

Documentazione Sistemi Embedded

Giorgio Farina
M63000861

Luca Giamattei
M63000825

Gabriele Previtera
M63000834

Indice

1 STM32: USART SMARTCARD	4
1.1 Introduzione	4
1.2 ISO 7816	4
1.3 Protocollo di trasmissione	5
1.4 Timing	7
1.5 BOARD e SETUP	8
1.6 HAL SMARTCARD	9
1.6.1 Sleep mode	12
2 STM32: USB	13
2.1 Introduzione	13
2.2 Configurazione Progetto	14
2.3 Codice	14
2.4 Uso di scanf e printf	17
2.5 Esempio d'uso della VCP	18
3 STM32: CAN	20
3.1 Introduzione	20
3.2 Driver con libreria HAL	20
3.3 Driver senza libreria HAL	21
3.3.1 Funzionamento libreria	21
3.3.2 Testing della libreria	25
4 Compilare e installare Linux su Zybo	27
4.1 Prerequisiti	27
4.1.1 Download dei file necessari	28
4.1.2 Configurare il build-environment	29
4.2 BOOT	29
4.2.1 FSBL	30
4.2.2 Bitstream	31
4.2.3 U-Boot	31

4.2.4	Realizzazione del file BOOT.bin	32
4.3	Kernel	34
4.3.1	Configurazione	34
4.3.2	Compilazione del kernel	35
4.4	DeviceTree	35
4.4.1	Modificare i bootargs	36
4.4.2	Specificare il driver da utilizzare per il controllo delle periferiche costum	37
4.4.3	Utilizzare le interruzioni	37
4.4.4	Compilare il dtb	37
4.5	Root FS	38
4.6	Preparare la microSD	39
4.7	Configurazione aggiuntiva	40
4.7.1	Aggiunta dei nuovi pacchetti a buildroot	41
4.7.2	Uso di gdbserver	45
5	GPIO	46
5.1	Introduzione	46
5.2	Hardware	47
5.2.1	Componente base	47
5.2.2	Interrupt Generator and Handler	47
5.2.3	AXI Wrapper	48
5.2.4	Design	52
5.3	Software	52
5.3.1	Libreria	53
5.3.2	Bare metal	55
5.3.3	No driver	55
5.3.4	UIO	56
5.3.5	Driver a livello kernel	57
6	UART	58
6.1	Introduzione	58
6.2	Hardware	58
6.2.1	Componente base	58
6.2.2	AXI wrapper	60
6.2.3	Design	62
6.3	Software	63
6.3.1	Libreria	63
6.3.2	Bare Metal	64
6.3.3	No Driver	65
6.3.4	UIO	66

Indice

6.3.5	Modulo Kernel	67
7	YOCTO	69
7.1	Introduzione	69
7.2	OpenEmbedded	69
7.2.1	Architettura	70
7.2.2	Workflow	75
7.3	Jumpstarting	77
7.3.1	Requisiti	77
7.3.2	Layer Xilinx	78
7.3.3	Aggiungere sorgenti personalizzati	78
7.3.4	Configurazione e compilazione	79
7.3.5	Caricamento sulla board	80
7.4	Aggiungere ulteriori pacchetti utilizzando layer esistenti	81

Capitolo 1

STM32: USART SMARTCARD

1.1 Introduzione

In questo breve capitolo si mostrerà un esempio di implementazione del protocollo riferito allo standard ISO 7816 per carte di identificazione a contatto. In particolare ci si riferirà ad una specifica board e se ne descriveranno le modalità di utilizzo a supporto del protocollo. Infine si mostrerà un esempio di utilizzo sviluppato tramite la libreria HAL messa a disposizione per l'interfacciamento e l'utilizzo del SoC.

1.2 ISO 7816

L'ISO 7816 come detto è uno standard relativo a carte di identificazione elettroniche, specialmente le smartcard. Lo standard prevede una interfaccia composta da 8 contatti elettronici mostrati in [Figura 1.1](#).

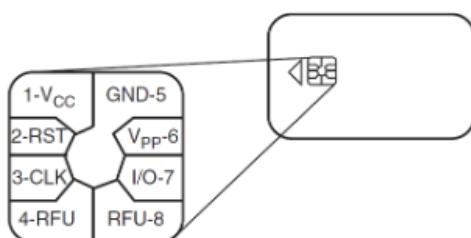


Figura 1.1: Contatti Smartcard

I pin mostrati in figura verranno connessi ad un lettore con interfaccia duale prima di avviare la trasmissione seriale. Tuttavia è possibile prevedere, senza l'ausilio di componenti aggiuntivi, un collegamento diretto ad una board rispettando i vincoli mostrati in [Figura 1.2](#) (Soluzione non consigliata).

STM32 pin / function	Smartcard pin	Function
USART_CK	CLK	Smartcard clock
USART_TX	IO	IO serial data: open drain
Any GPIO	RST	Reset to card
Any GPIO	V _{CC}	Supply voltage
Any GPIO	V _{PP}	Programming voltage

Figura 1.2: Tabella mapping

Secondo il protocollo, il modulo UART sulla board in modalità smartcard lavorerà da master su un canale Half duplex single wired. Sarà lui a iniziare la comunicazione, oltre che essere responsabile per l'alimentazione, il clocking e il reset della smartcard. Inoltre, come si può vedere dalla tabella, la programmazione dei bit GPIO in modalità open-drain collegherà il segnale dati USART_TX al pin Smartcard_IO e il generatore di clock al pin Smartcard_CLK configurato in push-pull.

1.3 Protocollo di trasmissione

Lo standard prevede che il protocollo si divida principalmente in due fasi:

- **Fase di setup:** questa fase è necessaria per effettuare una prima comunicazione tra board e smartcard e viene effettuata a frequenza e voltaggio standard. Nello specifico osservando la [Figura 1.3](#) è possibile osservare che, una volta che i pin sono messi in contatto, viene inviato verso la smartcard un segnale di reset (cold reset). A questo segnale la smartcard risponderà (Answer To Reset ATS) con una sequenza di Byte contenente tutte le informazioni necessarie per la futura comunicazione. Da qui può seguire una negoziazione dei parametri.
- **Fase di Comunicazione:** Per questa fase sono previsti due protocolli diversi. Il protocollo **T=0**, il più semplice dei due, che prevede un semplice set di comandi da poter inviare alla smartcard, dalla quale si riceveranno le relative risposte. Il protocollo **T=1** invece aggiunge complessità alla comunicazione aggiungendo più livelli di astrazione

(fisico, datalink, applicativo), consentendo però lo scambio di interi blocchi di informazioni con prologo, dati ed epilogo. Questa strutturazione del pacchetto permette la comunicazione tra più entità, alle quali può essere assegnato un identificativo, specificando nel prologo mittente e destinatario.

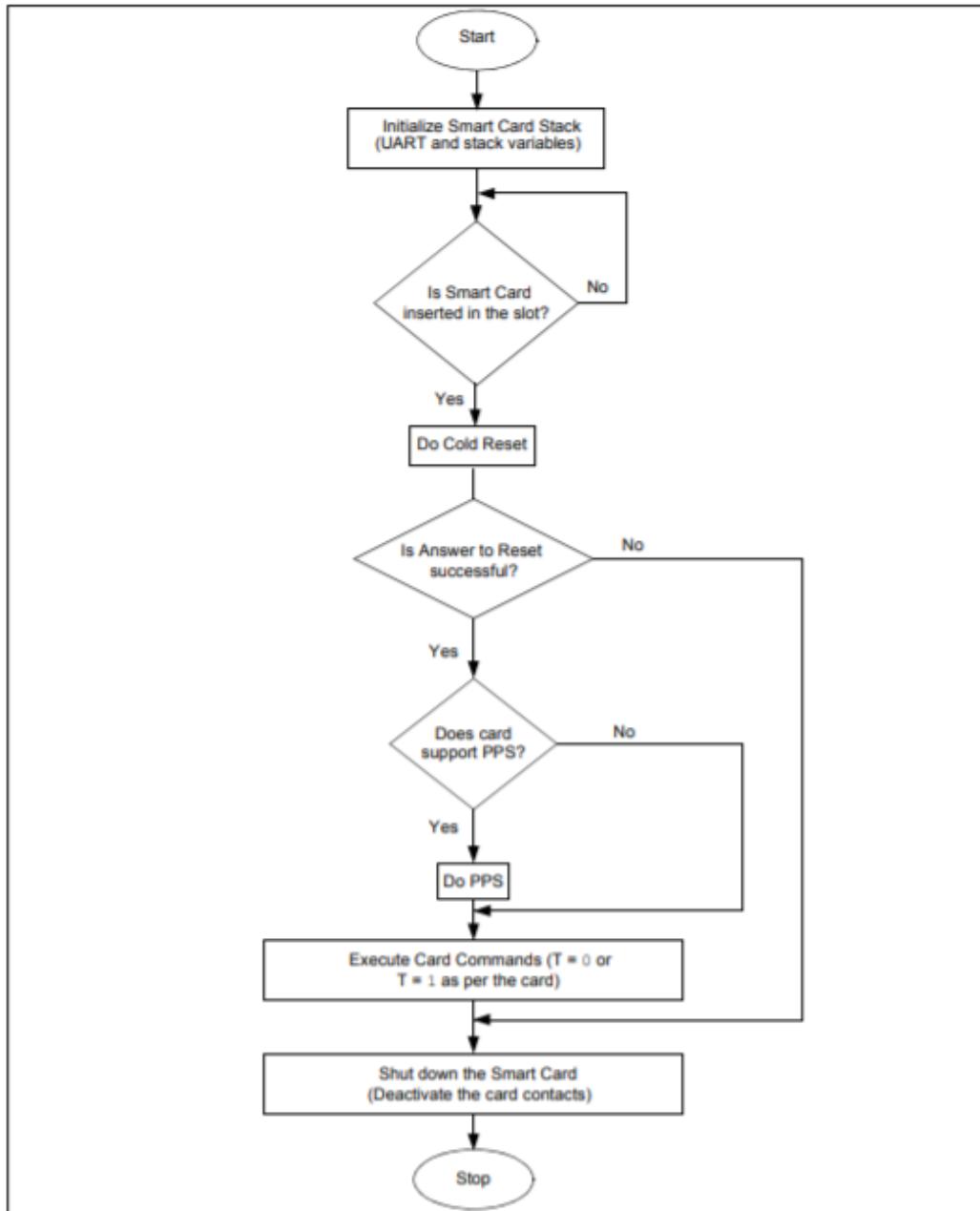


Figura 1.3: Software Flow

1.4 Timing

Il timing gioca un ruolo fondamentale nel funzionamento del protocollo. Il bit time è definito in termini di ETU (Elementary Time Unit). Come accennato precedentemente sarà la board a fornire il segnale di clock alla smartcard e questo sarà utilizzato per ricavare il Baud Rate per l'I/O seriale. Vengono definiti due Guard Time:

- **CGT (CHARACTER GUARD TIME)** è definito sia per T=0 sia per T=1 come il minimo ritardo tra i leading edges di due consecutivi caratteri nella stessa direzione di trasmissione. Durante l'ATR il CGT vale 12 ETU. Dopo l'ATR, il valore di CGT viene calcolato utilizzando il carattere TC1, che è uno dei byte ATR ricevuto dalla smart card.
- **BGT (BLOCK GUARD TIME)** esiste solo per T=1, è definito come il minimo ritardo tra i due leading edges di due caratteri consecutivi nella direzione opposta. Il BGT ha un valore standard di 22 ETU.

Inoltre si definisce il **Waiting Time (WT)** come il massimo ritardo tra due leading edge di un carattere trasmesso dalla card e il leading edge di precedente carattere (trasmesso dalla card o dal interface device). Questo viene utilizzato principalmente per rilevare una unresponsive card.

Di conseguenza è possibile definire il **CWT (Charachter Waiting Time)** ed il **BWT(Block Wait Time)**. Il CWT ([Figura 1.4](#)) è il massimo delay tra due leading edges di due caratteri consecutivi. Il delay **minimo** è il CGT. Il BWT ([Figura 1.5](#)) è il massimo delay tra il leading edge dell'ultimo carattere del blocco inviato alla smartcard e il leading edge del primo carattere del blocco inviato dalla smartcard. Come accennato questo viene utilizzato per valutare la unresponsivness di una smartcard. Questi parametri vengono calcolati con i valori negoziati in fase di setup.

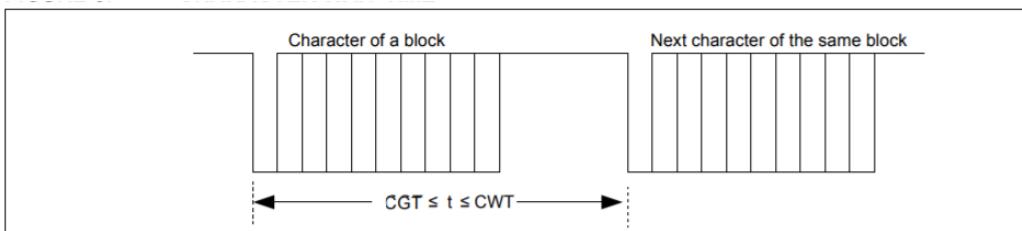


Figura 1.4: CWT

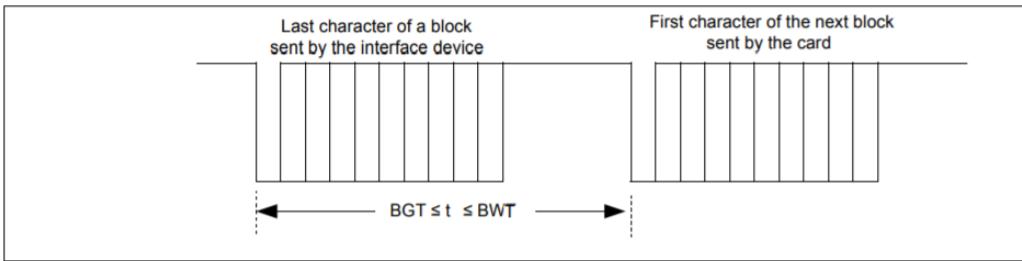


Figura 1.5: BWT

1.5 BOARD e SETUP

Nell'esempio sviluppato, si è deciso di utilizzare due board STM32F4, non avendo a disposizione tutto il materiale necessario ad interfacciarsi con una smartcard. Per questo motivo non è stato possibile implementare la trasmissione prevista dal protocollo. Tuttavia è stato comunque possibile utilizzare la modalità USART SMARTCARD messa a disposizione per la board ed effettuare una comunicazione seriale utilizzando questa modalità. In particolare, la board fornisce (con l'UART in modalità smartcard) la possibilità di impostare il **registro di guardtime (USART_GTPR)** attraverso il quale è possibile ritardare l'asserzione del TC flag. Mentre in UART il registro TC è asserito quando lo shift register di trasmissione è vuoto e non ci sono ulteriori richieste in trasmissione, in modalità Smartcard uno shift register di trasmissione vuoto triggerà un contatore guard time, il quale conterrà fino al valore di conteggio programmato nel guard time register. TC è forzato a low durante questo periodo e viene asserito allo scadere del guard time. In aggiunta al registro di guard time, tenendo presente che il canale è half duplex e che l'UART quando trasmette è anche in ricezione sul canale, allora se dovesse essere rilevato dal receiver (la smartcard) un parity error quest'ultimo potrà mandare un NACK abbassando la linea prima del concludersi dello stop bit (per 1 o 2 baud clock). L'UART essendo anche esso in ascolto sul canale appena vedrà l'abbassarsi della linea durante la lettura dello stop bit interpreterà tale condizione come un framing error. Il meccanismo è illustrato in [Figura 1.6](#).

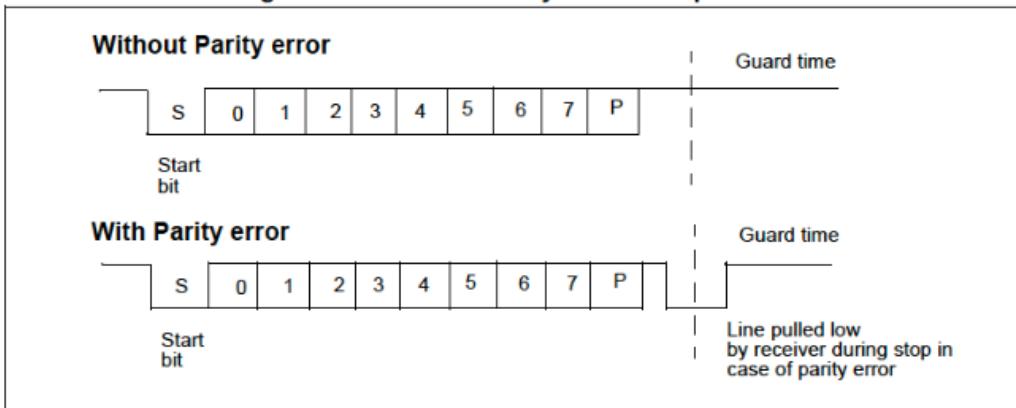


Figura 1.6: Parity/Frame error

1.6 HAL SMARTCARD

Con l'ausilio della libreria HAL è stato possibile sviluppare un piccolo esempio di utilizzo, di cui si è brevemente parlato in precedenza. In particolare si è deciso di utilizzare il paradigma di programmazione condizionale per far sì che un unico file di codice possa essere corretto sia per la board trasmittente che ricevente. Infatti per individuare la board che trasmette sarà sufficiente definire la macro **TRANSMITTER_BOARD**, o commentare la define nel caso opposto. Nell'esempio si è deciso di utilizzare la trasmissione e la ricezione con interruzioni. Per completezza e comprensione dell'esempio si riporta di seguito il readme che è possibile trovare nella directory dell'esempio.

