ALMA MATER STUDIORUM
UNIVERSITA' DI BOLOGNA

FACOLTA' DI INGEGNERIA
CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

Streaming di immagini via ethernet con Zynq con sistemi operativi Standalone e Linux

Tesi di laurea sperimentale in **Ingegneria Informatica**

CANDIDATO
Simone Mingarelli

RELATORE
Prof. Stefano Mattoccia

SESSIONE I
ANNO ACCADEMICO 2015-2016

INDICE

1. INTRODUZIONE 4

2. ARCHITETTURA ZYNQ-7000 6

3. SISTEMA ZYNQ STANDALONE 9
   - 3.1. Libreria LightWaight TCP/IP (lwIP) 10
   - 3.2. Streaming di immagini e dati 13
     - 3.2.1. File “constant.h” 14
     - 3.2.2. File “interrupt.h” e “frame_interrupt.h” 15
     - 3.2.3. File “platform.h”, “platform_config.h” e “pl_reset.h” 17
     - 3.2.4. File “main.c” 18
       - 3.2.4.1. Funzione “start_udp_application()” 19
       - 3.2.4.2. Interrup Handler 22
   - 3.3. Implementazione ricezione 23
     - 3.3.1. File “get_frame_network.c” 24
     - 3.3.2. File “my_opencv.c” e “show_frame.c” 25
     - 3.3.3. Il file “main.c” su PC 27
   - 3.4. Problematiche riscontrate 28

4. ZYNQ e LINUX 31
   - 4.1. Operazioni perliminari 32
     - 4.1.1. Formattazione SD Card 32
     - 4.1.2. Installazione Petalinux 33
     - 4.1.3. Comandi Petalinux 36
   - 4.2. Implementazione invio kernel_mode 38
     - 4.2.1. Funzioni base per la gestione del modulo 39
       - 4.2.1.1. Funzione “ov7670_read” 40
       - 4.2.1.2. Funzione “ov7670_write” 41
       - 4.2.1.3. Struttura “platform_driver” 42
       - 4.2.1.4. Funzione “ov7670_init” 43

4.2.1.5. Funzione “ov7670_exit” 45
4.2.2. Interrupt in Linux 46
4.2.3. Invio frame con UDP kernel socket 47
4.3. Implementazione alternativa user space 48
4.3.1. File “constant.h” e “main.c” 48
4.3.2. File “udp_app.c” 49
4.3.3. File “tcp_app.c” 51
4.4. Implementazione ricezione 52
4.4.1. File “my_tcp_app.h” 52
4.5. Problematiche 54

5. RISULTATI SPERIMENTALI E CONCLUSIONI 55

Bibliografia 57

1. INTRODUZIONE

Il seguente lavoro di tesi si inserisce all'interno di un progetto accademico volto alla realizzazione di un sistema capace elaborare immagini utilizzando una rete FPGA, acquisite da una telecamera stereo. Tale sensore (stero o mono), collegato a un sistema Zynq [2], sarà mappato sulla scheda FPGA in modo tale da scrivere direttamente sulla RAM condivisa con il processore ARM il flusso di frame acquisiti.

Per evitare problemi di letture non consistenti si è scelto di utilizzare un *buffer* contiguo di 8 frame, avente indirizzo di partenza 0x10000000 e un *frame_index* (mappato a 0x42100000), che indica l'ultimo frame completamente scritto in memoria. Ogni scrittura di un frame in memoria provoca l'invio di un *interrupt* alla CPU. Il progetto mappato sulla scheda FPGA è sviluppato da Riccardo Albertazzi nella Tesi [26].

L'obiettivo è creare un sistema client/server che permetta il trasferimento e la visualizzazione a video del flusso di frame. Lo schema completo della configurazione hardware è rappresentato in Figura 1.

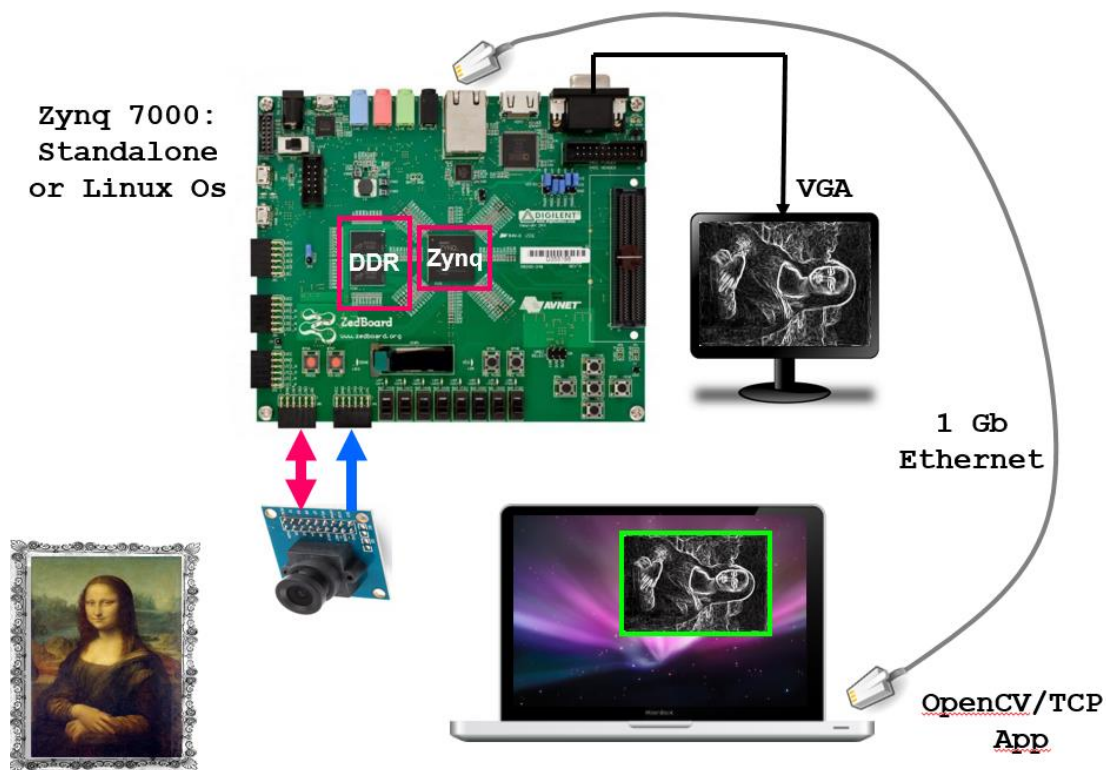

Figura 1: Schema funzionamento

Il progetto che sarà eseguito sulla ZedBoard è proposto in due versioni:

- la prima più minimale, in modalità *bare metal* o *standalone* (assenza di sistema operativo)
In questa modalità si è scelto di implementare solamente le funzioni di base: gestione della periferica ethernet (creando un apposito *driver* per gestire le operazioni su di essa), registrazione *interrupt* e invio dei frame utilizzando il protocollo UDP.

- la seconda versione, più evoluta, è eseguita sul sistema operativo *Petalinux* [3], una distribuzione Linux resa disponibile da Xilinx [14]. Rispetto alla precedente implementa anche una connessione TCP, destinata a inviare o ricevere dati che richiedono affidabilità come comandi o eventuali parametri di configurazione al modulo FPGA.

La ricezione delle immagini trasmesse dal sistema Zynq, eseguita su un PC, è stata implementata utilizzando le normali librerie c *socket.h*, il pacchetto *OpenCv* [1] per la visualizzazione a video dei dati ricevuti e la libreria *pthread* [11] per ottimizzare le prestazioni generali. Nella Figura 2 si può osservare la configurazione dell'intero sistema nel caso Linux, nel Capitolo successivo sarà descritta nel dettaglio anche la configurazione *standalone*.

Figura 2: Configurazione completa caso Linux

2. ARCHITETTURA ZYNQ-7000

La *evaluation board* ZedBoard prodotta da AVNET [5], è basata sul dispositivo Xilinx Zynq-7000 A*ll Programmable SoC.* Tale dispositivo [4], mostrato in Figura 1, mette in stretta correlazione due distinte unità funzionali: il *Processing System* (PS) e la *Programmable Logic* (PL).
La prima comprende un processore ARM *dual core* Cortex A9. Mentre la seconda è composta da una FPGA (*Field Programmable Gate Array*).

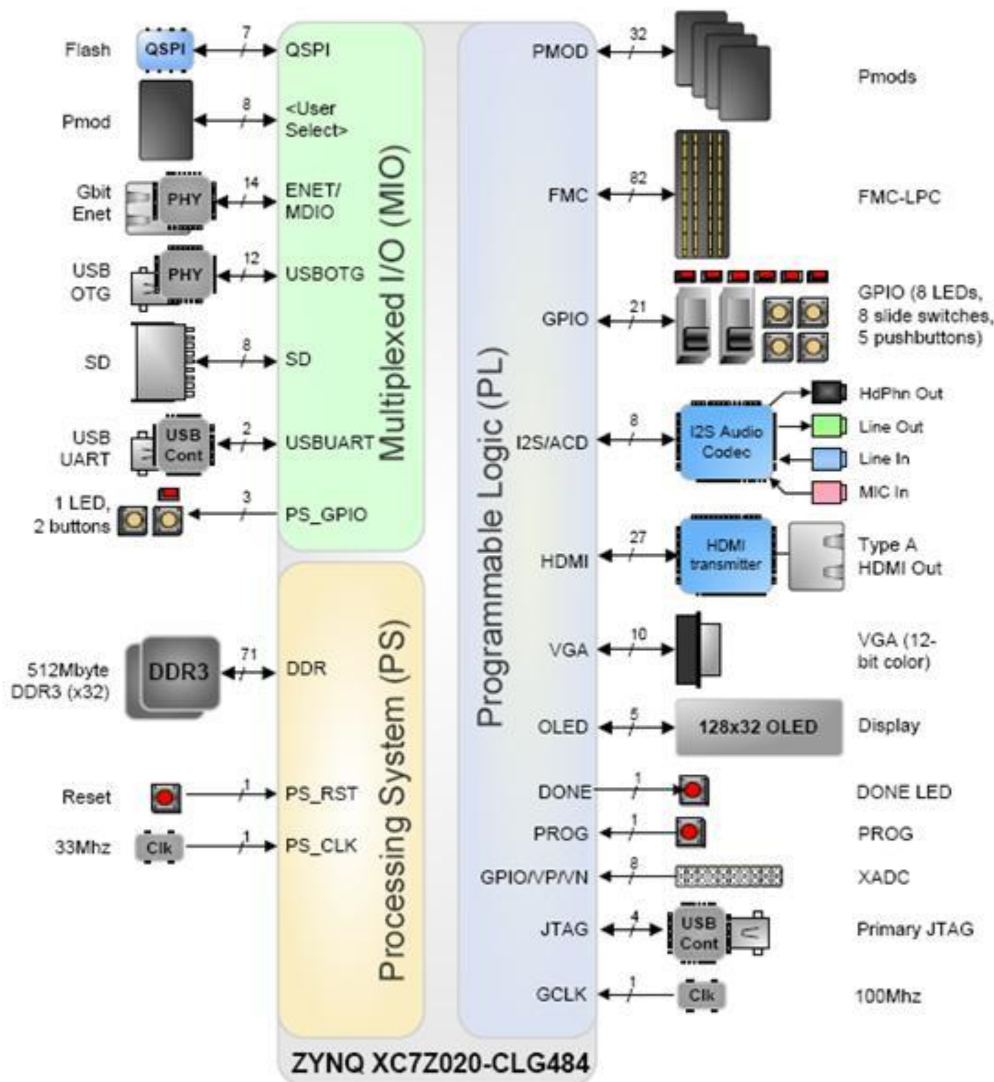

Figura 3: Architettura del sistema Zynq 7000

Il vantaggio di questa configurazione hardware, schema in Figura 3, è di avere in un unico dispositivo sia una CPU sia una FPGA. Quest'ultima può essere programmata per eseguire le

operazioni più onerose dal punto di vista computazionale migliorando le prestazioni, riducendo il carico di lavoro della CPU e diminuendo i consumi energetici. La Zedboard può essere configurata, utilizzando i *jumpers*, in diverse modalità. Nel nostro caso li useremo per specificare che il *boot* *a*vvenga dalla periferica SD. Maggiori informazioni sulla funzione dei *jumpers* sono disponibili al seguente link [6].

