prossima si attende che il pin RX non sia più utilizzato.

```

1 while (HAL_UART_GetState(&huart2) == HAL_UART_STATE_BUSY_RX) {
2 }
3 if (HAL_UART_Receive_IT(&huart2, (uint8_t *) UART_RxBuffer, BUFFER_SIZE) !=
    HAL_OK)
4     Error_Handler();

```

4.3.2.6 CRC

La periferica adibita a calcolare il CRC viene configurata in modo tale da utilizzare i polinomi descritti nella specifica

```

1 hcrc.Init.DefaultPolynomialUse = DEFAULT_POLYNOMIAL_DISABLE;
2 hcrc.Init.GeneratingPolynomial = CRC_Polynomial;
3 hcrc.Init.CRCLength = CRC_POLYLENGTH_32B;
4 hcrc.Init.DefaultInitValueUse = CRC_DefaultValue;

```

Abbiamo lasciato la possibilità di utilizzare un valore di inizializzazione nel caso in un successivo futuro questo dovesse essere richiesto.

Si noti che è necessario configurare due volte la periferica per utilizzare polinomi diversi.

```

1 MX_CRC_Init(CRC_POLYNOMIAL_1, CRC_DEFAULTVALUE_1);
2 uint32_t CRC32_1 = HAL_CRC_Calculate(&hcrc, (uint32_t *) ReceivedFrame,
    PAYLOAD_SIZE);
3 MX_CRC_Init(CRC_POLYNOMIAL_2, CRC_DEFAULTVALUE_2);
4 uint32_t CRC32_2 = HAL_CRC_Calculate(&hcrc, (uint32_t *) ReceivedFrame,
    PAYLOAD_SIZE);
5 if (CRC32_1 != ReceivedFrame[CRC1_OFFSET])
6     HAL_GPIO_WritePin(GPIOE, LED10_RED_Pin, GPIO_PIN_SET);
7 else
8     HAL_GPIO_WritePin(GPIOE, LED7_GREEN_Pin, GPIO_PIN_SET);
9 if (CRC32_2 != ReceivedFrame[CRC2_OFFSET])
10    HAL_GPIO_WritePin(GPIOE, LED3_RED_Pin, GPIO_PIN_SET);
11 else
12    HAL_GPIO_WritePin(GPIOE, LED6_GREEN_Pin, GPIO_PIN_SET);

```

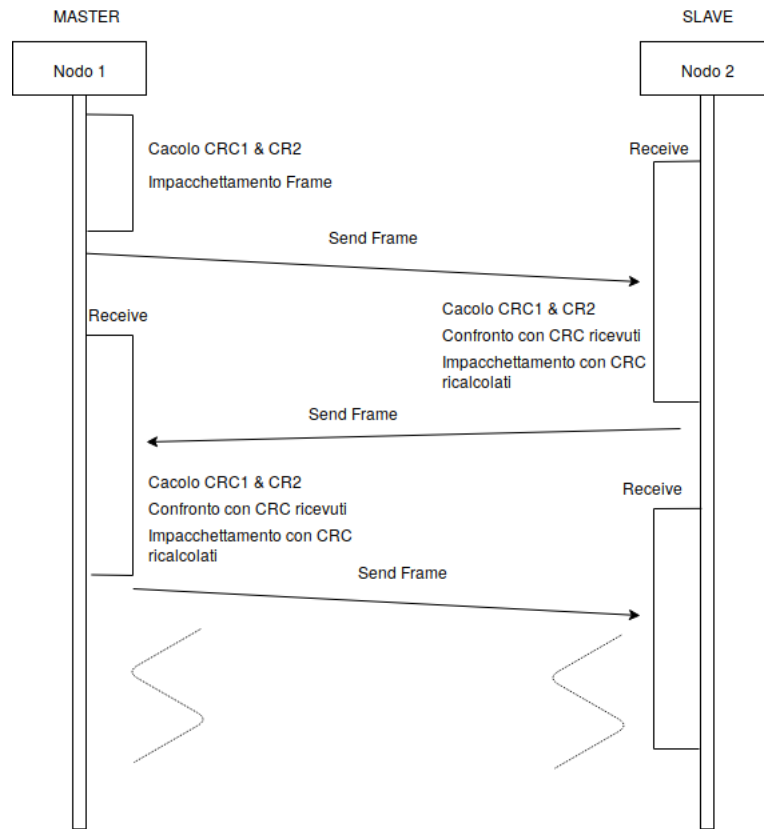
Alla ricezione di un messaggio vengono prelevati i valori dei CRC da una posizione stabilita nel messaggio (essendo il messaggio di dimensione fissa), si riefettua il calcolo con i polinomi stabili e confrontati per controllare se sono identici con quelli ricevuti.