Questo esempio descrive la trasmissione UART in modalità Smartcard tra due boards

All'inizio del main program l'HAL-Init() è chiamata per resettare tutte le periferiche, inizializzare la flash interface, e Systick.

La funzione SystemClock::config() è usata per configurare il clock del sistema /SYSCLK) per eseguire a 25 MHz.

La Transmitter board è in sleep mode finché l'user button non è premuto.

La pressione del bottone, gestita con interrupt, farà partire la trasmissione con la seconda board per poi mettersi in attesa di ricezione.

La Receiver Board è in Sleep Mode finché non viene svegliata da un interrupt per la ricezione.

Capitolo 1. STM32: USART SMARTCARD

Dopo aver ricevuto, trasmetterà un carattere verso la Receiver Board.

I LED della board sono utilizzati per monitorare lo stato dell'esecuzione:

- LED 6 (Blu) è ON quando il processo di trasmissione è completo
- LED 5 (Rosso) è ON quando è stato ricevuto il messaggio che ci si aspettava
- LED 4 (Verde) è ON quando il processo di ricezione del carattere inviato dalla controparte è completo
- LED 3 (Arancione) è utilizzato per indicare che il codice è in esecuzione sulla transmitter board.

Quando si spegne indica che è iniziato il processo d'invio e di ricezione e quando si riaccende indica che al board è pronta per inviare di nuovo il messaggio .

L'UART in modalità Smartcard è stato configurato come segue:

- BaudRate = 115200 baud
- Word Length = 8 Bits (7 data bit + 1 parity bit)
- 1.5 Stop Bit
- bit di parità pari
- Hardware flow control disabled (RTS and CTS signals)

Il canale è Half Duplex in modalità Smartcard sul pin PA2.

Listing 1.1: Smartcard README

Per comprendere al meglio l'esempio descritto si lascia nel [Listing 1.2](#) il codice che implementa le callback associate all'evento di ricezione e trasmissione sull'UART.

```
1 void HAL_SMARTCARD_TxCpltCallback(SMARTCARD_HandleTypeDef * hsc)
2 {
3     /**
4      * Set del flag SmartcardReady: transfer complete
5      */
6     SmartcardReady = SET;
7
8     /**
9      * LED 6 acceso: il trasferimento si è concluso
10     */
11    HAL_GPIO_WritePin(LED6_GPIO_Port, LED6_Pin, 1);
12 }
13
14 void HAL_SMARTCARD_RxCpltCallback(SMARTCARD_HandleTypeDef * hsc)
```

```

15 {
16 /**
17 * Set del flag SmartcardReady: receive complete
18 */
19 SmartcardReady = SET;
20
21 /**
22 * Incremento della variabile nReceived
23 */
24 nReceived++;
25
26 /**
27 * LED 4 acceso: è stato ricevuto il carattere inviato dalla controparte.
28 * Nel caso del transmitter board sta a significare: "La seconda ricezione consecutiva si
29 * è conclusa"
30 * Nel caso del Receiver Board sta a significare: "La prima ricezione consecutiva si è
31 * conclusa"
32 */
33 #ifdef TRANSMITTER_BOARD
34     if(nReceived == 2)
35 #endif
36
37 HAL_GPIO_WritePin(LED4_GPIO_Port, LED4_Pin, 1);
38
39 /**
40 * Si controlla se il carattere ricevuto è quello che ci si aspettava di ricevere dalla
41 * controparte.
42 * Nel caso in cui il controllo vada a buon fine, il LED 5 verrà acceso.
43 */
44
45 if(!Buffercmp(aRxBuffer,aRxExpected, RXBUFFERSIZE)){
46     HAL_GPIO_WritePin(LED5_GPIO_Port, LED5_Pin, 1);
47 }
48
49 #ifndef TRANSMITTER_BOARD
50     else if(nReceived !=2){
51         /**
52         * Nel Receiver board
53         * se il primo carattere ricevuto (quello della ricevuta dalla controparte) non è quello
54         * che ci si aspettava
55         * il led 5 viene spento
56         */
57         HAL_GPIO_WritePin(LED3_GPIO_Port, LED5_Pin, 0);
58     }
59 #endif
60 }

```

Listing 1.2: Smartcard_code

1.6.1 Sleep mode

Per evitare che la board sprecasse energia durante l'attesa dell'evento di terminazione di trasmissione o ricezione è stata configurata opportunamente la *Sleep Mode*. Questa modalità di funzionamento, caratteristica per i processori **ARM**, permette di ridurre il consumo energetico quando la board è in attesa di eventi interrompenti provenienti da sorgenti esterne. È importante precisare che quando si entra in modalità *Sleep* anche gli eventi di **systick**, cioè eventi generati con cadenza regolare da un timer configurato automaticamente dall'HAL, sono visti come eventi che risvegliano il processore dalla modalità. Pertanto il Tick di sistema deve essere sospeso prima di entrare in questa modalità e riattivato quando si esce dalla modalità per riprendere la normale esecuzione.

```
1 HAL_SuspendTick();  
2 HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON,  
                           PWR_SLEEPENTRY_WFI);  
3 HAL_ResumeTick();
```

Listing 1.3: Abilitazione Sleep Mode su STM32F4

Capitolo 2

STM32: USB

2.1 Introduzione

USB (Universal Serial Bus) è uno standard hot-pluggable per la comunicazione seriale tra dispositivi. Lo standard definisce le interfacce software, hardware e il protocollo per la connessione. Questo ha dato la possibilità di creare diversi prodotti che, utilizzando lo standard USB, possono interconnettersi ad altrettanti dispositivi.

USB è un protocollo che si basa sul paradigma Master/Slave e non prevede che gli slave possano generare interrupt, pertanto usa un meccanismo basato su polling con periodo di 1 ms per interrogare i dispositivi per capire se hanno qualcosa da inviare. Questo richiede che la gestione del protocollo sia fatta principalmente in hardware utilizzando i componenti chiamati **UTMI** (USB Transceiver Macrocell Interface).

L'ulteriore vantaggio introdotto da USB per gli sviluppatori sono le **Classi** che identificano un dispositivo per un particolare scopo e permettono di utilizzare dei driver generici per quella classe risparmiando al produttore del dispositivo e/o del sistema operativo l'onere dell'implementazione del driver.

Proprio quest'ultimo vantaggio è stato sfruttato insieme ad **OTG** (USB On-The-Go) che permette ad un qualsiasi dispositivo di agire come host per costruire il nostro esempio.

E' possibile infatti creare una seriale sull'USB OTG della scheda STM32F4 che, connessa al pc, può essere utilizzata per effettuare debug delle applicazioni.

2.2 Configurazione Progetto

Come prima cosa per poter utilizzare l'USB OTG sulla board è necessario creare un nuovo progetto su **Cube IDE** e aumentare la minimum heap size. Per farlo, aprire l'ioc e in *Project Manager* → *Project* cambiare il valore di *Minimum Heap Size* da 0x200 a 0x400 o 0x800. Questo è necessario perchè il footprint del codice con USB è molto grande e fa largo uso dell'allocazione dinamica.

Il passo successivo è configurare il device, in *Connectivity* → *USB_OTG_FS* impostare **Mode** come **Device_Only** e in **Parameter Settings** impostare **Speed** come Full Speed. In *Middleware* → *USB_DEVICE* è possibile scegliere la classe di device che si vuole assegnare alla periferica OTG sulla board. Il parametro da configurare è **Class For FS IP** e nel nostro caso scegliamo *Communication Device Class (Virtual Port Com)*, come mostrato in [Figura 2.1](#).

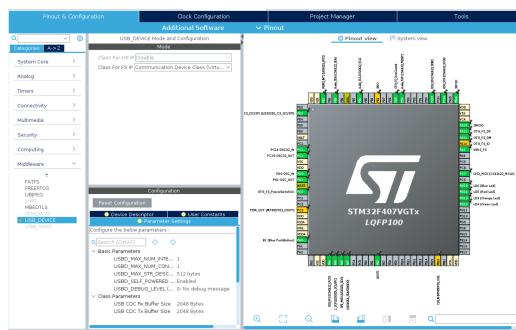


Figura 2.1: IOC USB CDC

Inoltre sulla medesima scheda è possibile configurare anche altri parametri come la dimensione dei buffer di ricezione e invio. In *Device Descriptor* è possibile cambiare alcuni parametri che vengono inviati dal device quando il dispositivo è connesso ad un PC o in generale a un master. Tra questi troviamo **MANUFACTURE_STRING** e **PRODUCT_STRING** che identificano rispettivamente il produttore e il prodotto specifico.

2.3 Codice

Una volta configurato opportunamente l'ioc e generato il codice, troviamo all'interno delle cartelle uno scheletro da modificare per poter implementare il comportamento richiesto quando si vuole inviare o ricevere qualcosa utilizzando USB. Il processo di generazione del codice crea una cartella chiamata

USB_DEVICE nella quale si trovano i file generati per poter implementare la classe selezionata nella fase di configurazione. Nel nostro caso, all'interno della cartella **App** sono presenti i file *usbd_cdc_if.h* e *usbd_cdc_if.c*, che implementano le funzioni per la classe CDC.

Le funzioni che dobbiamo modificare all'interno di *usbd_cdc_if.c* sono **CDC_Transmit_FS** e **CDC_Receive_FS**.

La funzione **CDC_Transmit_FS** è la funzione che invia il contenuto di un buffer sull'usb, come se si scrivesse su una seriale.

Nel Listing 2.1 è presente il codice utilizzato.

```

1 uint8_t CDC_Transmit_FS(uint8_t *Buf, uint16_t Len)
2 {
3     uint8_t result = USBD_OK;
4     /* USER CODE BEGIN 7 */
5
6     USBD_CDC_HandleTypeDef *hcdc = (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.
7         pClassData;
8
9     if(strlen(UserTxBufferFS) + Len > APP_TX_DATA_SIZE ){
10         tx_buf_len = 0;
11         memset(UserTxBufferFS, 0 , APP_TX_DATA_SIZE);
12         memcpy(UserTxBufferFS, "\r\nBuffer Error!!!\r\n", 21);
13         return BUFF_FULL;
14     }
15
16     if(Len && strlen(Buf)){
17         memcpy(UserTxBufferFS, Buf, Len);
18         tx_buf_len += Len ;
19     }
20
21     if (hcdc->TxState != 0){
22         return USBD_BUSY;
23     }
24
25     if(strlen(UserTxBufferFS)){
26
27         USBD_CDC_SetTxBuffer(&hUsbDeviceFS, UserTxBufferFS, strlen(UserTxBufferFS))
28         ;
29         result = USBD_CDC_TransmitPacket(&hUsbDeviceFS);
30
31         if(!result){
32             tx_buf_len = 0;
33             memset(UserTxBufferFS, 0 , APP_TX_DATA_SIZE);
34         }
35     }
36     /* USER CODE END 7 */
37     return result;
38 }
```

Listing 2.1: CDC_Transmit_FS

Il codice aggiunto copia il buffer che si vuole inviare in uno temporaneo in modo tale che se il dispositivo è occupato il messaggio non andrà perso ma verrà inviato successivamente quando viene invocata di nuovo la medesima funzione.

Per escludere il caso in cui il buffer è pieno(condizione indicata dal valore assunto dalla variabile `tx_buf_len`) e di conseguenza non può essere effettuata la `CDC_Transmitt_FS`, è stata modificata la callback di trasmissione completata in modo tale da invocare la `CDC_Transmst_FS`.

Il [Listing 2.2](#) mostra la modifica appena descritta.

```

1 static int8_t CDC_TransmitCplt_FS(uint8_t *Buf, uint32_t *Len, uint8_t epnum)
2 {
3     uint8_t result = USBD_OK;
4     /* USER CODE BEGIN 13 */
5     if (tx_buf_len > 0){
6         //Flush buffer
7         CDC_Transmit_FS(" ", 0);
8     }
9     UNUSED(Buf);
10    UNUSED(Len);
11    UNUSED(epnum);
12    /* USER CODE END 13 */
13    return result;
14 }
```

Listing 2.2: CDC_Transmit_FS

Inoltre è necessario modificare anche la funzione `CDC_Receive_FS`, questa è la callback chiamata quando viene ricevuto un pacchetto su USB.

Nel [Listing 2.3](#) è mostrato il codice aggiunto che copia quello che si è ricevuto nel buffer di ricezione.

```

1 static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)
2 {
3     /* USER CODE BEGIN 6 */
4     USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
5     USBD_CDC_ReceivePacket(&hUsbDeviceFS);
6
7     memcpy(UserRxBufferFS, Buf, *Len);
8     rx_buf_len += (*Len);
9
10    return (USBD_OK);
11
12    /* USER CODE END 6 */
13 }
```

Listing 2.3: CDC_Receive_FS

La variabile `rx_buf_len` indica il numero di byte attualmente presenti nel buffer di ricezione, questa sarà utile successivamente.

La libreria creata però non mette a disposizione una funzione per permettere all'utente di leggere il contenuto del buffer di ricezione e copiarlo in un proprio buffer. Pertanto è stata creata una funzione che copia il contenuto del buffer di ricezione RX in quello utente e aggiorna la variabile `rx_buf_len`. Nel [Listing 2.4](#) è presente il codice della funzione realizzata

```
1 uint32_t User_CDC_Receive_FS(uint8_t* Buf, uint32_t Len){  
2     if(Len > rx_buf_len){  
3         Len = rx_buf_len;  
4     }  
5  
6     memcpy(Buf, UserRxBufferFS, Len);  
7     rx_buf_len -= Len;  
8  
9     return Len;  
10 }
```

Listing 2.4: User_CDC_Receive_FS

2.4 Uso di `scanf` e `printf`

In alcuni casi, nei sistemi embedded, risulta impossibile compilare e caricare sul dispositivo la *C standard library* pertanto è stata realizzata una versione più compatta chiamata **C Nano Library**. Tra le principali funzioni presenti nella libreria quelle che ci interessano sono la `scanf` e la `printf`.

In particolare queste due funzioni permettono di leggere e scrivere su file, normalmente `stdio` e `stdout`. Tuttavia nel nostro caso le funzioni di lettura e scrittura sul file devono essere sostituite con le operazioni che ci permettono di leggere e scrivere utilizzando USB.

Per fare ciò è necessario modificare o ridefinire le funzioni `_write` e `_read`, presenti in **Core/Src/syscalls.c**. Questo è possibile poichè sono definite con l'attributo `WEAK` il che significa che il compilatore userà quella funzione soltanto quanto non è stata definita una funzione con la stessa firma.

La funzione `_write` è l'ultima funzione invocata della `printf` ed è quella che si occupa di scrivere il buffer d'ingresso sul file o sul dispositivo (nel nostro caso).

Come mostrato in [Listing 2.5](#) questa deve chiamare la `CDC_Trnasmitt_FS`.

```

1 int _write(int file, char *ptr, int len){
2     CDC_Transmit_FS(ptr, len);
3 }
```

Listing 2.5: `_write`

La funzione che deve essere modificata per effettuare la `scanf` è `_read`. Questa è la funzione invocata dalla `scanf` che si occupa di leggere dal dispositivo o dal file il buffer ricevuto. Nel nostro caso deve invocare la funzione che copia il buffer di ricezione nel buffer in ingresso alla `scanf`.

Per ottenere un comportamento analogo alla classica `scanf`, in cui si resta in attesa di un input, è necessario far precedere la lettura da un'attesa fino a quando non si riceve qualcosa, segnalato dalla variazione di `rx_buf_len`.

Nel Listing 2.6 è presente il codice della funzione implementata.

```

1 int _read(int file, char *ptr, int len){
2     while(rx_buf_len == 0);
3
4     return User_CDC_Receive_FS(ptr, len);
5 }
```

Listing 2.6: `_read`

2.5 Esempio d'uso della VCP

Prima di implementare le funzioni per utilizzare la `printf` e la `scanf` è stata implementata una semplice libreria che stampa dei messaggi sulla seriale che può essere utilizzata per effettuare il debugging.

