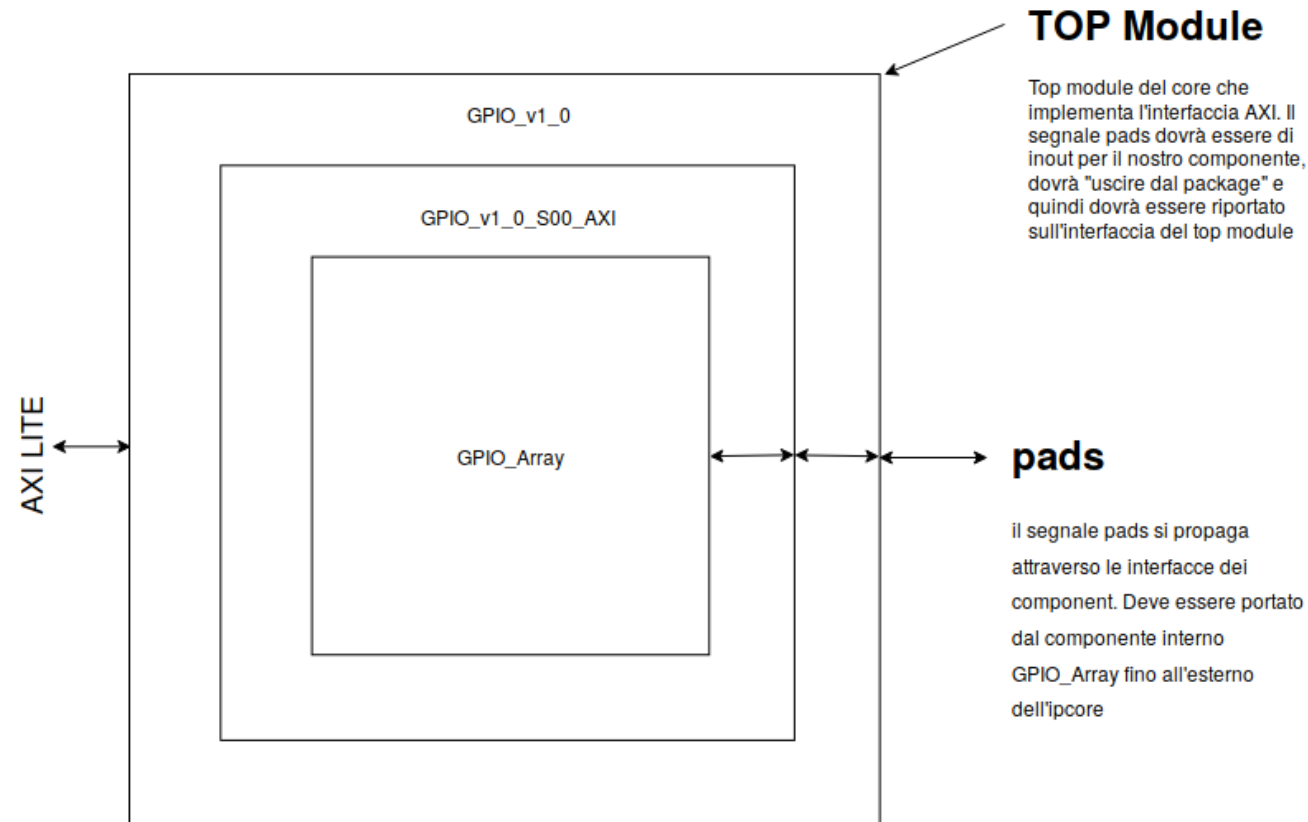


Custom IP AXI: GPIO

1. Realizzare un componente GPIO
2. Connettere il componente al bus AXI
3. Implementare i driver per la gestione del componente

Connessione del componente al bus AXI

Inserimento del componente GPIO realizzato all'interno di un **wrapper** fornito da Vivado per l'implementazione dell'interfaccia AXI



Connessione del componente al bus AXI

- Gestione registri:

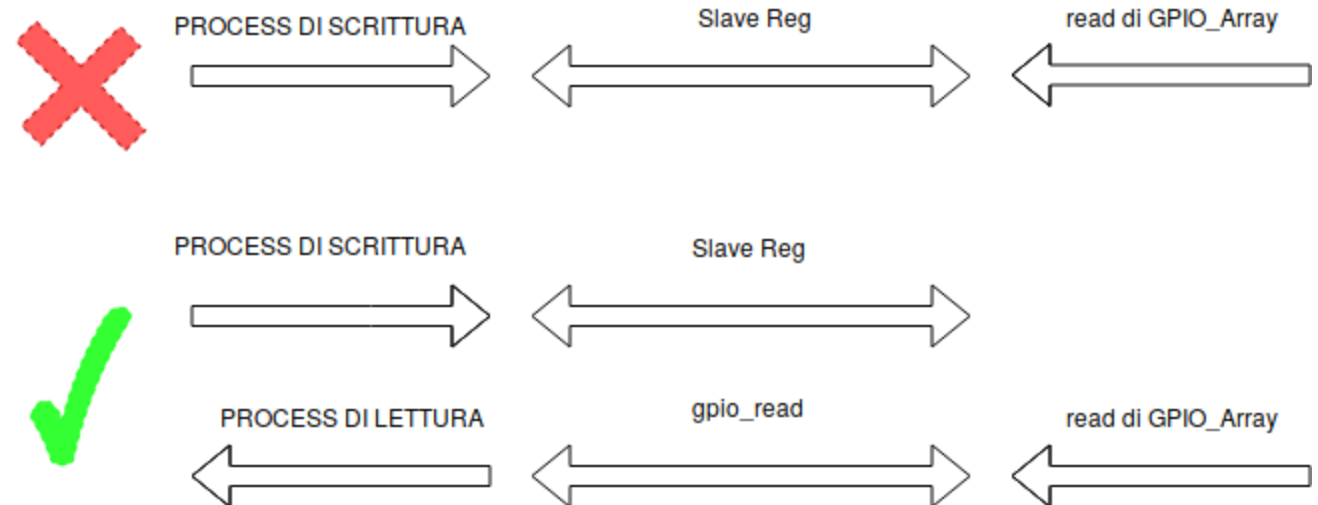
Connessione dei segnali del nostro componente GPIO al bus.

ENABLE -> slv_reg0

WRITE -> slv_reg1

READ -> gpio_read

Per il segnale di read non viene utilizzato uno degli *slv_reg* poiché essendo un segnale di output per il GPIO si genererebbe un conflitto in scrittura con il bus.



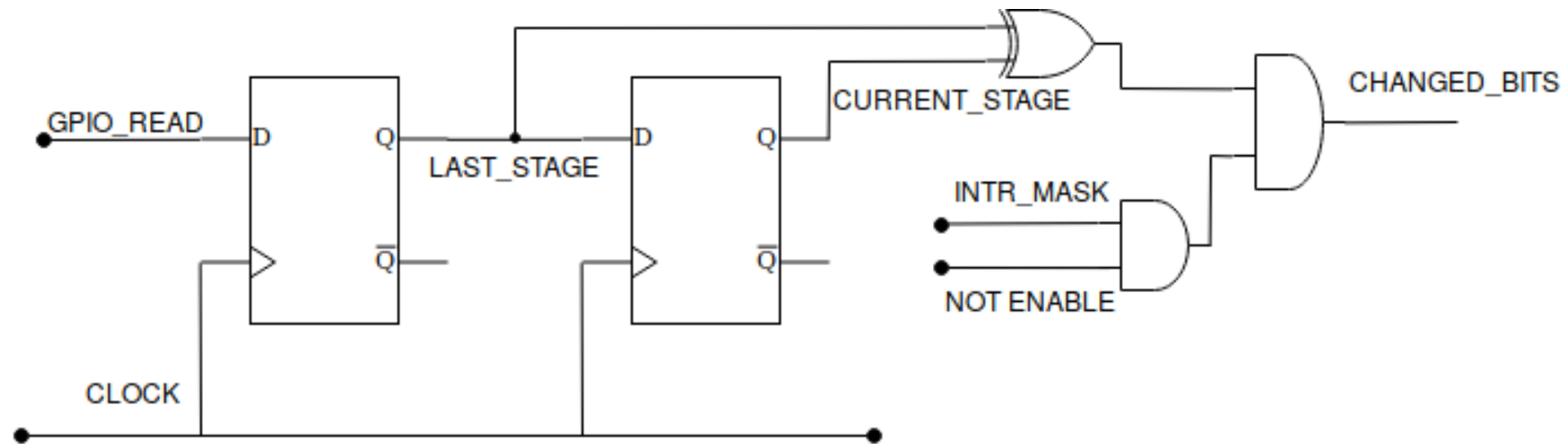
Gestione interruzioni

Siamo interessati a generare un evento di interruzione ogni qual volta vi sia *una variazione* del segnale di READ del componente 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

Gestione Interruzioni

Il *process vhdl* che gestisce la rilevazione dei fronti può essere schematizzato come segue:



Gestione Interruzioni

Gestione registro interruzioni pendenti:

change_detected <= global_intr and or_reduce(changed_bits)

