
0.1 Soluzione

0.1.1 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 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 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.

Con riferimento alla Figura 0.1, si stabilisca la seguente convezione: 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 visibilità.

0.1.1.1 Sezione Trasmissione

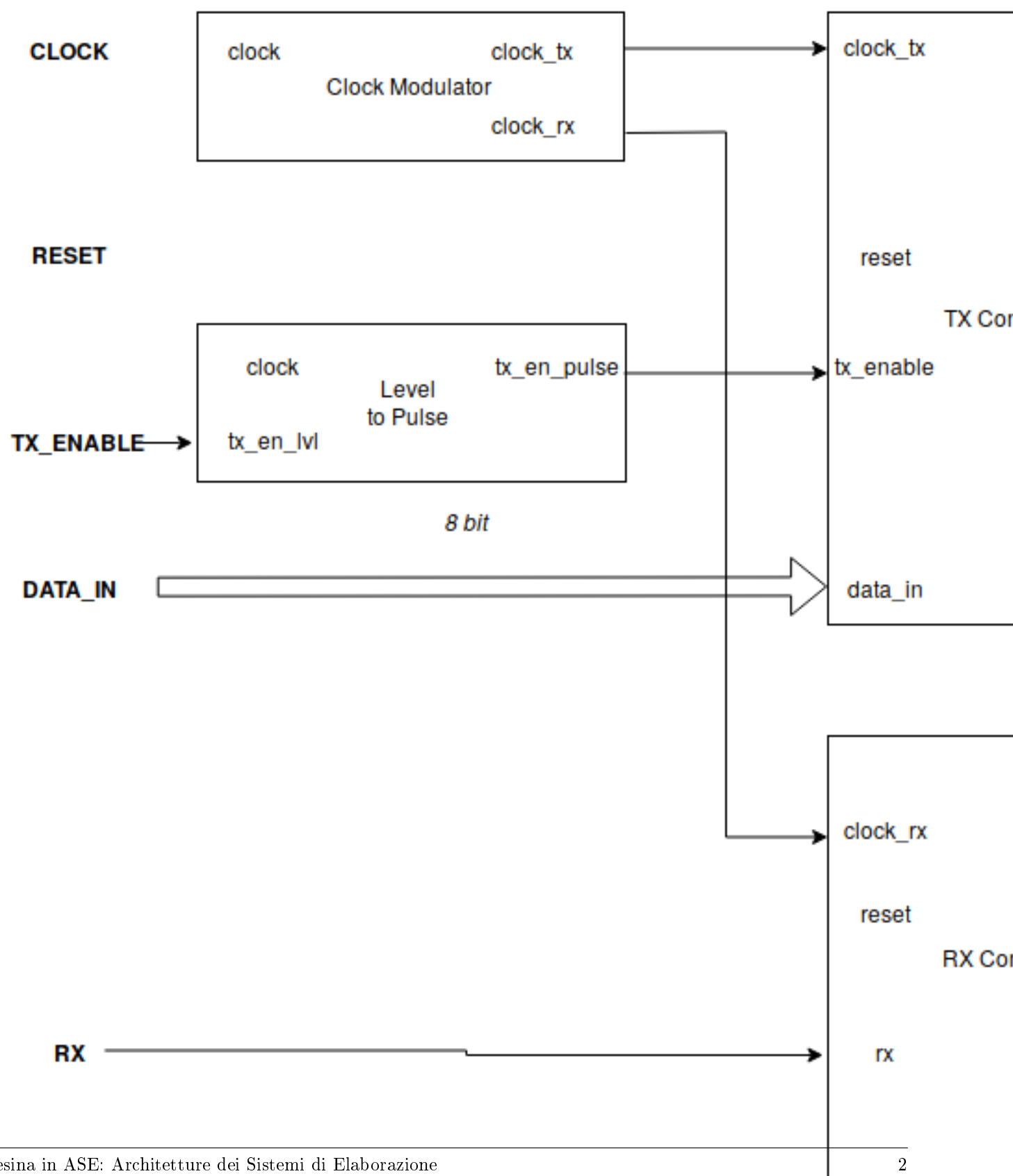
- Shift Register con scorrimento a destra, caricamento parallelo del dato da trasmettere ed uscita seriale per la trasmissione sul canale.
- Contatore Mod 11 incrementato ad ogni bit trasmesso il cui segnale di uscita counter_done viene utilizzato dalla control unit per verificare la fine della trasmissione.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