La principale funzione implementata dalla libreria è mostrata nel Listing 2.7 che in base al livello di debug scelto aggiunge alla firma, settata in fase di inizializzazione, un messaggio che indica il livello d'importanza del messaggio per poi chiamare la `CDC_Transmit_FS`.

```

1 uint8_t USB_VCP_DEBUGGER_print(vcp_debugger *debugger, const uint8_t
2                                 debug_level, const uint8_t *msg){
3
4     uint8_t *debug_msg = malloc(sizeof(uint8_t) * (MAX_MSG_LEN));
5     get_debug_msg_from_level(debug_level, debug_msg);
6
7     uint8_t msg_len = strlen(msg);
8     uint16_t tot_msg_len = msg_len + debugger->fingerprint.len + strlen(debug_msg) +
9                           3;
10    uint8_t *str = malloc(sizeof(uint8_t) * (tot_msg_len));
11
12    strncpy(str, debugger->fingerprint, debugger->fingerprint.len);
```

Capitolo 2. STM32: USB

```
11 strncat(str, debug_msg, strlen(debug_msg));
12 strncat(str, msg, msg_len);
13
14 strncat(str, "\r\n", 2);
15 str[tot_msg_len] = '\0';
16
17 int status = CDC_Transmit_FS(str, tot_msg_len );
18
19 free(str);
20 free(debug_msg);
21
22 return status;
23 }
24
25 Per ulteriori dettagli è possibile visionare i file presenti in \textit{ STM32F4/usb\_cdc\
    _vcp\_example/Core/Inc} e in \textit{ STM32F4/usb\_cdc\_vcp\_example/Core/
    Src}.
```

Listing 2.7: USB_VCP_DEBUGGER_print

Capitolo 3

STM32: CAN

3.1 Introduzione

Nel seguente capitolo si è sviluppato un esempio di utilizzo della periferica CAN integrata nel SoC Cortex M4 sulla board STM32F4. In particolare si sono sviluppati due esempi, uno con l'ausilio della libreria HAL, che fornisce un' astrazione di alto livello per il controllo dell'HW, agevolando lo sviluppo, e uno scrivendo direttamente sui registri della periferica. A tal proposito, per semplificare, nel secondo esempio si è comunque deciso di sviluppare una piccola libreria che implementi le funzioni necessarie per realizzare l'esempio. In entrambi gli esempi è stato utilizzato soltanto il dispositivo CAN1 in modalità loopback, la trasmissione e la ricezione si verranno effettuate sulla stessa board fisica. Inoltre è stato deciso di utilizzare soltanto la MAILBOX0 e il filtro FIFO0.

3.2 Driver con libreria HAL

Per lo sviluppo tramite libreria HAL si è deciso di partire dalla configurazione del SoC attraverso il wizard del file .ioc. A configurazione ultimata è stato possibile autogenerare lo scheletro del codice necessario ad implementare una semplice comunicazione via CAN. Infatti, dal momento che si è deciso di utilizzare la periferica con le interruzioni, è stato sufficiente ridefinire le funzioni ISR secondo la logica scelta. Nello specifico si è deciso di implementare un semplice contatore, incrementato ad ogni ricezione, che visualizza il valore sui led.

3.3 Driver senza libreria HAL

In questo esempio è stata pilotata la periferica CAN senza l'utilizzo della libreria HAL. Tale implementazione si è basata su alcune assunzioni: gli identificativi sono stati considerati esclusivamente standard e i frame sono solo di tipo DATA (non REMOTE). Per semplicità si è deciso di effettuare la configurazione delle altre periferiche (User button, HSE (clock esterno), GPIOD) con l'ausilio dell'interfaccia nel file .ioc fornita dall' IDE e di focalizzare quindi lo sviluppo sulla configurazione e utilizzo di CAN. Di seguito si spiegherà brevemente il modo di utilizzo e le funzionalità della libreria sviluppata.

3.3.1 Funzionamento libreria

Funzione di Setup

```
1 void setup (uint32_t enable_interrupts)
```

La funzione effettua il setup della periferica CAN e la porta nello stato di inizializzazione. Tale funzione abilita il clock per il CAN configurando il bus APB e il clock per il GPIOB, usato dai pin Tx e Rx, configurando il bus AHB. Inoltre imposta PB8 come Alternate functions per CANTx e PB9 come Alternate function per CANRx e imposta il bit time del CAN in modo da avere 1/17 di bit time per la rilevazione del bit change, 12/17 del bit time per il campionamento e la restante parte per la trasmissione.

La funzione prende in ingresso il parametro enable_interrupts dando la possibilità all'user di abilitare due tipologie d'interrupt:

- CAN_IER_FMPIE0 abilita l'interrupt quando vi è un messaggio pending nella FIFO0 di ricezione (segnala quando i bits FMP0 di stato nel CAN_RF0R register non sono '00');
- CAN_IER_TMEIE abilita l'interrupt quando il messaggio è stato trasmesso dalla MAILBOX0 di trasmissione (segnala quando il bit RQCP0 nel CAN_TSR register è asserito via HW).

Modalità di test

```
1 void set_testmode (unsigned int testmode)
```

La funzione permette di impostare la modalità di test durante la fase di inizializzazione del CAN. Per ripristinare il modo di operare standard del

CAN basterà passare zero come valore del parametro testmode. Il parametro 'testmode' accetta la o delle modalità di testmode scelte:

- **CAN_BTR_SILM** (SILENT) CANTx è sempre recessivo, e permette di spiare il canale tramite CANRx. Come mostrato in [Figura 3.1](#).
- **CAN_BTR_LBKM** (LOOPBACK) Viene letto il valore trasmesso sul bus senza leggere realmente il bus. CANTx è realmente connesso al BUS. Come mostrato in [Figura 3.2](#).

Nota: La o delle due modalità elimina completamente i contatti con il bus esterno, come mostrato in [Figura 3.3](#).

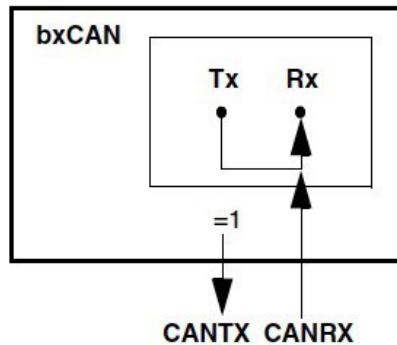


Figura 3.1: Silent Mode

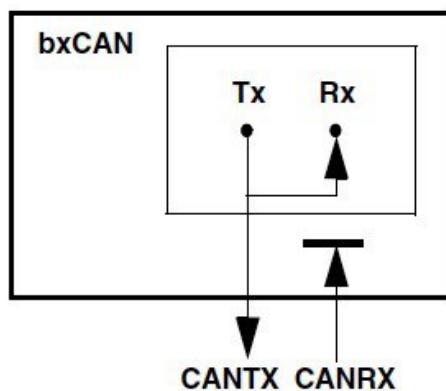


Figura 3.2: Loopback Mode

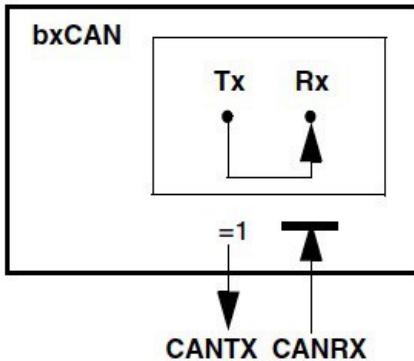


Figura 3.3: Combined Mode

Attese attiva

Sono state implementate due funzioni di attesa attiva.

La prima (`wait_trans_mail0_is_empty()`) è attiva sulla condizione transmit MAILBOX0 vuota mentre la seconda (`wait_msg_pending_fifo0_()`) sulla condizione messaggio pending nella FIFO0 di lettura.

Normal mode

La Funzione `start()` permette di lasciare l'initialization mode ed entrare nel normal mode. Per fare ciò è sufficiente che il bit INRQ, nel registro di controllo **MCR**, sia resettato. Successivamente si attende che il bit CAN_MCR_INRQ dello status register **MSR** sia stato resettato via hardware.

Filtering

Il CAN1 mette a disposizione 14 filter banks, che possono essere configurati per permettere al dispositivo di leggere/accodare i messaggi che transitano sul bus che hanno un identificativo associato a un filter bank. Per il settaggio di un banco di filtri si è scelto di fornire la configurazione di un filtro in modalità Lista (e non mascheramento) tramite la funzione:

```
1 uint8_t config_bank_filter (uint16_t* id, uint8_t num_id, uint32_t CAN_filterIdx)
```

Avendo considerato solo identificativi standard, si è scelto di utilizzare i due registri (32 bit) del banco CAN_filterIdx in modo tale da poter inserire 4 standard identifier. Per ottenere tale Filter bank scale configuration i bit di posizione CAN_filterIdx dei registri di modo (FM1R) e di scale (FS1R) sono stati settati nel seguente:

Capitolo 3. STM32: CAN

- FBMx=1: List Mode (e non Mask mode)
- FSCx=0: organizza i due registri da 32 bit (dedicati al banco di filtri) in 4 registri di 16 bit (di cui solo gli 11 più significativi di ciascuno sono utilizzati per l'identifier STD). Quindi due identificativi standard potranno essere scritti nei bit (31 ; 31-11) e (15 ; 15-11) del FxR1 e altri due nei bit (31 ; 31-11) e (15 ; 15-11) del FxR2.

Nella seguente figura è possibile osservare le diverse tipologie di configurazione possibili tra cui anche quella scelta e implementata in questa libreria.

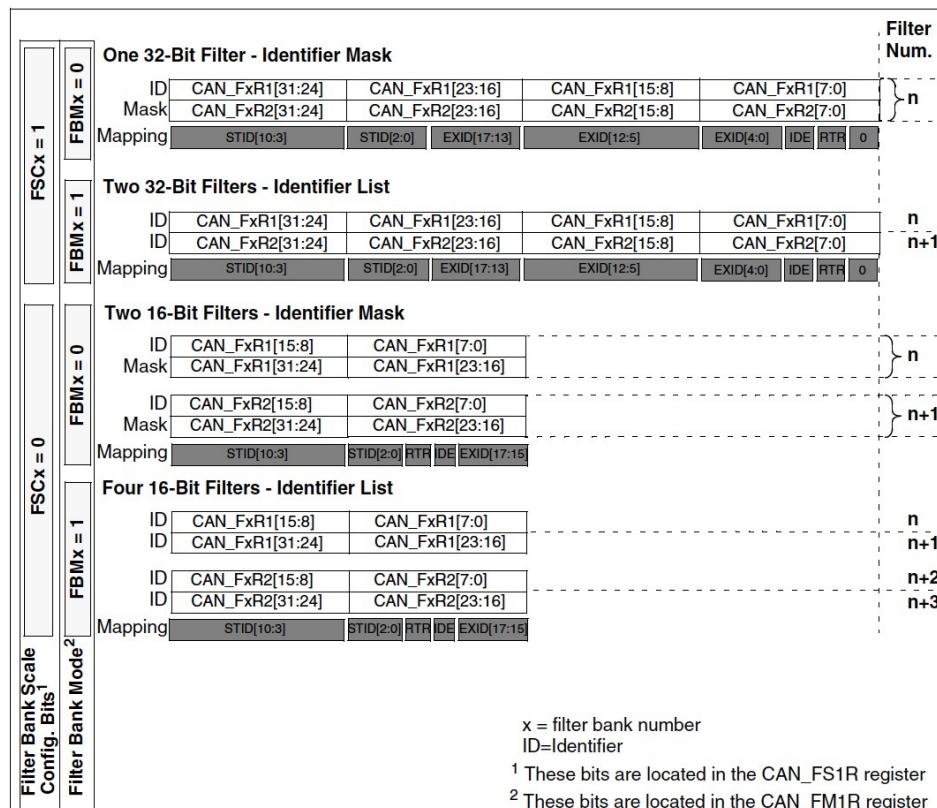


Figura 3.4: filter bank scale configuration-organization register

I parametri di ingresso della funzione sono:

- **id** può essere un vettore di al massimo 4 identificativi standard da filtrare nella ricezione;
- **num_id** deve corrispondere al numero di identifier presenti nel vettore id;

- **CAN_filterIdx** è filtro che si vuole settare (da 0 a 13).

Il valore di ritorno è -1 in caso di errata num_id o CAN_filterIdx, 0 altrimenti.

Funzione di trasmissione

```
1 uint8_t frame_transmit (CAN_msg* msg, uint8_t with_int_trasm)
```

La funzione invia un frame tramite la MAILBOX0 di trasmissione. In base al tipo di identifier del messaggio viene settato in modo opportuno il registro TIR ma in tale implementazione si è supposto di gestire solo identificativi Standard (bit2 di TIR forzato a zero), di conseguenza l'identificativo (11 bit) deve essere messo a partire dalla posizione 21 del registro TIR. Analogamente poiché in tale implementazione è stata considerata solo la trasmissione di frame di tipo DATA e non REMOTE, il bit 1 di TIR è stato fissato a 0. Il frame data è il frame che permette di inviare fino a 8 byte. Se si vuole abilitare il segnalamento di completamento della trasmissione bisogna passare '1' come valore del parametro di ingresso *with_int_trasm*. I parametri di ingresso sono:

- **msg** instanziazione di un messaggio CAN STANDARD e di tipo DATA;
- **with_int_trasm** valore 1 se si vuole abilitare l'interruzione al completamento della trasmissione.

Il valore di ritorno sarà '-1' se la lunghezza in byte del messaggio da trasmettere non è compresa tra 0 e 8 byte, '0' altrimenti.

Funzione di Ricezione

```
1 uint8_t frame_read (CAN_msg* msg)
```

La funzione permette di leggere il messaggio pending sulla FIFO0, inoltre rileva anche un eventuale segnalamento di overrun. L'unico parametro per la funzione è **msg** (Ingresso /uscita) che aggiorna la struttura con le informazioni del messaggio letto. Il valore di ritorno sarà '-1' in caso di overrun, '0' altrimenti

3.3.2 Testing della libreria

La libreria di pilotaggio del CAN è stata testata nel modo seguente. L'identificativo da filtrare è stato messo in ognuna delle quattro posizioni configurate

Capitolo 3. STM32: CAN

per il banco di filtri 1. Sono stati trasmessi da 1 a 8 byte e sia la funzione di trasmissione che di ricezione hanno risposto in modo corretto. Il flusso di interruzione sia per la trasmissione che per la ricezione è stato testato.

Capitolo 4

Compilare e installare Linux su Zybo

4.1 Prerequisiti

Per poter compilare e installare Linux sulla Zybo è necessario avere a disposizione una distribuzione Linux con installati i seguenti pacchetti:

- **Xilinx Vivado con SDK**, conosciuto come **Vitis**;
- **U-boot-tools**: per poter compilare U-Boot, bootload che caricherà il kernel e il root fs;
- **Device tree compiler**: per compilare il devicetree che utilizzerà il kernel per conoscere le periferiche presenti sulla board e i relativi driver;
- **Gparted**: per partizionare la scheda SD che conterrà il kernel, bootloader, devicetree e il root FS;
- **Git**: per clonare le repository;
- **Cross-compilatore ARM**: per poter compilare il materiale da eseguire sulla board;
- **build-essential**; per avere a disposizione tutti i tools per compilare;
- **libncurses**: per poter utilizzare l’interfaccia grafica da terminale, che usano i menuconfig;
- **libssl-dev**: libreria contenente gli header di ssl per poter compilare correttamente il u-boot. **Nota** in alcuni casi è necessario avere la versione 1.0.

- **libgcrypt-dev**: libreria contenente header di funzioni crittografiche usate per la compilazione di u-boot.

Per installare i pacchetti necessari è possibile usare il seguente comando:

```
sudo apt install -y u-boot-tools libncurses-dev device-tree-compiler gparted git build-essential libssl-dev libgcrypt-dev
```

Nota: Se si riscontrano problemi durante la compilazione di U-Boot, probabilmente sono dovuti alla versione di **libssl-dev**, per risolvere l'inconveniente è necessario disinstallare la libreria e installare la versione 1.0.

Per disinstallare, usare il comando:

```
sudo apt remove --purge libssl-dev
```

Scaricare la versione di libssl-dev 1.0 e installarla, usando i seguenti comandi:

```
wget http://archive.ubuntu.com/ubuntu/pool/main/o/openssl/1/libssl1.0.0_1.0.2g-1ubuntu4_amd64.deb  
dpkg -i libssl1.0.0_1.0.2g-1ubuntu4_amd64.deb
```

Nota: se si usa una distribuzione diversa da Ubuntu, potrebbe essere necessario individuare il pacchetto nei repository della distribuzione che si sta utilizzando.

4.1.1 Download dei file necessari

Creare una cartella dove si cloneranno tutti i repository necessari per poter realizzare la nostra distribuzione personalizzata. Per la realizzazione del root filesystem si è scelto di utilizzare buildroot, ma è possibile utilizzare qualsiasi altro strumento integrato che permetta di realizzare root filesystem personalizzati, come yocto o petalinux, oppure è possibile compilare un root filesystem di una nota distribuzione come Ubuntu o Debian.

I repository da clonare sono:

- **linux-digilent**: kernel linux, già patchato per poter essere cross-compilato e eseguito sulla zybo;
- **u-boot-digilent**: versione di U-Boot pronta per essere cross-compilata ed eseguita sulla zybo;
- **buildroot**: tool per generare un root filesystem personalizzato;
- **device-tree-xlnx**: file necessari alla compilazione del device tree.