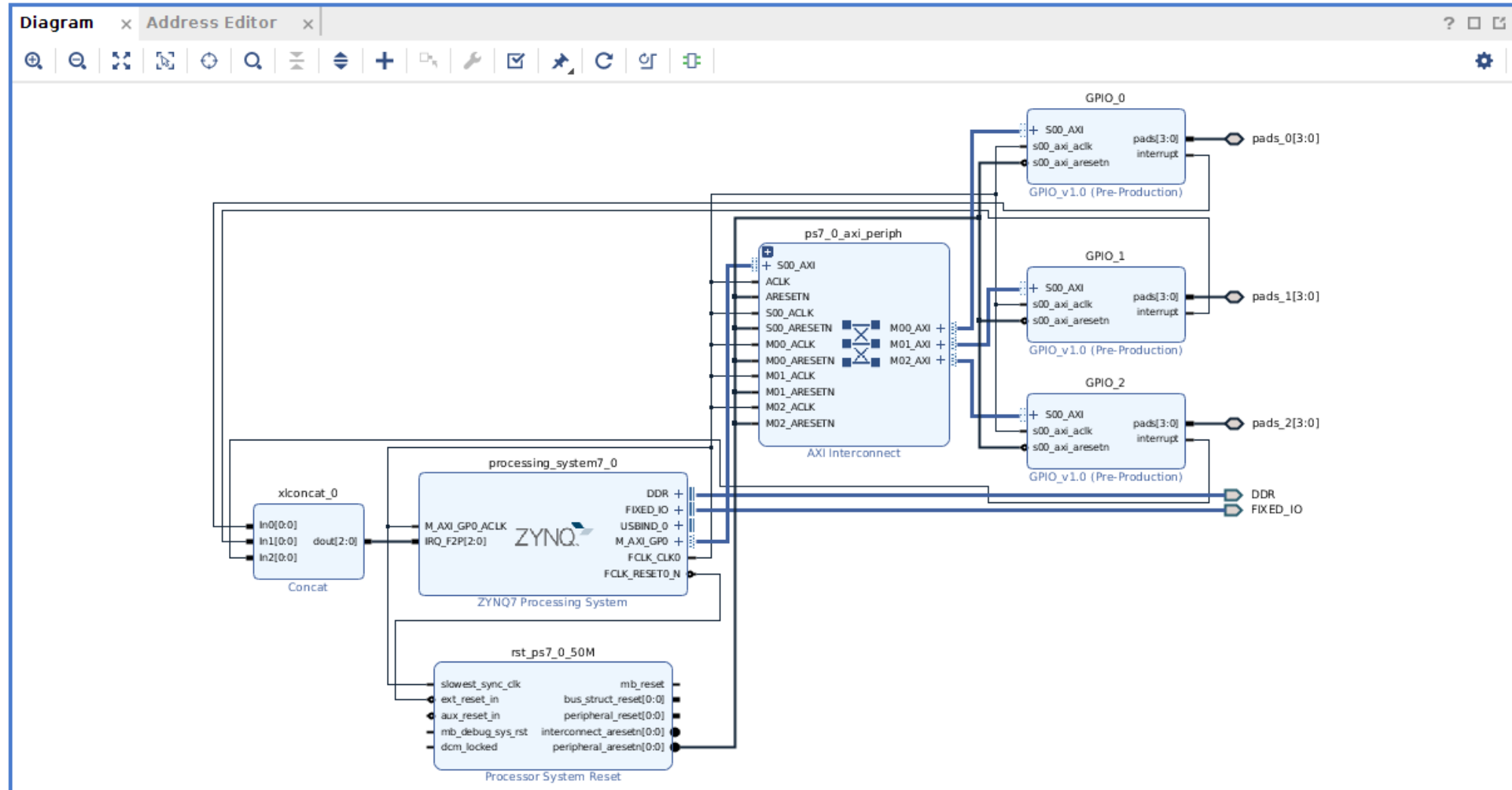
```
intr_pending : process (S_AXI_ACLK, change_detected, ack_intr, pending_intr_tmp)
begin
  if (rising_edge (S_AXI_ACLK)) then
    if (change_detected = '1') then
      pending_intr <= pending_intr_tmp or changed_bits;
    elsif (or_reduce(ack_intr)='1') then
      pending_intr <= pending_intr_tmp and (not ack_intr);
    else
      pending_intr <= pending_intr_tmp;
    end if;
  end if;
end process;
```

Gestione Interruzioni

- Gestione segnale interruzione verso il processore

```
inst_irq : process(S_AXI_ACLK,pending_intr, global_intr)
begin
    if (rising_edge (S_AXI_ACLK)) then
        if ( S_AXI_ARESETN = '0' ) then
            interrupt <= '0';
        else
            if (or_reduce(pending_intr) = '1' and global_intr = '1') then
                interrupt <= '1';
            else
                interrupt <= '0';
            end if;
        end if;
    end if;
end process;
```

Block Design

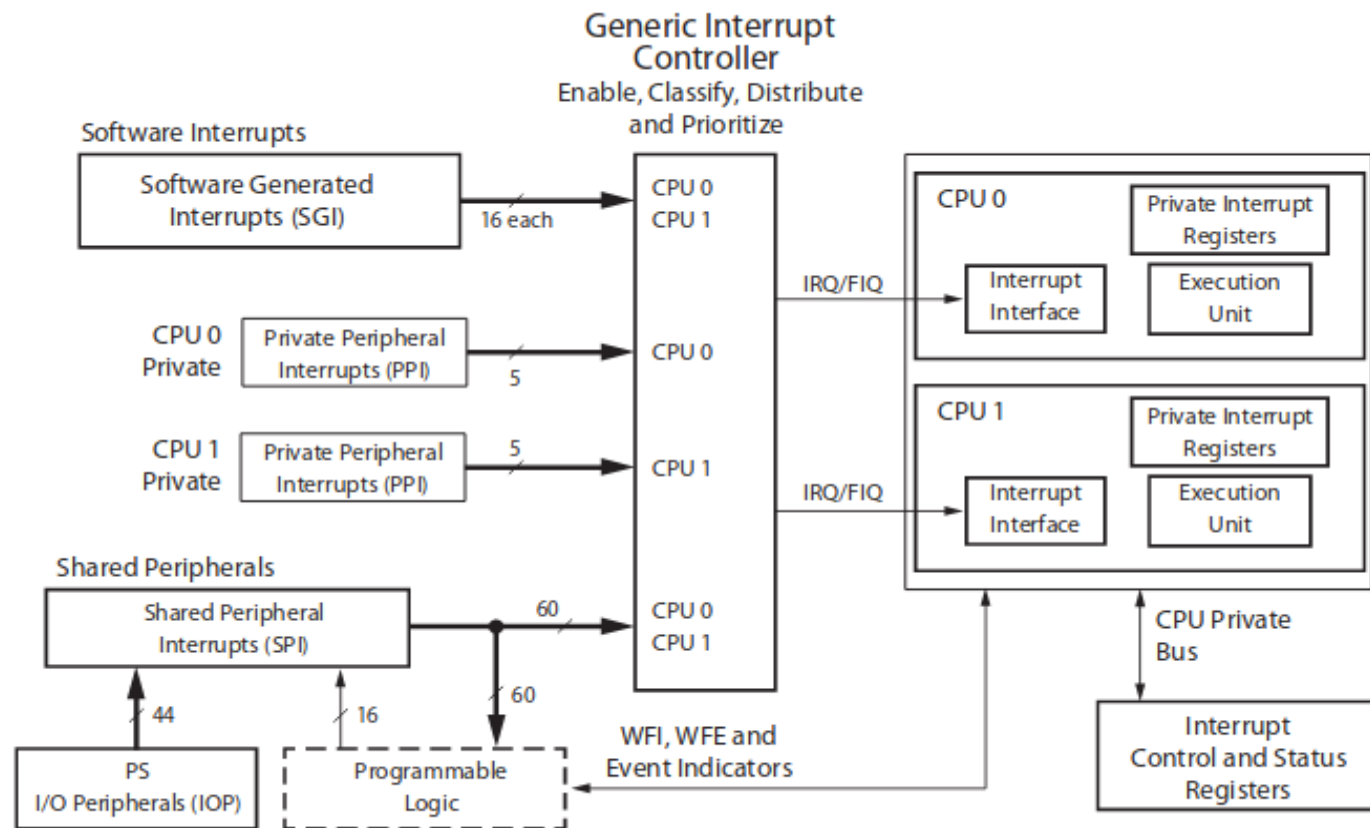


Driver

- Driver Standalone
- Driver con supporto del SO Linux:
 - Kernel Mode
 - Userspace I/O

Driver Standalone

- Interazione diretta con l'hardware e la PS della Zynq 7000, composta da Cortex-A9 e un GIC pl390 interrupt controller



Driver Standalone - Configurazione

1. Configurazione del GIC tramite driver forniti da *Xilinx* nella libreria *scugic*
2. Abilitare la gestione delle eccezioni relative al GIC (opzionale)
3. Registrare gli **handler** alle 3 linee di interruzione
4. Abilitazione linee di interruzione

Driver Standalone - ISR

1. Disabilitazione interruzioni globali
2. Verifica quale delle linee chiedono di essere servite
3. Da l'ACK alle linee pendenti
4. Riabilita le interruzioni globali del componente

```
void SwitchISR() {  
    XGPIO_GlobalDisableInterrupt(&GPIO_Switch, 0x01);  
    InterruptProcessed = TRUE;  
    print("\n\n*****ISR SWITCH*****\n\n");  
    uint8_t pendingReg = XGPIO_GetPending(&GPIO_Switch);  
    XGPIO_ACK(&GPIO_Switch, pendingReg);  
    XGPIO_GlobalEnableInterrupt(&GPIO_Switch, 0x01);  
}
```

Driver Linux – Kernel Mode

- Driver scritto sotto forma di modulo kernel e inserito *dinamicamente* all'interno del kernel fornendo più flessibilità rispetto al «*building statico*».
- Nei SO UNIX le periferiche (*/dev*) sono rappresentate da due tipologie speciali di file:
 - Device a **Blocchi**: dispositivi che effettuano operazioni di I/O per blocchi di bytes (memorie di massa)
 - Device a **Caratteri**: dispositivi seriali/paralleli che comunicano a caratteri.

Modulo Kernel

```
/**
 * @brief Identifica il device all'interno del device tree
 *
 */
static const struct of_device_id __test_int_driver_id[]={
    {.compatible = "GPIO"},
    {}
};

/**
 * @brief Struttura che specifica le funzioni che agiscono sul device
 *
 */
static struct file_operations GPIO_fops = {
    .owner      = THIS_MODULE,
    .llseek     = GPIO_llseek,
    .read       = GPIO_read,
    .write      = GPIO_write,
    .poll       = GPIO_poll,
    .open       = GPIO_open,
    .release    = GPIO_release
};
```

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.

Modulo Kernel

```
/**
 * @brief Definisce le funzioni probe() e remove() da chiamare al caricamento del driver.
 */
static struct platform_driver GPIO_driver = {
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(__test_int_driver_id),
    },
    .probe = GPIO_probe,
    .remove = GPIO_remove
};

/**
 * @brief la macro module_platform_driver() prende in input la struttura platform_driver ed implementa le
 * funzioni module_init() e module_close() standard, chiamate quando il modulo viene caricato o
 * rimosso dal kernel.
 *
 * @param GPIO_driver struttura platform_driver associata al driver
 */
module_platform_driver(GPIO_driver);
```

Astrazione del componente GPIO

```
typedef struct {
/** Major e minor number associati al device
 * (M: identifica il driver associato al device;
 * m: utilizzato dal driver per discriminare il singolo device tra quelli a lui associati)*/
    dev_t Mm;
/** Puntatore a struttura platform_device cui l'oggetto GPIO si riferisce */
    struct platform_device *pdev;
/** Struttura per l'astrazione di un device a caratteri */
    struct cdev cdev;
/** Puntatore alla struttura che rappresenta l'istanza del device*/
    struct device* dev;
/** Puntatore a struttura che rappresenta una vista alto livello del device*/
    struct class* class;
/** Interrupt-number a cui il device è connesso*/
    uint32_t irqNumber;
/** Puntatore alla regione di memoria cui il device è mappato*/
    struct resource *mreg;
/** Device Resource Structure*/
    struct resource res;
/** Maschera delle interruzioni interne attive per il device*/
    uint32_t irq_mask;
/** res.end - res.start; numero di indirizzi associati alla periferica.*/
    uint32_t res_size;
/** Indirizzo base virtuale della periferica*/
    void __iomem *vrtl_addr;
/** wait queue per la sys-call read() */
    wait_queue_head_t read_queue;
/** wait queue per la sys-call poll()*/
    wait_queue_head_t poll_queue;
/** Flag che indica, quando asserito, la possibilità di effettuare una chiamata a read*/
    uint32_t can_read;
/** Spinlock usato per garantire l'accesso in mutua esclusione alla variabile can_read*/
    spinlock_t slock_int;
} GPIO;
```

Il modulo dispone di una lista per la gestione di più device.

Probe

Il kernel effettuerà una chiamata alla funzione **probe** per ciascun device che presenta il campo *compatible* uguale a quello specificato all'interno della struttura *of_device_id*. La funzione ha il compito di allocare la lista se essa è ancora vuota, effettuare tutte le operazioni necessarie per l'inizializzazione del device chiamando la funzione GPIO_Init e infine di aggiungere l'oggetto alla lista.