Caso sistema *bare metal*:

JP2 – CHIUSO

JP3 – CHIUSO

JP7 - GND

JP8 - GND

JP9 - GND

JP10 - GND

JP11 - GND

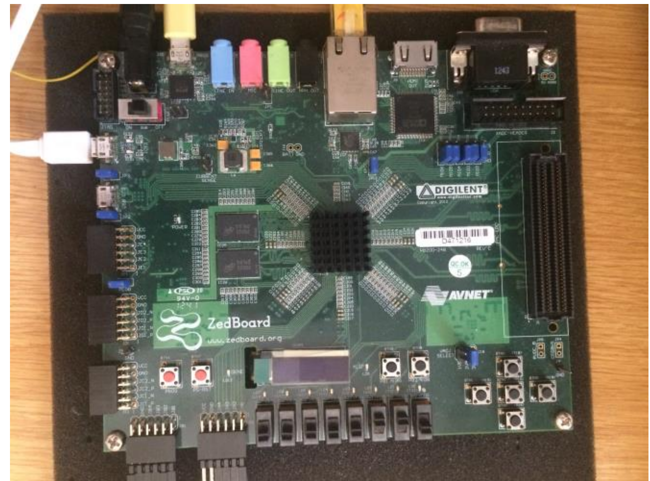

Figura 4: Configurazione bare metal

La Figura 4 mostra la configurazione bare metal. Per programmare l'FPGA ed eseguire codice/debug utilizzando SDK [15] è necessario collegare le due porte UART in alto a sinistra. (La microUSB bianca è utilizzata solamente per le stampe su consolle).

Caso sistema *Linux*:

JP2 – APERTO

JP3 - APERTO

JP7-GND

JP8-GND

JP9-3V3

JP10-3V3

JP11-GND

Figura 5: Configurazione boot SD Card

In Figura 5 è mostrata la configurazione per eseguire un sistema operativo Unix. In questo caso l'unica porta UART collegata sarà utilizzata per avviare il *boot* di Petalinux [3]. Tale procedura può

essere automatizzata modificando le impostazioni del sistema operativo utilizzato, utilizzando SSH [16] da un terminale remoto per eseguire le operazioni desiderate.

3. SISTEMA ZYNQ STANDALONE

L’ambiente *standalone* Zynq supporta lo sviluppo di applicazioni in linguaggio C, mettendo a disposizione alcune librerie per compiti specifici [7].

Come anticipato, l’obiettivo del progetto è la trasmissione di immagini acquisite mediante la PL (FPGA) e invio mediante PS a un *host*, via ethernet, per una successiva elaborazione e/o visualizzazione in tempo reale. In particolare, in questa tesi si è focalizzata l’attenzione sulla trasmissione e visualizzazione delle immagini ricevute dall’host sfruttando un progetto hardware oggetto di una contemporanea tesi di laurea.

Per la realizzazione della nostra applicazione adotteremo una gestione a *interrupt* per quanto riguarda la notifica della scrittura di un nuovo frame in memoria. Per l’invio sono stati utilizzati i metodi offerti dalla libreria lwIP [8] un’implementazione indipendente del protocollo TCP/IP, che utilizza meno risorse rispetto alla libreria socket TCP/UDP Unix [9] ed è ampiamente diffusa nei sistemi *embedded*.
Il codice ricevente, eseguito sul PC e scritto in C, utilizza la libreria OpenCV [1] per la visualizzazione delle immagini, le normali socket “<socket.h>” in ambiente Linux e le *WinSock* [10] (libreria che permette di utilizzare le stesse API di Linux su Windows) per l’ambiente Windows.

Per ottimizzare e limitare l’utilizzo delle risorse hardware utilizzeremo i *pthread* [11], disponibili sia in Linux sia in Windows, assicurandoci l’atomicità delle operazioni di lettura e scrittura delle variabili condivise con il MUTEX_LOCK già implementato dalla libreria.

3.1. Libreria LightWeight TCP/IP

La libreria LightWeight TCP/IP [8], meglio nota come lwIP, nasce come un'implementazione ridotta, ma sufficiente per gli obiettivi del presente lavoro, del protocollo TCP/IP. Gli obiettivi di questa implementazione del protocollo TCP/IP sono l'ottimizzazione delle risorse utilizzate in modo da aumentare l'efficienza e ridurre i consumi della piattaforma hardware che esegue il codice. Queste caratteristiche sono molto utili nel nostro caso, poiché avendo una banda molto elevata (fino a 1 Gbit/s) se l'invio dei dati non avvenisse abbastanza velocemente potrebbero verificarsi criticità nella gestione degli *interrupt*.

Lwip è multi-piattaforma che può essere utilizzata sia in modalità *bare metal* sia con un sistema operativo standard come Linux e comprende la gestione dei seguenti protocolli:

- IP (Internet Protocol), tra cui l'inoltro di pacchetti su più interfacce di rete
- ICMP (Internet Control Message Protocol) per la manutenzione della rete e il debug
- IGMP (Internet Group Management Protocol) per la gestione del traffico multicast
- UDP (User Datagram Protocol) comprese le estensioni sperimentali UDP-Lite
- TCP (Transmission Control Protocol) con controllo della congestione, la stima RTT e il recupero veloce e la ritrasmissione veloce
- Raw/API socket native per migliorare le prestazioni
- Berkeley API socket
- DNS (Domain Name Resolver)
- SNMP (Simple Network Management Protocol)
- DHCP (Dynamic Host Configuration Protocol)
- PPP (Point-to-Point Protocol)
- ARP (Address Resolution Protocol) per Ethernet

Il protocollo scelto per il progetto è UDP per via della sua maggiore velocità di trasmissione rispetto alla controparte TCP, che esegue un controllo sull'invio/ricezione dei dati. Inoltre per la natura dell'applicazione (streaming real-time di immagini) la soluzione basata su UDP risulta più indicata perché in caso di perdita di dati non è previsto il re-invio di alcun pacchetto. Questo implica la

perdita del frame anche nel caso non sia ricevuto un solo pacchetto. Tuttavia, questo non rappresenta un problema nel contesto applicativo considerato purché la perdita di pacchetti/immagini sia un evento raro.
Di seguito è riportato un elenco delle strutture e dei metodi della libreria utilizzati per l'invio dei singoli frame.
Al seguente link è possibile trovare una esaustiva documentazione inerente lwIP [12].

- La prima struttura di fondamentale importanza è ***netif***

Ogni struttura *netif* rappresenta l'interfaccia di rete di ogni singolo dispositivo. I vari campi della struttura tengono traccia delle informazioni necessarie come il nome, l'indirizzo IP e i puntatori alle funzioni che il driver dovrà chiamare quando sono ricevuti e inviati i pacchetti di dati. È inoltre possibile definire più *netif* su un singolo dispositivo per semplificare la gestione di diversi protocolli.

- I pacchetti sono rappresentati dalla struttura ***pbuf***:

Essa è possibile considerarla come i *datagram* della libreria TCP/IP standard. Presenta alcune funzionalità particolari come la possibilità di specificare il loro metodo di allocazione.

- PBUF_RAM: la memoria è allocata in RAM, come un unico blocco che comprende anche gli *headers*
- PBUF_ROM: si suppone che sia utilizzato come se la memoria fosse molto simile a una ROM (invariata). non è allocata memoria durante la creazione di un nuovo *pbuf*, ma si suppone che sia inserito in una catena di *pbuf*, dove l'intestazione è contenuta nei precedenti
- PBUF_REF: non è allocata memoria durante la creazione di un nuovo *pbuf*. È usato nelle applicazioni a singolo *thread*, è necessario richiamare *pbuf_take* per copiare un pbuf dal POOL creato precedentemente e assegnarlo al PBUF_REF
- PBUF_POOL: è creato un *pool* di *pbuf* durante la *pbuf_init()* che saranno utilizzati successivamente.

- La struttura **udp_pcb:**

Essa può essere interpretata come una *socket*. Tuttavia presenta alcune differenze, come la possibilità di eseguire una *udp_connect(...)* che permette di memorizzare nei campi appositi della struttura l'indirizzo (IP, porta) a cui inviare e ricevere pacchetti.

In questo modo si può utilizzare *udp_send(udp_pcb,pbuf)* al posto della tradizionale *udp_sendto(udp_pcb, pbuf, ip, port)*.
A più basso livello il metodo *udp_send(...)* richiamerà *udp_sendto(...)*passando come IP e porta di destinazione i rispettivi campi contenuti della struttura *udp_pcb* precedentemente impostati utilizzando *udp_connect(...)*.

Per poter essere utilizzata, la libreria lwIP deve essere inizializzata. A tal proposito è fondamentale richiamare il metodo *lwip_init()* prima di utilizzare la libreria. Inoltre, trattandosi di una implementazione del protocollo standard TCP/IP che mette a disposizione molti parametri di configurazione per ottimizzare le risorse, è importante eseguire una corretta gestione della memoria in particolare per quanto riguarda l'utilizzo dei *pbuf*. È lasciata al programmatore la completa gestione dell'allocazione e de-allocazione di questi ultimi. Ogni *pbuf* è *"one shot"* (può essere utilizzato per un solo invio e ricezione), contiene campi nel suo *header* che perdono significato una volta utilizzato. È inoltre possibile modificare le dimensioni dei *buffer* di ricezione e invio, oltre che ai *timeout* e la priorità di gestione di diversi tipi di pacchetti (distinti per protocollo o *netif*). Queste caratteristiche rendono lwIP molto versatile e in grado di adattarsi a utilizzi che richiedono specifiche differenti.

3.2. STREAMING DI IMMAGINI E DATI

Il progetto hardware utilizzato è configurato per salvare in memoria DDR (condivisa tra FPGA e ARM) un *frame-buffer* composto da 8 immagini. Ognuna di esse occupa 307.200 byte (640x480 pixels). Appena un *frame* è scritto in memoria, è immediatamente generato un *interrupt* che avvisa il processore ARM di tale evento. Inoltre all'indirizzo di memoria 0x42100000, è aggiornato dal modulo implementato su FPGA un registro che indica quale tra le 8 immagini che compongono il frame-buffer è stata scritta in memoria, permettendo al sistema ARM di sapere qual è l'ultima immagine da estrarre dalla DDR e da inviare via ethernet.

Di seguito, in Figura 6, è mostrata la struttura del progetto *standalone* eseguito sulla parte di PS dello Zynq.