Per clonare i repository citati è possibile usare i seguenti comandi:

```
git clone https://github.com/Digilent/linux-digilent.git  
git clone https://github.com/Digilent/u-boot-digilent.git  
git clone https://github.com/buildroot/buildroot.git  
git clone https://github.com/Xilinx/device-tree-xlnx.git
```

4.1.2 Configurare il build-environment

Per poter cross-compilare correttamente e generare il file di boot da inserire sulla zybo è necessario esportare delle variabili d'ambiente. La prima **ARCH** che indica l'architettura di riferimento durante la compilazione del kernel.

```
export ARCH=arm
```

Per poter utilizzare gli strumenti di vivado, utili per la generazione del devicetree partendo dal file xsa, è necessario caricare il suo enviroment, usando il comando:

```
source /tools/Xilinx/Vitis/2019.2/settings64.sh
```

Aggiungere il path della tool chain di Vitis al **PATH** di sistema, in modo tale di poter utilizzare il cross compilatore.

```
export PATH=${PATH}:/tools/Xilinx/Vitis/2019.2/gnu/aarch32/lin/  
gcc-arm-linux-gnueabi/bin
```

Esportare la variabile d'ambiente che indica il cross compilatore da usare, il nome potrebbe cambiare se si usano versioni diverse di Vivado:

```
export CROSS_COMPILE=arm-linux-gnueabihf-
```

Nota: il path `/tools/Xilinx/` è il path di installazione di Vivado e Vitis e 2019.2 è la versione usata, pertanto se si installa Vivado in un altro path o si usa un'altra versione è necessario aggiornare il path usato nei comandi precedenti.

Inoltre è fondamentale che tutte le operazioni di compilazione vengano fatte utilizzando il terminale nel quale si sono eseguiti i precedenti comandi, altrimenti i binari generati verranno compilati per l'architettura del pc Host e non per quella del sistema Target.

4.2 BOOT

Il processo di boot della zybo, si articola in tre fasi, eseguite da 3 componenti diversi:

- Fase 0: Appena la board viene accesa viene eseguito il codice presente nella **BootROM**, presente sulla board, che a seconda della configurazione del jumper JP5, si prepara per eseguire il boot dalla **MicroSD** o per preparare la scheda per essere flashata con **JTAG** o con **QSPI Flash**.
- Fase 1: Se la board è configurata come boot dalla MicroSD, viene eseguito il First Stage Boot Loader (**FSBL**), realizzato da noi partendo dall'XSA. Il FSBL non è altro che del codice che viene eseguito dal processing system, che ha come compito quello di flashare il nostro bitstream e lanciare U-Boot;
- Fase 2: U-Boot carica in Ram il Kernel e poi il root file system.

Per eseguire la Fase 1 e la Fase 2 del boot, è necessario caricare sulla partizione di Boot della microSD un file chiamato **BOOT.bin** (è importante rispettare il nome), che deve essere realizzato utilizzando gli strumenti messi a disposizione da Vitis.

Il file **BOOT.bin** contiene al suo interno in ordine: il binario del FSBL, il bitstream e il binario di U-Boot.

4.2.1 FSBL

Per realizzare il binario del FSBL che verrà eseguito durante la fase di boot è necessario disporre di un file **xsa**, che descrive il nostro HW.

1. Aprire Vitis e creare un nuovo **Platform Project**, creandolo da un'Hardware Specification e selezionando il file xsa di riferimento.
2. Aprire il progetto e assicurarsi che sia spuntata la voce Generate boot components, come in [Figura 4.1](#)

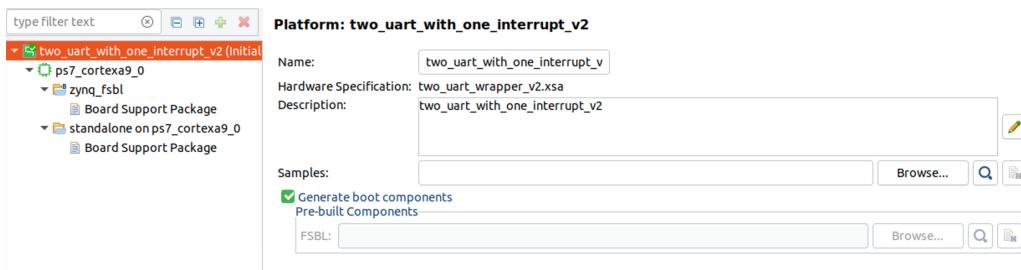


Figura 4.1: Generate boot components

3. Fare la build del progetto;

4. Alla fine della compilazione, aprire la cartella del progetto e andare in **zynq_fslb** e copiare il file **executable.elf** rinominandolo **fsbl.elf**(per comodità) in una cartella temporanea, che conterrà anche gli altri file per creare il **BOOT.bin**.

4.2.2 Bitstream

E' possibile trovalo nella cartella di Vivado, dopo che ha eseguito la sintesi e l'implementazione del design. In alternativa, se è presente nel file xsa, è possibile ottenerne il bitstream usando **xsct**. Per estrarre il bitstream è necessario aprire xsct, lanciando il seguente comando da terminale (nota: è necessaria la source del file settings64.sh di Vitis):

```
xsct
```

Nella console di xsct dare il seguente comando:

```
hsf open_hw_design path_del_file_xsa_compreso_di_nome  
exit
```

Usciti da xsct, nella cartella corrente, se incluso nell xsa, sarà presente il file con estensione **.bit** (se incluso nel xsa). Copiamo il bitstream nella stessa cartella temporanea in cui è presente il FSBL.

4.2.3 U-Boot

Prima di compilare U-Boot è necessario configurarlo, in modo tale che carichi il root FS dalla SD e che non utilizzi ramdisk. In particolare sarà necessario modificare il file **uboot_dir/includes/zynq_common.h** nel seguente modo:

- Individuare la riga dove inizia la definizione di **sdboot**

```
"sdboot;if mmcinfo; then " \\
```

e all'ultima riga, dove si trova bootm sostituirla **\$ramdisk_load_address** con **-**.

Quindi la riga

```
"bootm ${kernel_load_address} ${ramdisk_load_address} ${  
devicetree_load_address}; " \  
\\
```

diventa

```
"bootm ${kernel_load_address} - ${devicetree_load_address}\n}; " \
```

- Cancellare la riga immediatamente precedente a quella modificata, cioè quella contenente:

```
"load mmc 0 ${ramdisk_load_address} ${ramdisk_image} && \"\
```

- Cancellare la riga che contiene la seguente stringa:

```
"ramdisk_image=uramdisk.image.gz\0" \
```

E' poi necessario creare la configurazione per il **Makefile** che verrà utilizzata durante la compilazione, lanciando il seguente comando:

```
make zynq_zybo_config
```

Infine è necessario compilare usando **make**, ricordandosi di poter usare l'opzione **-j4** per poter parallelizzare la compilazione:

```
make -j4
```

Il risultato della compilazione, che ci interessa, sarà il file presente nella cartella di U-Boot, chiamato **u-boot**, spostiamo anche questo nella cartella temporanea dove è presente il bitstream e il FSBL.

4.2.4 Realizzazione del file **BOOT.bin**

1. Aprire Vitis
2. Aprire lo strumento per creare la Boot Image, cliccando sul menu *Xilinx* → *Create Boot Image*
3. In 'Output Bif file path', aprire il path della cartella temporanea e dare come nome file **BOOT.bif**
4. In **boot image partitions** cliccare sul pulsante **Add** e aggiungere il file **fsbl.elf**, assicurandosi che sotto la voce **Partition type** sia presente **bootloader**, come in [Figura 4.2](#);

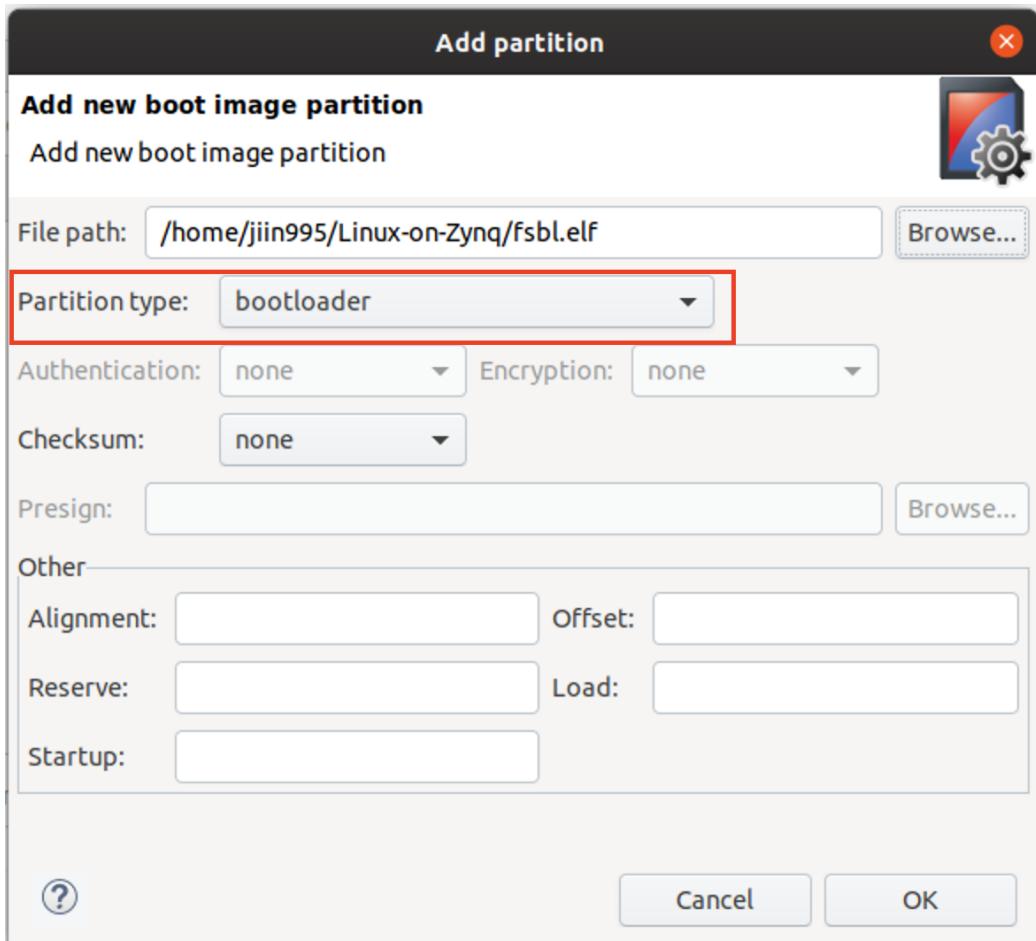


Figura 4.2: Aggiunta di FSBL al boot image

5. Aggiungere il bitstream, assicurandosi che sotto la voce **Partition type** sia presente **datafile**;
6. Aggiungere il file u-boot.elf, assicurandosi che sotto la voce **Partition type** sia presente **datafile**.
7. Cliccare su **Create Image**

Nota: è importante rispettare l'ordine delle add. Il risultato deve essere simile a quello in [Figura 4.3](#).

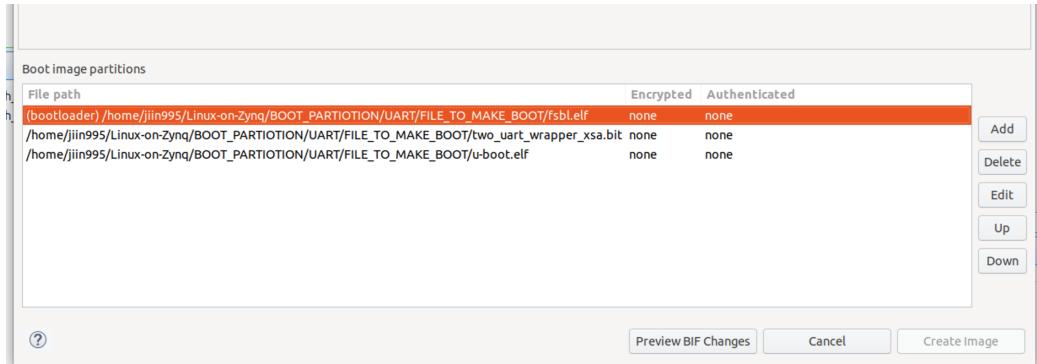


Figura 4.3: Risultato boot

Il file **BOOT.bin**, presente nella cartella selezionata all'inizio, dovrà essere caricato nella partizione di boot della microSD.

4.3 Kernel

La compilazione del Kernel è estremamente facile in quanto già patchato e l'unica modifica necessaria è legata all'abilitazione del supporto per i driver **UIO**.

4.3.1 Configurazione

Posizionarsi nella cartella dove si è scaricato il kernel e caricare la configurazione di default:

```
make xilinx_zynq_defconfig
```

Abilitare il supporto ai driver UIO, aprendo il menu config:

```
make menuconfig
```

Aprire il menù **Device Drivers** e cercare **Userspace I/O drivers** e mettere l'asterisco, premendo space, come mostrato in figura.

Capitolo 4. Compilare e installare Linux su Zybo

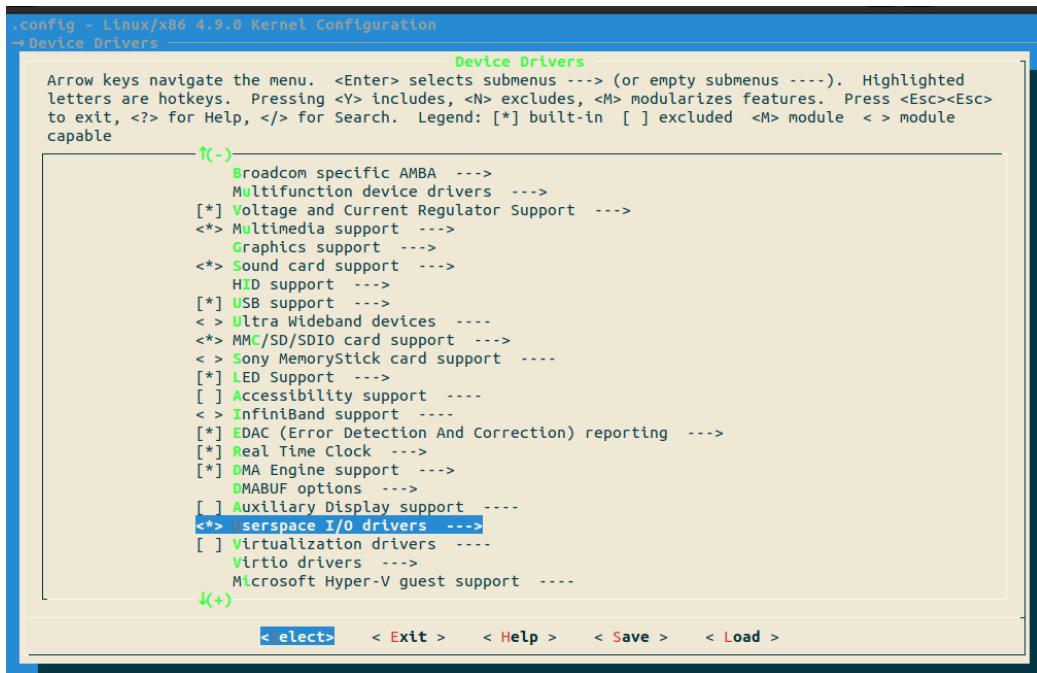


Figura 4.4: Risultato boot

Successivamente eseguire 'make -j4' o il massimo numero di CPU disponibili

4.3.2 Compilazione del kernel

```
make UIMAGE\_LOADADDR=0x8000 uImage -j4
```

Al termine della compilazione nella cartella **linux-digilent/arch/arm/boot/** sarà generato il file **uImage**, che dovrà essere portato anch'esso nella partizione di boot della microSD.

4.4 DeviceTree

Il Device Tree è utilizzabile per descrivere l'hardware di sistema in una struttura dati ad albero. In questa struttura, ogni nodo dell'albero descrive un dispositivo. Il codice sorgente del Device Tree viene compilato dal Device Tree Compiler (**DTC**) per formare il Device Tree Blob (**DTB**), leggibile dal kernel all'avvio. Tale file è utilizzato per fornire al kernel le informazioni Platform Specific, come ad esempio:

- Quali periferiche sono connesse al device, la loro configurazione e il driver da utilizzare per controllarle;
- configurazione del processore;
- setup della console.

Per realizzare il **DTB**, opportunamente configurato, è necessario creare un nuovo software design usando **xsct**.

Posizionarsi nella cartella **device-tree-xlnx** e lanciare **xsct**. Da xsct digitare i comandi:

```
hsı open_hw_design path_del_file_xsa_compresso_di_nome  
hsı set_repo_path path_della_repo_device_tree_xlnx  
hsı create_sw_design device-tree -os device_tree -proc  
    ps7_cortexa9_0  
hsı generate_target -dir customdt  
exit
```

Nota: *ps7_cortexa9_0* è il processore per la Zynq 7000, se si volesse usare un'altra board è necessario cambiare questo valore.