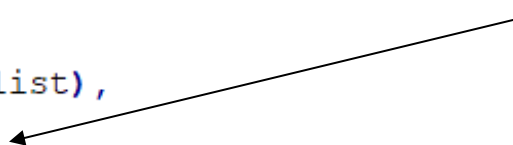
```
/**
 * @brief Inizializza una struttura GPIO per il corrispondente device
 *
 * @param GPIO_device puntatore a struttura GPIO, corrispondente al device su cui operare
 * @param owner puntatore a struttura struct module, proprietario del device (THIS_MODULE)
 * @param pdev puntatore a struct platform_device
 * @param driver_name nome del driver
 * @param device_name nome del device
 * @param serial_numero seriale del device
 * @param f_ops puntatore a struttura struct file_operations, specifica le funzioni che agiscono sul device
 * @param irq_handler puntatore irq_handler_t alla funzione che gestisce gli interrupt generati dal device
 * @param irq_mask maschera delle interruzioni attive del device
 *
 * @retval "0" se non si è verificato nessun errore
 *
 * @details
 */
int GPIO_Init(
    GPIO* GPIO_device,
    struct module *owner,
    struct platform_device *pdev,
    struct class* class,
    const char* driver_name,
    const char* device_name,
    uint32_t serial,
    struct file_operations *f_ops,
    irq_handler_t irq_handler,
    uint32_t irq_mask) {
```

Probe – Chiamata a GPIO_Init

La probe effettua una chiamata alla funzione GPIO_Init per inizializzare la struct che astrae il device GPIO, puntata da **GPIO_ptr**, passando i seguenti parametri:

```
if ((ret = GPIO_Init( GPIO_ptr,
                      THIS_MODULE,
                      pdev,
                      GPIO_class,
                      DRIVER_NAME,
                      DRIVER_FNAME,
                      GPIO_list_device_count(device_list),
                      &GPIO_fops,
                      (irq_handler_t) GPIO_irq_handler,
                      INTR_MASK)) != 0) {
    printk(KERN_ERR "%s: GPIO_Init() ha restituito %d\n", __func__, ret);
    kfree(GPIO_ptr);
    return ret;
}
```

Handler
dell'interruzione;
ridefinito all'interno
del modulo kernel



```
GPIO_list_add(device_list, GPIO_ptr);
```

Il puntatore alla struct class **Gpio_class** deve essere stato precedentemente inizializzato mediante la chiamata a:

```
GPIO_class = class_create(THIS_MODULE, DRIVER_NAME);
```

GPIO_Init

1. **Allocare un range di Major e minor numbers** per il device a caratteri

```
if ((error = alloc_chrdev_region(&GPIO_device->Mm, 0, 1, file_name)) != 0) {  
    printk(KERN_ERR "%s: alloc_chrdev_region() ha restituito %d\n", __func__, error);  
    return error;  
}
```

2. **Inizializzare la struttura cdev specificando la struttura file operations**
associata al device a caratteri

```
cdev_init (&GPIO_device->cdev, f_ops);  
GPIO_device->cdev.owner = owner;
```

GPIO_Init

3. **Creazione del device** all'interno del **filesystem** assegnandogli i numbers richiesti in precedenza. La funzione ritorna un puntatore alla struct device

```
if ((GPIO_device->dev = device_create(class, NULL, GPIO_device->Mm, NULL, file_name)) == NULL) {  
    printk(KERN_ERR "%s: device_create() ha restituito NULL\n", __func__);  
    error = -ENOMEM;  
    goto device_create_error;  
}
```

4. **Aggiungere** il device a caratteri al sistema. Se l'operazione va a buon fine sarà possibile vedere il device sotto **/dev**

```
if ((error = cdev_add(&GPIO_device->cdev, GPIO_device->Mm, 1)) != 0) {  
    printk(KERN_ERR "%s: cdev_add() ha restituito %d\n", __func__, error);  
    goto cdev_add_error;  
}
```

GPIO_Init

5. Inizializzare la struct resource con i valori recuperati dal nodo corrispondente al device all'interno del device tree

```
dev = &pdev->dev;
if ((error = of_address_to_resource(dev->of_node, 0, &GPIO_device->res)) != 0) {
    printk(KERN_ERR "%s: address_to_resource() ha restituito %d\n", __func__, error);
    goto of_address_to_resource_error;
}
```

6. Allocare una quantita *res_size* di memoria fisica per il dispositivo I/O a partire dall'indirizzo *res.start*

```
GPIO_device->res_size = GPIO_device->res.end - GPIO_device->res.start + 1;

if ((GPIO_device->mreg = request_mem_region(GPIO_device->res.start, GPIO_device->res_size, file_name)) == NULL) {
    printk(KERN_ERR "%s: request_mem_region() ha restituito NULL\n", __func__);
    error = -ENOMEM;
    goto request_mem_region_error;
}
```

GPIO_Init

7. **Mappare la memoria** fisica del dispositivo I/O appena allocata nello spazio degli indirizzi virtuali del kernel. La funzione restituisce **l'indirizzo virtuale** corrispondente al *base_address* in memoria fisica del device.

```
if ((GPIO_device->vrtl_addr = ioremap(GPIO_device->res.start, GPIO_device->res_size))==NULL) {  
    printk(KERN_ERR "%s: ioremap() ha restituito NULL\n", __func__);  
    error = -ENOMEM;  
    goto ioremap_error;  
}
```

8. **Cercare le specifiche dell'interrupt nel device tree.** La funzione restituisce il suo **numero identificativo**

```
GPIO_device->irqNumber = irq_of_parse_and_map(dev->of_node, 0);
```

GPIO_Init

9. Allocare la linea di interrupt e registrare l'handler ad essa associato. Se l'operazione va a buon fine si potrà osservare il numero della linea associata all'IRQ del device sotto */proc/interrupts* (comprese altre informazioni quali il numero di interruzioni rilevate o il tipo di interrupt)

```
if ((error = request_irq(GPIO_device->irqNumber , irq_handler, 0, file_name, NULL)) != 0) {  
    printk(KERN_ERR "%s: request_irq() ha restituito %d\n", __func__, error);  
    goto irq_of_parse_and_map_error;  
}  
GPIO_device->irq_mask = irq_mask;
```

GPIO_Init

10. Inizializzare wait_queue, spinlock e variabile can_read

```
Inizializzazione della wait-queue per la system-call read() e poll() */
```

```
init_waitqueue_head(&GPIO_device->read_queue);  
init_waitqueue_head(&GPIO_device->poll_queue);
```

```
Inizializzazione degli spinlock */
```

```
spin_lock_init(&GPIO_device->slock_int);  
GPIO_device->can_read = 0;
```

11. Abilitare interruzioni del device

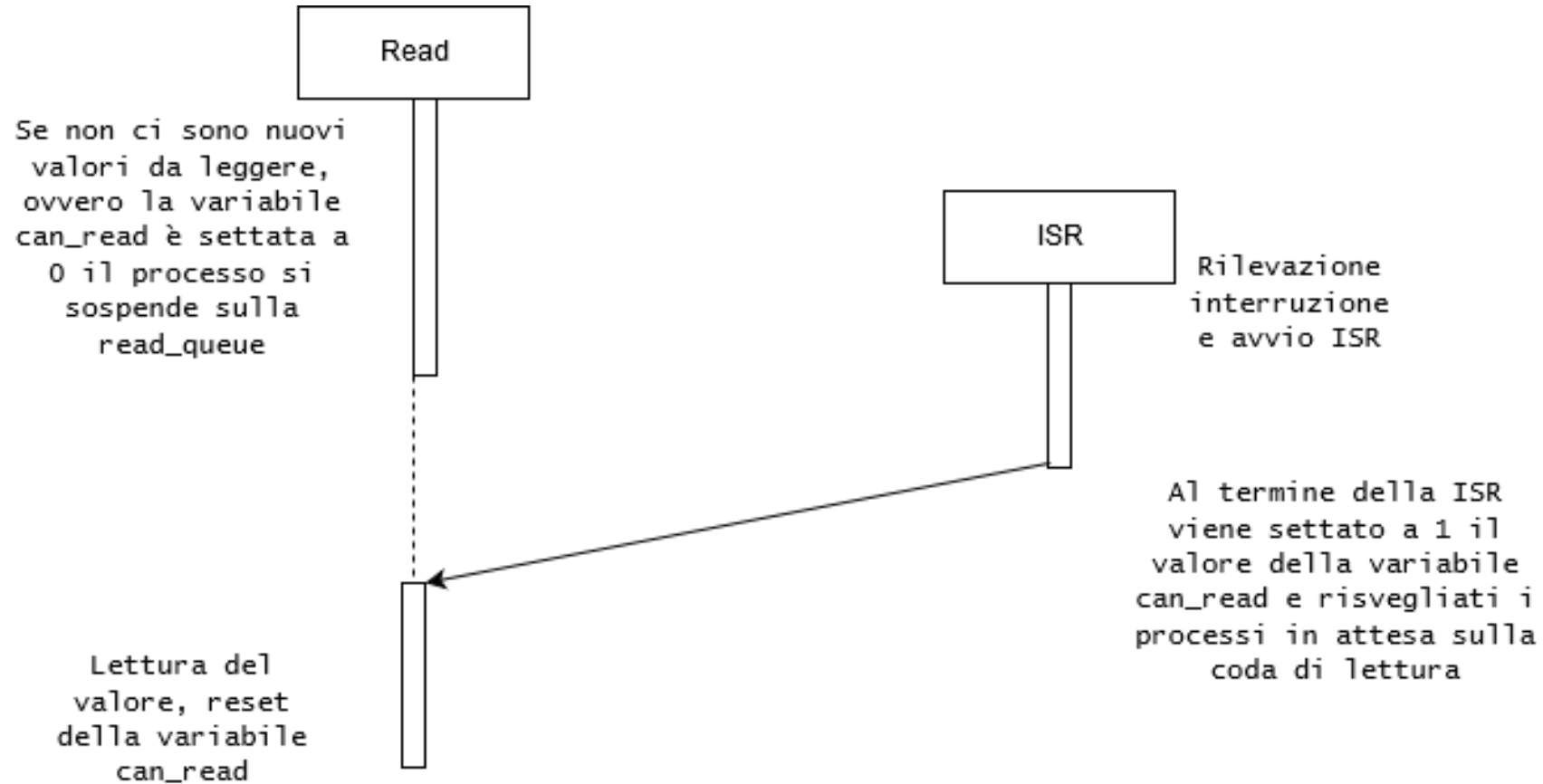
```
GPIO_GlobalInterruptEnable(GPIO_device);  
GPIO_PinInterruptEnable(GPIO_device, GPIO_device->irq_mask);
```

```
printk(KERN_INFO " IRQ registered as %d\n", GPIO_device->irqNumber);  
printk(KERN_INFO " Driver succesfully probed at Virtual Address 0x%08lx\n", (unsigned long) GPIO_device->vrtl_addr);
```