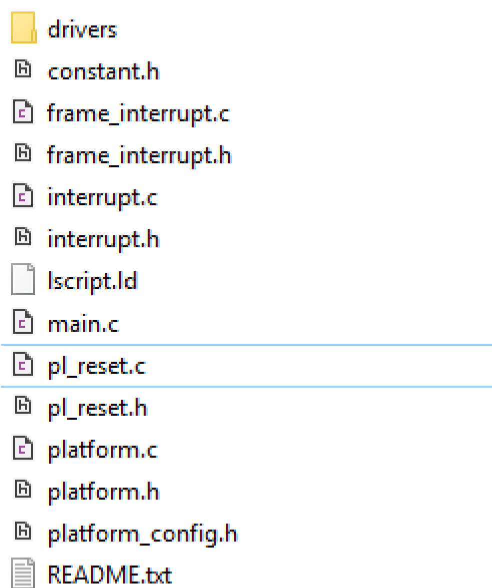

Figura 6: Struttura progetto standalone

3.2.1. Il file "constant.h"

Nel file `constant.h`, Figura 7, sono dichiarate tutte le costanti e le strutture dati utilizzate dal software.

```
#define MTU_SIZE 1500
#define FRAME_SIZE 307200

/*
 * Rappresenta la dimensione inviata nella struttura per pacchetto dati
 */
#define FRAME_UDP_FRAGMENT_SIZE 7680 // Nel caso windows 30720

#define LOCAL_PORT 5454
#define REMOTE_PORT 5555

#define FRAME_BUFFER_DIM 307200
#define FRAME_BUFFER_BASE_ADDR 0x10000000
#define FRAME_BUFFER_NUM 8

typedef unsigned char BYTE;

typedef struct{
    int count;
    int fragment;
    int frame_index;
    BYTE data[FRAME_UDP_FRAGMENT_SIZE];
}packet_data;
```

Figura 7: File "constant.h"

In particolare:

- FRAME_SIZE: rappresenta la dimensione di un frame in memoria
- FRAME_UDP_FRAGMENT_SIZE: rappresenta la dimensione del frammento di frame che è inviato in ogni pacchetto UDP
- BYTE: tipo di dato definito per comodità, corrisponde a un *unsigned char*
- packet_data: struttura che rappresenta il payload del pacchetto UDP

3.2.2. File “interrupt.h” e “frame_interrupt.h”

I due file `interrupt.h` e `frame_interrupt.h` contengono le definizioni delle funzioni necessarie a gestire gli interrupt e registrare la funzione di *callback*, ovvero l’*interrupt handler*. Nella implementazione corrente, l’interrupt handler che registreremo invierà un singolo frame, prendendo come parametro di ingresso il *frame_index* letto dalla periferica, mappata in memoria all’indirizzo 0x42100000, che rappresenta l’ultimo frame scritto.

In particolare nel file `interrupt.c`, come mostrato in Figura 7, contiene il metodo per istanziare l’interrupt controller che risulta essere un *singleton*.

```c
if (!inizializzato)
{
    int Status;
    XScuGic_Config *IntcConfig; /* Instance of the interrupt controller */

    Xil_ExceptionInit();

    /*
     * Initialize the interrupt controller driver so that it is ready to
     * use.
     */
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (NULL == IntcConfig) {
        inizializzato = -1;
        return NULL;
    }

    Status = XScuGic_CfgInitialize(&InterruptController, IntcConfig, IntcConfig->CpuBaseAddress);

    if (Status != XST_SUCCESS) {
        inizializzato = -1;
        return NULL;
    }


    /*
     * Connect the interrupt controller interrupt handler to the hardware
     * interrupt handling logic in the processor.
     */

    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
                (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                &InterruptController);

    Xil_ExceptionEnable();

    inizializzato = 1;
    return &InterruptController;
```

Figura 8: File “interrupt.c”

```c
int frame_interrupt_init(XScuGic* gicInst)
{
    if (gicInst == NULL || gicInst->IsReady == 0)
    {
        xil_printf("Parametro gicInst invalido\n");
        return XST_FAILURE;
    }

    xil_printf("Configurazione GPIO\n");
    if (XGpio_Initialize(&frame_intr_gpio, XPAR_AXI_GPIO_FRAME_INTR_DEVICE_ID) != XST_SUCCESS)
    {
        xil_printf("Il GPIO del frame_index non e' stato trovato\n");
        return XST_FAILURE;
    }

    if (XScuGic_Connect(gicInst, FRAME_INDEX_INTR, frame_index_interrupt_handler, (void*) &frame_intr_gpio) != XST_SUCCESS)
    {
        xil_printf("XScuGIc_Connect failed!\n");
        return XST_FAILURE;
    }


    XGpio_InterruptEnable(&frame_intr_gpio, 0xFFFFFFFF);
    XGpio_InterruptGlobalEnable(&frame_intr_gpio);

    //u32 enabled = XGpio_InterruptGetEnabled(&frame_intr_gpio);
    //xil_printf("GPIO interrupts enabled = %d\n\n", enabled);

    gic = gicInst;

    return XST_SUCCESS;

}
```

Figura 9: File “frame_interrupt.c” registrazione interrupt

Come mostrato in Figura 9, il file `frame_interrupt.c` implementa la configurazione del *gpio_frame_index*, registrando la funzione di *callback*, Figura 10, che verrà richiamata ogni volta si verifica l’interrupt desiderato.

```c
void frame_interrupt_register_callback(void (*FrameCallback)(int frame_index))
{
    XScuGic_Enable(gic, FRAME_INDEX_INTR);
    FrameWrittenCallback = FrameCallback;
}
```

Figura 10: File “frame_interrupt.c” registrazione interrupt handler

3.2.3. File “platform.h”, “platform_config.h” e “pl_reset.h”

I file `platform.h` e `platform_config.h` contengono le impostazioni di configurazione della ZedBoard. Per future espansioni del progetto con altre *evaluation board* è possibile implementare in questi due file la gestione di hardware differenti.

```c
int pl_reset_enable()
{
    int init = pl_reset_initialize();
    if (init == 1)
    {
        XGpio_DiscreteWrite(&resetGPIO, 1, RESET_ENABLED);
        return 0;
    }
    else return init;
}

int pl_reset_disable()
{
    int init = pl_reset_initialize();
    if (init == 1)
    {
        XGpio_DiscreteWrite(&resetGPIO, 1, RESET_DISABLED);
        return 0;
    }
    else return init;
}
```

Figura 11: File “pl_reset” invio reset software

Il file `pl_reset.h` mette a disposizione i metodi per inviare, via software, un reset alla parte FPGA. Per reset si intende il caricamento di una configurazione nota per permetterle il suo normale funzionamento, non l’intera riprogrammazione del dispositivo FPGA. Le due funzioni mostrate in Figura 9 implementano l’invio del reset. Esso è eseguito scrivendo in un registro del *gpio_pl_reset* (componente del progetto Vivado [23]) il valore 0 o 1.

NOTA: prima di poter utilizzare i metodi descritti bisogna richiamare *pl_reset_initialize()*, implementato nello stesso modo del file `frame_interrupt.c` (Figura 11), che inizializza il gestore dell’interrupt del componente *pl_reset*.

3.2.4. File “main.c”

Nel file `main.c` è implementa l’applicazione utilizzando i metodi definiti in precedenza al fine di ricevere le immagini, leggere il *frame_index* a 0x42100000 e inviare l’immagine via ethernet.

Trovandoci in un ambiente senza un vero e proprio sistema operativo, come prima cosa, sarà necessario inizializzare i componenti hardware con cui interagiremo, registrando i segnali e le variabili necessarie al loro funzionamento.

```
struct ip_addr netmask, gw;

/* the mac address of the board*/
unsigned char mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };

udp_netif = &server_netif;

init_platform();

//Reset di 1 ms alla PL
pl_reset();

if (inizializza_writer() != XST_SUCCESS)
    xil_printf("Errore inizializzazione writer\n");

if (inizializza_reader() != XST_SUCCESS)
    xil_printf("Errore inizializzazione reader\n");

/* initliaze IP addresses to be used */
IP4_ADDR(&ipaddr,  192, 168,   1, 10);
IP4_ADDR(&netmask, 255, 255, 255,  0);
IP4_ADDR(&gw,      192, 168,   1,  1);

/* Remote ip address */
IP4_ADDR(&d_addr,  192, 168,   1, 100);

//netif_add(udp_netif, &ipaddr, &netmasck, &gw, NULL, NULL, ethernet_input());
lwip_init();


/* Add network interface to the netif_list, and set it as default */
if (!xemac_add(udp_netif, &ipaddr, &netmask, &gw, mac_ethernet_address, PLATFORM_EMAC_BASEADDR))
{
    xil_printf("Error adding N/W interface\n\r");
    return -1;
}

udp_netif->mtu = MTU_SIZE;
netif_set_default(udp_netif);
netif_set_up(udp_netif);
netif_set_link_up(udp_netif);

xil_printf("Setting netif\n");
print_ip_settings(&udp_netif->ip_addr, &udp_netif->netmask, &udp_netif->gw);

start_udp_application(LOCAL_PORT, REMOTE_PORT);
enable_interrupts();
```

Figura 12: Prima parte del file main.c

Come mostrato nel codice riportato in Figura 12, la prima operazione effettuata è l’inizializzazione del *writer* e del *reader*. Essi si occuperanno di gestire le aree di memoria in uso della DDR.

```
int inizializza_writer()
{
    int status;

    XAxis_to_ddr_writer writer;
    status = XAxis_to_ddr_writer_Initialize(&writer, XPAR_XAXIS_TO_DDR_WRITER_0_DEVICE_ID);

    if (status != XST_SUCCESS)
        return status;

    XAxis_to_ddr_writer_Set_base_ddr_addr(&writer, FRAME_BUFFER_BASE_ADDR);
    XAxis_to_ddr_writer_Set_frame_buffer_dim(&writer, FRAME_BUFFER_DIM);
    XAxis_to_ddr_writer_Set_frame_buffer_number(&writer, FRAME_BUFFER_NUM);
    XAxis_to_ddr_writer_Set_frame_buffer_offset(&writer, FRAME_BUFFER_DIM);
    XAxis_to_ddr_writer_EnableAutoRestart(&writer);
    XAxis_to_ddr_writer_Set_update_intr(&writer, 1);
    XAxis_to_ddr_writer_Start(&writer);

    return XST_SUCCESS;

}
```

Figura 13: funzione inizializza_writer()

Queste due funzioni: *inizializza_writer()* in Figura 13 e *inizializza_reader()*, implementata nello stesso modo della prima, utilizzano i metodi offerti dai file auto-generati della cartella *drivers* in Figura 6 per raccogliere gli indirizzi delle zone di memoria sia RAM sia registri delle periferiche GPIO, modificati dalla FPGA.

La successiva funzione *xemac_add(…)* si occupa dell'inizializzazione della periferica ethernet in modo quasi completamente trasparente. Essa non si limita solamente ad assegnare *IP* e *MAC address*, ma registrerà i gestori agli eventi generati dalla periferica hardware (arrivo pacchetti, invio pacchetti, ecc) trasferendo a livello applicativo solo quelli di interesse e richiamando tramite interrupt le funzioni destinate alla loro gestione. In seguito ad aver dichiarato la struttura *netif* come gestore di default della periferica, la funzione *start_udp_application()* inizializza la socket UDP. Infine, il metodo *enable_interrupts()* dichiara il gestore dell'*interrupt_frame_index*, Figura 9, e registrata la funzione di *callback* utilizzando il metodo mostrato nella Figura 10, in questo momento l'applicazione inizierà ad inviare le immagini utilizzando il protocollo UDP.

3.2.4.1. Funzione “start_upd_application()”

In lwIP la struttura che *udp_pcb* sostituisce il comportamento di una socket C tradizionale. Il suo procedimento di inizializzazione ha lo stesso procedimento logico, mostrato in Figura 14, di una normale socket. Per semplicità di gestione è stato scelto di inviare i pacchetti in BROADCAST, in quanto l’invio di un pacchetto ad un indirizzo specifico prevede la risoluzione dell’IP utilizzando ARP. Passando al livello 2 di ISO/OSI, nell’*heather* di un *frame ethernet* devono essere presenti gli indirizzi MAC di mittente e destinatario. Per eseguire questa operazione il mittente del pacchetto deve conoscere il MAC *address* di destinazione (ottenuto tramite ARP). La gestione della coda di ingresso di una periferica di rete è implementata con un sistema *a interrupt* la cui inizializzazione è completamente trasparente, ed eseguita dal metodo *xemac_add(..)* precedentemente descritto, Figura 10.