4.3.3 Applicativo Utente

Per testare le corretto funzionamento dell'architettura sottostante e eseguire una limitata simulazione del sistema è stato realizzato un applicativo che permettere la comunicazione fra un master e uno slave del sistema. L'algoritmo eseguito ciclicamente è il seguente:

1. Ricezione Messaggio
2. Ricalcolo, confronto CRC e sostituzione dei CRC nel messaggio
3. Invio del nuovo Messaggio

Nell'applicativo la differenza fra nodo Master e nodo slave è che il primo inizia il suo workflow con un init con valori random del messaggio da mandare, un calcolo dei due CRC, imballaggio del messaggio e una successiva Send allo slave. Segue uno schema del flusso:

Algoritmo 4.1 Worklow test Main

Il main test è facilmente configurabile in:

- Dimensione del messaggio da inviare: Il messaggio viene generato casualmente in un ciclo for iniziale. Per settarne la dimensione è sufficiente definire la dimensione del `PAYLOAD_SIZE` nel `main.h` indicando come valore `(numero_byte_da_inviare*4)`.
- Periferiche sulle quali effettuare la trasmissione:
 - Ogni periferica è codificata da un opportuna define `[UART_MODE, I2C_MODE, SPI_MODE]`
 - Il parametro `SERIAL_SELECT` di ingresso della `Sand_CRC` può essere settato tramite `#define` mettendo in *OR bitwise* le periferiche se si vogliono usare. ES. `[UART_MODE | I2C_MODE]` per utilizzare UART e I2C

La modalità CAN viene utilizzata in modalità di test, in quanto il componente è configurato in `LOOP_BACK` (allacciato su se stesso) e inserendo come indirizzo della send uno dei valori (`NODE_ADDRESS` e `GROUP_ADDRESS`) passati alla funzione di configurazione delle periferiche. Inoltre deve essere effettuata solo la send altrimenti la receive rimarrebbe bloccata in attesa di ricezione. Il codice per un eventuale funzionamento non in loopback è comunque commentato nella receive. Le altre periferiche possono essere utilizzate in contemporanea. SPI potrebbe generare malfunzionamenti dovuti alla connessione, UART e I2C risultano stabili.

4.3.3.1 Connessioni fische broad

Si riporta in seguito la connessione delle board utilizzate nell'attuale configurazione dei driver. Per SPI è necessario collegare esclusivamente PD8 sul master con PD9 dello slave e non viceversa.

