## 0.1 Soluzione

### 0.1.1 Dispostivo UART

L'implementazione del componente UART è stata realizzatta seguendo lo schema di un generico dispositivo commerciale, dividendo dunque la logica nei seguenti blocchi (Figura 0.1):

- sezione ricevitore: implementa la logica di ricezione. Quanto un byte è ricevuto viene copiato nell'Holding Register e vi rimane fino alla completa riecezione 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.

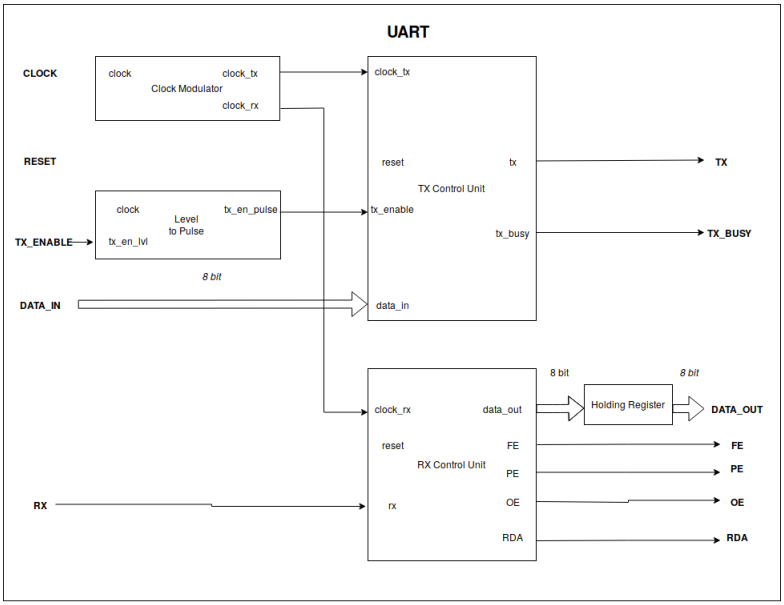


Figure 1: Schema a blocchi componente UART

Con riferimento alla Figura 0.1, si stabilisca la seguente convezione: i segnali in grasseto rappresentano l'interfaccia del componente, quelli sulla sinistra indicano segnali di ingresso, quelli sulla desta di uscita. Alcuni sengali (clock, reset), avento multiple destinazioni, non sono collegati per pura questione di visibiltà.

#### 0.1.1.1 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:

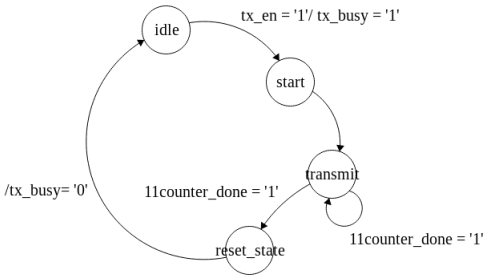


Figure 2: Diagramma Stati FSM trasferimento

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

#### 0.1.1.2 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 a ricezione completata e successivamente disabilitato per preservare il dato fino all'arrivo dei 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:

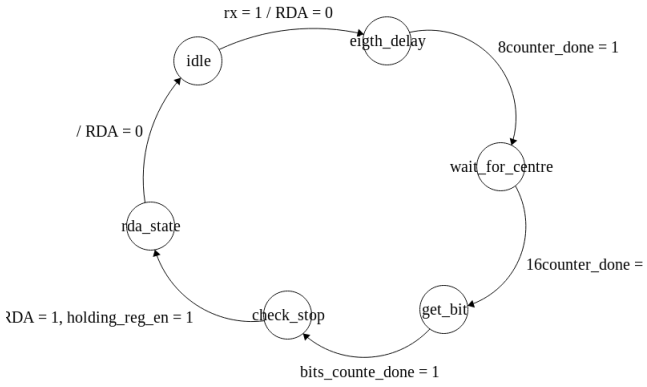


Figure 3: FSM ricezione

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 completatamente 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.

#### 0.1.1.3 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 componete 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.

### 0.1.2 Custom AXI IP Core

Si procede dunque alla crazione di un costum IP Core come mostrato nel precedente capitolo. Verranno instanziati due componenti:

1. **my_uart_intv1**: top modul 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 modul. Gestisce i la logica di interruzione della periferica

#### 0.1.2.1 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* |

Table 1: Mapping indirizzi

Si omette la connessione del componente UART in quanto basilare e riassumibile tramite la Tabella 0.1.