Durante l’implementazione dell’invio verso un singolo indirizzo IP ho riscontrato criticità di gestione dei pacchetti in ingresso alla ZedBoard. Nello specifico eseguendo il metodo *enable_interrupts()* la periferica di rete non processava eventuali pacchetti in entrata, comprese le risposte ARP. Nell’implementazione a *interrupt handler* è stato quindi utilizzato l’invio in BROADCAST.

```c
int start_udp_application(unsigned local_port, unsigned remote_port)
{
    pcb = udp_new();
    if (!pcb)
    {
        xil_printf("Error creating PCB. Out of Memory\n\r");
        return XST_FAILURE;
    }

    /* bind to specific port */
    if(udp_bind(pcb, IP_ADDR_ANY, local_port) != ERR_OK)
    {
        xil_printf("Unable to bind to port %d: err = %d\n\r", local_port, err);
        return XST_FAILURE;
    }

    if(udp_connect(pcb, IP_ADDR_BROADCAST, remote_port) != ERR_OK)
    {
        xil_printf("Unable to connect to port %d: err = %d\n\r", remote_port, err);
        return XST_FAILURE;
    }

    /* Registro la funzione di callBack */
    udp_recv(pcb, recv_callback, NULL);

    xil_printf("UDP server started Local@ port %d ", pcb->local_port);
    print_ip(" @ and Ip: ", &pcb->local_ip);
    xil_printf("UDP server started Remote @ port %d", pcb->remote_port);
    print_ip(" @ and Ip: ", &pcb->remote_ip);

    return XST_SUCCESS;
}
```

Figura 14: Funzione start_udp_application(…)

Per specializzare il *pcb* nella comunicazione con un indirizzo (IP e porta) si utilizza la funzione *udp_connect(...)*. Come è possibile vedere in Figura 14 i campi passati come parametri saranno salvati in variabili interne della struttura del *pcb*. In questo modo è possibile richiamare la funzione *udp_sand(pcb, pbuf)*, la quale si occuperà automaticamente di inviare il pacchetto al destinatario impostato precedentemente, senza la necessità di usare la funzione *udp_sendto(pcb, pbuf, ip_dest, porta_dest)*.

3.2.4.2. Interrupt Handler

Il metodo *enable_interrupt()* richiamato nel main, inizializza il *gpio_frame_index* e registra la funzione di *callback* (*interrupt handler* mostrato in Figura 15) utilizzando i metodi precedentemente descritti nel Paragrafo 3.2.2.

```
void frame_written_callback(int frame_index)
{
    xemacif_input(udp_netif);
    Xil_DCacheFlushRange(start_buffer(), buffer_length());

    for(i = 0; i < FRAME_SIZE / FRAME_UDP_FRAGMENT_SIZE; i++)
    {
        struct pbuf* p = pbuf_alloc(PBUF_TRANSPORT, sizeof(packet_data), PBUF_RAM);
        if(!p)
        {
            xil_printf("errore allocazione pbuf: %d\n", i);
            exit(1);
        }

        packet_data* buff = (packet_data*) p->payload;
        buff->count = counter++;
        buff->fragment = i;
        buff->frame_index = frame_index;

        BYTE* ddr_address = ((BYTE*) FRAME_BUFFER_BASE_ADDR) + frame_index*FRAME_SIZE + i*FRAME_UDP_FRAGMENT_SIZE;
        memcpy(buff->data, ddr_address, FRAME_UDP_FRAGMENT_SIZE);

        if(udp_send(pcb, p) != ERR_OK)
            xil_printf("Errore nell' invio del pacchetto %d\n", i);
        usleep(50);
        pbuf_free(p);
    }
}
```

Figura 15: Interrupt Handler

Analizzando il codice dell'*interrupt handler* mostrato in Figura 15 è possibile notare che prima di effettuare qualsiasi operazione è disabilitata la cache della CPU nella zona del buffer. Questa operazione viene effettuata per evitare che, leggendo ripetitivamente dalla stessa zona di memoria, la CPU ci fornisca un dato non aggiornato che è salvato in cache, al posto del frame che si sta inviando. Una volta eseguita questa operazione si procede all'invio del frame frammentandolo per non superare la dimensione massima del payload UDP (64k).

È stato riscontrato che è necessario inserire una *usleep* dopo la *udp_send(...),* in quanto è stata verificata la possibilità che per operazioni temporalmente molto lunghe non sia garantita l'atomicità. In particolare l'area di memoria del *pbuf* era rilasciata prima del termine della *udp_send(...)* causando un errore nella lettura della memoria durante l'invio.

3.3. IMPLEMENTAZIONE RICEZIONE

Il progetto eseguito su PC, o su qualsiasi dispositivo di ricezione con un sistema operativo standard, si occupa della ricezione e della visualizzazione delle immagini.

Per la sua realizzazione sono utilizzate le normali socket.c, la libreria OpenCV [1] per visualizzare le immagini a video e la libreria pthread [11] che ci permetterà di ottimizzare il codice e migliorarne la leggibilità. Nella Figura 16 è rappresentata la struttura del progetto.

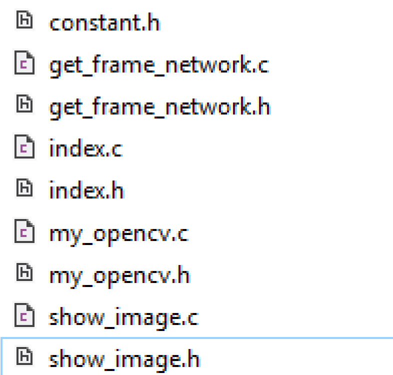

Figura 16: Struttura progetto ricezione

Il file `constant.h` è la copia dell'omonimo file presente sulla ZedBoard, in più presenta `#include "../os.h"` che contiene una costante che identifica il sistema operativo in cui ci troviamo, in quanto il client è compilabile per diversi sistemi operativi e in particolare per Linux e Windows.

3.3.1. File “get_frame_network.c”

Il codice contenuto nel file `get_frame_network.h` consente di astrarre completamente dai livelli sottostanti mettendo a disposizione pochi metodi:

- *start_udp_application()*: inizializza tutte le componenti necessarie alla ricezione delle immagini.
- *stop_udp_application()*: rilascia tutte le risorse utilizzate.
- *get_net_frame()*: fornisce un nuovo frame quando disponibile; nel caso in cui sia già stato fornito il frame corrente mette in *wait()* il processo fino all’arrivo di un nuovo frame.

All’interno del codice incluso in `get_frame_network.c` sono utilizzati i pthread [11]. In particolare, *start_udp_application()* crea un *thread* che esegue la funzione *udp_app()* Figura 17. Il codice mostrato dichiara una socket e si mette in ascolto in attesa di pacchetti UDP. Un nuovo frame è memorizzato solo se arriva completamente integro, ovvero senza perdita di nessun pacchetto. In caso contrario (mancata ricezione di almeno un pacchetto) il frame è scartato e si attende la ricezione del successivo.

```c
void udp_app(){
#ifdef WINDOWS
    WSADATA wsa;
    //Initialise winsock
    printf("\nInitialising Winsock...");
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("Failed. Error Code : %d", WSAGetLastError());
        exit(EXIT_FAILURE);
    }
    printf("Initialised.\n");
#endif
    /* Preparazione indirizzi */
    /* Inizializzazione indirizzo locale, porta statica*/
    memset((char *)&local_udp_addr, 0, sizeof(struct sockaddr_in));
    local_udp_addr.sin_family = AF_INET;
    local_udp_addr.sin_addr.s_addr = INADDR_ANY;
    local_udp_addr.sin_port = htons(local_udp_port);

    /* La zedboard si trova a un indirizzo statico (da modificare se si implementa un discover con broadcast */
    memset((char *)&remote_udp_addr, 0, sizeof(struct sockaddr_in));
//  remote_udp_addr.sin_family = AF_INET;
//  remote_udp_addr.sin_addr.s_addr = inet_addr(remote_udp_ip);
//  remote_udp_addr.sin_port = htons(remote_udp_port);

    /* Creazione socket */
    sd=socket(AF_INET, SOCK_DGRAM, 0);
    if(sd<0) {perror("apertura socket"); exit(3);}
    printf("Creata la socket sd=%d\n", sd);

    /* Bind */
    if(bind(sd,(struct sockaddr *) &local_udp_addr, sizeof(local_udp_addr))<0)
    {
        perror("Bind Socket non riuscita: ");
        exit(1);
    }
    printf("Bind socket ok, alla porta %i\n", ntohs(local_udp_addr.sin_port));
```

Figura 17: Inizializzazione socket UDP funzione “udp_app()”

3.3.2. File “my_opencv.c” e “show_frame.c”

Nel file `my_opencv.c` sono implementate le funzioni necessarie alla visualizzazione, estraendo completamente il resto del progetto da OpenCV [1]. La Figura 18 sono riportate le API da utilizzare per la visualizzazione delle immagini.

```
#ifndef MY_OPENCV_H_
#define MY_OPENCV_H_

#include "constant.h"
#include <opencv/cv.h>
#include <opencv/highgui.h>

void start_my_opencv();
void release_image();
void show_frame(char * name);
void push_new_frame(BYTE * frame_start);
void write_text(char * msg);
IplImage * get_IplImage();


#endif /* MY_OPENCV_H_ */
```

Figura 18: File “my_opencv.h”

In particolare, all’avvio, è necessario richiamare *start_my_opencv()* per poter utilizzare gli altri metodi, quest’ultimo inizializza le componenti necessarie alla libreria.

- show_frame(): visualizza l’ultimo frame scritto nella libreria
- write_text(): scrive il testo passato come parametro sul frame

Per creare un *thread* che si occupi ciclicamente di richiedere un nuovo frame e lo visualizzi a video è possibile utilizzare le API mostrate in Figura 19.

```
#ifndef SHOW_IMAGE_H_
#define SHOW_IMAGE_H_

#include <pthread.h>
#include "constant.h"
#include "my_opencv.h"
#include "get_frame_network.h'

int start_view_image();
void stop_view_image();

#endif /* SHOW_IMAGE_H_ */
```

Figura 19: File “show_image.h”

- *start_view_image():* crea un *thread* che visualizza le immagini richiedendole a `get_frame_network.c`. Implementazione della funzione in Figura 20
- *stop_view_image()*: rilascia le risorse utilizzate

Il codice in `show_frame.c` funziona esattamente come il codice in `get_frame_network.c` (già descritto in precedenza), utilizzando un *thread* che ciclicamente richiede un nuovo frame e lo visualizzerà a video. Il codice che implementa questa funzione è molto semplice e riportato in Figura 20.

```
void opencv_app()
{
    while(1)
    {
        get_new_frame(get_IplImage()->imageData);
        show_frame("OV");
    }
}
```

Figura 20: Implementazione “show_image.c”

3.3.3. Il file “main.c” su PC

Il *main*, mostrato in Figura 21 utilizza le API descritte in precedenza e può essere utilizzato come esempio per un loro futuro riutilizzo.

```
#include "index.h"

int main(int argc, char **argv)
{
    start_my_opencv();
    start_udp_application();
    start_view_image();

    //start_tcp_application();

    printf("Premere un tasto per terminare\n");
    getchar();

    stop_view_image();
    stop_udp_application();
    //stop_tcp_application();
    return(0);
}
```

Figura 21: File “main.c”

È molto importante osservare che è necessario richiamare *start_udp_application()* e *start_my_opencv()* prima di *start_view_image(),* in quanto l’ultima funzione andrà ad utilizzare componenti inizializzate dalle altre (Figura 21).

3.4. PROBLEMATICHE RISCONTRATE

Durante lo sviluppo del codice sono emerse situazioni di particolare interesse.

La funzione `htons` e la rispettiva `ntohs` non sono disponibili in formato standard nell'ambiente di sviluppo *standalone*.
Infatti, il valore passato come parametro, è cambiato con uno standard diverso dalle omonime funzioni disponibili su ambienti con sistema operativo. Dopo aver eseguito varie prove, monitorando gli *header* dei pacchetti inviati utilizzando Wireshark [24], sono giunto alla conclusione che non è strettamente necessario utilizzare le due funzioni precedentemente citate durante l'inizializzazione degli indirizzi IP, Figura 12 e 14.