User application: test modulo

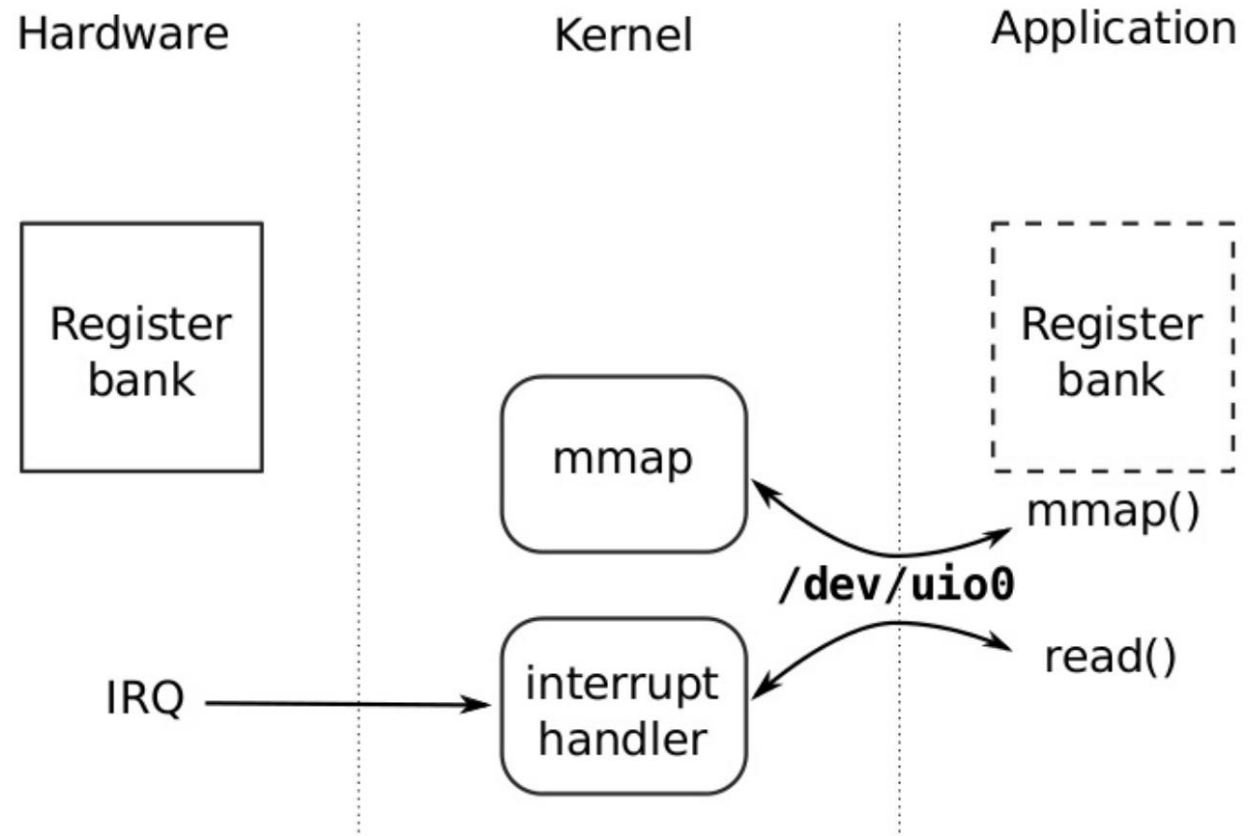
- Per testare il funzionamento del modulo Kernel è stata realizzata un user application interattiva che permette all'utente di scegliere quale device utilizzare.
- L'user app suppone che *pads* sia pilotato dall'esterno e che si voglia esclusivamente leggerne il valore quando vi è una variazione (es. premere un button).
- Semplicemente l'user app esegue le seguenti operazioni:
 - OPEN *sul file descriptor* del GPIO selezionato
 - Si mette in attesa di un'interruzione con una READ bloccante.

User application: workflow



Driver Userspace I/O

- Framework per la gestione dei driver nell'*userspace*.
- Gestione dell'interruzione demandata *all'userspace*.



Workflow esempio user application - UIO

1. Apertura descrittore del file sul device `/dev/uio0`
2. Effettuare il mapping con l'indirizzo virtuale **mmap**
3. Attesa di interruzioni dal device tramite chiamata bloccante/non bloccante a **read** (o in alternativa a **poll**)
4. Quando il *sottosistema UIO* rileva un'interruzione provvederà a risvegliare il processo
5. Chiamata **write** per indicare al sottosistema che l'interruzione è stata gestita e che può riabilitare le interruzioni

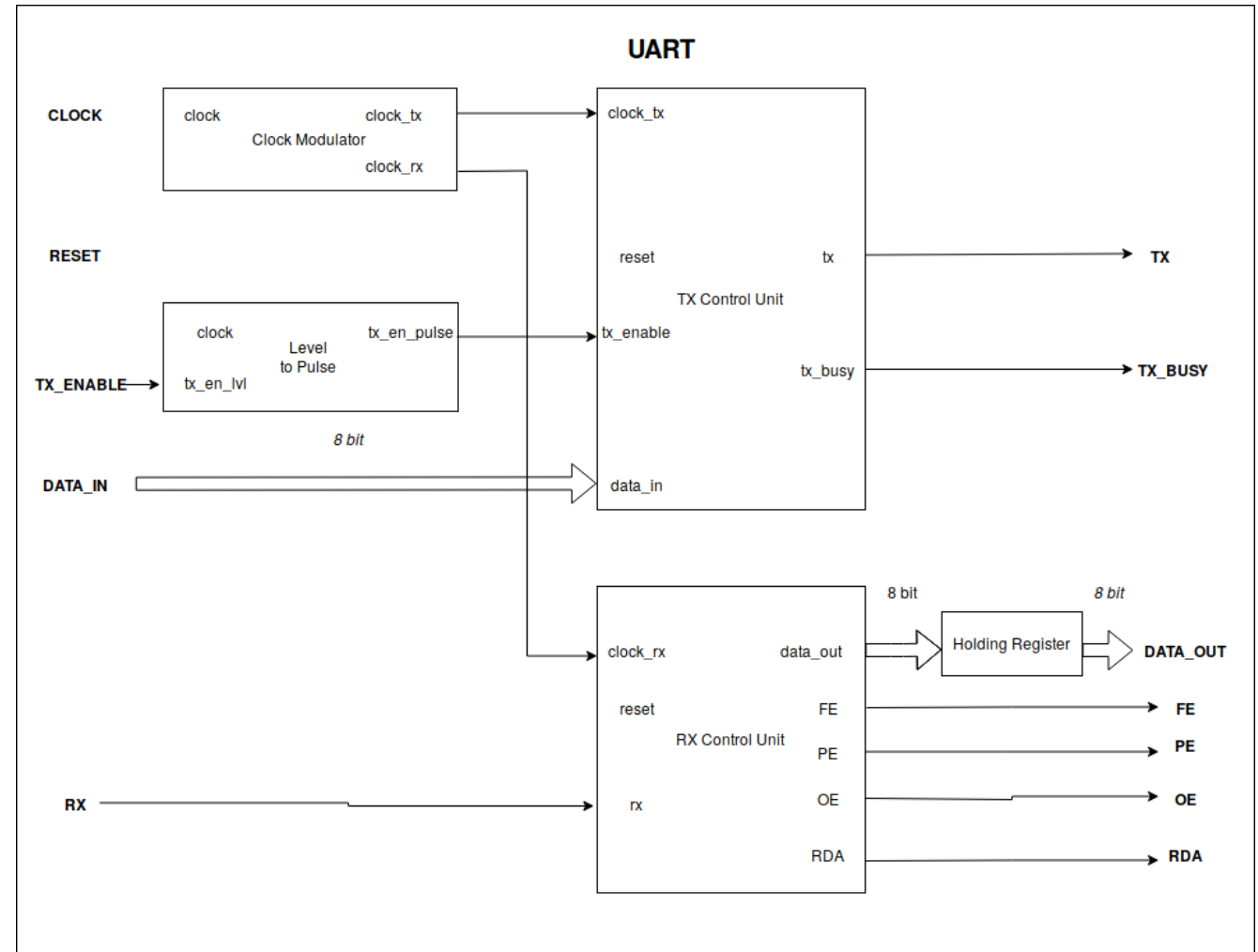
Custom IP AXI: UART

1. Realizzare un componente UART
2. Connettere il componente al bus AXI
3. Implementare i driver per la gestione del componente

Componente UART

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

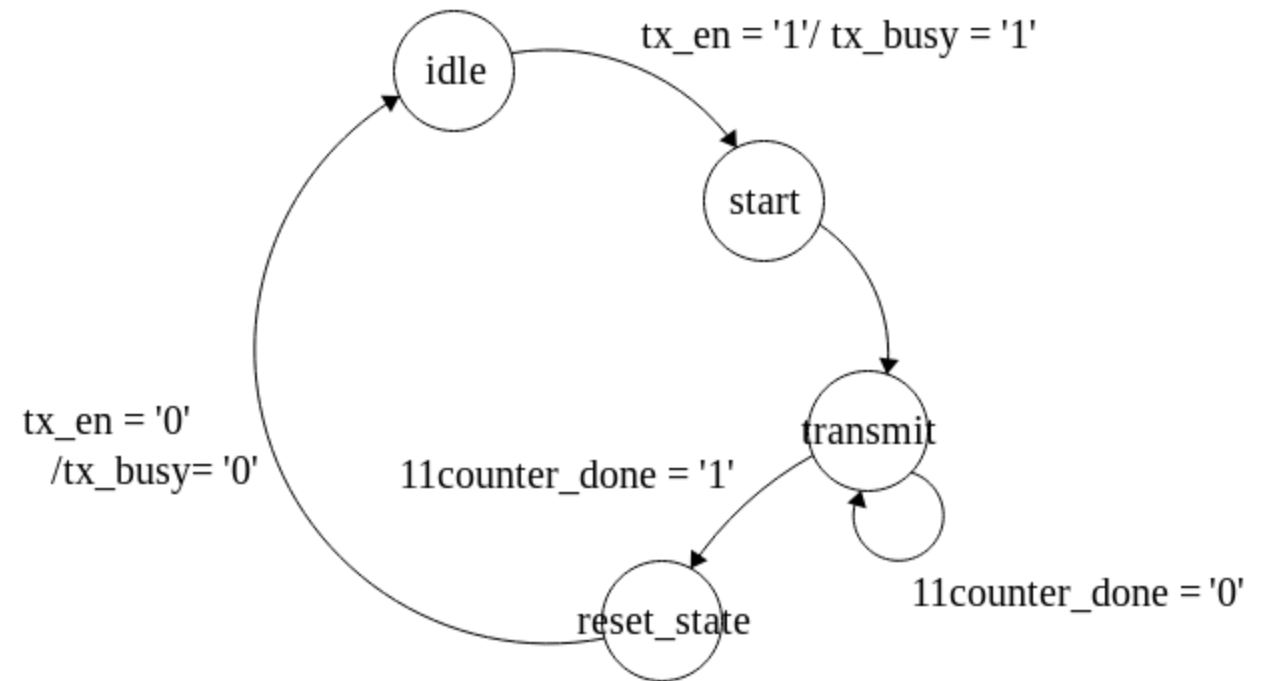
- sezione trasmettitore
- sezione ricevitore
- modulazione del clock