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 varizione 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
```

```
            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);    --! detect falling edge tx_busy
    rx_rising_detect <= not current_stage(0) and last_stage(0);           --! detect rising edge RDA

  changed_bits <= (rx_rising_detect & tx_busy_falling_detect)
          and intr_mask;                    --! and con la intr_mask perchè sono interessato a vedere l'edge del sengale
                                                                          --! solo se la relativa interruzione è abilitata

    change_detected <= global_intr or_reduce(changed_bits);               --! 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 pendendi e dell'ack. Se viene rilevata una nuova richiesta di interruzione su una delle due linee essa viene aggiunta alle precedenti pendenti. Se viene dato un ack per la specifica interruzione essa viene rimossa da quelle pendenti.

```

   pending_intr_tmp <= pending_intr;

    --! process per la gestizione della logica di interruzione pedente
    --! e meccanismo di ack per rimuovere l'interruzione pendente
    intr_pending : process (S_AXI_ACLK, change_detected, ack_intr)
    begin
    if (rising_edge (S_AXI_ACLK)) then
        if (change_detected = '1') then                                   --! se c'è richiesta di interruzione su una delle due line
            pending_intr <= pending_intr_tmp or changed_bits;             --! aggiungi la richiesta alle interruzioni pendenti
        elsif (or_reduce(ack_intr)='1') then                              --! se viene dato un ack
            pending_intr <= pending_intr_tmp and (not ack_intr);          --! rimuovi la richiesta pendente relativa
        else
      pending_intr <= pending_intr_tmp:                --! altrimenti il segnale resta al suo valore corrente
        end if;
    end if;
    end process;
```