Il progetto iniziale prevedeva l'utilizzo dei *Jumbo frame* [17], frame ethernet con *payload* espandibile fino a 9000 byte, molto più grande dei convenzionali 1500. Questo avrebbe permesso di ridurre il numero di operazioni effettuate con la conseguente riduzione dei tempi di invio e risorse utilizzate.
In LightWaigthIP è data la possibilità di impostare MTU (*maximum transmission unit*) per evitare la frammentazione dei dati su un numero eccessivo di pacchetti. Questa caratteristica non è da confondere con i 64k di dati che rappresentano il massimo trasportabile da un pacchetto IP.
Implementando questa caratteristica ho riscontrato un problema utilizzando le librerie di lwIP, superando il valore di circa 5000byte (ben sotto alla soglia di 9000 rappresentata dai *Jumbo frame*) il processo di invio dei pacchetti fallisce. Allo stesso modo, anche rimanendo sotto a tale soglia e riuscendo in questo modo a inviare i pacchetti, la periferica di rete in ricezione blocca i pacchetti in entrata non passandoli a livello applicativo, poiché marcati come *Malformed Packet*, anche se sono abilitati i *Jumbo frame*.

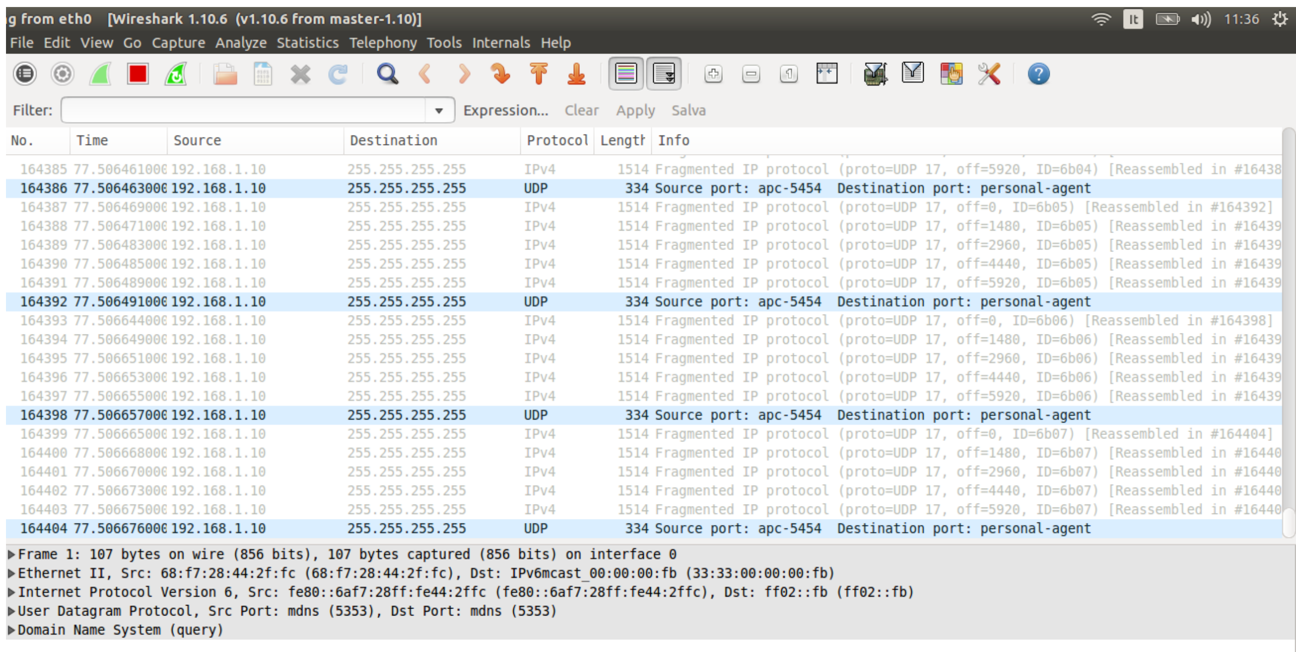

Figura 22: *Screenshot* rilevamento pacchetti con Wireshark [24]

È comunque possibile inviare una quantità di dati superiore a 1500 byte grazie alla frammentazione del protocollo UDP, è stata scelta la dimensione di 7680byte. Durante il passaggio fra i livelli di OSI il nostro pacchetto applicativo è suddiviso in più pacchetti, ognuno con payload massimo i 1500 byte.

Come mostrato in Figura 22 con questa configurazione i pacchetti UDP vengono riconosciuti come *GigE (Gigabit Ethernet)* [13].

È comunque interessante notare che le prestazioni fra diversi sistemi operativi variano enormemente.

In ambiente Linux ho riscontrato prestazioni massime utilizzando un *payload* di 7860 byte.

Mentre in ambiente Windows è stato possibile aumentare questo parametro fino a 30720 byte. Scambiando queste configurazioni abbiamo situazioni ben distinte:

- utilizzando un *payload* di 30720 byte su Linux i pacchetti non vengono nemmeno processati dal client in ricezione
- utilizzando un *payload* di 7680 byte in ambiente Windows si ha una perdita di pacchetti media del 15,7%, mentre con la seconda configurazione si passa a solo il 0,01%

NOTA - INVIO: è necessario allocare un nuovo *pbuf* ad ogni ciclo, risulta inutilizzabile una volta

passato come parametro a una *send*, *sendto* o *sendtoif*.

NOTA - RICEZIONE: il firewall Windows blocca completamente i pacchetti in entrata dell'applicazione. Per un corretto funzionamento è necessario disattivarlo o aggiungere il processo dell'applicazione all'elenco delle eccezioni.

Su entrambi i sistemi operativi, in ricezione, è necessario impostare un indirizzo IP statico che sarà cablato nel codice in esecuzione sullo Zynq/ARM.

Come già accennato in precedenza, risulta problematica la gestione di eventuali pacchetti in entrata sulla ZedBoard sia UDP che TCP. In particolare, quando arriva un pacchetto, è inserito in una coda di entrata in attesa di essere assegnato al *netif* deputato alla sua gestione. Per poter passare il controllo delle risorse a questa parte del codice viene creato un interrupt a livello software che avvisa lwIP ogni volta che la coda in entrata deve essere gestita. Durante lo sviluppo del software non è stato possibile rendere funzionante questo meccanismo se contemporaneamente viene gestito l'*interrupt* relativo al *frame_index*, con la conclusione di avere una coda in entrata che non è mai gestita a livello applicativo.

Possiamo identificare la causa di questo problema in un conflitto nel controllore di interrupt *XGpio*. Per tale ragione, l'implementazione di una connessione TCP per la trasmissione di dati per la configurazione di moduli mappati su FPGA o comandi/dati per il sistema ARM è stata implementata solamente in ambiente Linux, lasciando al sistema *standalone* primitivo solamente l'invio del flusso di dati.

4. ZYNQ e LINUX

Sulla periferica ZedBoard è possibile installare sistemi operativi basati su Linux (Linario, Petalinux ecc). Nel mio caso ho scelto di utilizzare Petalinux [3], una distribuzione sviluppata da Xilinx per i propri dispositivi. Proprio come un normale progetto *standalone* necessita di un BSP (*Board Support Packages*) per essere installato.

Dopo aver configurato la ZedBoard (jumpers) per essere avviata dalla scheda SD, è necessario creare l'immagine di Linux, includere il progetto FPGA e caricarla su una partizione fat32 della scheda SD ricordandosi di configurare il *rootfs* in modo tale che programmi la FPGA durante il *boot* del sistema operativo.

Il compito del *device driver* è mappare kernel del sistema operativo la periferica FPGA, istruendolo delle aree di memoria che utilizza e registrare eventuali interrupt.

L'obiettivo finale è quello di avere un file in /dev/… che rappresenti la periferica, in questo modo può essere trattato come un normale *device* del sistema.

È necessario creare un modulo kernel, il quale si occuperà di tradurre le operazioni ad alto livello (*open, close*, *read* e *write*) che possono essere eseguite sul file su /dev/ov7670, in operazioni di più basso livello eseguibili direttamente sui registri dell'hardware mappato, schema in Figura 23.

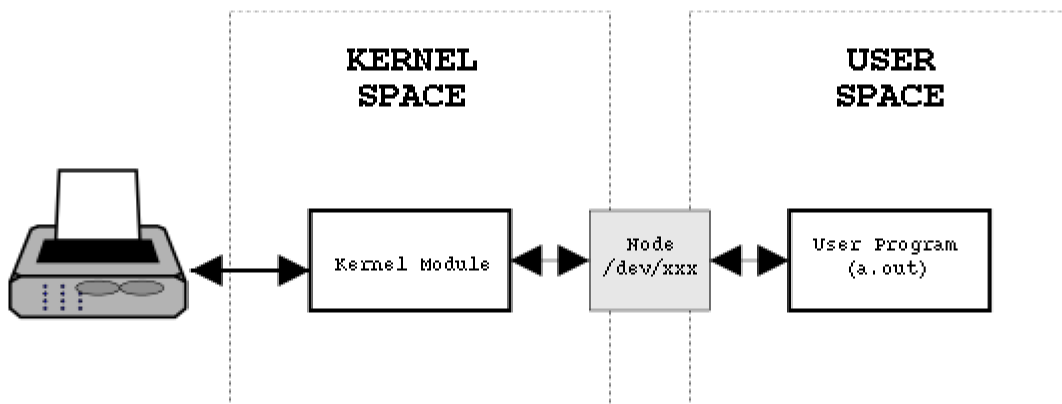

Figura 23: Schema stack chiamate user space

4.1. OPERAZIONI PRELIMINARI

Come accennato in precedenza per poter avviare un sistema operativo da SD bisogna eseguire alcune operazioni preliminari. Come prima cosa è necessario configurare la ZedBoard in metodo opportuno come descritto nel Capitolo 2.

4.1.1. FORMATTAZIONE SD CARD

L'immagine di Linux deve essere copiata in una partizione fat32. La Figura 24 mostra il partizionamento della scheda di memoria.

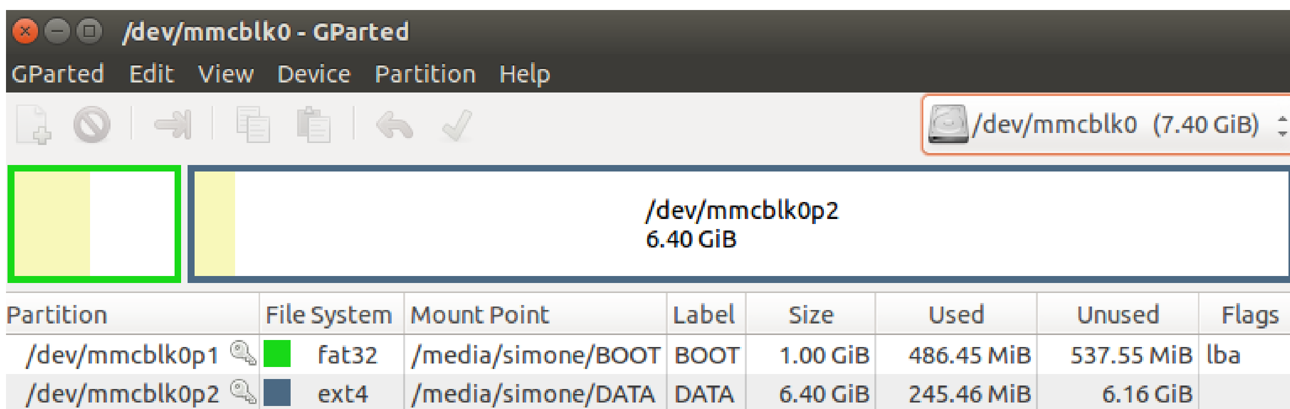

Figura 24: Partizionamento scheda SD

4.1.2. INSTALLAZIONE PETALINUX

Questo progetto di tesi è stato sviluppato sul sistema operativo Ubuntu 14.04.4 LTS [18]. L'utilizzo di questa versione è consigliato per la sua compatibilità con il kernel Petalinux 2014.4 [19]. Prima di poter utilizzare creare l'immagine *boot* eseguibile dalla ZedBoard, Petalinux richiede un processo di installazione come un normale software di sviluppo.
Di seguito sono riportati i passaggi da eseguire per l'installazione del pacchetto Petalinux 2014.4, per maggiori informazioni consultare la guida riportata in [20].