Sezione Trasmissione

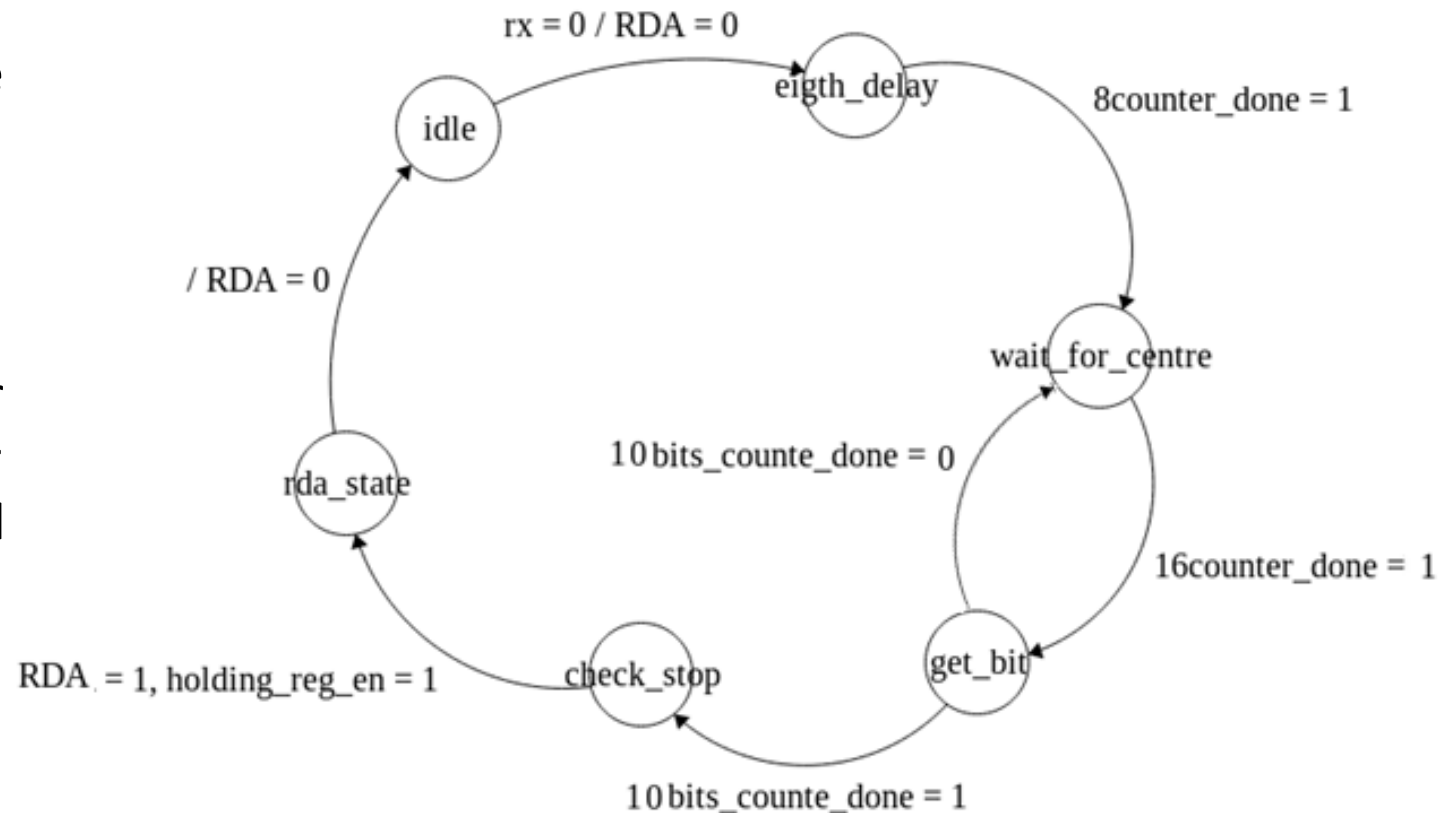
Divisa secondo la logica PO/PC. Quest'ultima è governata da una macchina a stati finiti che evolve seguendo il seguente grafo:

- **tx_en**: abilita la trasmissione, permette alla macchina di uscire dallo stato di idle. Quando questo avviene il segnale **tx_busy** diventa alto per indicare che è iniziato un trasferimento.
- **11counter done**: asserito quando il contatore modulo 11 termina il conteggio indicando che sono stati trasmessi tutti i bit (start, dati, parity, stop)
- Quando la trasmissione è completa la macchina torna nello stato di idle e il segnale tx_busy torna al valore basso



Sezione Ricevitore

- Alla ricezione del bit di start viene avviato un contatore modulo 8 la cui terminazione indica che il campionamento del segnale in ingresso avviene proprio al centro del bit.
- Il campionamento dei 10 bit (dati, parity, stop) avviene ogni qual volta il contatore modulo 16 asserisce il suo segnale di terminazione del conteggio.
- Una volta terminate le 10 ricezioni viene settato ad 1 il segnale **holding_reg_en** per trasferire i dati ricevuti dallo shift register ad un registro esterno e viene portato ad 1 il valore di **RDA** il quale indica la terminazione della ricezione



Gestione Interruzioni

Siamo interessati agli eventi:

- Trasmissione completata = **tx_busy 1->0**
- Ricezione completata = **RDA 0->1**

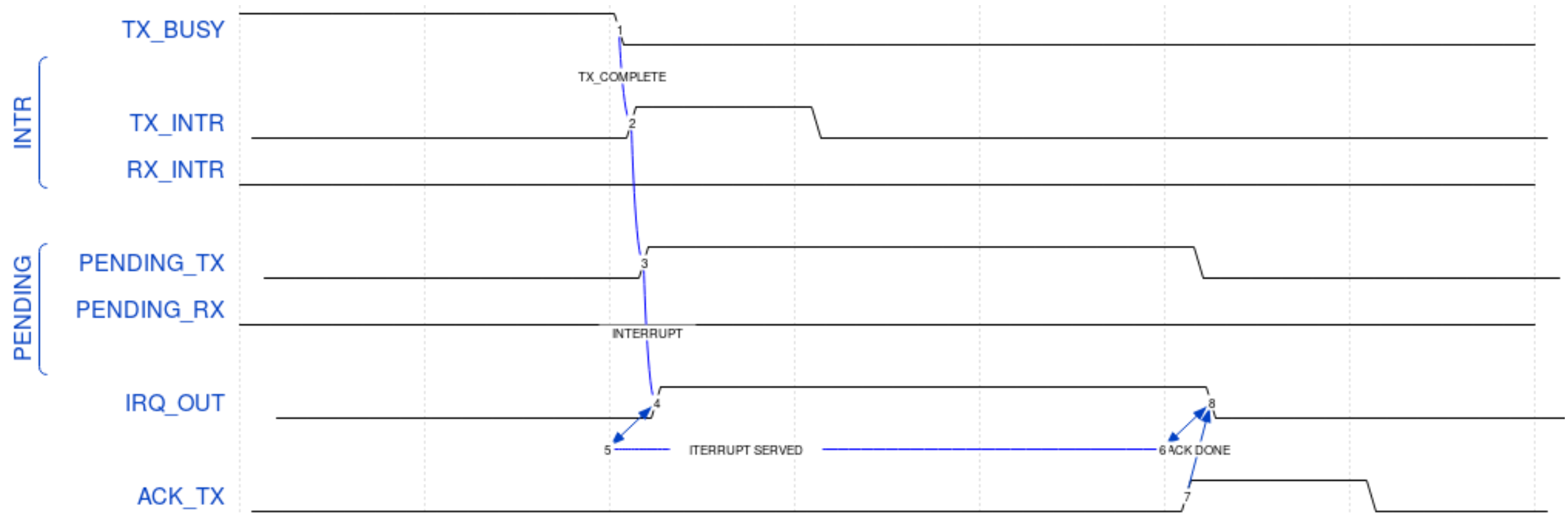
```
status_reg_sampling : process (S_AXI_ACLK,uart_status_reg)
begin
  if (rising_edge (S_AXI_ACLK)) then
    if ( S_AXI_ARESETN = '0' ) then
      last_stage <= (others => '0');
      current_stage <= (others => '0');
    else
      last_stage <= uart_status_reg(4 downto 3);
      current_stage <= last_stage;
    end if;
  end if;
end process;

tx_busy_falling_detect <= not last_stage(1) and current_stage(1);
rx_rising_detect <= not current_stage(0) and last_stage(0);

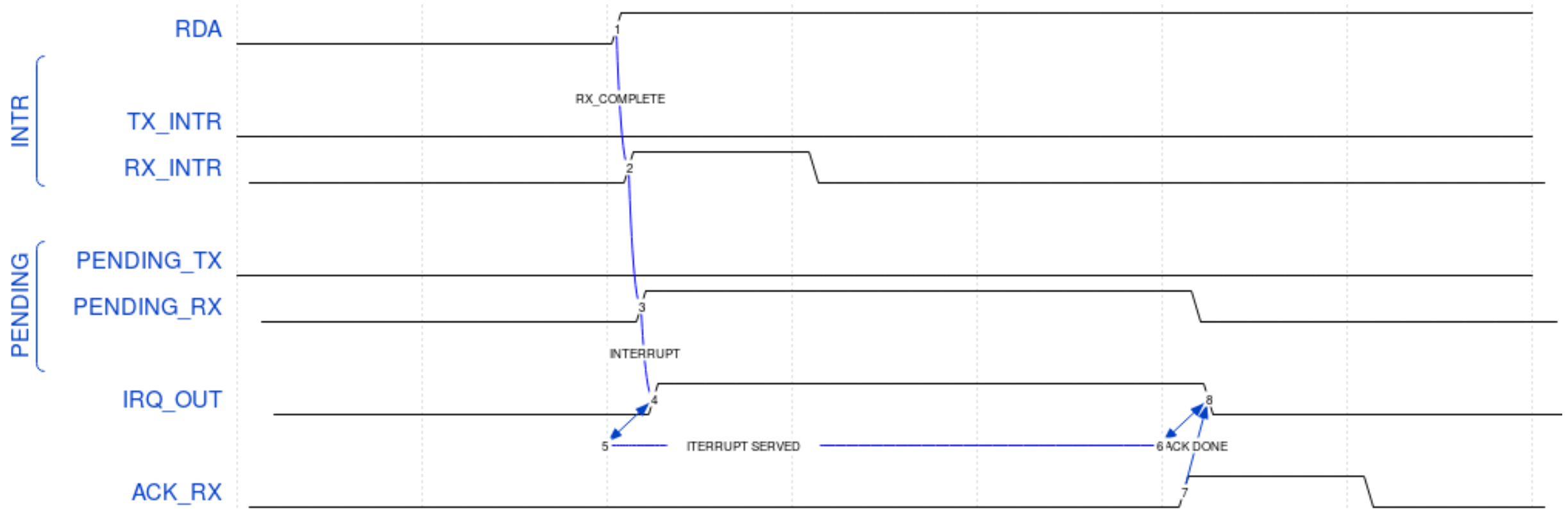
changed_bits <= (rx_rising_detect & tx_busy_falling_detect) and intr_mask;

change_detected <= global_intr and or_reduce(changed_bits);
```