Il segnale 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 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:

UART



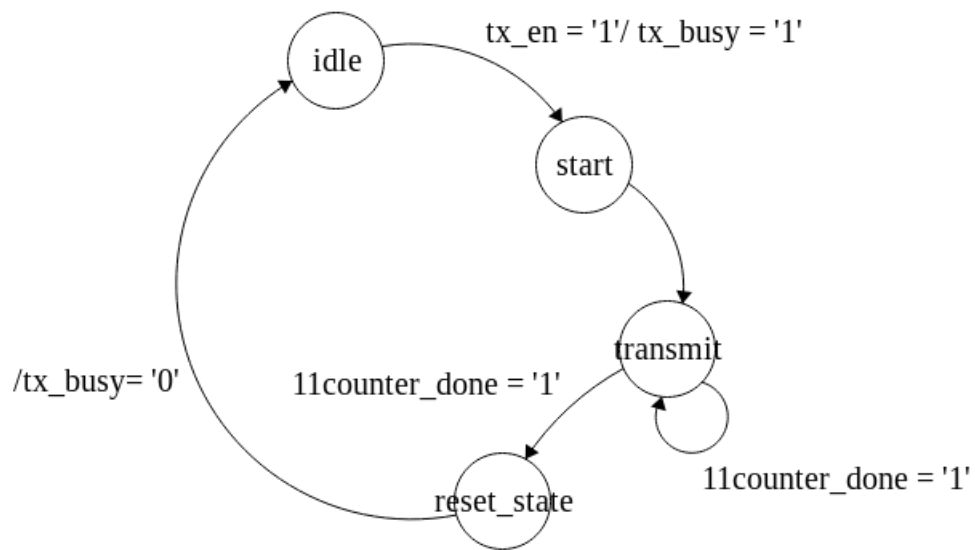


Figure 2: Diagramma Stati FSM trasferimento

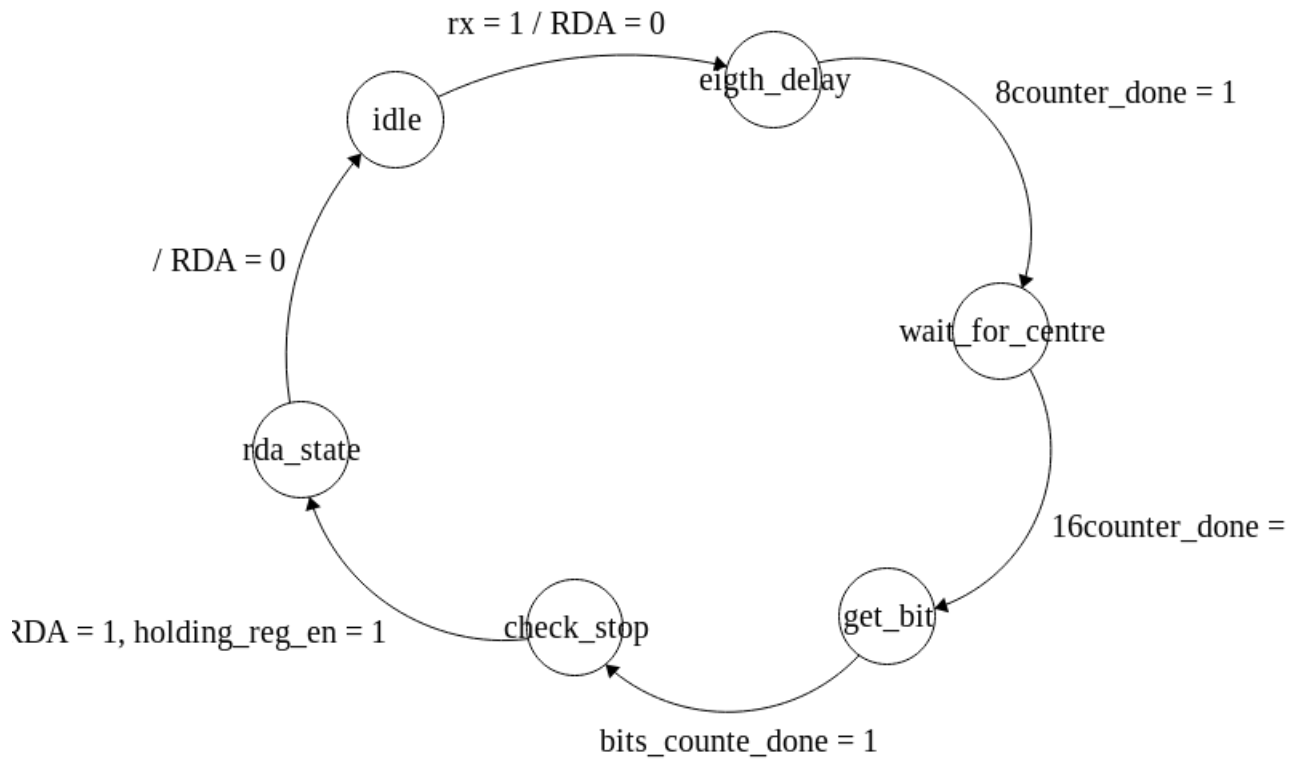


Figure 3:

Si è scelto, per questioni di temporizzazione e per fornire un modello di programmazione adeguato, aggiungere 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 dare 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 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.

0.1.2 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 modul dell'IP. Il segnale TX è in out all'ip mentre quello di RX in ingresso.

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

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

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 \begin{lstlisting}[language=VHDL]
2   --! process utilizzato per captare varizione dei segnali RDA(bit 3) e tx_busy(bit 4)
3   --! la sintesi da due FF in cascata
4   status_reg_sampling : process (S_AXI_ACLK,uart_status_reg)
5   begin
6     if (rising_edge (S_AXI_ACLK)) then
7       if ( S_AXI_ARESETN = '0' ) then
8         last_stage <= (others => '0');
9         current_stage <= (others => '0');
10      else
11        last_stage <= uart_status_reg(4 downto 3);
12        current_stage <= last_stage;
13      end if;
14    end if;
15  end process;
16
17
18  tx_busy_falling_detect <= not last_stage(1) and current_stage(1);    --! detect falling
    edge tx_busy
19  rx_rising_detect <= not current_stage(0) and last_stage(0);          --! detect rising
    edge RDA
20
21  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
22
23
24  change_detected <= global_intr and or_reduce(changed_bits);          --! Segnale
    che indica se è stato rilevata una variazione di tx_busy o RDA
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

Siamo interessati a rilevare uno dei due edge solo se è stato richiesto che il componente lavori con le interruzioni (GLOBAL_INTR_EN) e le due linee di interruzione sono abilitate (INTR_MASK). 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 pendenti. Se viene dato un ack per la specifica interruzione essa viene rimossa da quelle pendenti.

```