- **Installazione servizi tftp:** questo pacchetto è necessario solamente se si desidera utilizzare i servizi *network tftp* per trasferire l'immagine di sistema di Linux sulla ZedBoard.
  NOTA: Nel caso si scegliesse di non installare questo pacchetto disabilitare o ignorare i messaggi di errore "WARNING: No tftp server found". Inoltre omettere il pacchetto tftp nei comandi seguenti.
  1. Modificare il fine "/etc/xinetd.d/tftp" inserendo le seguenti linee:
     ```
     service tftp
     {
         protocol = udp
         port = 69
         socket_type = dgram
         wait = yes
         user = nobody
         server = /usr/sbin/in.tftpd
         server_args = /tftpdboot
         disable = no
     }
     ```
     In questo file sono descritte le proprietà del servizio *tftp*. Il campo *server_args* è di particolare interesse, il primo argomento indica il *path* in cui saranno copiati tutti i file necessari al corretto funzionamento dell'immagine di boot di Petalinux.

- **Installazione pacchetti e librerie di supporto:** per il suo corretto funzionamento Petalinux utilizza alcune librerie, di seguito sono riportati i comandi da eseguire per installare i pacchetti interessati.
  1. sudo apt-get install tofrodos iproute gawk git-core net-tools
  2. sudo apt-get install ncurses-dev libncurses5-dev xinetd tftpd tftp

3. sudo apt-get install zlib1g-dev flex bison lib32z1 lib32ncurses5
4. sudo apt-get install lib32bz2-1.0 lib32stdc++6 libselinux1 lib32ncursesw5
5. sudo -i
6. cd /etc/apt/sources.list.d
7. echo "deb http://archive.ubuntu.com/ubuntu/ precise main restricted universe multiverse" > ia32-libs-raring.list
8. sudo apt-get install ia32-libs
9. rm ia32-libs-raring.list
10. sudo apt-get update
11. sudo chmod -R 777 /etc/xinetd.d/

NOTA: Eseguire il commando 11 solo nel caso in cui si abbia installato anche il pacchetto *tftp*.

- **Pacchetti opzionali:** in alcuni PC per la corretta esecuzione del comando "*petalinux-build*" è necessario installare manualmente l'architettura i386 necessario durante la creazione dell'immagine di boot di Petalinux.

1. sudo dpkg --add-architecture i386
2. sudo apt-get update
3. sudo apt-get install libbz2-1.0:i386
4. sudo apt-get -y update
5. sudo apt-get install libselinux1:i386

A questo punto è possibile installare Petalinux. I comandi che seguiranno vanno eseguiti da *superuser* (root) per evitare problemi di permessi e autorizzazioni di accesso a file o directory.

1. mkdir Petalinux

   Come prima cosa creare la cartella nel *path* desiderato, per esempio "/opt/Petalinux".
2. ./petalinux-v2014.4-final-installer.run /opt/Petalinux/

   Eseguire il file .run passando come parametro il *path* di installazione.
3. sudo dpkg-reconfigure dash

   Riconfigurare la dash, selezionare **NO** nel menu.
4. Per importare automaticamente i comandi di Petalinux all'apertura di una nuova *bash* modificare il file .bashrc presente nella cartella home utente inserendo la seguente riga:

source /opt/Petalinux/petalinux-v2014.4-final/settings.sh

NOTA: il *path* del commando precedente rappresenta il percorso di installazione.

4.1.3. COMANDI PETALINUX

In questo paragrafo saranno elencati alcuni dei comandi base di Petalinux, come la creazione di un nuovo progetto, l'importazione di una periferica mappata nell'FPGA, inserimento di un nuovo modulo del kernel, configurazione di *rootfs* e *kernel*.

- **Creare nuovo progetto:** per creare un nuovo progetto (immagine del sistema operativo per ZedBoard) Petalinux necessita di una *board support package* (BSP), insieme di librerie e driver che formano lo strato più basso dello *stack* software.
  1. petalinux-create -t project -n name -s bsps/Avnet-Digilent-ZedBoard-v2014.4-final.bsp

- **Compilazione progetto:** per compilare un progetto Petalinux è sufficiente portarsi nella directory base del progetto ed eseguire il comando
  1. petalinux-build.
  Le informazioni dettagliale sul risultato della *build* sono situate nel file /build/buld.log.

- **Opzioni kernel e rootfs:** i due comandi per modificare le opzioni di kernel e rootfs sono rispettivamente:
  1. petalinux-config -c kernel
  2. petalinux-config -c rootft.

  Per abilitare il servizio SSH includere nella configurazione del kernel entrambi le voci del menù: Filesystem Packages -> console/network -> dropbear
  NOTA: per applicare le modifiche è necessario ricompilare il progetto.

- **Importare progetto hardware vivado:** per importare un progetto vivado è necessario ottenere il suo bitstream.
  1. petalinux-config --get-hw-escription=/%path%/design_1_wrapper_hw_platform_0
  Questo comando (1) autogenera anche il *device three* che rappresenta la piattaforma.
  2. source /opt/Xilinx/Vivado/2016.2/settings64.sh
  3. petalinux-package --boot --fsbl images/linux/zynq_fsbl.elf --fpga ./subsystems/linux/hw-description/design_1_wrapper.bit --uboot –force

I comandi 2 e 3 inseriranno il progetto precedentemente importato nel file imagine .elf di Petalinux: Il sistema si occuperà autonomamente di programmare la FPGA al momento dell'accensione della ZedBoard.

- **Creare modulo kernel:** per il corretto funzionamento della periferica occorre una parte di software che "traduca" le funzioni richiamate ad alto livello in una serie di operazioni eseguite sull'hardware. In questo caso il modulo kernel "ov7670" si occuperà del mapping del GPIO (interrupt), gestione delle aree di memoria utilizzate, inizializzazione della parte di network necessaria alla comunicazione via rete e registrazione del device nel sistema operativo.

  1. petalinux-create -t modules -n nome –enable

  Questo comando crea un nuovo modulo kernel utilizzando un template di esempio. L'oprione –enable inserisce il modulo nella build di Petalinux. È possibile scegliere i moduli attivi dalla pagina di configurazione del kernel.

  Per inserire automaticamente il modulo nel kernel al boot di Linux eseguire il comando 2.

  2. $(TARGETINST) -d -a "ov7670" /etc/modules

  NOTA: I moduli attivi saranno automaticamente compilati e inclusi nell'immagine durante l'esecuzione del comando "petalinux-build".

- **Avviare Petalinux utilizzando Putty [21]:** se risulta impossibile collegarsi alla board utilizzando SSH (impostazioni della periferica di rete della ZedBoard non corrette) è possibile aprire un terminale remoto utilizzando Putty. È sufficiente impostare Putty come mostrato in Figura 23 e modificare le configurazioni della periferica di rete.

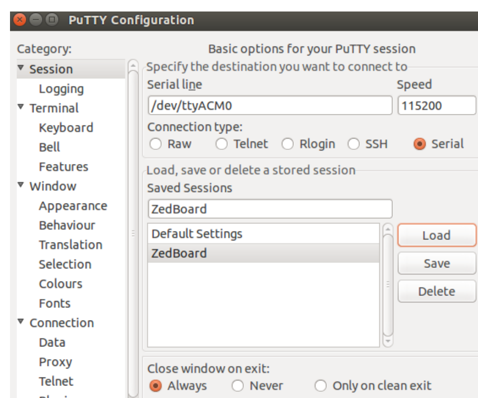

Figura 25: Configurazione Putty

4.2. IMPLEMENTAZIONE INVIO KERNEL-MODE

In questo capitolo sarà illustrata la strategia utilizzata per creare il *device driver*. Nei sistemi operativi Linux il *mapping* delle periferiche nel kernel avviene tramite un match fra due campi **char* denominate classi di compatibilità.
I dispositivi hardware presenti nel sistema sono descritti da un file chiamato *device tree* al cui interno sono contenute le caratteristiche di ciascuna periferica. Per esempio: aree di memoria utilizzate, clock, presenza di interrupt e *interrupt parents* (gestore), classi di compatibilità, altri segnali di input/output, ecc.
È possibile dichiarare nel modulo kernel (software) la lista delle classi di compatibilità con cui è compatibile. Quando il sistema operativo trova un mach fra il campo *compatible* del *device tree* e la lista di compatibilità di un modulo kernel, associa la periferica hardware al modulo software, eseguendo il metodo *probe* (implementato all'interno del modulo kernel). Quest'ultimo ha il compito di inizializzare le risorse necessarie al funzionamento dell'hardware, occupandosi anche della richiesta di specifiche regioni di memoria (*mapping*) e registrazione di eventuali interrupt.

Tuttavia, in questo progetto di tesi, non è stato possibile né identificare un irq (numero rappresentativo di un interrupt) statico né abbinare le classi di compatibilità fra *device tree* e *kernel modul*. Per maggiori informazioni consultare il Capitolo 4.5.

4.2.1. FUNZIONI BASE PER LA GESTIONE DEL MODULO

Una volta che il modulo è attivo e inserito nel kernel di Linux comparirà nel *path* /dev/ov7670. Qualsiasi applicazione *user-space* si interfaccerà a esso come a un normale file di testo. Di seguito saranno mostrate le funzioni base (read, write, open, close) richiamate quando si eseguono le rispettive operazioni sul file /dev/ov7670 e le funzioni (init, exit e probe) richiamate rispettivamente durante l'inserimento, uscita, installazione del modulo nel kernel di Linux.

```
static char cmd_buff[50];
static int ov7670_open(struct inode *inod, struct file *fil);
static ssize_t ov7670_read(struct file *filep, char *buff, size_t len, loff_t *off);
static ssize_t ov7670_write(struct file *filep, const char *buff, size_t len, loff_t *off);
static int ov7670_release(struct inode *inod, struct file *fil);
static void send_udp_frame(int control);

static struct class *driver_class = NULL;
static dev_t first;
static struct cdev c_dev;   //Global variable for character device structure


int local_udp_port = 5454;
int remote_udp_port = 5555;
char * local_udp_ip = "192.168.1.100";
char * remote_udp_ip = "192.168.1.10";

struct socket *sock;
struct sockaddr_in local_udp_addr, remote_udp_addr;

static struct file_operations flops = {
    .read = ov7670_read,    //Function called when reading on device
    .write = ov7670_write,  //Function called when writing on device
    .open = ov7670_open,    //Function called when open the device
    .release = ov7670_release,  //Function called when release the device
};
```

Figura 26: ov7670 funzioni base

Queste funzioni o anche dette *file_operations* (flops) possono essere modificate, la loro assegnazione alle operazioni effettuate sul file avviene attraverso la struttura flops (Figura 26). Di particolare importanza sono le funzioni di *read* e *write* che permettono di scambiare dati fra kernel e applicazione *user space*.

4.2.1.1 FUNZIONE “ov7670_read”

La funzione “ov7670_read” è richiamata ogni volta che si esegue una lettura dal *device file* /dev/ov7670.

```
static ssize_t ov7670_read(struct file *fil, char *buf, size_t len, loff_t *off)
{
    //Copy drom kernel to user space
    printk(KERN_ALERT "Reading ov7670 device: %d\n", len);
    send_udp_frame(1);
    strcpy(cmd_buff, "frame inviato");
    copy_to_user(buf, cmd_buff, strlen(cmd_buff));
    return strlen(cmd_buff);
}
```

Figura 27: Funzione ov7670_read

Con questa configurazione, come mostrato in Figura 27, è inviato un frame a ogni lettura dal *device file*. Questa situazione è temporanea, proposta in seguito al problema in precedenza descritto riguardante l’impossibilità di gestire gli interrupt.

La funzione *copy_tu_user(…)* è una speciale *memcpy(…)* specializzata nel copiare dati da *kernel-space* a *user-space*. La variabile *buf rappresenta il messaggio inviato da *user space* verso il *device*.