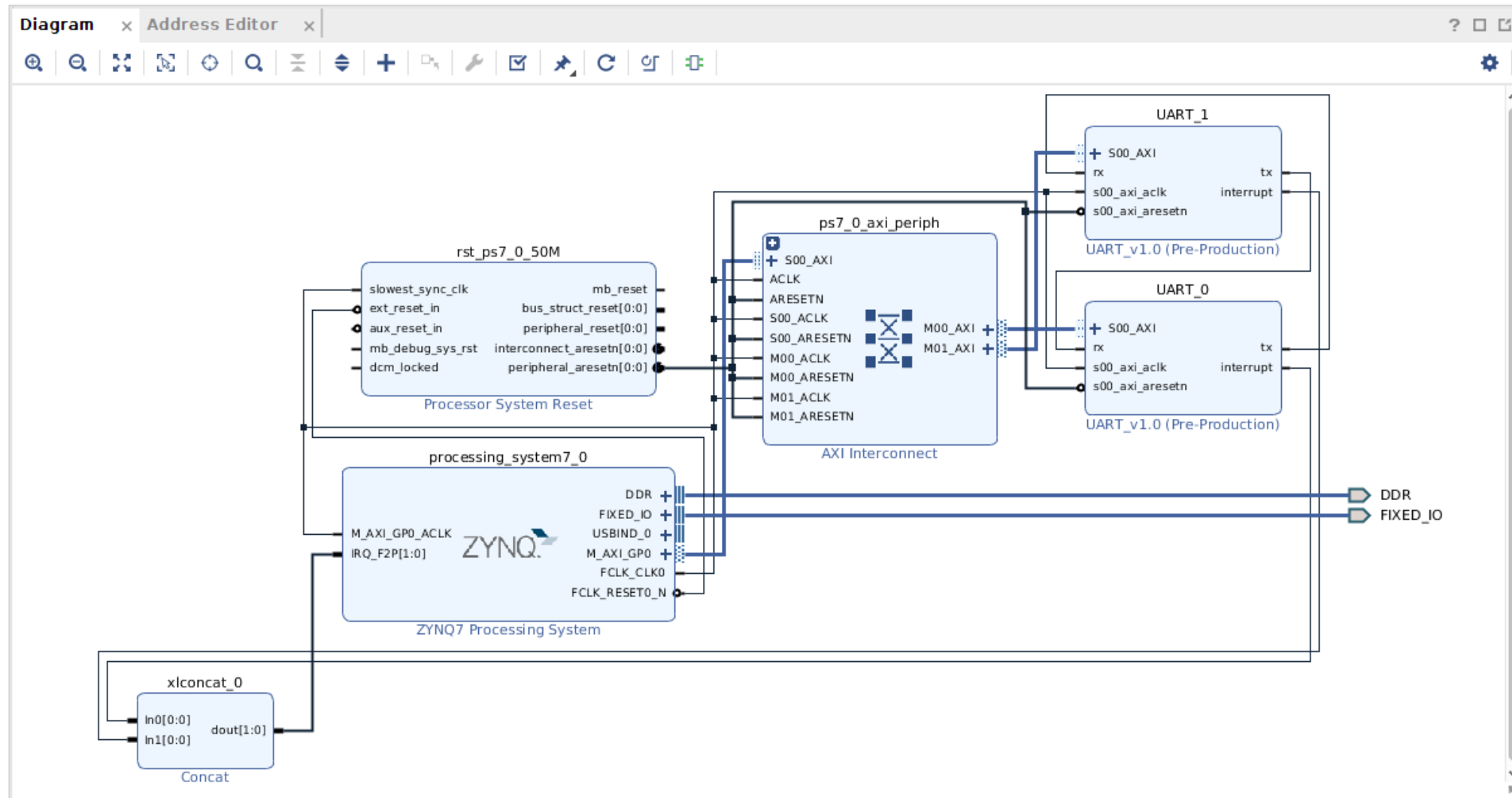
Trasmissione



Ricezione



Block Design – Connessione componenti



Driver Standalone

```
int SetupInterrupt(){  
    int Status;  
  
    //inizializzazione driver xscugic per la gestione del gic  
    GicConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);  
    Status = XScuGic_CfgInitialize(&InterruptController, GicConfig, GicConfig->CpuBaseAddress);  
    if ( Status != XST_SUCCESS) return XST_FAILURE;  
  
    //abilita la gestione delle eccezioni relative alla linea di interruzione in ingresso.  
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,  
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler,&InterruptController);  
    Xil_ExceptionEnable();  
  
    //Associa l'handler definito dall'utente alla linea di interruzione in ingresso al gic  
    //relativa al componente.  
    Status = XScuGic_Connect(&InterruptController, XPAR_FABRIC_UART_0_INTERRUPT_INTR,  
                            (Xil_ExceptionHandler)DeviceDriverHandler0, (void *)&InterruptController);  
    if ( Status != XST_SUCCESS) return XST_FAILURE;  
  
    Status = XScuGic_Connect(&InterruptController, XPAR_FABRIC_UART_1_INTERRUPT_INTR,  
                            (Xil_ExceptionHandler)DeviceDriverHandler1, (void *)&InterruptController);  
    if ( Status != XST_SUCCESS) return XST_FAILURE;  
  
    //Abilita la linea di interruzione del gic relativa al componente mappato  
    XScuGic_Enable(&InterruptController, XPAR_FABRIC_UART_0_INTERRUPT_INTR);  
    XScuGic_Enable(&InterruptController, XPAR_FABRIC_UART_1_INTERRUPT_INTR);  
  
    return Status;  
}
```

il GIC per risolvere eventuali conflitti nel caso in cui le due interrupt si verifichino insieme, assegna priorità maggiore alla linea con ID più basso (parametro XPAR_FABRIC_UART_X_INTERRUPT_INTR)

Ricevitore (UART_0) -> linea 61 GIC -> maggiore priorità

Driver Standalone – Interrupt Handler

Avendo una sola linea di interruzione diretta verso il processore è necessario identificare quale delle due linee interne ha attivato la linea IRQ. Nel fare questo è necessario esplicitare uno schema di priorità interno di gestione delle interruzioni. Viene gestita prima l'interruzione relativa alla linea RX.

```
void DeviceDriverHandler0()
{
    count0++;
    UART_GlobalDisableInterrupt(&UARTInstance0,0x1);
    pendingReg = UART_GetPending(&UARTInstance0);
    printf("PENDING REG :%08x \n\n",pendingReg);
    if((pendingReg & 0x00000002) == 0x00000002)
        ISR_RX(UARTInstance0);
    else if((pendingReg & 0x00000001) == 0x00000001){
        ISR_TX(UARTInstance0);
    }
    UART_GlobalEnableInterrupt(&UARTInstance0,0x1);
}
```

Driver Linux - Kernel Mode

```
typedef struct {
    /** Major e minor number associati al device (M: identifica il driver associato al device; m:
    utilizzato dal driver per discriminare il singolo device tra quelli a lui associati) */
    dev_t Mm;
    /** Puntatore a struttura platform_device cui l'oggetto UART si riferisce */
    struct platform_device *pdev;
    /** Struttura per l'astrazione di un device a caratteri */
    struct cdev cdev;
    /** Puntatore alla struttura che rappresenta l'istanza del device */
    struct device* dev;
    /** Puntatore a struttura che rappresenta una vista alto livello del device */
    struct class* class;
    /** Interrupt-number a cui il device è connesso */
    uint32_t irqNumber;
    /** Puntatore alla regione di memoria cui il device è mappato */
    struct resource *mreg;
    /** Device Resource Structure */
    struct resource res;
    /** res.end - res.start; numero di indirizzi associati alla periferica. */
    uint32_t res_size;
    /** Indirizzo base virtuale della periferica */
    void __iomem *vrtl_addr;
    /** wait queue per la sys-call read() */
    wait_queue_head_t read_queue;
    /** wait queue per la sys-call poll() */
    wait_queue_head_t poll_queue;
    /** wait queue per la sys-call write() */
    wait_queue_head_t write_queue;
    /** Flag che indica, quando asserito, la possibilità di effettuare una chiamata a read */
    uint32_t can_read;
    /** Flag che indica, quando asserito, la possibilità di effettuare una chiamata a write */
    uint32_t can_write;
    /** Spinlock usato per garantire l'accesso in mutua esclusione alla variabile can_read */
    spinlock_t slock_int;
    /** Spinlock usato per garantire l'accesso in mutua esclusione alla variabile can_write */
    spinlock_t write_lock;
    /** Buffer utilizzato per contenere i caratteri da trasmettere */
    uint8_t * buffer_tx;
    /** Buffer utilizzato per contenere i caratteri da ricevere */
    uint8_t * buffer_rx;
} UART;
```

Analogamente a quanto visto in precedenza per il GPIO i device UART sono gestiti come un **dispositivo a caratteri**.

Il modulo dispone di una lista per la gestione di più device.

Write gestita analogamente alla read: se un trasferimento è in corso il processo chiamante viene sospeso sulla write_queue e viene risvegliato dalla ISR all'avvenuto completamento della trasmissione.

Workflow esempio user application - UIO

1. Apertura descrittore del file sui device /dev/uio0 e /dev/uio1

```
/* Abilitazione interruzioni globali */
write_reg(uart_rx_ptr, GLOBAL_INTR_EN, 1);
/* Abilitazione interruzioni */
write_reg(uart_rx_ptr, INTR_EN, RX);

/* Abilitazione interruzioni globali */
write_reg(uart_tx_ptr, GLOBAL_INTR_EN, 1);
/* Abilitazione interruzioni */
write_reg(uart_tx_ptr, INTR_EN, TX);

/* Settaggio del primo carattere da mandare */
write_reg(uart_tx_ptr, DATA_IN, buffer_tx[0]);

/* Abilitazione del trasferimento */
write_reg(uart_tx_ptr, TX_EN, 1);

poll_fds[0].fd = rx_file_descr;
poll_fds[0].events = POLLIN;

poll_fds[1].fd = tx_file_descr;
poll_fds[1].events = POLLIN;
```