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

```

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

```

1 \begin{lstlisting}[language=VHDL]
2   --! process per gestire l'unica linea di interruzione
3   --! in uscita dal componente
4   inst_irq : process (S_AXI_ACLK, pending_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') then                    --! Se c'è
11          almeno un interruzione pendente
12          interrupt <= '1';                                         --! interrupt
13          = '1'
14        else
15          interrupt <= '0';                                         --!
16          altrimenti 0
17        end if;
18      end if;
19    end if;
20  end process;

```

0.1.3 Design

Per testare il componente, la linea TX è stata collegata alla linea RX allacciando il componente su se stesso. Dunque ad ogni trasmissione corrisponderà una ricezione.

0.1.4 Driver Standalone

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. 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 uno 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 avrà il compito di settare la variabile per poter rendere possibile la lettura e risvegliare i processi dalla coda, solo quando il trasferimento è completato. 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 avrà il compito di settare la variabile per poter rendere possibile la scrittura e risvegliare i processi dalla coda, solo quando il trasferimento è completato. 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 non risulteranno essere 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" 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. Viene aperto dunque il descrittore del file associato al device, invocata una write per inserire i caratteri nel buffer di trasmissione e infine una read per leggerli dal buffer di ricezione.
2. poll_user_app.c : l'utente indica da riga di comando la stringa che vuole trasmettere tramite il device UART. Viene aperto dunque il descrittore del file associato al 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 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

```

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 GPIO 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 update the entries */