4.2.1.2. FUNZIONE “ov7670_write”

La funzione “ov7670_write” è richiamata ogni volta che si esegue una scrittura sul *device file* /dev/ov7670.

```
static ssize_t ov7670_write(struct file *fil, const char *buf, size_t len, loff_t *off)
{
    //Get data from user space to kernel space
    printk(KERN_ALERT "Writing ov7670 device: %d\n", len);
    copy_from_user(cmd_buff, buf, strlen(buf));


    /* Interrupt */
/*
    if (request_irq(135, ov7670_irq, IRQF_DISABLED, DRIVER_NAME, NULL)){
        printk(KERN_ALERT "Not Registered IRQ. \n");
        return -EBUSY;
    }
    printk(KERN_ERR "Registered IRQ. \n");
*/


    /* Creazione struttura da inviare */
    buff = (packet_data *) kmalloc(sizeof(packet_data), GFP_KERNEL);
        if (!lp) {
                printk(KERN_ALERT "Cound not allocate buffer to send\n");
                return -ENOMEM;
        }
```

Figura 28: Prima parte ov7670_write

La funzione *copy_from_user(…)* è la complementare della precedente *copy_to_user(…)*, in questo caso è specializzata nel copiare dati da *user-space* a *kernel-space*. Come mostrato nella Figura 28, nel caso del modulo kernel ov7670 non si occupa solo della comunicazione con applicazioni *user space,* ma per i problemi precedentemente descritti con le classi di compatibilità si occupa anche dell’inizializzazione delle risorse di rete Figura 29.

```
/* Creazione socket */
    if(sock_create(AF_INET, SOCK_DGRAM, IPPROTO_UDP, &sock) < 0)
    {
            printk(KERN_INFO "ov7670 : Errore nella creazione della socket\n");
            goto out;
    }
    printk(KERN_ALERT "Creata la socket\n");


    /* Bind */
    if(sock->ops->bind(sock, (struct sockaddr *) &local_udp_addr, sizeof(struct sockaddr))< 0)
    {
            printk(KERN_INFO "ov7670 : Errore nella bind\n");
            goto out;
    }

    if(sock->ops->connect(sock, (struct sockaddr *) &remote_udp_addr, sizeof(struct sockaddr), 0) < 0)
    {
            printk(KERN_INFO "ov7670 : Errore nella connect\n");
            goto out;
    }
    printk(KERN_ALERT "Bind socket ok, alla porta %i\n", ntohs(local_udp_addr.sin_port));

return strlen(buf);

out:
    sock_release(sock);
    return ENXIO;
```

Figura 29: Seconda parte ov7670_write

La Figura 29 mostra le primitive da invocare per l'inizializzazione di una socket in kernel space. Le librerie utilizzate sono <linux/in.h>, <linux/ip.h>, <linux/netdevice.h> e <linux/errno.h>.

4.2.1.3. STRUTTURA PLATFORM_DRIVER

```
static struct platform_driver ov7670_driver = {
    .probe          = ov7670_probe,
        .remove         = ov7670_remove,
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = ov7670_of_match,
    },
};
```

Figura 30: platform_driver di ov7670

Similarmente alle *flops_operations*, la struttura *platform_driver* contiene i riferimenti alle caratteristiche descrittive del modulo (nome, proprietario e riferimento alla tabella di compatibilità). In questa struttura, Figura 30, vengono anche dichiarati i riferimenti alle funzioni di *probe* e *remove*, richiamate rispettivamente all'inserimento e disinstallazione del modulo.

4.2.1.4. FUNZIONE OV7670_INIT

La funzione *ov7670_init* è richiamata nel momento in cui il modulo viene installato nel kernel, sia che avvenga automaticamente al boot del sistema operativo o con *insmod*.

```
static int __init ov7670_init(void)
{
    printk("ov7670 module loaded.\n");

    //Register a range of char device numbers
    if(alloc_chrdev_region(&first, 0, 11, "Simone") < 0){
        return -1;
    }

    driver_class = class_create(THIS_MODULE, CLASS_NAME);
    if(driver_class == NULL){
        printk(KERN_ALERT "Create class falied\n");
        unregister_chrdev_region(first, 1);
        return -1;
    }

    if(device_create(driver_class, NULL, first, NULL, "ov7670") == NULL){
        printk(KERN_ALERT "Create device falied\n");
        class_destroy(driver_class);
        unregister_chrdev_region(first, 1);
        return -1;
    }

    cdev_init(&c_dev, &flops);
    if(cdev_add(&c_dev, first, 1) == -1){
        printk(KERN_ALERT "Create character device falied\n");
        device_destroy(driver_class, first);
        class_destroy(driver_class);
        unregister_chrdev_region(first, 1);
        return -1;
    }
```

Figura 31: Prima parte ov7670_init

La Figura 31 mostra i passaggi necessari a automatizzare la creazione del file /dev/ov7670, dichiarando le risorse che verranno utilizzate dal modulo e dal kernel durante il funzionamento. In questo metodo vengono anche richieste le aree di memoria a indirizzi fisici utilizzate dalla periferica, proteggendole da altri accessi. La sola entità autorizzata ad accedere e gestire queste risorse è il modulo kernel che le ha richieste.

```
/* GPIO */
if (!request_mem_region(lp->mem_start, lp->mem_end - lp->mem_start - 1, DRIVER_NAME)){
    printk(KERN_ALERT "ov7670 GPIO Couldn't lock memory region at %p\n", (void *)lp->mem_start);
    rc = -EBUSY;
    goto error1;
}

lp->base_addr = ioremap(lp->mem_start, lp->mem_end - lp->mem_start - 1);
if (!lp->base_addr) {
    printk(KERN_ALERT "ov7670 GPIO: Could not allocate iomem\n");
    rc = -EIO;
    goto error2;
}
```

Figura 32: mapping memoria GPIO

La funzione *request_mem_region(...)* si occupa di richiedere un'area di memoria dello spazio di indirizzamento del processore (in cui è presente la periferica che vogliamo mappare) Figura 32. La funzione *ioremap(...)* rimappa l'indirizzo fisico utilizzato dalla *request_mem_region(...)* restituendo l'indirizzo logico di partenza del blocco di memoria rimappato.

```
/* PL_RESET */
if (!request_mem_region(lp->pl_reset_start, lp->pl_reset_end - lp->pl_reset_start, DRIVER_NAME)){
            printk(KERN_ALERT "ov7670 pl_reset  Couldn't lock memory region at %p\n",
        (void *) lp->pl_reset_start);
            rc = -EBUSY;
            goto error1;
    }

    lp->pl_base_addr = ioremap(lp->pl_reset_start, lp->pl_reset_end - lp->pl_reset_start);
    if (!lp->pl_base_addr) {
            printk(KERN_ALERT "ov7670 pl_reset: Could not allocate iomem\n");
            rc = -EIO;
            goto error2;
    }
printk(KERN ALERT "ov7670 at 0x%08x mapped to 0x%08x, pl reset at 0x%08x mapped to 0x%08x\n",
```

Figura 33: mapping memoria pl_reset

In Figura 32 e Figura 33 sono mostrati i *mapping* in memoria dei due GPIO presenti nel progetto Vivado [23] utilizzato in questa tesi.

```
/* FRAME BUFFER */
buffer_start_address = phys_to_virt(FRAME_BUFFER_BASE_ADDR);
if (buffer_start_address == NULL)
{
        printk(KERN_ALERT "Errore durante la phys_to_virt\n");
    goto error2;
}
else
    printk(KERN_ALERT "Frame Buffer mappato a 0x%08x\nMapping della mem
    (unsigned int __force) buffer_start_address);
```

Figura 34: phys_to_virt address

Il *frame_buffer*, trovandosi in un'area di indirizzamento già utilizatta dal device DDR, non è possibile richiederla utilizzando una *req_mem_region(...)*. Utilizzando *phys_to_virt()* è possibile recuperare l'indirizzo logico corrispondente a un determinato indirizzo fisico, Figura 34.

4.2.1.5. FUNZIONE OV7670_EXIT

La funzione *ov7670_exit* è richiamata nel momento in cui il modulo viene disinstallato dal kernel. Il suo compito è di rilasciare tutte le risorse utilizzate, liberando lo spazio di indirizzamento della periferica, Figura 35.

```
static void __exit ov7670_exit(void)
{
    free_irq(lp->irq, lp);
    free_irq(lp->irq_pl, lp);
    release_mem_region(lp->mem_start, lp->mem_end - lp->mem_start - 1);
    release_mem_region(lp->pl_reset_start, lp->pl_reset_end - lp->pl_reset_start);
    kfree(lp);

    cdev_del(&c_dev);
    device_destroy(driver_class, first);
    class_destroy(driver_class);
    printk(KERN_ALERT "ov7670 unloaded.\n");
}
```

Figura 35: Funzione ov7670_exit

4.2.2. INTERRUPT IN LINUX

Nel sistema operativo Linux gli Interrupt sono rappresentati da un numero intero. Quest'ultimo può essere assegnato automaticamente dal kernel oppure, per alcune configurazioni hardware, assegnato staticamente. La funzione *request_irq(...)* permette di registrare una funzione di *callback* a un determinato *irq* (numero rappresentativo di in interrupt).

```
if (request_irq(135, ov7670_irq, IRQF_DISABLED, DRIVER_NAME, NULL)){
    printk(KERN_ALERT "Not Registered IRQ. \n");
    return -EBUSY;
}
printk(KERN_ERR "Registered IRQ. \n");
```

Figura 36: esempio request_irq

La nomenclatura completa è mostrata in Figura 36, il numero 135 rappresenta *irq* al quale ci si vuole registrare. L'*interrupt handler* è o*v7670_irq.*

4.2.3. INVIO FRAME CON UDP KERNEL SOCKET

Al contrario delle *user space* socket, le *kernel sock* necessitano dell'impostazione di alcuni campi dell'header del pacchetto UDP, Figura 37.

```
struct iovec iov;
mm_segment_t oldfs;

static void send_udp_frame(int control)
{
    frame_index =  *(lp->base_addr);
    flush_icache_range(buffer_start_address, buffer_start_address + FRAME_BUFFER_SIZE);
    for(i = 0; i < FRAME_SIZE / FRAME_UDP_FRAGMENT_SIZE; i++)
    {
        buff->count = counter++;
        buff->fragment = i;
        buff->frame_index = frame_index;

        memcpy(buff->data, (buffer_start_address+(frame_index*FRAME_SIZE/4)+(i*FRAME_UDP_FRAGMENT_SIZE/4)), FRAME_UDP_FRAGMENT_SIZE);

        iov.iov_base = buff;
            iov.iov_len = sizeof(packet_data);

            msg.msg_flags = 0;
            msg.msg_name = &remote_udp_addr;
            msg.msg_namelen  = sizeof(struct sockaddr_in);
            msg.msg_control = NULL;
            msg.msg_controllen = 0;
            msg.msg_iov = &iov;
            msg.msg_iovlen = 1;
            msg.msg_control = NULL;

            oldfs = get_fs();
            set_fs(KERNEL_DS);
            sock_sendmsg(sock,&msg,sizeof(packet_data));
            set_fs(oldfs);
    }
}
```

Figura 37: Funzione send_udp_frame(…)

Figura 37 riporta il codice della funzione del modulo kernel che invia un singolo frame utilizzando il protocollo UDP. La definizione delle costanti utilizzate si può trovare nel file `constant.h`.

4.3. IMPLEMENTAZIONE ALTERNATIVA USER SPACE

In questo capitolo è descritta una versione alternativa al modulo kernel. In questo caso, restando in user space, verrà implementato lo straming UDP e la connessione sicura TCP senza preoccuparsi di gestire l'interrupt. Per rilevare la riscrittura di un nuovo frame in memoria si controlla a loop la variazione del *frame_index*, accedendo alle aree di memoria fisica attraverso la funzione *mmap(...)* [22]. Per il debug è stata utilizzata l'opzione TCF Agent, nella documentazione [25] da pagina 82 in poi è presente una guida per il suo utilizzo.