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

```

  --! process per gestire l'unica linea di interruzione
```

```
    --! in unscita dal componente
     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  --! Se c'è
                     almeno un interruzione pendente e globali sono abilitate
                      interrupt <= '1';                                          --! interrupt
                          = '1'
                  else
                      interrupt <= '0';                                          --!
                         altrimenti 0
                  end if;
              end if;
          end if;
     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.

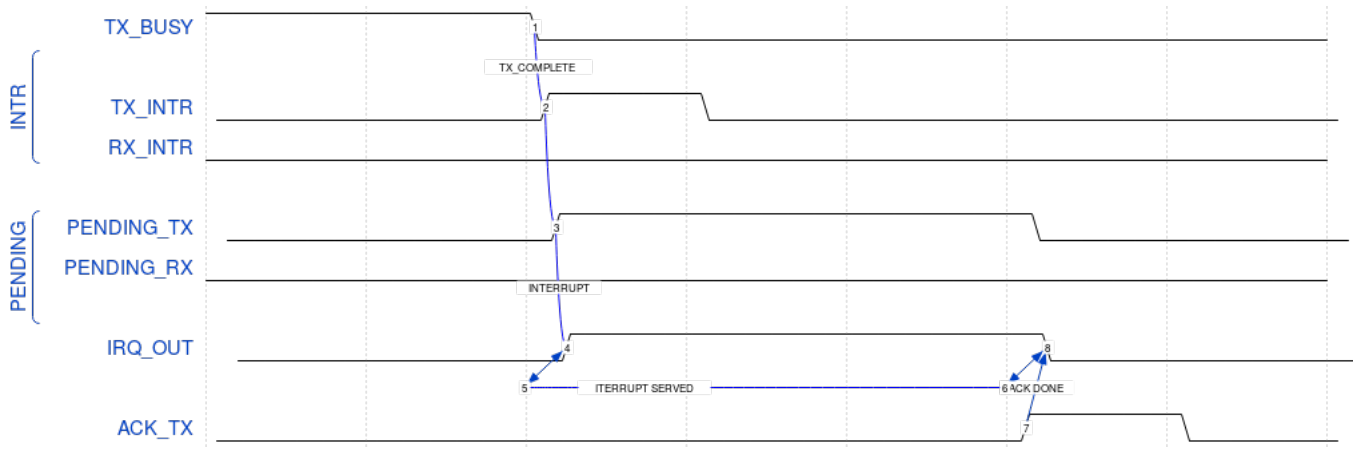


Figure 4: Gestione interruzioni trasmissione

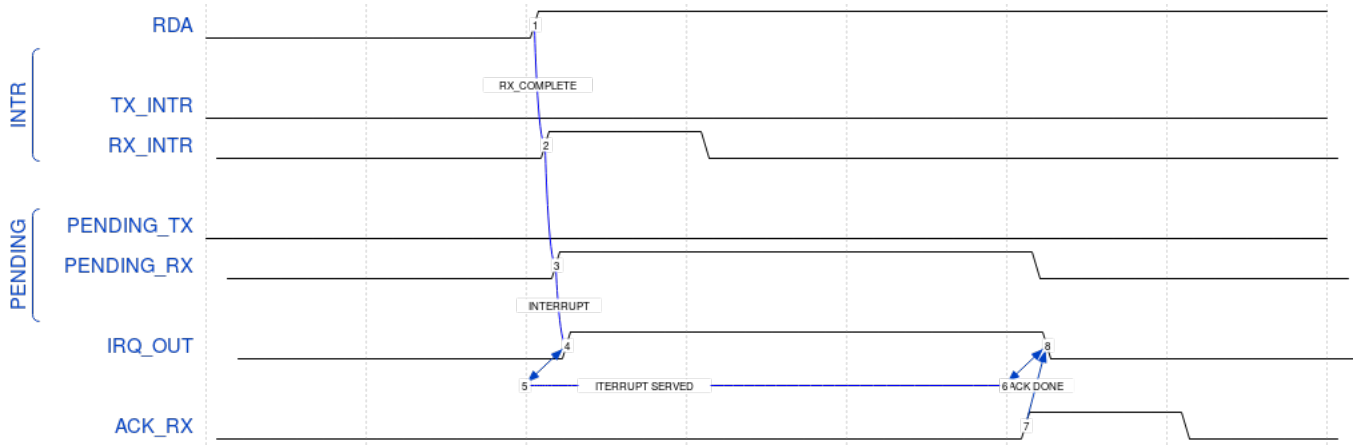


Figure 5: Gestione interruzioni ricezione

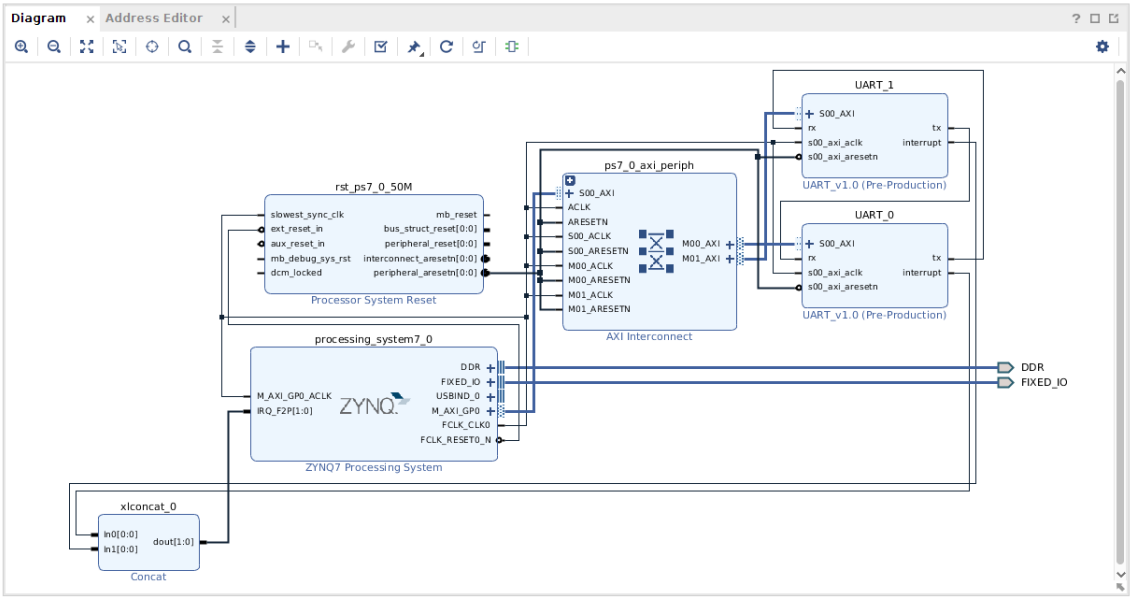


Figure 6: Block Design

### 0.1.3 Design

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

### 0.1.4 Driver Standalone

Il driver Standalone è stato realizzato per verificare il comportamento basilare del dispositovo. Come applicativo di test si è scelto di realizzare un main che esegue cinque invii dello stesso carattere.Il corretto funzionamento può essere verificando controllando le varibaili "count" di debug e i registri della periferica. Per la stesura delle librerie di gestione delle periferiche si rimanda alla documentazione html 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 verifichino 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 seguiro il main del programma.