```



```

9      clock-names = "s00_axi_aclk";
10     clocks = <&clkc 15>;
11     compatible = "xlnx,UART-1.0";
12     interrupt-names = "interrupt";
13     interrupt-parent = <&intc>;
14     interrupts = <0 29 4>;
15     reg = <0x43c00000 0x10000>;
16     xlnx,s00-axi-addr-width = <0x5>;
17     xlnx,s00-axi-data-width = <0x20>;
18 };
19 };
20 };

```

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 potrà osservare sotto /dev il device uio0 corrispondente al nostro 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 "UART_interrupt_uio.h"
11
12 #define DATA_IN      0  // DATA TO SEND
13 #define TX_EN         4  // TRANSFER ENABLE (0)
14 #define STATUS_REG    8  // OE(0) FE(1) DE(2) RDA(3) TX_BUSY(4)
15 #define RX_REG        12 // DATA RECEIVED
16 #define GLOBAL_INTR_EN 16 // GLOBAL INTERRUPT ENABLE
17 #define INTR_EN        20 // LOCAL INTERRUPT ENABLE
18 #define INTR_ACK_PEND  28 // PENDING/ACK REGISTER
19
20 #define TX             1
21 #define RX             2
22
23 #define INTR_MASK 3
24 /**
25  * @file UART_interrupt_uio.c
26  * @page driver_UART_UIO
27  * @brief funzioni per gestire la trasmissione e la ricezione dei
28  *       dati utilizzando il protocollo UART
29  */
30
31 int tx_count, rx_count = 0;
32 int buffer_size = 0;
33 char * buffer_tx;
34 char * buffer_rx;
35
36 /**
37  *
38  * @brief Utilizzata per scrivere un valore all'interno di un registro
39  *       della periferica, specificando l'indirizzo base virtuale e
40  *       l'offset del registro in cui scrivere
41  *
42  * @param addr indirizzo virtuale della periferica
43  * @param offset offset del registro a cui scrivere

```

```

44  * @param valore da scrivere
45  *
46  * @return
47  *
48  * @note
49  *
50  */
51  void write_reg(void *addr, unsigned int offset, unsigned int value)
52  {
53      *((unsigned*)(addr + offset)) = value;
54  }
55  /**
56  *
57  * @brief Utilizzata per leggere un valore da un registro
58  *        della periferica, specificando l'indirizzo base virtuale e
59  *        l'offset del registro da cui leggere
60  *
61  * @param addr indirizzo virtuale della periferica
62  * @param offset offset del registro a cui leggere
63  *
64  * @return valore presente all'interno del registro
65  *
66  * @note
67  *
68  */
69  unsigned int read_reg(void *addr, unsigned int offset)
70  {
71      return *((unsigned*)(addr + offset));
72  }
73
74  /**
75  *
76  * @brief Attende l' arrivo di un interrupt utilizzando la read
77  *        su un device UIO
78  *
79  * @param file_descr descrittore del UIO driver
80  * @param addr indirizzo virtuale della periferica
81  *
82  * @return
83  *
84  * @note
85  *
86  */
87  void wait_for_interrupt(int file_descr, void *addr)
88  {
89
90      printf("Waiting for interrupts.....\n");
91
92      int i = 0;
93      int pending = 0;
94      int reenable = 1;
95      u_int32_t pending_reg = 0;
96      u_int32_t reg_sent_data = 0;
97      u_int32_t reg_received_data = 0;
98
99      /** Attesa sul file descriptor in attesa di ricevere un'interruzione*/
100      read(file_descr, (void *)&pending, sizeof(int));
101      /** Interruzione ricevuta*/
102
103      /** Disasseresisce il transfer enable e disabilita le interruzioni */

```