4.3.1. File "constant.h" e "main.c"

Come negli altri casi, nel file `constant.h`, sono dichiarate tutte le costanti e strutture utilizzate dall'applicazione. Il suo contenuto è identico all'omonimo file del progetto *standalone*.

```
int main()
{

    start_udp_application();
    start_tcp_application();

    printf("Premere un tasto per terminare\n");
    getchar();

    stop_udp_application();
    stop_tcp_application();
    return 0;

}
```

Figura 38: implementazione "main.c"

L'implementazione del file `main.c`, Figura 38, mostra come richiamare i file del progetto. Le due funzioni *start_udp_application()* e *start_tcp_application()* creeranno due *pthread* assegnando al primo il compito dell'invio dei frame e al secondo l'inizializzazione di una connessione TCP con il PC a cui è collegata la ZedBoard.

4.3.2. File “udp_app.c”

```
int start_udp_application()
{
    iret_udp = pthread_create(&udp_thread, NULL, udp_app, NULL);
    if(iret_udp)
    {
        printf("Error - pthread_create() return code: %d\n",iret_udp);
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Figura 39: Funzione start_udp_application()

Per avviare l’invio del flusso di immagine è sufficiente richiamare questa funzione, Figura 39, il *pthread* creato si occuperà anche dell’inizializzazione delle risorse di rete utilizzate (stesso procedimento usato per la ricezione UDP).

```
fd = open("/dev/mem", O_RDWR | O_SYNC);
frame_index = mmap(NULL,4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, FRAME_INDEX_BASE_ADDR & ~(4096-1));
if (frame_index == MAP_FAILED)
            perror ("Errore durante la mmap del frame_index!\n");
    else
        printf("Mapping frame_index avvenuto correttamente!\n");

buffer_start_address = mmap(NULL,FRAME_BUFFER_DIM*FRAME_BUFFER_NUM, PROT_READ | PROT_WRITE, MAP_SHARED, fd, FRAME_BUFFER_BASE_ADDR & ~(FRAME_BUFFER_DIM*FRA
if (buffer_start_address == MAP_FAILED)
        perror ("Errore durante la mmap!\n");
else
    printf("Mapping della memoria avvenuto correttamente!\n");
buffer_start_address = (buffer_start_address + (FRAME_BUFFER_BASE_ADDR & (FRAME_BUFFER_DIM*FRAME_BUFFER_NUM-1)));
buff = (packet_data *) malloc (sizeof(packet_data));
counter = 0;
old_frame_index = 0;
printf("Pronto per inviare frame...\n");

while(1)
{
    while(old_frame_index == *frame_index);
    old_frame_index = *frame_index;
    sand_frame(old_frame_index);
    //printf("Frame: %d inviato \n", old_frame_index);
    //usleep(10000);
}
```

Figura 40: mmap e loop controllo frame_index

Per accedere a indirizzi di memoria di cui conosciamo solo il *physical address* è utilizzata la funzione *mmap* in combinazione con il *file descriptor* del device /dev/mem, il quale contiene il mapping fra indirizzi logici e fisici. In Figura 40 è mostrato come viene utilizzata la *sistem call mmap*. Viste le difficoltà ad agganciarsi all’n*iterrupt*, nella seconda parte viene implementato un ciclo in cui si controlla, a polling, la variazione del *frame_index*. Ogni volta che il *frame_index* del frame appena inviato è diverso dal corrente verrà richiamata la funzione *send_frame(…)* Figura 41 che si occuperà di inviare il frame identificato dal *frame_index* passato come parametro.

```
void sand_frame(int frame_index)
{
    for(i = 0; i < FRAME_SIZE / FRAME_UDP_FRAGMENT_SIZE; i++)
    {
        buff->count = counter++;
        buff->fragment = i;
        buff->frame_index = frame_index;

        memcpy(buff->data, (buffer_start_address+(frame_index*FRAME_BUFFER_DIM/4)+(i*FRAME_UDP_FRAGMENT_SIZE/4)), FRAME_UDP_FRAGMENT_SIZE);
        len = sizeof(remote_udp_addr);
        if(sendto(sd, buff, sizeof(packet_data), 0, (struct sockaddr *)&remote_udp_addr, len)<0)
            perror("Errore nella sandto!\n");

    }

    //xil_printf("Inviato frame %d\n", frame_index);
}
```

Figura 41: Funzione send_frame

4.3.3. File “tcp_app.c”

Nel file `tcp_app.c` viene implementata la connessione TCP. Proprio come nel caso di `udp_app.c` è sufficiente richiamare la funzione *start_tcp_application()* in Figura 42.

```
int start_tcp_application()
{
    iret_tcp = pthread_create(&tcp_thread, NULL, tcp_app, NULL);
    if(iret_tcp)
    {
        printf("Error - pthread_create() return code: %d\n",iret_tcp);
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Figura 42: Funzione start_tcp_application()

Essa si occuperà di inizializzare tutte le risorse utilizzate, lasciando l’implementazione dell’invio/ricezione di dati a una futura implementazione.

```
/* SELECT */
len = sizeof(struct sockaddr_in);
while((connsd = accept(tcp_sd, (struct sockaddr *) &remote_tcp_addr, &len)))
{
    if(connsd < 0 )
    {
        perror("connect\n");
        continue;
    }
    printf("Gestisco la richiesta TCP\n");
    /* GESTISTO LA RICHIESTA */
    int i;
    read(connsd, &i, sizeof(int));
    printf("ricevuto %d\n", i);
    char * risp = "ricev";
    if(write(connsd, risp, 6)<0)
        perror("write");
    sleep(1);
    printf("Inviato");
```

Figura 43: Implementazione proposta echo server

L’implementazione TCP proposta, Figura 43, è un server echo, in cui a ogni ricezione di un messaggio viene inviata la stringa “ricev” al mittente.

L’uso di questa connessione è riservata a scopi futuri da implementare, il codice attuale ha la sola funzione di esempio e test.

4.4. IMPLEMENTAZIONE RICEZIONE

La ricezione è implementata su PC, esattamente come nel caso del sistema *standalone*, con in più l'introduzione del modulo TCP, che si occupa della creazione di una connessione affidabile con la ZedBoard per inviare e ricevere dati che richiedono la garanzia di ricezione.

4.4.1. File "my_tcp_app.h"

Nel file `my_tcp_app.h` sono fornite alcune funzioni che permettono di aprire una connessione TCP con la ZedBoard, fornendo una struttura che compie una scrittura e lettura dalla connessione per dimostrare il suo funzionamento.

```
/*      Mingarelli Simone
 *      0000654082
 *      my_tcp_app.h
 */

#ifndef MY_TCP_APP_H_
#define MY_TCP_APP_H_

#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/types.h>
#include <signal.h>
#include <arpa/inet.h>
#include <pthread.h>
#include "constant.h"

int start_tcp_application();
void stop_tcp_application();
```

Figura 44: File "my_tcp_app.h"

Come possiamo notare dalla Figura 44, che mostra solo il contenuto del file *header* abbiamo a disposizione solo due funzioni:

- *start_tcp_application():* al suo interno è creato un *pthread* al quale viene dato il compito di instaurare una connessione TCP con la ZedBoard
- *stop_tcp_application():* se richiamata termina forzatamente il *pthread* assegnato alla

gestione della connessione TCP, preoccupandosi di de-allocare e liberare eventuali risorse utilizzate.

All'interno del file `my_tcp_app.c` corrispondente possiamo trovare l'implementazione delle funzioni:

```
        }
        printf("Connect ok\n");

        int i = 1;
        char * rt[6];
        if (write(tcp_sd, &i, sizeof(int))<0)
        {
                perror("write");
        }

        if (read(tcp_sd, rt, 6)<0)
        {
                perror("read");
        }

        printf("%s\n", rt);

}
..
```

Figura 45: File “my_tcp_app.c”

Il blocco di codice in Figura 45 mostra con una semplice lettura e scrittura come inviare dati utilizzando TCP. I dati da trasmettere sono dipendenti dal tipo di necessità e potrebbero consistere in tabelle, comandi per il sistema ARM o eventuali parametri per la configurazione delle periferiche mappate sulla rete FPGA.

4.5. PROBLEMATICHE

Durante l'implementazione del modulo kernel si sono riscontrate criticità nel match fra classi di compatibilità. Nonostante le stringhe identificative contenute nel *device tree* e nella *compatible_list* (struttura contenuta nel codice del modulo kernel nella quale sono elencate tutte le classi di compatibilità del modulo) coincidessero, durante l'esecuzione di *insmod* il kernel non considerava compatibili le due componenti, non eseguendo il metodo *probe*.

Inoltre importando l'*hardware description* del progetto Vivado (eseguendo i comandi descritti nel Paragrafo 4.2.3) non viene specificato il numero identificativo dell*'interrupt*. In questo modo al posto di un numero costante da passare come parametro alla *request_irq(...) descritta nel Paragrafo 4.3.2, irq corrispondente all'interrupt viene assegnato dinamicamente dal kernel al boot di Linux.*

Provando a "cablare" nel codice l'irq, come in Figura 36, non è stato possibile intercettare l'*interrupt*.

5. RISULTATI SPERIMENTALI E CONCLUSIONI

Eseguendo misurazioni con l'utilizzo di timer inseriti nel codice è stato possibile calcolare i tempi di invio dei frame, stimando le prestazioni massime raggiungibili con queste configurazioni.

Caso Standalone:
Tempo di invio medio: 4.252 ms.
Tempo di invio massimo: 4.342 ms.

Caso Linux con soluzione alternativa:
Tempo di invio medio: 5.532 ms.
Tempo di invio massimo: 5.619 ms.

Questi dati sono ottenuti misurando la durata dell'*interrupt handler* avendo una cadenza di *interrupts* a 30 FPS (Frame Per Secondo), inviando *frame* di dimensione 640x480 della dimensione di 307200 Byte.
Sulla base di queste misurazioni, utilizzando il tempo massimo rilevato è possibile stimare, approssimando per eccesso, la quantità massima di dati inviabile dalla ZedBoard in 1 sec.

Standalone:
Bitrate massimo: 552 Mbits
FPS massimo: 230 FPS

Linux:
Bitrate massimo: 427 Mbits
FPS massimo: 177 FPS

Questi dati sono approssimati, in quanto ottenuti sulla base del tempo massimo rilevato misurando il tempo impiegato per eseguire l'intero *interrupt handler*. Questo implica un *overhead* di operazioni effettuate prima del vero e proprio invio dei dati che in realtà non si eseguirebbero se aumentasse la dimensione del frame.
Queste misurazioni sono state eseguite collegando direttamente con un cavo ethernet la ZedBoard a

un PC, utilizzando i software precedentemente descritti.

Nelle immagini seguenti, Figura 46 e Figura 47, sono riportati due *screenshot* del risultato ottenuto in modalità Standalone e Linux.

Figura 46: frame caso Standalone

Figura 47: frame caso Linux

Bibliografia

[1] http://opencv.org/

[2] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[3] http://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[4] http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[5] http://www.avnet.com/en-us/Pages/default.aspx

[6] http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf

[7] http://www.wiki.xilinx.com/Standalone+Drivers+and+Libraries

[8] http://savannah.nongnu.org/projects/lwip/

[9] http://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html

[10] https://msdn.microsoft.com/en-us/library/windows/desktop/ms740632(v=vs.85).aspx

[11] https://computing.llnl.gov/tutorials/pthreads/

[12] http://www.nongnu.org/lwip/

[13] https://en.wikipedia.org/wiki/Gigabit_Ethernet

[14] http://www.xilinx.com/

[15] http://www.xilinx.com/products/design-tools/embedded-software/sdk.html

[16] http://linuxcommand.org/man_pages/ssh1.html

[17] https://en.wikipedia.org/wiki/Jumbo_frame

[18] http://releases.ubuntu.com/14.04/

[19]
http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/2014-4.html

[20] http://www.xilinx.com/support/documentation/sw_manuals/petalinux2014_2/ug976-petalinux-installation.pdf

[21] http://www.putty.org/

[22] http://man7.org/linux/man-pages/man2/mmap.2.html

[23] http://www.xilinx.com/products/design-tools/vivado.html

[24] https://www.wireshark.org/

[25] http://www.xilinx.com/support/documentation/sw_manuals/petalinux2015_4/ug1144-petalinux-tools-reference-guide.pdf

[26] Sistema di visione stereo su architettura Zynq