```
#include "myuart.h"
#include <stdio.h>
#include <stdlib.h>
#include "xil_io.h"
#include "xil_exception.h"
#include "xparameters.h"
#include "xil_cache.h"
#include "xil_printf.h"
#include "xil_types.h"
#include "xscugic.h"
#include "xil_cache_l.h"

/************************** Constant Definitions *****************************/

#define INTC_DEVICE_ID    XPAR_SCUGIC_0_DEVICE_ID

#define UART0_BASE_ADDR   0x43C00000
```

```c
#define UART1_BASE_ADDR   0x43C10000

#define INT_MASK_TX     0x1
#define INT_MASK_RX     0x2
/************************** Type Definitions *****************************/

void DeviceDriverHandler0();
void DeviceDriverHandler1();
void ISR_TX(UART UARTInstance);
void ISR_RX(UART UARTInstance);

UART UARTInstance0,UARTInstance1;

u32 pendingReg, dataReg;
volatile static int count0,count1, tx_count, rx_count;
XScuGic InterruptController;
XScuGic_Config *GicConfig;
int i=1;


/**
 *
 * @brief Effettua la configurazione e l'abilitazione del GIC.
 *
 * @return  XST_SUCCESS se la configurazione è avvenuta correttamente
 *      XST_FAILURE altrimenti
 *
 * @note
 *
 */
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 lina 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);

```

```
77      return Status;
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 intrruzioni 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 interruzioini.
131     In questo caso viene gestita prima l'interruzione relativa alla linea RX
132   */
133
134   if((pendingReg & 0x00000002) == 0x00000002)
135     ISR_RX(UARTInstance0);
```

```
  else if((pendingReg & 0x00000001) == 0x00000001){
    ISR_TX(UARTInstance0);
    }


  //abilitazione interruzioni globali del componente
  UART_GlobalEnableInterrupt(&UARTInstance0,0x1);
}


void DeviceDriverHandler1()
{

  count1++;
  //vengono disabilitate le intrruzioni globali del componente
  UART_GlobalDisableInterrupt(&UARTInstance1,0x1);
  pendingReg = UART_GetPending(&UARTInstance1);
  printf("PENDING REG :%08x \n\n",pendingReg);

  /*  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 interruzioini.
    In questo caso viene gestita prima l'interruzione relativa alla linea RX
  */

  if((pendingReg & 0x00000002) == 0x00000002)
    ISR_RX(UARTInstance1);
  else if((pendingReg & 0x00000001) == 0x00000001){
    ISR_TX(UARTInstance1);
    if(tx_count<5){
    UART_SetData(&UARTInstance1,0xF);
    UART_Start(&UARTInstance1);}
  }
  //abilitazione interruzioni globali del componente
  UART_GlobalEnableInterrupt(&UARTInstance1,0x1);
}


void ISR_TX(UART UARTInstance){
  printf("******** ISR TX*************\n\n");
  tx_count++;
  //ACK interruzione relativa a TX
  UART_ACK(&UARTInstance,0x1);
}

void ISR_RX(UART UARTInstance){
  printf("******** ISR RX*************\n\n");
  rx_count++;
  //ACK interruzione relativa a RX
  UART_ACK(&UARTInstance,0x2);
}
```

### 0.1.5 Driver Linux

#### 0.1.5.1 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.