Dopo il comando *generate_target*, nella cartella corrente, viene creata la cartella *customdt*. In questa cartella sono presenti i file che permettono la generazione del device tree nel formato Blob, tuttavia è prima necessario effettuare delle modifiche.

4.4.1 Modificare i bootargs

Nel file **system-top.dts** (descrive la top-level entity del nostro sistema), è necessario modificare i bootargs, che vengono usati dal kernel durante la fase di boot.

In particolare ci interessa fornire al kernel le seguenti informazioni:

- qual'è il terminale di I/O e il Baud rate da usare per la seriale;
- in quale partizione è presente il root filesystem, qual'è il tipo di partizione e le opzioni di mount;
- abilitazione del supporto alle interruzioni per l'UIO.

Per fornire queste informazioni è sufficiente sostituire la stringa **bootargs** con la seguente:

```
bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw  
earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1  
uio_pdrv_genirq.of_id=generic-uio"
```

4.4.2 Specificare il driver da utilizzare per il controllo delle periferiche costum

Se si utilizzano periferiche custom e si vuole utilizzare un particolare driver per il controllo del device è necessario modificare il device tree. In particolare nel file **pl.dtsi** sono presenti tutti i device che fanno parte della programmable logic definiti nella fase di design, sotto al nome delle periferiche è necessario modificare il parametro **compatible**, assegnandogli il nome del driver necessario all'utilizzo della periferica. Ad esempio per poter controllare il dispositivo utilizzando il driver UIO è necessario sostituire sostituire la riga contenente **compatible** con la seguente:

```
compatible = "generic-uio";
```

Se invece volessimo utilizzare il driver chiamato myGPIOK è necessario sostituire la medesima riga con la riga:

```
compatible = "myGPIOK";
```

4.4.3 Utilizzare le interruzioni

Per poter utilizzare le interruzioni, se correttamente configurate nella fase di design hardware, è necessario assicurarsi che nel file **pl.dtsi** siano presenti le voci

```
interrupt-names = "interrupt";
interrupt-parent = <&intc>;
interrupts = <0 29 4>;
```

In particolare, in caso di più periferiche, è necessario che il secondo parametro nell'ultima riga sia diverso per ogni periferica. Questo valore indica la linea dell'interruzione, tuttavia il valore letto dal kernel corrisponde a questo valore + 32, questo perché le prime 32 linee vengono usate dal kernel per mappare le altre interruzioni.

4.4.4 Compilare il dtb

Prima di compilare il dtb è necessario precompilare il dts usando gcc. Posizionarsi nella cartella **device-tree-xlnx** e lanciare il seguente comando:

```
gcc -I costumdt -E -nostdinc -undef -D__DTS__ -x assembler-with-
    -cpp -o costumdt/system-top.dts.tmp costumdt/system-top.dts
```

Dopo è possibile compilare il dtb:

```
dtc -I dts -O dtb -o costumdt/devicetree.dtb costumdt/system-top.dts.tmp
```

Il file **devicetree.dtb** è il Device Tree Blob che deve essere caricato nella partizione di boot della microSD.

4.5 Root FS

Come per il kernel, anche per il root filesystem è necessario modificare la configurazione prima di avviare la compilazione. Posizionarsi nella cartella di buildroot e lanciare il menuconfig:

```
make menuconfig
```

In target impostare

- **Target Architecture (ARM(little endian))**: endianness del processore;
- **Target Architecture Variant (cortex-A9)**: variante del processore utilizzato;
- **Enable NEON**: abilitazione della Floating Point Unit di ARM;
- **Istruction mettere ARM**: selezione dell'istruzione set da utilizzare

Il risultato dovrebbe essere analogo a quello mostrato in [Figura 4.5](#)

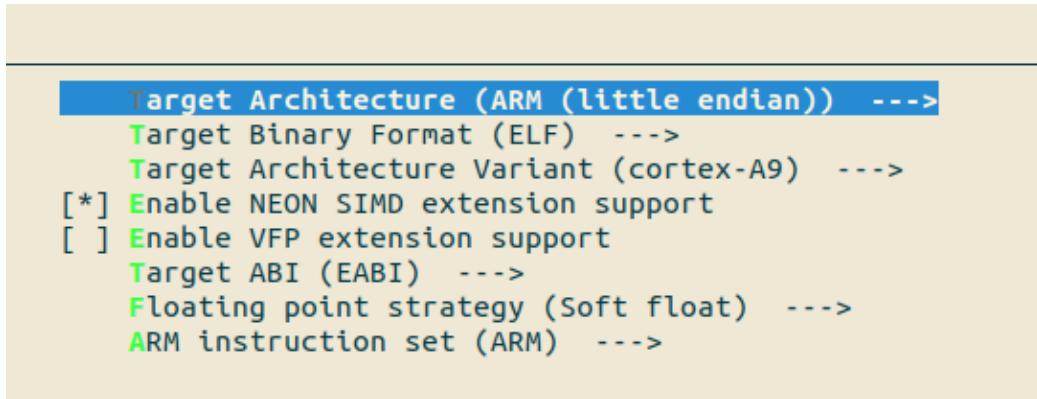


Figura 4.5: Buildroot target config

Per configurare la seriale è necessario tornare alla pagina iniziale del menuconfig e in **System configuration**, sotto la voce **Run a getty (login prompt) after boot** configurarla nel seguente modo:

- **TTY port:** ttyPS0
- **Boundrate:** 115200

Per aggiungere ulteriori pacchetti è necessario recarsi nel menu **Target packages** e selezionare quelli da compilare e includere nel filesystem.

Nota: Per alcuni pacchetti è necessario risolvere alcune dipendenze, in particolare per fare ciò è molto utile la funzione di ricerca nel menuconfig, attivabile premendo il tasto '/'.

Inoltre è necessario tener presente che se si cambiano delle impostazioni in **Build options**, **Toolchain** e in **System configuration** e si è già effettuata una build è necessario effettuare una clean preventiva.

Una volta configurato il tutto, è possibile compilare il root filesystem, ricordandosi anche questa volta dell'opzione `-j`

```
make -j4
```

Al termine del processo di compilazione, nella cartella **buildroot_dir/output/images/** sarà presente un file chiamato **rootfs.tar** contenente il root filesystem da estrarre nella partizione di root della microSD.

4.6 Preparare la microSD

Configurando opportunamente il jumper **JP5**, posizionandolo nella posizione **microSD**, il bootloader presente nella **bootROM** sarà in grado di leggere una partizione **FAT32** presente sulla microSD con l'obiettivo di caricare e poi eseguire il **FSBL** presente nel file **BOOT.bin**.

Come detto precedentemente il FSBL dopo aver flashato il bitstream, lancerà Uboot che a sua volta avvierà il kernel. Secondo quanto specificato nei bootargs, il kernel caricherà dalla seconda partizione presente sulla microSD, con filesystem ext4, il root filesystem.

Pertanto sarà necessario creare due partizioni sulla microSD, una FAT32 di pochi Mb e una ext4 con lo spazio rimanente, lasciando 4Mb di spazio non allocato prima della prima partizione.

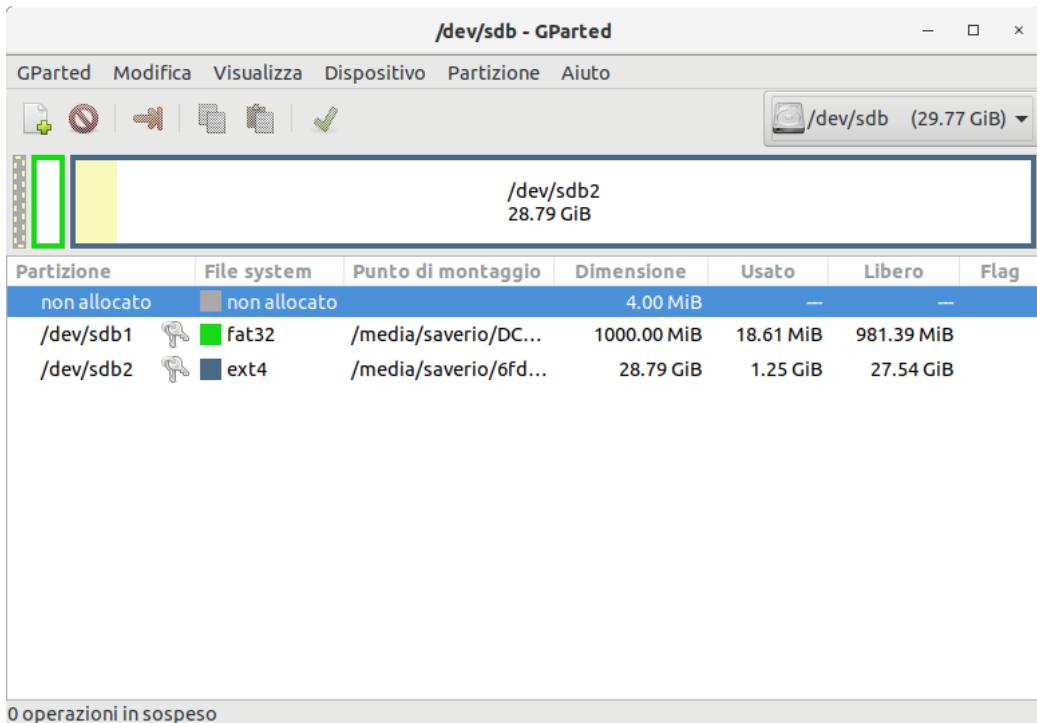


Figura 4.6: Partizionamento della microSD con GParted

Create le partizioni, è necessario copiare i file generati nelle fasi precedenti nelle due partizioni nel seguente modo.

Partizione di **boot** (**FAT32**):

- BOOT.bin
- uImage
- devicetree.dtb

Partizione del **root filesystem** (**EXT4**):

- il contenuto dell'archivio **rootfs.tar**

4.7 Configurazione aggiuntiva

Per il testing e debugging dei programmi da eseguire sulla board è risultato molto utile ed efficiente eseguire un ulteriore passo di configurazione del filesystem. Infatti, se fosse possibile caricare i file eseguibili a runtime, senza bisogno di rimuovere fisicamente la micro sd e caricarli tramite pc, sarebbe molto più agevole ed efficiente. Ciò è risultato possibile modificando la

configurazione base di buildroot, includendo nel filesystem anche **ssh** e **gdb-server**.

Sfruttando il collegamento della board alla rete, tramite cavo ethernet, è stato possibile accedervi tramite ssh per copiare ed eseguire i comandi.

Inoltre gdbserver ha consentito di effettuare il debug dei programmi e di ottenere ulteriori informazioni in caso di malfunzionamento.

4.7.1 Aggiunta dei nuovi pacchetti a buildroot

I pacchetti aggiunti nella versione finale di filesystem utilizzata durante lo sviluppo sono:

- gdbserver
- openssh
- dhcpcd
- vim
- nano

gdbserver

Per abilitare gdbserver è necessario risolvere prima le seguenti dipendenze:

- **toolchain w/**: dal menuconfig di buildroot, in Toolchain, abilitare *Enable Wchar support*, come mostrato in [Figura 4.7](#);
- **threads debug**: dal menuconfig di buildroot, in Toolchain, spuntare *Thread library debugging*, come mostrato in [Figura 4.7](#);
- **C++ support**: dal menuconfig di buildroot andare in Toolchain e abilitare *Enable C++ support*, come mostrato in [Figura 4.7](#).

Capitolo 4. Compilare e installare Linux su Zybo



The screenshot shows the Buildroot configuration interface with the title "Toolchain". It displays a list of toolchain options. The "Toolchain type (Buildroot toolchain) --->" option is selected. Under it, there are sections for "Toolchain Buildroot Options", "custom toolchain vendor name", "library (uClibc-ng)", "Kernel Header Options", "Kernel Headers (Linux 5.4.x kernel headers)", "uClibc Options", and "uClibc configuration file to use?". Other options include "Enable WCHAR support", "Enable toolchain locale/i18n support", "Thread library implementation (linuxthreads)", "Thread library debugging", "Enable stack protection support", "Compile and install uClibc utilities", "Binutils Options", "Binutils Version (binutils 2.32)", "Additional binutils options", "GCC Options", "GCC compiler Version (gcc 8.x)", "Additional gcc options", and "Enable C++ support". Some options like "Enable C++ support" are marked with an asterisk (*) indicating they are selected.

Figura 4.7: Configurazione buildroot toolchain

Risolte le dipendenze, sarà possibile includere il pacchetto **gdbserver** in *Target Packages* → *Debugging, profiling and benchmark* e spuntare le voci **gdb** e **gdbserver** come mostrato in Figura 4.8.

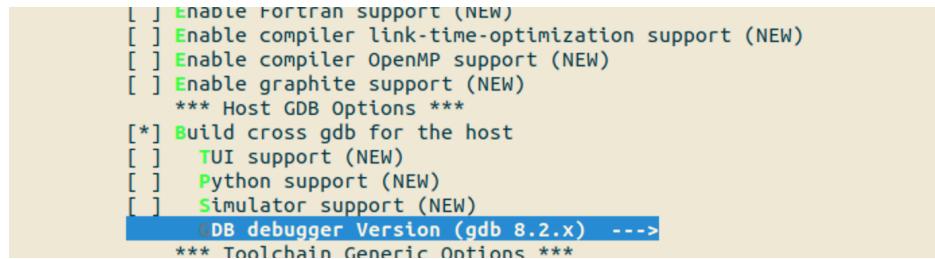


The screenshot shows the Buildroot configuration interface with the title "Target Packages". It lists several packages: fio, gdb, gdbserver, full debugger (NEW), google-breakpad, iozone, and kexec. The "gdb" package is selected, indicated by a green highlight.

Figura 4.8: Aggiungere gdb al root FS

Per poter debuggere un programma compilato per un'altra architettura, come nel nostro caso, è necessario avere una versione particolare di **gdb**. Infatti proprio come per il cross-compilatore che compila per un'altra architettura, **gdb** sulla macchina host (il nostro pc) dovrà interpretare e seguire il flusso di esecuzione per l'architettura target, quindi è necessario avere un cross-gdb che buildroot può compilare durante il processo di build. Per indicare a buildroot che necessitiamo di un cross-gdb è necessario spuntare

in Toolchain la voce **Build cross gdb for the host**, come mostrato in [Figura 4.9](#)



```

[ ] Enable Fortran support (NEW)
[ ] Enable compiler link-time-optimization support (NEW)
[ ] Enable compiler OpenMP support (NEW)
[ ] Enable graphite support (NEW)
*** Host GDB Options ***
[*] Build cross gdb for the host
[ ] TUI support (NEW)
[ ] Python support (NEW)
[ ] Simulator support (NEW)
CDB debugger Version (gdb 8.2.x) --->
*** Toolchain Generic Options ***

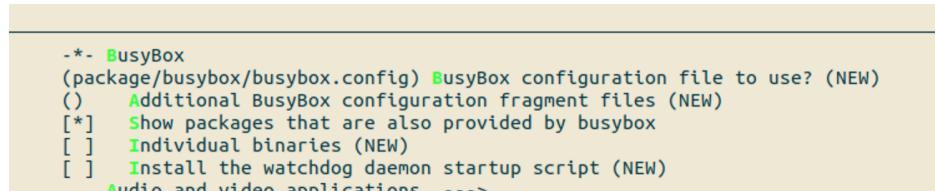
```

Figura 4.9: Build dross gdb for the host

Questa operazione genererà il cross-debugger che sarà salvato nella cartella **'buildroot-dir'/output/host/bin/**.

Per includere openssh e dhcpcd (demone dhcp) è necessario flaggare in *Target Packages* → *Network applications* le relative voci.

Infine per includere vim è necessario prima spuntare in *Target packages* la voce **Show packages that are also provided by busybox** per far comparire la voce da spuntare, come mostrato in [Figura 4.10](#), poi è possibile spuntare vim come mostrato in [Figura 4.11](#).



```

-- BusyBox
(package/busybox/busybox.config) BusyBox configuration file to use? (NEW)
() Additional BusyBox configuration fragment files (NEW)
[*] Show packages that are also provided by busybox
[ ] Individual binaries (NEW)
[ ] Install the watchdog daemon startup script (NEW)
Audio and video applications --->

```

Figura 4.10: Mostrare ulteriori pacchetti di busybox

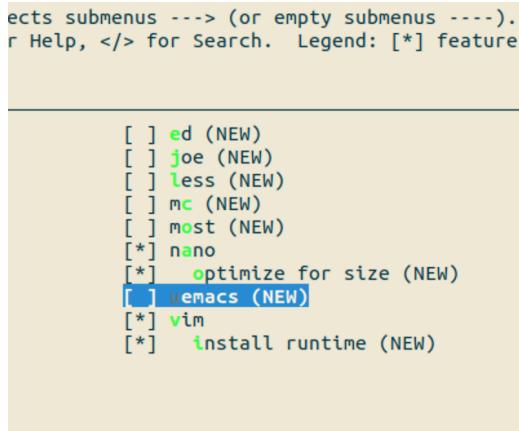


Figura 4.11: Aggiungere vim

Configurazione

Aggiunti i nuovi pacchetti, per poter utilizzare la board come descritto è necessario configurare opportunamente il sistema operativo. Per prima cosa è necessario configurare l'interfaccia di rete e nel nostro caso abbiamo scelto di utilizzare un IP assegnato staticamente.