```

104 write_reg(addr, TX_EN, 0);
105 write_reg(addr, GLOBAL_INTR_EN, 0);
106
107 printf("IRQ detected!\n");
108
109 pending_reg = read_reg(addr, INTR_ACK_PEND);
110 printf("Pending reg: %08x\n", pending_reg); // STAMPA DEBUG
111
112 if((pending_reg & 0x00000002) == 0x00000002){
113
114     printf("---ISR RX---\n");
115
116     if(rx_count < buffer_size){
117         rx_count++;
118         reg_received_data = read_reg(addr, RX_REG);
119         printf("ISR RX - value received: %c\n", reg_received_data);
120         buffer_rx[rx_count] = reg_received_data;
121
122
123     }else if(rx_count == buffer_size){
124         printf("Trasmissione/Ricezione completata, valore ricevuto: ");
125         for(i=0; i<rx_count; i++)
126             printf("%c",buffer_rx[i]);
127
128     }
129     /** ACK ricezione */
130     write_reg(addr, INTR_ACK_PEND, RX);
131     sleep(1);
132     write_reg(addr, INTR_ACK_PEND, 0);
133
134     /** Riabilitazione interruzioni*/
135     write_reg(addr, GLOBAL_INTR_EN, 1);
136 }
137 else if((pending_reg & 0x00000001) == 0x00000001){
138
139     printf("---ISR TX---\n");
140     tx_count++;
141
142     if(tx_count < buffer_size){
143
144         reg_sent_data = read_reg(addr, DATA_IN);
145         printf("ISR TX - value sent: %c\n", reg_sent_data);
146         printf("ISR TX - start sending next value: %c\n", buffer_tx[tx_count]);
147         write_reg(addr, DATA_IN, buffer_tx[tx_count]);
148
149     /** ACK trasmissione*/
150     write_reg(addr, INTR_ACK_PEND, TX);
151     sleep(1);
152     /** Riabilitazione interruzioni*/
153     write_reg(addr, INTR_ACK_PEND, 0);
154     write_reg(addr, GLOBAL_INTR_EN, 1);
155     /** Abilitazione del trasferimento */
156     write_reg(addr, TX_EN, 1);
157 }
158 else{
159     write_reg(addr, INTR_ACK_PEND, TX);
160
161 sleep(1);
162     write_reg(addr, INTR_ACK_PEND, 0);
163     write_reg(addr, GLOBAL_INTR_EN, 1);

```

```

164     }
165 }
166
167 /** Riabilita l'interrupt nell'interrupt controller attraverso il sottosistema UIO */
168
169 int ret = write(file_descr, (void *)&reenable, sizeof(int));
170
171 }
172
173 int main(int argc, char *argv[]){
174
175     int j;
176     void *uart_ptr;
177
178     int DIM = strlen(argv[1]);
179     buffer_size = DIM;
180     buffer_tx = malloc(sizeof(char)*DIM);
181     buffer_rx = malloc(sizeof(char)*DIM);
182
183     buffer_tx = argv[1];
184
185     int file_descr = open("/dev/uio0", O_RDWR);
186     if (file_descr < 1){
187         printf("Errore nell'accesso al device UIO.\n");
188         return -1;
189     }
190
191     unsigned dimensione_pag = sysconf(_SC_PAGESIZE);
192
193     uart_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED, file_descr, 0);
194
195     printf("L'utente ha chiesto di mandare la stringa: %s, di %d caratteri.", buffer_tx, DIM);
196
197     /** Abilitazione interruzioni globali */
198     write_reg(uart_ptr, GLOBAL_INTR_EN, 1);
199
200     /** Abilitazione interruzioni */
201     write_reg(uart_ptr, INTR_EN, INTR_MASK);
202
203     /** Settaggio del primo carattere da mandare */
204     write_reg(uart_ptr, DATA_IN, buffer_tx[0]);
205
206     /** Abilitazione del trasferimento */
207     write_reg(uart_ptr, TX_EN, 1);
208
209     while(tx_count < buffer_size){
210         wait_for_interrupt(file_descr, uart_ptr);
211     }
212
213     /** Fa l'unmap del device UART */
214     munmap(uart_ptr, dimensione_pag);
215
216     free(buffer_rx);
217     return 0;
218
219 }

```

La prima operazione del driver è quella di aprire il file descriptor relativo al device uio0. Successivamente calcola la dimensione della pagina e effettua il mapping tramite chiamata a `mmap()`. Viene invocata una `write_reg` sul registro `DATA_IN` per inserire il primo carattere da inviare nel registro di trasmissione e successivamente viene dato

il segnale di TX_EN. Iterativamente, finchè non sono stati trasmessi un numero di caratteri pari alla dimensione della stringa da trasmettere, viene effettuata una read per mettersi in attesa di eventi interrompenti dal device. Una volta risvegliato dalla chiamata il processo si occupa della gestione dell'interruzione in maniera analoga alla parte applicativa della ISR vista nel capitolo precedente.