```c
/**
 * @brief Stuttura che astrae un device UART in kernel-mode.
 * Contiene ciò che è necessario al funzionamento del driver.
 */
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;
/** Stuttura 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 effettuale una chiamata a read*/
  uint32_t can_read;
/** Flag che indica, quando asserito, la possibilità di effettuale 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;
```

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:

```c
/**
 * @brief Struttura che specifica le funzioni che agiscono sul device
 *
 */
static struct file_operations GPIO_fops = {
    .owner    = THIS_MODULE,
    .llseek   = UART_llseek,
    .read     = UART_read,
    .write    = UART_write,
    .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 funzionalià 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 le tutte 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:

```
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 rimuove il modulo impartire il comando:

```
rmmod my_kernel_UART.ko
```

#### 0.1.5.2 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:

```
/ {
  amba_pl: amba_pl {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    ranges ;
    UART_0: UART@43c00000 {
      /* This is a place holder node for a custom IP, user may need to update the entries */
      clock-names = "s00_axi_aclk";
      clocks = <&clkc 15>;
      compatible = "xlnx,UART-1.0";
      interrupt-names = "interrupt";
      interrupt-parent = <&intc>;
      interrupts = <0 29 4>;
      reg = <0x43c00000 0x10000>;
      xlnx,s00-axi-addr-width = <0x5>;
      xlnx,s00-axi-data-width = <0x20>;
    };
    UART_1: UART@43c10000 {
      /* This is a place holder node for a custom IP, user may need to update the entries */
      clock-names = "s00_axi_aclk";
      clocks = <&clkc 15>;
      compatible = "xlnx,UART-1.0";
      interrupt-names = "interrupt";
      interrupt-parent = <&intc>;
      interrupts = <0 30 4>;
      reg = <0x43c10000 0x10000>;
      xlnx,s00-axi-addr-width = <0x5>;
      xlnx,s00-axi-data-width = <0x20>;
    };
```

```
  };
};
```

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:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <string.h>
#include <poll.h>
#include "UART_interrupt_uio.h"
/**
 * @file UART_interrupt_uio.c
 * @brief permette la gestione della periferica UART utilizzando un driver di tipo UIO
 */
#define DATA_IN       0 // DATA TO SEND
#define TX_EN         4 // TRANSFER ENABLE (0)
#define STATUS_REG      8   // OE(0) FE(1) DE(2) RDA(3) TX_BUSY(4)
#define RX_REG          12  // DATA RECEIVED
#define GLOBAL_INTR_EN  16  // GLOBAL INTERRUPT ENABLE
#define INTR_EN         20  // LOCAL INTERRUPT ENABLE
#define INTR_ACK_PEND   28  // PENDING/ACK REGISTER

#define TX        1
#define RX        2

#define TIMEOUT     5000

/**
 * @file UART_interrupt_uio.c
 * @page driver_UART_UIO
 * @brief funzioni per gestire la trasmissione e la ricezione dei
 *        dati utilizzando il protocollo UART
 */

int tx_count, rx_count =0;
int buffer_size = 0;
char * buffer_tx;
char * buffer_rx;
struct pollfd poll_fds [2];

/**
 *
 * @brief Utilizzata per scrivere un valore all'interno di un registro
 *      della periferica, specificando l'indirizzo base virtuale e
 *      l'offset del registro in cui scrivere
 *
 * @param addr indirizzo virtuale della periferica
 * @param offset offset del registro a cui scrivere
 * @param valore da scrivere
 *
 *
 *
```

```
 */
void write_reg(void *addr, unsigned int offset, unsigned int value)
{
  *((unsigned*)(addr + offset)) = value;
}
/**
 *
 * @brief Utilizzata per leggere un valore da un registro
 *    della periferica, specificando l'indirizzo base virtuale e
 *    l'offset del registro da cui leggere
 *
 * @param addr indirizzo virtuale della periferica
 * @param offset offset del registro a cui leggere
 *
 * @return  valore presente all'interno del registro
 *
 *
 */
unsigned int read_reg(void *addr, unsigned int offset)
{
  return *((unsigned*)(addr + offset));
}

/**
 *
 * @brief Attende l' arrivo di un interrupt utilizzando la read
 *      su un device UIO
 *
 * @param poll_fds struct contenente i due descrittori del file per
 *      i due device UART
 * @param uart_rx_ptr indirizzo virtuale della periferica UART utilizzata
 *      in ricezione
 * @param   uart_tx_ptr indirizzo virtuale della periferica UART utilizzata
 *      in trasmissione
 *
 */
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){

/** Se vi è un'interruzione sul device UIO0 associato all'UART per la ricezione */
    if(poll_fds[0].revents && POLLIN){

      read(poll_fds[0].fd, (void *)&pending, sizeof(int));

/* Disabilita le interruzioni */
      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");
```

```
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/uio0", 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/uio1", 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.