Per effettuare ciò è stato modificato il file **etc/network/interfaces** aggiungendo le seguenti righe:

```

iface inet eth0 static
    address 192.168.1.222
    netmask 255.255.255.0
    gateway 192.168.1.254
    nameserver 8.8.8.8

```

È importante notare come questa configurazione sia specifica per la rete a cui è stata connessa la board.

L'ulteriore modifica per quanto riguarda la parte di rete è stata quella disabilitare IPv6, in quanto nella nostra configurazione impediva alla board di comunicare con gli altri host della rete.

Per disabilitare IPv6 è necessario modificare **/etc/sysctl.conf** (se non esiste è necessario crearlo) aggiungendo le seguenti righe:

```

net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1

```

L'ultimo passo è abilitare la possibilità di connettersi in ssh al server ssh in esecuzione sulla board utilizzando l'utente root. Per farlo è necessario modificare il file `/etc/ssh/sshd_config` e aggiungere alla fine del file:

```
PermitRootLogin yes
```

Infine, per sicurezza, è necessario impostare una password per l'utente root. Il modo più semplice per fare ciò è avviare la board, connettersi alla seriale e dare il comando **passwd** e digitare la nuova password.

Seguiti questi passi la board sarà in grado di connettersi alla rete e sarà possibile connettersi in ssh per poter copiare file e eseguire comandi.

4.7.2 Uso di gdbserver

Nelle versioni più recenti di gdb è possibile eseguire il debug di applicazione in esecuzioni su host remoti, questo ci ha permesso di eseguire il debug di applicazioni che erano in esecuzione sulla Zybo senza dover necessariamente aggiungere build-essentials al root FS.

Per effettuare il cross-debugging dove il debugger è in esecuzione sul nostro pc e comunica con il **gdbserver** in esecuzione sulla board che esegue il codice è necessario seguire i seguenti passi:

1. Copiare l'eseguibile sulla board;
2. Avviare gdbserver sulla zybo lanciando il comando

```
gdbserver localhost:2000 nome\_eseguibile
```

che avvia il server mettendolo in ascolto sulla porta 2000;

3. Avviare la versione di gdb presente in `<buildroot-dirt>/output/host/bin/` seguito dal path dell'eseguibile copiato sulla zybo

```
<buildroot-dirt>/output/host/bin/<tuple>-gdb nome\
_eseguibile
```

4. Connettersi al server gdb in esecuzione sulla zybo lanciando il seguente comando nella console di gdb:

```
(gdb) target remote 192.168.1.222:2000
```

5. Eseguire il debugging.

Capitolo 5

GPIO

5.1 Introduzione

Il **GPIO** (General Purpose Input Output) è un componente hardware che permette di utilizzare un pin fisico presente su una board sia per effettuare delle operazioni di lettura che di scrittura. In particolare è necessario disporre di una porta tri-state che connessa al pin, di ingresso/uscita, si possa trovare in una condizione di alta impedenza quando è necessario effettuare delle operazioni di lettura, in modo tale da non influenzare il valore letto con quello che si vorrebbe scrivere.

Facendo riferimento alla [Figura 5.1](#), che mostra la struttura logica del singolo gpio, il comportamento del singolo GPIO sarà il seguente:

- *GPIO_read* segue sempre il valore presente su *GPIO_inout*;
- Quando *GPIO_enable* assume valore logico alto, il valore presente su *GPIO_write* viene propagato su *GPIO_inout*.

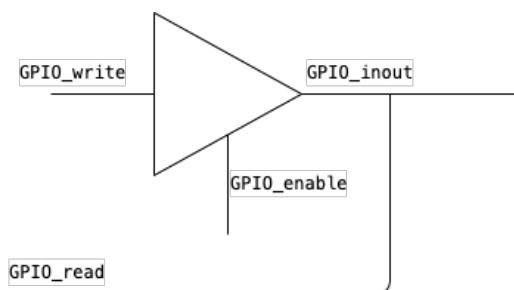


Figura 5.1: GPIO

5.2 Hardware

5.2.1 Componente base

Il componente fondamentale che implementa un singolo **GPIO** è molto semplice da realizzare, nel [Listing 5.1](#) è possibile trovare il codice che descrive il componente.

```
1 architecture Structural of GPIOsingle is
2 begin
3   GPIO_read <= GPIO_inout;
4   with GPIO_enable select
5     GPIO_inout <=   GPIO_write when '1',
6                   'Z' when others;
7 end Structural;
```

Listing 5.1: GPIO VHDL

Per poter stanziare più GPIO è stato realizzato un ulteriore componente, che utilizzando gli opportuni costrutti del VHDL permette di istanziare un numero arbitrario, tra 4 e 32, di GPIO, chiamato **GPIO_Array**.

Tutti i file, con la relativa documentazione possono essere trovati nella cartella *Zybo/GPIO/Hardware/VHDL/*.

5.2.2 Interrupt Generator and Handler

Per la generazione dell'interruzione, in lettura, è stato realizzato un componente separato dal GPIO, chiamato **Interrupt_Generator_Handler**. Il componente realizzato controlla la generazione del segnale d'interruzione per la linea d'ingresso di un singolo GPIO offrendo la possibilità di scegliere la modalità di generazione dell'interruzione: generata sul fronte di salita o di discesa della linea d'ingresso `gpio_inout`, oppure generata in base al livello della linea. Il generatore dell'interruzione fa anche da gestore dell'interruzione, infatti in ingresso c'è il segnale `IACK` che permette di abbassare l'interruzione. Il componente è stato realizzato come una macchina sequenziale e nel [Listing 5.2](#) è presente il codice VHDL della parte che implementa la logica dello stato prossimo e dei segnali d'uscita.

```
1 process(GPIO_INOUT, MODE, EDGE, IACK, state, int_tmp) begin
2   next_int_tmp <= int_tmp;
3   -- rising edge
4   if (((not state) and GPIO_INOUT ) and MODE) and EDGE) =
      '1' then
```

```

5      next_int_tmp <= '1';
6  -- falling edge
7  elsif ((state and (not GPIO_INOUT) and MODE) and (not EDGE)
8 )) = '1' then
9      next_int_tmp <= '1';
10 -- level
11 elsif MODE = '0' then
12     next_int_tmp <= GPIO_INOUT;
13 end if;
14
15 -- Reset del segnale di interrupt
16 if IACK = '1' then
17     next_int_tmp <= '0';
18 end if;
19
20 next_state <= GPIO_INOUT;
21
22 end process;
23 INT <= next_int_tmp;

```

Listing 5.2: Interrupt Generation and Handler

Di seguito è descritto sinteticamente il comportamento di generazione dell'interruzione:

- Quando **MODE** è uguale a 0, il componente alza il segnale d'interrupt quando il pin assume un valore logico alto e lo mantiene fino a quando non si abbassa la linea d'interruzione. **Nota:** Se il pin non viene abbassato, anche se venisse abbassata dall'IACK, il segnale d'interruzione verrebbe immediatamente rialzata e quindi il segnale di IACK in questa modalità ha senso solo quando la linea d'ingresso assume il livello basso.
- Quando **MODE** è alto e **EDGE** è basso, il componente alza la linea di interrupt solo quando rileva un fronte di discesa sulla linea d'ingresso.
- Quando **MODE** è alto e **EDGE** è alto, il componente alza la linea di interrupt solo quando rileva un fronte di salita sulla linea d'ingresso.

5.2.3 AXI Wrapper

La connessione della periferica al Processing System è realizzata con il protocollo **AXI** (Advanced eXtensible Interface). Pertanto è stato istanziato un

componente che implementa un’interfaccia AXI Lite e che wrappa il componente *GPIO_Array* in modo che gli applicativi in esecuzione sul PS possano utilizzare la periferica costum come se fosse una classica periferica memory mapped.

Lo scheletro di questo componente è messo a disposizione direttamente da **Xilinx** usando il suo wizard di generazione di un IP Core. Quest’ultimo funge da anello di congiunzione della nostra periferica con il PS.

Di seguito sono elencati i registri con la modalità a cui è possibile accedere, l’offset da aggiungere al base address della periferica e il loro significato:

- **MODE**: (R/W, offset +0x0): consente di impostare i singoli pin del device come ingressi o uscite; solo i *GPIO_width* bit meno significativi del registro hanno significato, agire sui restanti bit non produce nessun effetto; Il valore che i singoli pin possono assumere sono:
 - ’1’: il pin viene configurato come pin di uscita;
 - ’0’: il pin viene configurato come pin di ingresso.
- **WRITE**: (R/W, offset +0x4): consente di imporre un valore ai pin del device, qualora essi siano configurati come uscite; solo i *GPIO_width* bit meno significativi del hanno significato, agire sui restanti bit non produce nessun effetto;
- **READ**: (R, offset +0x8): consente di leggere il valore dei pin del device, indipendentemente dal mode; solo i *GPIO_width* bit meno significativi del registro hanno significato, gli altri saranno fissi a zero;
- **GIES**: (Global Interrupt Enable/Status, R/W, offset 0xC): Consente di abilitare/disabilitare gli interrupt globali della periferica; sono utilizzati solo i due bit meno significativi del registro:
 - IE (bit 0): interrupt enable, abilita gli interrupt, può essere scritto e letto; se posto ad ’1’ la periferica potrà generare interrupt a seconda di come viene configurata la modalità di interrupt per il singolo bit; se posto a ’0’ il device non genererà mai interruzioni;
 - IS (bit 1): interrupt status, settato dalla periferica ad ’1’ nel caso in cui la periferica abbia generato interrupt; settato a ’0’ quando tutte le interruzioni sono servite; replica del segnale interrupt diretto verso il processing-system, che segnala al GIC il sollevamento di un’interruzione.

- **PIE:** (Pin Interrupt Enable, R/W, offset 0x10): consente di abilitare/disabilitare gli interrupt per i singoli pin. Con GIES(0)=’1’ e MODE(n)=’0’ (interrupt globale abilitati e il pin n-esimo è configurato come input), se PIE(n)=’1’ allora il device ha generato un interrupt verso il processing-system in seguito alla variazione dello stato del pin (variazione configurabile usando i registri IRQ_MODE e IRQ_EDGE), mentre, se PIE(n)=’0’ non verrà generata nessuna interruzione;
- **IRQ:** (Interrupt Request, R, offset 0x14): IRQ(n)=’1’ indica che il pin n-nessimo è la sorgente di interruzione; i singoli bit del registro sono pilotati dall’Interrupt_Generator_Handler, che in base alla configurazione, scelta in fase di setup, alza il segnale di interrupt sulla specifica linea; la or-reduce di tale registro costituisce il flag interrupt (IS) di GIES.
- **IACK:** (Interrupt Ack, W, offset 0x18): imponendo IACK(n)=’1’ è possibile segnalare al device che l’interruzione generata dal pin n-esimo è stata servita; i bit IRQ(n) e IACK(n) saranno resettato automaticamente, permettendo di gestire interrupt consecutive quando la linea funziona con un interrupt sul livello del pin.
- **IRQ_MODE:** (Interrupt Mode, R/W, offset 0x1C): seleziona la modalità con vengono generate le interruzioni per singolo pin, quando GIES(1) = ’1’ e MODE(n)=’0’ e PIE(n) = ’1’ se IRQ_MODE(n)=’0’ il dispositivo genererà interruzioni consecutive (vengono abbassate da IACK(n)) fin quando READ(n) = ’1’, invece se IRQ_MODE(n)=’1’ il dispositivo genera interruzioni solo quando c’è una variazione del clock, il cui fronte è configurabile utilizzando il registro IRQ_EDGE(n).
- **IRQ_EDGE:** (Interrupt EDGE, R/W, offset 0x1F): indica il fronte su cui è sensibile il dispositivo per la generazione dell’interrupt; tale bit assume significato nel sollevamento dell’eccezione solo se il pin è configurato in modo tale da sollevare interruzioni sulla variazione del fronte, ovvero se GIES(1) = ’1’ e MODE(n)=’0’ e PIE(n) = ’1’ e IRQ_MODE(n) = ’1’; quando il dispositivo è sensibile sul fronte se IRQ_EDGE(n) = ’0’ l’interrupt si alza quando si verifica un fronte di discesa; quando il dispositivo è sensibile sul fronte e IRQ_EDGE(n) = ’1’ l’interrupt si alza quando si verifica un fronte di salita;

Oltre ai registri appena elencati sono stati aggiunti anche i seguenti segnali di supporto:

- **GPIO_inout_masked**: segnale utilizzato per dare in input all’Interrupt_Generator_Handler solo i pin per cui è abilitato l’interrupt;
- **interrupt_tmp**: è il segnale ottenuto come la or bit a bit di tutti i bit che compongono il registro IRQ. Questo bit viene messo in and con GIES(1) determina il livello del segnale interrupt che viene collegato al PS.

Per poter utilizzare la periferica è necessario connettere i registri dell’AXI Wrapper, letti e scritti dall’interfaccia, al GPIO_Array istanziato e all’Interrupt_Generation_Handler, come mostrato nel [Listing 5.3](#).

```

1 GPIOarray_inst : GPIOarray
2   Generic map ( GPIO_width      => GPIO_width)
3   Port map (  GPIO_enable     => MODE(GPIO_width-1 downto
0),
4               GPIO_write      => WRITE(GPIO_width-1 downto 0),
5               GPIO_inout      => GPIO_inout,
6               GPIO_read       => READ(GPIO_width-1 downto 0)
7 );
8
9 GPIO_inout_masked <= GPIO_inout and PIE(GPIO_width-1
downto 0);
10
11 interrupt <= interrupt_tmp and GIES(0);
12
13 gen:for i in 0 to (GPIO_width-1) generate
14   inst_int_gh:Interrupt_Generator_Handler
15   Port map (  GPIO_inout => GPIO_inout_masked(i),
16               CLK => S_AXI_ACLK,
17               MODE=> IRQ_MODE(i),
18               RESET => S_AXI_ARESETN,
19               EDGE =>IRQ_EDGE(i),
20               INT => IRQ(i),
21               IACK => IACK(i)
22 );
23 end generate;

```

Listing 5.3: GPIOarray_inst

5.2.4 Design

Per poter utilizzare la periferica definita nel capitolo, è necessario creare un design in cui essa viene istanziata e connessa all'ip core del processing system. Nel design mostrato in [Figura 5.2](#), sono stati istanziati 3 GPIO connessi al processing system attraverso il bus AXI. Ogni GPIO è stato configurato in modo da poter essere connesso a 4 pin, in modo tale da connettere un GPIO ai led, uno agli switch e uno ai bottoni presenti sulla baard, cosicché da poter utilizzarli per poter testare le librerie e gli esempi sviluppati.

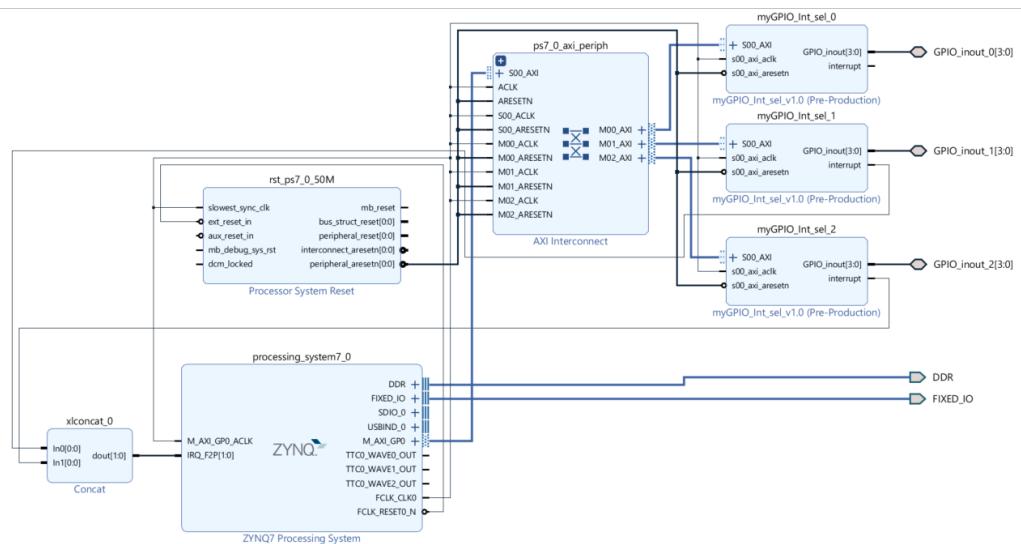


Figura 5.2: GPIO block diagram

Nel design in [Figura 5.2](#) oltre al processing system e alle istanze dei GPIO sono presenti anche altri elementi necessari ai fini dell'interconnessione tra PL e PS, tra questi troviamo il componente *Concat* (chiamato *xlconcat* nel design) realizzato dalla **Xilinx**. Quest'elemento permette di concatenare bus di segnali di dimensioni diverse, nel nostro caso è stato utilizzato per interconnettere la linea di interrupt del processore con le linee di interruzione generate dai nostri componenti.