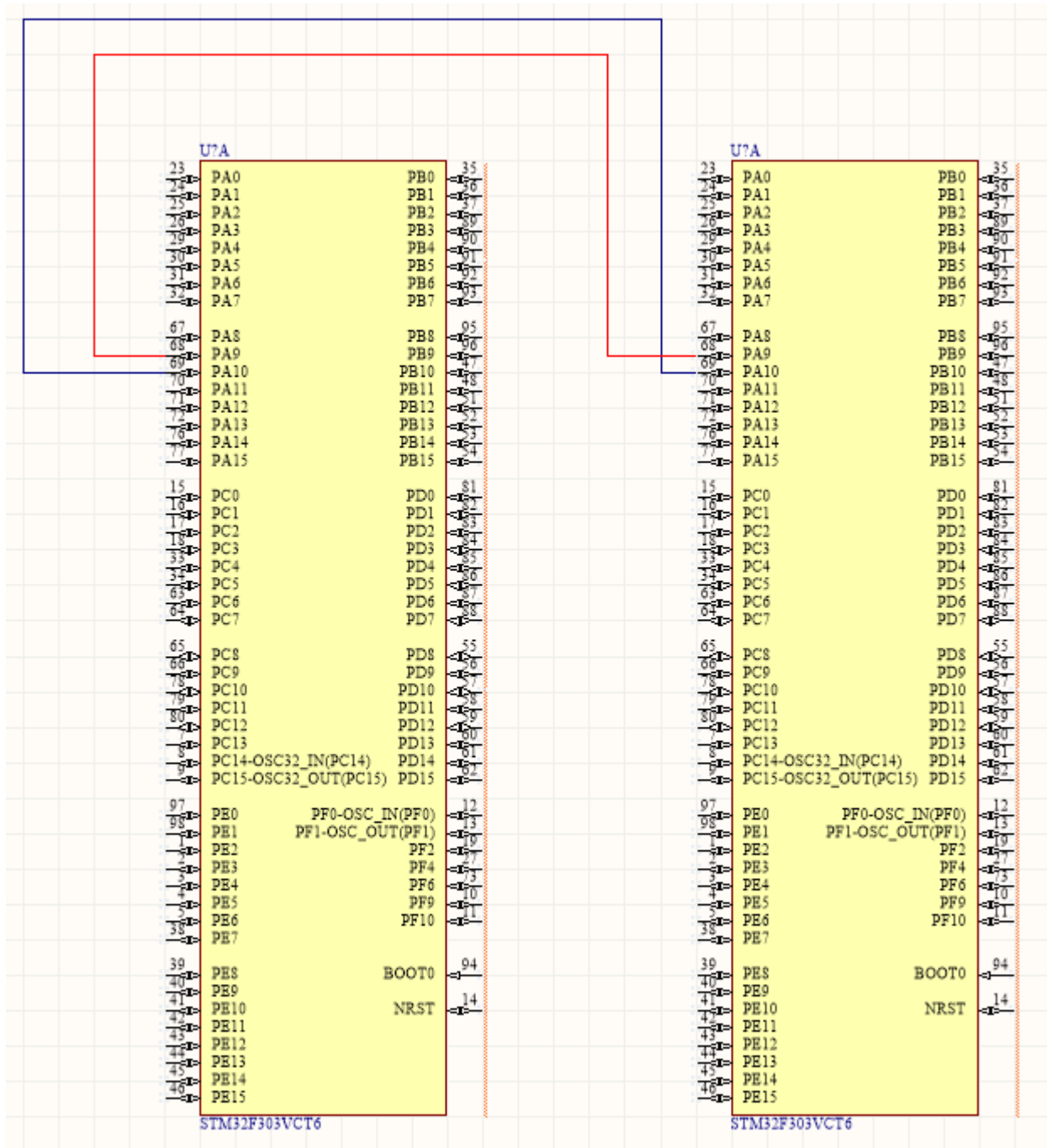


Figura 4.6: I2C

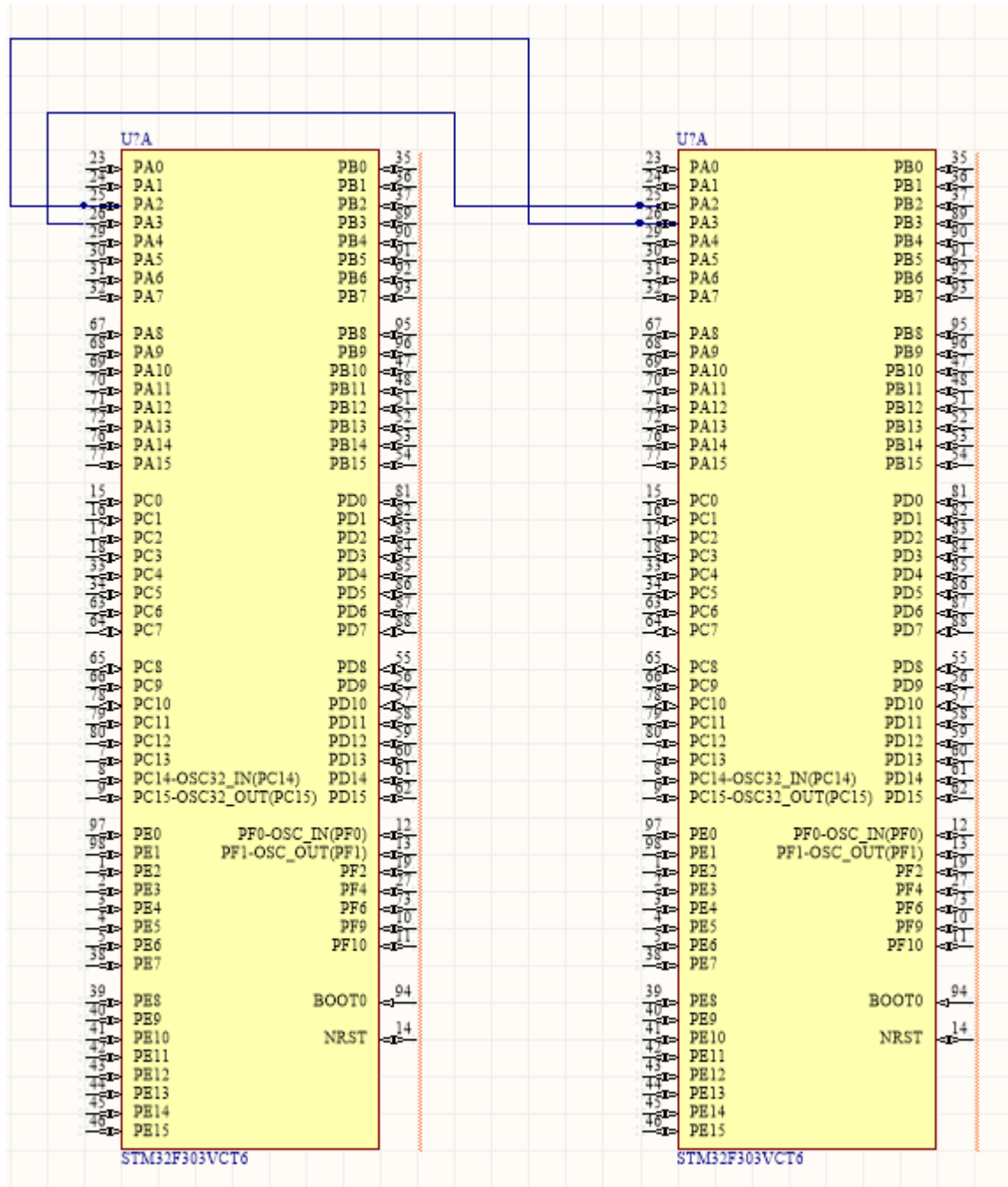