2. Abilitazione interruzioni e inserimento dei descrittori e degli eventi a cui siamo interessati nella struct pollfd

3. Attesa di interruzioni dai device tramite chiamata non bloccante a **poll**

Workflow esempio user application - UIO

4. Al termine del *TIMEOUT* specificato, il *sottosistema UIO* provvederà a risvegliare il processo fornendo una maschera indicante gli eventi rilevati sui descrittori dei file inseriti nella struct.
5. Se l'evento rilevato è quello a cui eravamo interessati, ovvero la presenza di nuovi dati da leggere (POLLIN) significa che è stata rilevata un'interruzione.
6. La chiamata a **read** sul corrispondente descrittore del file NON sarà bloccante.
7. Gestione interruzione e chiamata **write** per indicare al sottosistema che l'interruzione è stata gestita e che può riabilitare le interruzioni

```

void wait_for_interrupt(struct pollfd * poll_fds, void *uart_rx_ptr, void *uart_tx_ptr)
{
    int pending = 0;
    int reenable = 1;
    u_int32_t pending_reg = 0;
    u_int32_t reg_sent_data = 0;
    u_int32_t reg_received_data = 0;
    int ret = poll(poll_fds, 2, TIMEOUT);
    if (ret > 0){
        if(poll_fds[0].revents && POLLIN){
            read(poll_fds[0].fd, (void *)&pending, sizeof(int));
            write_reg(uart_rx_ptr, GLOBAL_INTR_EN, 0);
            pending_reg = read_reg(uart_rx_ptr, INTR_ACK_PEND);
            if((pending_reg & RX) == RX){
                printf("ISR RX detected!\n");
                if(rx_count <= buffer_size){
                    rx_count++;
                    reg_received_data = read_reg(uart_rx_ptr, RX_REG);
                    printf("ISR RX - value received: %c\n", reg_received_data);
                    buffer_rx[rx_count] = reg_received_data;
                }
                write_reg(uart_rx_ptr, INTR_ACK_PEND, RX);
                write_reg(uart_rx_ptr, INTR_ACK_PEND, 0);
                write_reg(uart_rx_ptr, GLOBAL_INTR_EN, 1);
            }
            write(poll_fds[0].fd, (void *)&reenable, sizeof(int));
        }
        if(poll_fds[1].revents && POLLIN){
            read(poll_fds[1].fd, (void *)&pending, sizeof(int));
            write_reg(uart_tx_ptr, TX_EN, 0);
            write_reg(uart_tx_ptr, GLOBAL_INTR_EN, 0);
            pending_reg = read_reg(uart_tx_ptr, INTR_ACK_PEND);
            if((pending_reg & TX) == TX){
                printf("ISR TX Detected\n");
                tx_count++;
                if(tx_count <= buffer_size){
                    reg_sent_data = read_reg(uart_tx_ptr, DATA_IN);
                    printf("ISR TX - value sent: %c\n", reg_sent_data);
                    write_reg(uart_tx_ptr, INTR_ACK_PEND, TX);
                    write_reg(uart_tx_ptr, INTR_ACK_PEND, 0);
                    write_reg(uart_tx_ptr, GLOBAL_INTR_EN, 1);
                    if(tx_count != buffer_size){
                        printf("ISR TX - start sending next value: %c\n", buffer_tx[tx_count]);
                        write_reg(uart_tx_ptr, DATA_IN, buffer_tx[tx_count]);
                        write_reg(uart_tx_ptr, TX_EN, 1);
                    }
                }
            }
        }
        write(poll_fds[1].fd, (void *)&reenable, sizeof(int));
    }
}
}

```

Corso di Sistemi Embedded

Progetto Finale 2019

Specifiche progetto

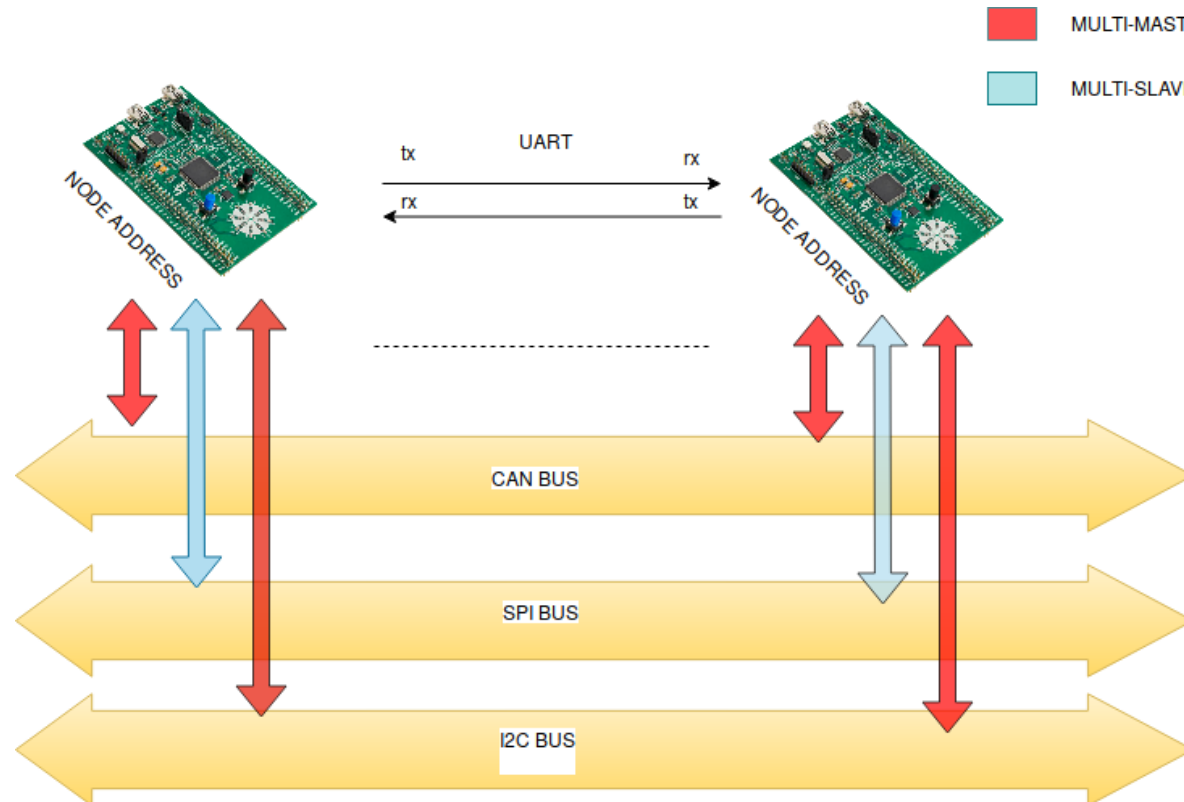
- Scambio messaggi fra più board *STM32F3 Discovery*
- Una board master e più board Slave
- Connessioni punto-punto realizzate con UART
- Connessione con bus seriali I2C, SPI, CAN
- SPI utilizzato con 3 fili (clock, MISO, MOSI) e *k slave select*
- I2C e CAN discriminano le board tramite un **campo indirizzo** nel frame
- Calcolo del CRC sui frame ricevuti e trasmetti secondo standard PVS

Specifiche progetto

- Architettura software basata su due livelli:
 - **Livello I** : si interfaccia con lo strato software HAL realizzando i driver per i quattro protocolli utilizzati
 - **Livello II**: fornisce all'applicativo utente delle API che astraggono l'utilizzo dei driver del livello sottostante. Le primitive SEND e RECEIVE permettono di scegliere uno o più canali per la comunicazione.

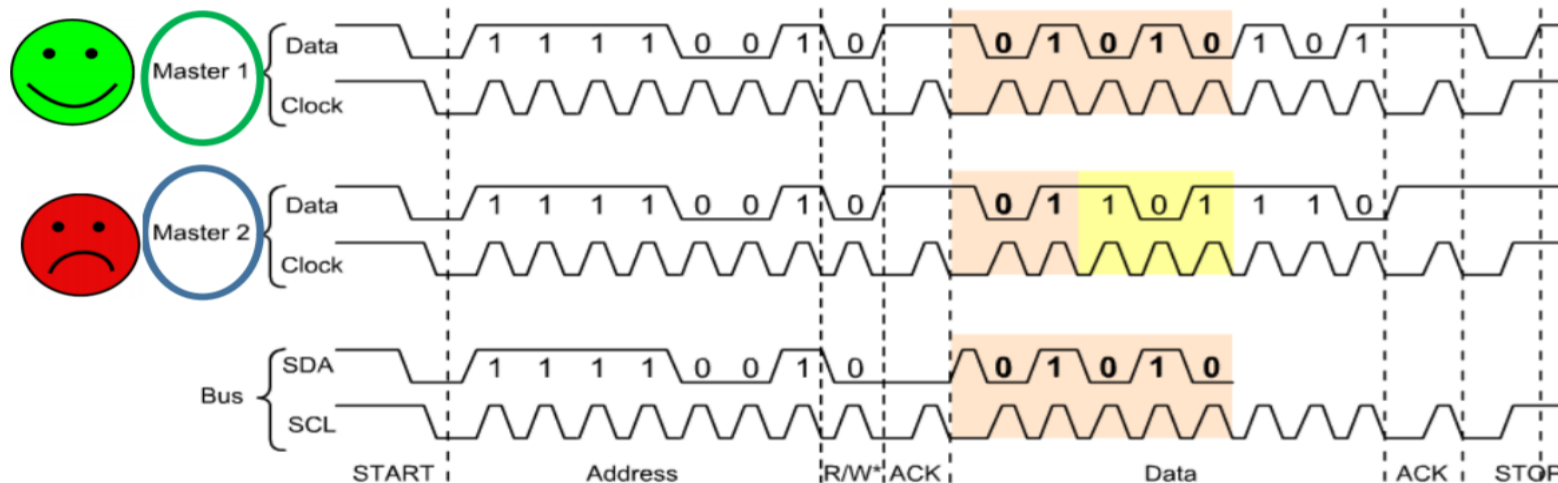
Architettura e topologia della rete

- Tutti i nodi sono collegati ai 3 bus e ci sono collegamenti fra coppie di nodi con periferiche UART



Interfacciamento al bus: I2C

- I2C è un bus nativamente *Multimaster-Multislave*. Più nodi possono operare in modalità master o slave.
- Le STM32 permettono di utilizzare le periferiche in questa modalità. Ogni nodo deve monitorare il bus

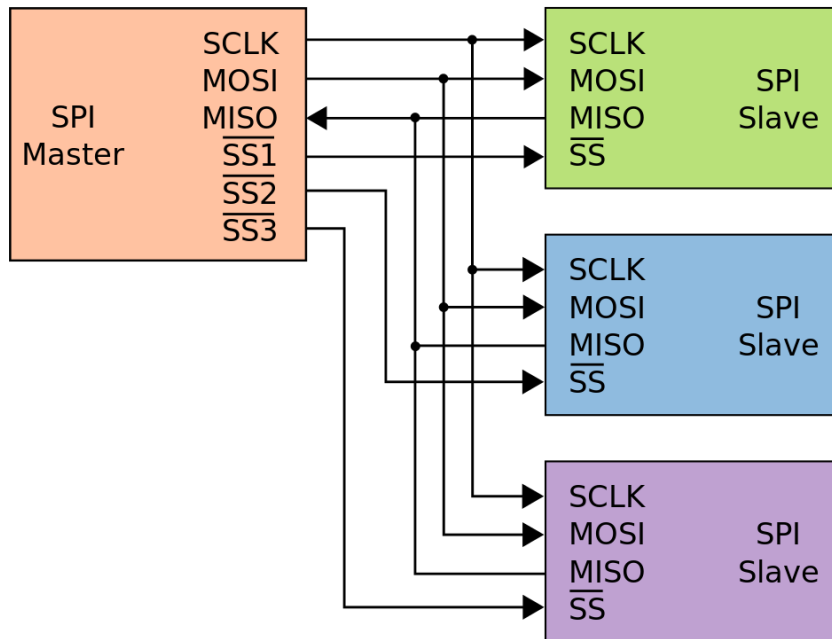


Interfacciamento al bus : I2C

- Nella soluzione realizzata si è scelto di realizzare lo schema *Multimaster*
- La comunicazione è stata caratterizzata in modo che ogni nodo si comporti da:
 - **Master I2C** quando vuole effettuare un trasferimento, necessita di conoscere l'indirizzo del nodo destinatario.
 - **Slave I2C** quando vuole effettuare una ricezione, non necessita dell'indirizzo del nodo mittente.