5.3 Software

Nella seguente sezione verrà posta l'attenzione sugli aspetti più rilevanti nella realizzazione e utilizzo del codice fornito a corredo del documento. In particolare si eviterà di riportare il codice e la descrizione delle funzioni in

quanto già presenti all'interno dei file forniti a corredo del presente documento, situati all'interno della cartella **Zybo/GPIO/Software** che da questo momento chiameremo **\$GPIO_SW_PATH**.

5.3.1 Libreria

Per poter utilizzare al meglio la periferica realizzata è stata definita una libreria che raccoglie tutte le funzioni per configurare e accedere alla periferica, astralendo il processo d'accesso ai registri e di generazione delle maschere per settare i bit nel modo corretto. La libreria è stata realizzata in modo tale da avere un unico pacchetto di file di riferimento da aggiungere ai propri progetti per applicazioni, a livello utente, che usino il dispositivo indipendentemente dalla presenza o meno di driver.

La libreria è stata realizzata seguendo il principio della programmazione condizionale, ovvero definendo delle macro (elencate di seguito), le quali consentiranno di includere nel processo di compilazione solo il sottoinsieme di funzioni relative al tipo di interfacciamento scelto.

- **MYGPIO_BARE_METAL:** Se definita verranno compilate solo le funzioni per guidare la periferica senza sistema operativo;
- **MYGPIO_NO_DRIVER:** Se definita verranno compilate solo le funzioni per guidare la periferica su un sistema operativo ma senza l'ausilio di un driver a livello kernel per il controllo;
- **MYGPIO_UIO:** Se definita verranno compilate solo le funzioni per pilotare la periferica su un sistema operativo attraverso l'asilio del driver generico **UIO**;
- **MYGPIO_KERNEL:** Se definita verranno compilate solo le funzioni per pilotare la periferica su un sistema operativo in cui è caricato il driver modulo kernel presente in **\$GPIO_SW_PATH/kernel_module**.

Nel processo di compilazione è possibile definire queste macro con l'opzione **-D** di *gcc*.

All'interno dell'header di libreria, *mygpio.h*, della cartella **\$GPIO_SW_PATH/lib/**, sono presenti le macro, con nomi autoesplicativi, che permettono di configurare la periferica limitando la possibilità di commettere errori. Nel [Listing 5.4](#) sono presenti alcune macro contenute nel file *mygpio.h*. In particolare nel listato sono mostrate le macro che permettono all'utente di definire quale pin della periferica configurare o utilizzare, quelle per scegliere la modalità d'utilizzo e delle macro che indicano i valori logici letti e scritti sui pin esterni.

Capitolo 5. GPIO

```
1 #define GPIO_PIN_0 (uint32_t) 1 << 0
2 #define GPIO_PIN_1 (uint32_t) 1 << 1
3 #define GPIO_PIN_2 (uint32_t) 1 << 2
4 #define GPIO_PIN_3 (uint32_t) 1 << 3
5 #define GPIO_PIN_4 (uint32_t) 1 << 4
6 #define GPIO_PIN_5 (uint32_t) 1 << 5
7 #define GPIO_PIN_6 (uint32_t) 1 << 6
8 #define GPIO_PIN_7 (uint32_t) 1 << 7
9
10#define READ_MODE 0
11#define WRITE_MODE 1
12
13#define LOW 0
14#define HIGH 1
```

Listing 5.4: Alcune macro definite mygpio.h

Queste macro sono sempre incluse indipendentemente o meno dalla presenza delle macro per la compilazione condizionale citate prima.

Oltre alla libreria *mygpio* è stata realizzata anche un'altra libreria *utils* che contiene delle funzioni di utilità sia per la libreria *mygpio* che per l'applicazione che userà il device. Anche qui è stato adoperato l'approccio della compilazione condizionale e tra le funzioni più importanti troviamo quelle che permettono l'inizializzazione dei device:

- **configure_no_driver**: questa funzione mappa la pagina contenente l'indirizzo fisico della periferica nello spazio degli indirizzi del processo e ritorna l'indirizzo virtuale della periferica e l'indirizzo virtuale della pagina appena mappata.

```
1 void* configure_no_driver(int file_descriptor, void** vrt_page_addr, uint32_t
phy_address)
```

- **configure_uio**: permette di ottenere l'indirizzo virtuale della periferica. In particolare la funzione mappa la periferica nello spazio virtuale del processo e ritorna l'indirizzo virtuale della periferica. Tale mapping è possibile grazie alle informazioni sull'indirizzo fisico ricavate dal file aperto **/dev/uio** associato alla periferica, il cui descrittore è passato in ingresso.

```
1 void* configure_uio(int file_descriptor)
```

- **open_device**: questa funzione viene utilizzata quando la periferica viene utilizzata usando il driver kernel, presente in *\$GPIO_SW_PATH/kernel_module*. In particolare la funzione effettua la open del file associato alla periferica nel momento in cui il driver **myGPIOK.ko** viene caricato.

```
1 int open_device(int* file_descriptor, char* device_file)
```

5.3.2 Bare metal

Per utilizzare il dispositivo senza nessun sistema operativo è stata realizzata un' astrazione del device a livello di codice realizzando una struct come mostrato in [Listing 5.5](#).

```
1 typedef struct{
2     uint32_t MODE;
3     uint32_t WRITE;
4     uint32_t READ;
5     uint32_t GIES;
6     uint32_t PIE;
7     uint32_t IRQ;
8     uint32_t IACK;
9     uint32_t IRQ_MODE;
10    uint32_t IRQ_EDGE;
11 } myGPIO;
```

[Listing 5.5:](#) myGPIO struct

Questa struttura astrae completamente l'accesso ai registri.

Per inizializzare il device è possibile utilizzare la funzione *myGPIO_init*. Per poter usufruire del meccanismo delle interruzioni, la libreria offre la possibilità di configurare in modo opportuno il GIC (generic interrupt controller); è possibile registrare gli handler delle funzioni da richiamare quando si solleva la linea d'interruzione.

A tal fine sono mostrate nel [Listing 5.6](#) le seguenti funzioni.

```
1 uint32_t gic_enable(uint32_t gic_id,XScuGic* gic_inst);
2 uint32_t gic_register_interrupt(XScuGic* gic_inst, uint32_t interrupt_line, void*
    interrupt_handler);
```

[Listing 5.6:](#) funzioni per abilitare il GIC e registrare un handler di gestione dell'interruzione

E' importante ricordare che nella funzione che gestisce l'interruzione è necessario abbassare la linea di interruzione utilizzando la funzione *myGPIO_clear_irq*.

E' possibile trovare un esempio d'utilizzo, documentato, della libreria in *\$GPIO_SW_PATH/Userspace_examples/bare_metal*.

5.3.3 No driver

Per utilizzare la periferica con un sistema operativo ma senza utilizzare alcun driver a livello kernel è necessario mappare la pagina di memoria che contiene

l'indirizzo fisico della periferica nello spazio d'indirizzamento del processo per poi fare la **init** utilizzando l'indirizzo virtuale ottenuto in seguito al mapping, dopodiché sarà possibile usare la periferica come nel caso bare metal ma senza la possibilità di poter utilizzare le interruzioni che richiedono la presenza di un driver a livello kernel.

Nel Listing 5.7 è mostrato come è stata inizializzata la periferica nel caso no driver nell'esempio presente nella cartella `$GPIO_SW_PATH/Userspace_examples/no_driver`.

```

1 int descriptor = open ("/dev/mem", O_RDWR);
2 if (descriptor < 1) {
3     perror(argv[0]);
4     return -1;
5 }
6 void* vrt_page_addr;
7 void* vrt_gpio_addr = configure_no_driver(descriptor,&vrt_page_addr,ADDR_LED);
8 myGPIO* led = myGPIO_init(vrt_gpio_addr);

```

Listing 5.7: Inizializzazione periferica nel esempio no driver

5.3.4 UIO

Un modo semplice per gestire le interruzioni di un dispositivo senza dover necessariamente realizzare un proprio driver a livello kernel consiste nell'utilizzo del driver generico UIO (Userspace I/O).

Questo driver permette di gestire le interruzioni della periferica in un modo molto semplice e in alcuni casi limitato.

Ottenuto l'indirizzo virtuale della periferica tramite la `configure_uio` mostrata prima, il pilotaggio della periferica è analogo al caso no driver. In più al caso no_driver è possibile usufruire del meccanismo delle interruzioni.

Ciò è possibile perché il modulo prevede di gestire la periferica come se fosse un dispositivo a caratteri, generico, con l'eccezione che l'accesso ai registri avviene in modo diretto e non utilizzando le primitive *read* e *write*, che però vengono utilizzate rispettivamente per mettere il processo in attesa di interruzione e per riabilitare l'interruzione dopo che è stata gestita. Anche per effettuare le operazioni appena citate sono state realizzate delle funzioni che permettono all'utilizzatore finale di guidare il dispositivo senza dover conoscere necessariamente il meccanismo delle interruzioni con il driver UIO.

```

1 int32_t wait_interrupt(int uio_descriptor, int32_t* interrupt_count);
2 int32_t reenable_interrupt(int uio_descriptor, int32_t* reenable);

```

Listing 5.8: Funzioni per utilizzare le interruzioni quando si utilizza il driver UIO per controllare la periferica.

Nella cartella `$GPIO_SW_PATH/Userspace_examples/uio` sono presenti due esempi che permettono comprendere come utilizzare la periferica usando il driver uio sia con che senza le interruzioni.

5.3.5 Driver a livello kernel

Per poter utilizzare al meglio tutti i vantaggi offerti dalla presenza di un sistema operativo è stato realizzato un modulo kernel sulla base di quello fornito dall'Ing. Barone Salvatore.

Il modulo kernel permette di controllare con una grana più fine la periferica e le code d'attesa per le interruzioni. Nell'elaborato si è effettuata una piccola modifica al modulo, spostando la **IACK** dalla funzione di read a quella di gestione dell'interruzione per poi realizzare un'estensione della libreria *myGPIO* affinché le funzionalità offerte fossero disponibili anche con l'ausilio del modulo kernel.

Differentemente dalla versione classica della libreria in cui la funzione di inizializzazione prevede in ingresso un puntatore all'indirizzo virtuale o fisico della periferica, nell'estensione con modulo kernel è previsto in ingresso alle funzioni il descrittore del file `/dev/MyGPIOK`.

Nel Listing 5.9 sono elencate le principali funzioni per leggere il registro READ, le prime due implementano una read non bloccante, quindi con l'invocazione si leggerà direttamente il registro READ della periferica, mentre l'ultima effettua la read solo dopo che la periferica ha sollevato un'interruzione.

```
1 uint8_t myGPIO_read_pin_k(int descriptor, uint32_t GPIO_pin);  
2 uint32_t myGPIO_read_k(int descriptor);  
3 uint32_t myGPIO_read_bloc_k(int descriptor);
```

Listing 5.9: Funzioni utilizzate per leggere il registro READ della periferica.

Nella cartella `$GPIO_SW_PATH/Userspace_examples/kernel_mod` è presente un esempio UserSpace che permette di comprendere il flusso di invocazione delle funzioni della libreria per controllare la periferica.

Nota: è necessario configurare opportunamente il dtb e caricare il driver con insomod.

Capitolo 6

UART

6.1 Introduzione

Quando si parla di UART ci si riferisce ad un dispositivo di comunicazione seriale. Nell'esempio specifico è stato realizzato in un ip core in VHDL a partire dal codice diffuso dalla Digilent. In questo modo, interconnettendo il dispositivo correttamente, è stato possibile sviluppare diverse tipologie di esempi di utilizzo, a partire da quello privo di sistema operativo (bare metal), per poi sviluppare esempi utilizzando la periferica da un sistema operativo con una logica no driver oppure tramite driver uio, per terminare con lo sviluppo e utilizzo di un modulo kernel appositamente progettato per gestire la periferica realizzata.

6.2 Hardware

6.2.1 Componente base

Come anticipato, il componente base è fornito dalla Digilent, e in [Figura 6.1](#) è mostrata l'interfaccia del componente fondamentale per capirne il funzionamento.

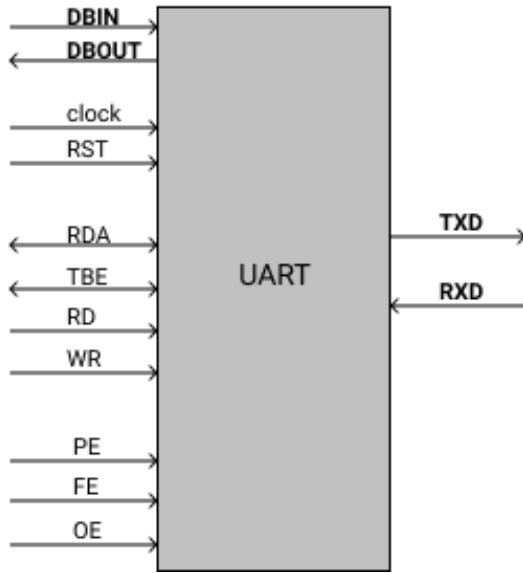


Figura 6.1: UART

In particolare:

- **TXD, RXD**: linee di trasmissione e ricezione verso l'esterno, collegate direttamente alle scan chain collegate ai bus DBOUT e DBIN;
- **DBIN, DBOUT**: linee di 8 bit che conterranno rispettivamente il byte da trasmettere o ricevuto. Direttamente collegate alle scan chain connesse a RXD e TXD;
- **Clock, RST**: clock proveniente dall'esterno e segnale di Reset.
- **RDA (Read Data Available)**: flag asserito al completamento della ricezione di un carattere;
- **TBE (Transfer Bus Empty)**: flag che indica che il contenuto di DBIN è stato trasmesso e che è possibile effettuare una nuova trasmissione. E' asserito al termine di una trasmissione.
- **RD, WR**: Segnali di controllo, quando RD è asserito, dall'esterno, indica che il dato ricevuto è stato consumato ed è quindi possibile effettuare una nuova ricezione. WR invece viene alzato per segnalare che il dato da trasmettere è stato caricato sul registro connesso al bus DBIN ed è iniziare la trasmissione. **NB.** il flag TBE non potrà tornare al livello logico alto fintanto che WR è asserito.

- **PE, FE, OE (Parity, Framing, Overrun Errors)**: flag di errore.

NB. tutti i flag vengono resettati quando RD viene posto al livello logico alto.

6.2.2 AXI wrapper

Come per il GPIO, anche in questo caso l'interfacciamento tra PL e PS avviene tramite AXI (Advanced eXtensible Interface) utilizzando il componente wrapper messo a disposizione dalla Xilinx e collegando opportunamente l'UART con l'interfaccia. Per prima cosa è stato necessario mappare in memoria tutti i segnali utili a pilotare il dispositivo, per effettuare ciò si è scelto di raggruppare i segnali nei registri come mostrato in figura [Figura 6.2](#).

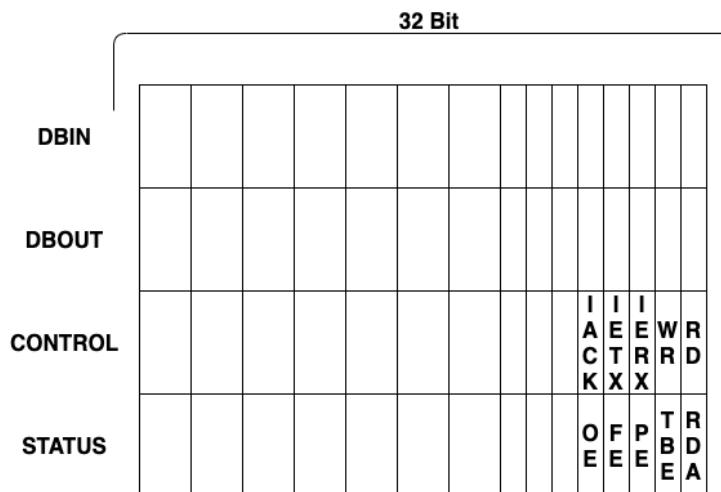


Figura 6.2: Registri Uart

Come si può notare si è deciso di utilizzare 4 registri:

- **DBIN** (R/W, offset +0x0): contiene il dato da trasmettere presente nella parte bassa;
- **DBOUT** (R, offset +0x4): contiene il dato letto, utilizza solo gli ultimi 8 bit;
- **CONTROL** (R/W, offset +0x8): registro di controllo contenente tutti i segnali utili a comandare ed abilitare le funzionalità del dispositivo. In particolare *IACK* è il segnale di interrupt ack e dovrà essere posto '1' alla fine del servizio di una interruzione di **trasmissione**, l'effetto sarà appunto quello di abbassare la linea di interruzione relativa al

termine dell'invio. *IETX* e *IERX* sono rispettivamente i due segnali di abilitazione delle interruzioni in trasmissione (quindi al termine di un invio) e di ricezione;

- **STATUS** (R, offset +0xC): registro di stato in sola lettura, qui si trovano tutti i flag di errore insieme a TBE e RDA, citati nel paragrafo precedente.