Figura 4.5: UART

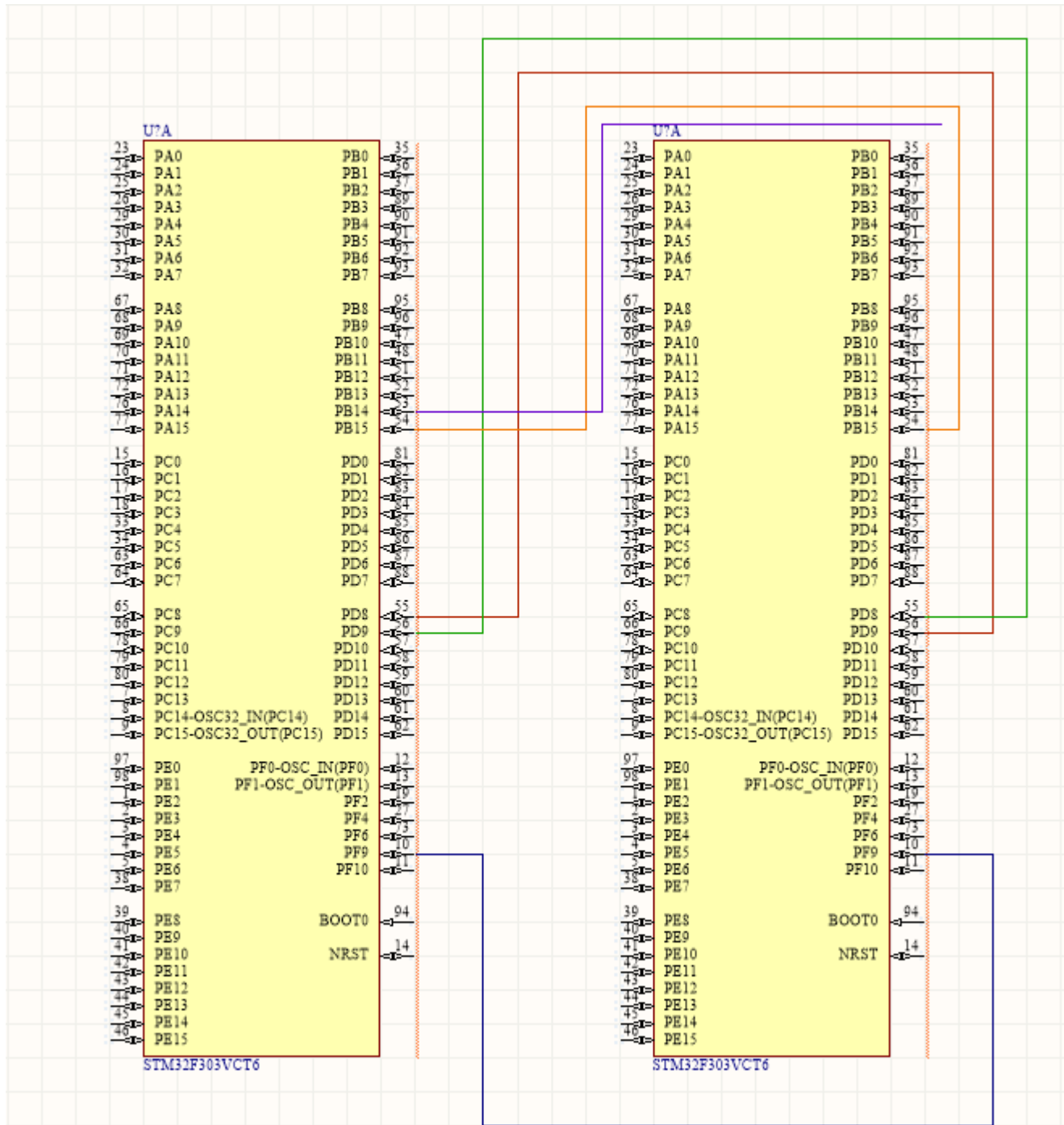


Figura 4.7: SPI