Interfacciamento al bus: SPI

- SPI è un bus nativamente *SingleMaster-MultiSlave*.
- Le board STM32 permettono di realizzare una soluzione Multislave con k nodi oppure Multimaster (solo due nodi).



Interfacciamento al bus: SPI

- Nella soluzione presentata si è scelto lo schema Multislave.
- Solo il master necessita di conoscere l'indirizzo dello slave, sia nel caso di trasmissione che di ricezione.
- Il master utilizza più GPIO per selezionare gli Slave con SlaveSelect
- In ogni slave, alternativamente al pin NSS è stato utilizzato un GPIO che ne emula la funzionalità, gestita però in software.

Arbitraggio dei bus: CAN

- Il bus CAN è un bus esclusivamente Multimaster.
- Le competizione in trasmissione fra i nodi sono risolte con una *Non-destructive bitwise arbitration* (0 dominante). Nella pratica «vince» il nodo che sta trasmettendo il messaggio con ID numericamente più basso.

Modalità di trasmissione

- Le trasmissioni del sistema possono essere:
 - **Unicast**: il destinatario del messaggio è unico. Supportato da tutti i bus utilizzati
 - **Multicast**: il messaggio viene inviato a tutti i nudi appartenenti ad un «gruppo». Solo CAN supporta questa modalità di trasmissione
 - **Broadcast**: il nodo manda il messaggio a tutti i nodi collegati al bus. La trasmissione in questa modalità è supportata da I2C (genericall call address) e da CAN, ma si è scelto di realizzarla solo con CAN.

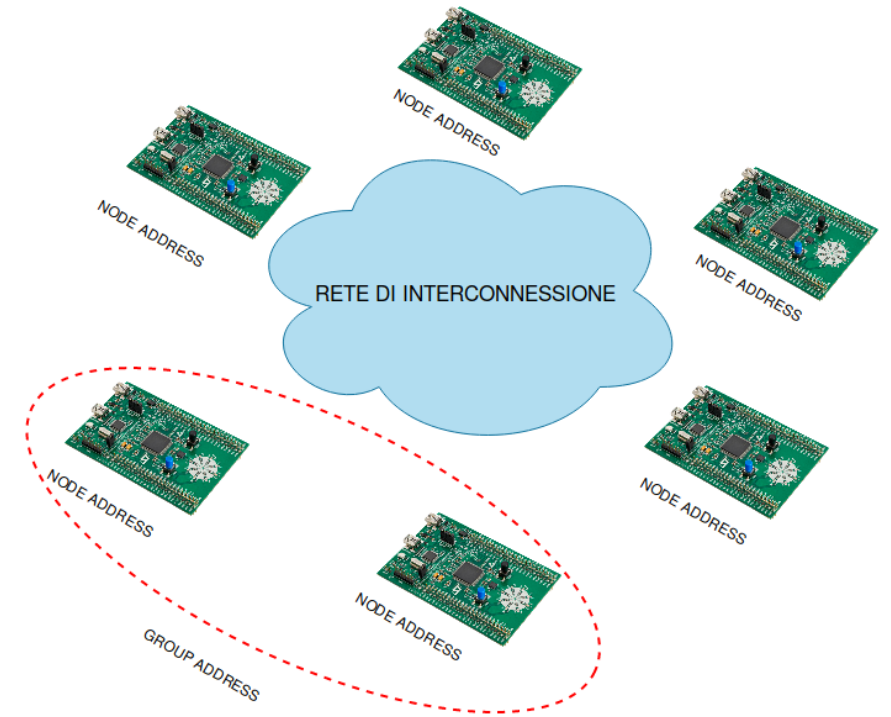
Gestione degli indirizzi

- Ogni periferica gestisce il proprio indirizzo in maniera differente
- Le comunicazioni delle periferiche devono rispondere ad un interfaccia unica che richiede un indirizzo unico
- E' necessario creare uno «spazio degli indirizzi» comune a tutte le periferiche

Gestione degli indirizzi

Ogni periferica risponde a due indirizzi:

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



Gestione degli indirizzi: vincoli periferiche

- **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 corrispondano all'indirizzo della periferica.
- **SPI**: non utilizza indirizzi per identificare i nodi ma collegamenti fisici.
- **CAN**: non associa un indirizzo univoco al nodo poiché il messaggio è inviato sempre in broadcast e spetta al singolo nodo stabilire se è interessato o meno al messaggio mediante dei filtri (***MASK*** o ***IDLIST***)

Gestione degli indirizzi

Gli indirizzi ammissibili per i nodi sono su 10 bit (limite superiore I2C).

CAN utilizza la modalità ***IDLIST*** con filtri a 16 bit (6 bit inutilizzati). Sono necessari dunque due filtri:

- Uno con l'id che funge da indirizzo del nodo
- Uno con l'id che funge da indirizzo del gruppo.

Per realizzare il broadcast è necessario un terzo filtro con id comune per tutti i nodi che funge da *broadcast address*

Gestione degli indirizzi

La codifica degli indirizzi dei gruppi non differisce dagli indirizzi dei nodi.

Lo spazio degli indirizzi disponibile dunque va diviso fra indirizzi dei nodi e indirizzi dei gruppi:

$$NodeAddr + GroupAddr = 2^{10} - BroadcastAddrCan - \text{indirizzi riservati I2C}$$

Questa scelta, rispetto a realizzare il multicast usando maschere per gli indirizzi, offre maggiore flessibilità nel partizionamento dei nodi in gruppi.

Architettura software

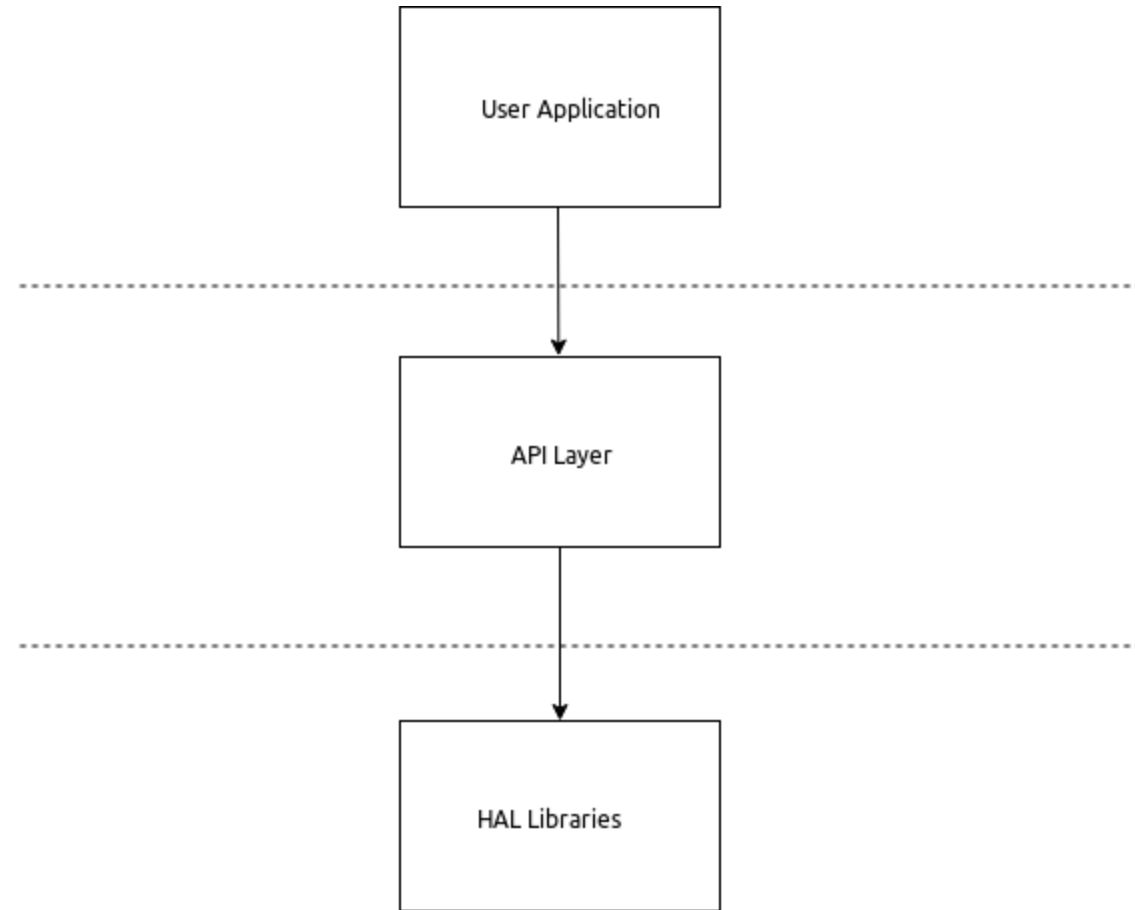
Secondo Livello

Incapsula le funzioni di

- Send
- Receive
- Configure Peripheral

Primo Livello

Incapsula i driver che gestiscono le periferiche interfacciandosi con lo strato HAL fornito da STM32.



API Layer

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

- **CRC_Check**: prende in ingresso il frame ricevuto composto da [payload + CRC1 + CRC2], ricalcola i due CRC sul payload, li confronta con quelli presenti nel frame e li sostituisce.
- **Send_CRC**: prende in ingresso il Messaggio completo da inviare, la maschera dei canali su cui trasmettere, l'indirizzo del destinatario (nodo o gruppo) e la modalità di trasmissione (unicast, multicast, broadcast). Viene effettuato controllo sulla coerenza fra modalità di trasmissione e canale scelto
- **Receive_CRC**: prende in ingresso il buffer dove ricevere il messaggio, i canali su cui riceve e l'indirizzo del nodo dal quale effettuare la ricezione (se questa avviene in master mode per qualche periferica)
- **Configure_peripheral**: prende in ingresso la maschera delle periferiche da inizializzare

Send & Receive: indirizzamento SPI

SPI non utilizza indirizzi per la selezione dello slave con cui comunicare, ma pin fisici. Il master utilizza tanti GPIO quanti sono gli slave per controllare i segnali NSS degli stessi.

E' necessario che il master possa effettuare la corrispondenza indirizzo-pin GPIO collegato al NSS dello slave. Questa «traduzione» deve essere effettuata in software da una funzione che ha una conoscenza completa dei nodi della rete. A tal proposito è stato utilizzato lo STUB ***getSSPinByAddress(uint16_t address)***

Applicativo di prova

Vengono utilizzati due nodi. L'algoritmo eseguito ciclicamente dai due nodi è:

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

Viene utilizzato un nodo Master ed un nodo Slave. Le periferiche, per come si è scelto di utilizzarle, sono identiche nel funzionamento fra i due nodi, fatta eccezione per SPI.

Applicativo di prova

Il nodo Master, prima di entrare nel ciclo, inizializza il Payload del messaggio con valori random e calcola i due CRC impacchettandoli nel frame.

L'applicativo, è genericamente configurabile nelle periferiche da utilizzare e nella dimensione del payload del messaggio.