Questi registri sono stati collegati tramite AXI ed i segnali contenuti al loro interno opportunamente associati a quelli dell'interfaccia fornita dal componente UART. Per maggiori dettagli consultare il codice VHDL a disposizione in *Zybo/UART/Hardware/VHDL/*.

Interruzioni

E' necessario porre particolare attenzione al caso di utilizzo dell'UART con interruzioni abilitate. I segnali d'interruzione esterni sono connessi ai flag 'RDA' e 'TBE'. In particolare 'RDA' ha esattamente il comportamento di un segnale di interruzione, infatti sarà alto al termine di una ricezione, e sarà abbassato al consumo del dato con 'RD', che può allora essere assimilato come una IACK in **ricezione**. Per quanto riguarda 'TBE' invece è necessario fare alcune precisazioni. 'TBE' sarà sempre alto quando non si trasmette nulla, invece sarà basso durante una trasmissione. Per poter ricavare un segnale di interruzione da 'TBE' è stato allora necessario aggiungere un processo al componente wrapper, che si riporta di seguito.

```

1 -- Process che genera l' interruzione in trasmissione
2 process (state_reg(1), control_reg(4))
3 begin
4     -- Se il bit TBE si è alzato allora è possibile sollevare l'
5     -- 'interruzione
6     if rising_edge(state_reg(1)) then
7         IRQ_T <= '1';
8     end if;
9     -- Se l'utente alza IACK allora il bit di interruzione deve
10    -- essere abbassato
11    if control_reg(4) = '1' then
12        IRQ_T <= '0';
13    end if;
14 end process;

```

Listing 6.1: UART_process

Come si può notare, il segnale di interruzione in trasmissione '*IRQ_T*', sarà alzato sul fronte di salita di 'TBE' (state_reg(1)) e, come accennato in precedenza, sarà posto a '0' con 'IACK' (control_reg(3)).

Ai fini della comprensione del materiale fornito è utile fare una ulteriore precisazione. Si sono sviluppate due tipologie diverse di interfacce, *UART_1_interrupt* e *UART_2_interrupt*. La differenza risiede semplicemente nel numero di segnali di interrupt forniti all'esterno.

Infatti nel primo componente vi sarà un solo segnale di interruzione (ponendo in OR i due segnali interni) e dunque sarà necessario consultare il registro di stato per comprendere la natura di una interruzione.

Nel secondo design invece vengono forniti due segnali di interruzione separati, uno in ricezione e uno in trasmissione.

6.2.3 Design

In seguito al Packaging dell'ipcore creato si è costruito il Block diagram riportato in [Figura 6.3](#) che mostra le interconnessioni per una *UART_1_interrupt*.

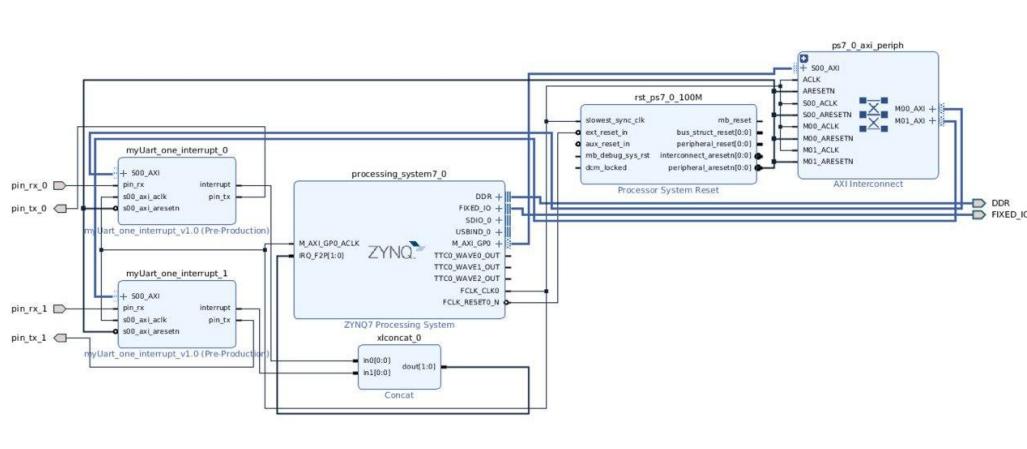


Figura 6.3: UART block diagram

Dato l'utilizzo di 2 dispositivi UART si è dovuto ricorrere ad un ulteriore componente **concat** che prende le singole linee d'interruzione dei componenti per, appunto, concatenarle. L'uscita sarà posta in ingresso al PS e, dopo aver abilitato le interruzioni, verranno mappate le vere e proprie linee di interruzione.

E' infine necessaria la scrittura dei constraints che consentono di mappare i pin di uscita in trasmissione/ricezione del design con i pin fisici della board. A tal proposito risulta utile la consultazione del Reference Manual.

6.3 Software

Dalla creazione del design Hardware è stato possibile esportare tramite Vivado il file **.xsa** che fornire una descrizione dell'hardware realizzato precedentemente.

Successivamente si è pensato di pilotare la periferica seguendo 4 approcci distinti, che tratteremo di seguito.

6.3.1 Libreria

La libreria utilizzata per la gestione del dispositivo è disponibile e consultabile, compresa di documentazione, nella cartella *Zybo/UART/UART-1-Interrupt/Software/lib*.

Così come per il GPIO anche per l'UART si è deciso di sviluppare un'unica libreria compatta per tutte le tipologie di utilizzo ma che consentisse di compilarne solo la parte necessaria. A tal proposito definendo una delle macro **MYUART_NO_DRIVER**, **MYUART_UIO**, **MYUART_KERNEL** è possibile decidere il set di funzioni da utilizzare.

Per poter definire la macro è possibile utilizzare l'opzione **-D** di gcc.

Consultando il file *myuart.h* è possibile accedere alla definizione della struttura myUART e di tutte le macro che identificano gli offset dei registri e dei singoli bit nei registri, così come quelle per l'abilitazione ho disabilitazione delle interruzioni.

Per comodità del lettore si è deciso di riportarle anche in questo documento.

```
1 #define CTR_RD      (uint32_t) 1 << 0
2 #define CTR_WR      (uint32_t) 1 << 1
3 #define CTR_IERX    (uint32_t) 1 << 2
4 #define CTR_IETX    (uint32_t) 1 << 3
5 #define CTR_IACK    (uint32_t) 1 << 4
6
7 #define ST_RDA      (uint32_t) 1 << 0
8 #define ST_TBE      (uint32_t) 1 << 1
9 #define ST_PE       (uint32_t) 1 << 2
10 #define ST_FE      (uint32_t) 1 << 3
11 #define ST_OE      (uint32_t) 1 << 4
12
13 #define DBIN_OFFSET      0x00
14 #define DBOUT_OFFSET     0x04
15 #define CONTROL_REG_OFFSET 0x08
16 #define STATUS_REG_OFFSET 0x0C
17
18 #define INT_EN      1
19 #define INT_DIS     0
20
```

Capitolo 6. UART

```
21 |  
22 |     typedef struct{  
23 |         uint32_t DBIN;  
24 |         uint32_t DBOUT;  
25 |         uint32_t CONTROL_REG;  
26 |         uint32_t STATUS_REG;  
27 |     } myUART;
```

Listing 6.2: UART_macro

Di seguito si fa notare che nel file config.h sono presenti le **define** utili ad individuare il dispositivo o tramite indirizzo (nel caso no_driver) o tramite nome dei file relativi agli altri due casi.

Grazie alla programmazione condizionale sarà sufficiente allora definire la giusta macro per il caso di utilizzo.

```
1 #ifdef MYUART_KERNEL  
2 #define UART_1_FILE "/dev/myUARTK0"  
3 #define UART_2_FILE "/dev/myUARTK1"  
4 #endif  
5  
6 #if defined MYUART_NO_DRIVER || defined MYUART_BARE_METAL  
7 #define UART1_ADDR 0x43c00000  
8 #define UART2_ADDR 0x43c10000  
9 #endif  
10  
11 #ifdef MYUART_UIO  
12 #define UIO_FILE_UART1 "/dev/uio0"  
13 #define UIO_FILE_UART2 "/dev/uio1"  
14 #endif
```

Listing 6.3: UART_config

Infine è necessario precisare che il caso Bare_metal è stato testato esclusivamente con interfaccia UART_2_interrupt (riferirsi alla rispettiva directory) ma che è ugualmente possibile utilizzare la libreria anche per l'altra interfaccia definendo la macro MYUART_BARE_METAL.

NB. Gli indirizzi UART1_ADDR e UART1_ADDR sono relativi all'esempio specifico e al mapping effettuato da Vivado, da modificare nel caso siano stati utilizzati indirizzi differenti.

6.3.2 Bare Metal

Il primo approccio, detto Bare Metal, non fa utilizzo di sistema operativo. Ciò implica la necessità di scrivere un programma di gestione della periferica che esegua direttamente sul processore. Ulteriore implicazione è che la ge-

stione delle interruzioni, compresa l'abilitazione e la configurazione del GIC risultano a carico dello sviluppatore del software.

A tal proposito è possibile utilizzare le funzioni della libreria realizzata che astrae questi dettagli implementativi e consente di sviluppare facilmente un programma che utilizzi le interruzioni.

Fare riferimento alla documentazione del codice fornita per ulteriori dettagli.

Si riportano di seguito le firme delle funzioni principali da utilizzare per la configurazione del GIC, in comune con il GPIO e per questo collocate nel file **utils.h**.

```
1 uint32_t gic_enable(uint32_t gic_id, XScuGic* gic_inst);
2 uint32_t gic_register_interrupt(XScuGic* gic_inst, uint32_t interrupt_line, void*
    interrupt_handler);
```

Listing 6.4: `UART_gic`

6.3.3 No Driver

L'approccio No Driver a differenza del precedente, e come tutti i successivi, fa utilizzo del sistema operativo.

Come suggerisce il nome non è un vero e proprio driver ma consente di interagire ugualmente con i registri della periferica in memoria tramite la chiamata di sistema **mmap** che consente di effettuare il mapping tra indirizzi virtuali e fisici. Tuttavia la libreria anche in questo caso consente di ignorare i dettagli e tramite una semplice funzione di inizializzazione è possibile ottenere un descrittore utile a scrivere sui registri in memoria.

NB. in questo caso non sarà possibile l'utilizzo delle interruzioni.

Da un punto di vista implementativo questo esempio non è molto differente da quello bare_metal. Come detto infatti la principale differenza risiede nell'effettuare il mapping indirizzo virtuale-fisico. A tal proposito si mostra di seguito la funzione utilizzata.

```
1 uint32_t gic_enable(uint32_t gic_id, XScuGic *gic_inst);
2 void* configure_no_driver(int file_descriptor, void** vrt_page_addr, uint32_t phy_address)
3 {
4     uint32_t page_size = sysconf(_SC_PAGESIZE);
5     uint32_t page_mask = ~(page_size-1);
6     uint32_t page_addr = phy_address & page_mask;
7     uint32_t offset = phy_address - page_addr;
8     *vrt_page_addr = mmap(NULL, page_size, PROT_READ | PROT_WRITE,
    MAP_SHARED, file_descriptor, page_addr);
```

```

9   if (*vrt_page_addr == MAP_FAILED) {
10      printf("Mapping indirizzo fisico - indirizzo virtuale FALLITO!\n");
11      return NULL;
12  }
13
14  void* vrt_addr = *vrt_page_addr + offset;
15  return vrt_addr;
16}

```

Listing 6.5: `UART_configure_no_driver`

Si fa notare che anche in questo caso la funzione è comune all'esempio GPIO e dunque definita in **utils.h**.

6.3.4 UIO

Per molti tipi di dispositivi, la creazione di un driver del kernel Linux è eccessiva. Tutto ciò che è veramente necessario è un modo per gestire un **interrupt** e fornire accesso allo spazio di memoria del dispositivo. Inoltre spesso la logica di controllo del dispositivo non deve necessariamente essere all'interno del kernel, in quanto il dispositivo non deve sfruttare nessuna delle altre risorse fornite dal kernel.

Per far fronte a questa situazione, è stato progettato, realizzato e introdotto all'interno del kernel un driver generico chiamato I/O userspace (UIO). Questo semplifica lo sviluppo e riduce il rischio di indrodurre gravi bug all'interno di un modulo kernel sviluppato da zero per il controllo della periferica.

Associato nel device tree il driver uio al controllo della periferica, verrà associato a ciascun dispositivo un file, chiamato `/dev/uio0` per il primo dispositivo, `/dev/uio1`, `/dev/uio2` e così via per i dispositivi successivi (vedi file config.h).

Si è detto quindi che con l'utilizzo di uio risulta possibile la gestione delle interruzioni senza dover utilizzare un driver specifico per la periferica. Ciò risulta possibile con l'ausilio delle funzioni *read* e *write*, chiamate sul file associato alla periferica, che rispettivamente mettono in attesa il processo in attesa di interruzione e riabilitano le interruzioni.

Come per gli altri esempi questa complessità è stata mascherata realizzando le seguenti funzioni di libreria, ancora una volta comuni a quelle realizzate per il GPIO.

```

1 int32_t wait_interrupt(int uio_descriptor, int32_t *interrupt_count){
2
3     if (read(uio_descriptor, interrupt_count, sizeof(uint32_t)) != sizeof(uint32_t)) {
4         perror("Read error!");
5         return -1;
6     }

```

```

7     return 0;
8 }
9
10 int32_t reenable_interrupt(int uio_descriptor, int32_t *reenable){
11     if (write(uio_descriptor, (void*)reenable, sizeof(uint32_t)) != sizeof(uint32_t)) {
12         perror(" Write error!");
13         return -1;
14     }
15     return 0;
16 }
```

Listing 6.6: `UART_read_write`

6.3.5 Modulo Kernel

L'ultimo approccio che è stato implementato è il più 'potente' ma di conseguenza anche il più complesso. Infatti poiché il modulo realizzato opera appunto in spazio kernel, questo rende potenziali bug molto più rischiosi poiché possono portare al fallimento dell'intero sistema, al contempo però permette una completa personalizzazione nella gestione del dispositivo.

I moduli kernel estendono le funzionalità del kernel senza bisogno di riavviare il sistema, questi infatti possono essere caricati e rimossi dal kernel a runtime.

A tal proposito si è deciso di sviluppare un modulo kernel per la gestione della periferica come se fosse un **device a caratteri**, con il quale è possibile interagire in userspace utilizzando alcune system call (es. `read`, `write`, `open` ecc...).

Tuttavia, per mantenere lo standard di sviluppo seguito per la realizzazione degli esempi precedenti, molte di queste funzioni sono incapsulate ed astratte da funzioni di più alto livello, con un nome uguale alle medesime utilizzate per gli esempi precedenti ma con `_k` aggiunto alla firma delle funzioni per distinguerle. Questo rende più semplice lo sviluppo ad un potenziale utilizzatore.

Di seguito si riportano a scopo esemplificativo le funzioni di trasmissione e ricezione a disposizione.

```

1 void myUART_transmit_int_k(int descriptor, uint8_t transmit_data);
2 void myUART_transmit_k(int descriptor, uint8_t transmit_data);
3 uint8_t myUART_read_k(int descriptor, uint32_t* status_reg);
4 void myUART_read_DBOUT_bloc_k(int descriptor, uint32_t* read_value);
```

Listing 6.7: `UART_read_write`

Capitolo 6. UART

Come si può notare le funzioni sono duplicate in modo da consentire all’utente di poter utilizzare il dispositivo sia in modalità polling che in modalità con interruzioni.

Capitolo 7

YOCTO

7.1 Introduzione

Yocto project è un progetto Umbrella¹, open source, esso ha lo scopo di costruire e mantenere validi tools e componenti necessari a realizzare SO Linux based per sistemi embedded. Il cuore di Yocto è OpenEmbedded. Nella prima sezione del paragrafo sarà presentata l'architettura di OpenEmbedded e il suo workflow di build che permette, tramite bitbake, la creazione di una distribuzione Linux per un sistema embedded. Successivamente sarà presentato un esempio pratico nel quale si creerà una distribuzione linux con sopra un package software compilato direttamente da Yocto per la board Zybo Zynq 7000.

7.2 OpenEmbedded

Poky è la distribuzione di riferimento di Yocto. Essa include l'OpenEmbedded build system, il quale mette a disposizione tutti i tools, recipes e configuration data richiesti per costruire lo stack del SO Linux.

Il workflow di OpenEmbedded integra la compilazione dei singoli software packages con il processo di creazione di una distribuzione Linux completa. Il processo di build dei software packages, in generale, segue il seguente pattern:

- **Fetch:** ottenere il codice sorgente;
- **Extract:** estrazione del codice sorgente;

¹racchiude più tecnologie e tools

- **Patch:** applicazione delle patch per fixare dei bug o aggiungere funzionalità;
- **Configure:** preparazione del processo di build per lo specifico sistema target (passaggio importante quando si vuole cross compilare);
- **Build:** compilazione;
- **Install:** copia dei binari e file ausiliare nelle opportune directory del sistema target. (ad esempio in Linux i file di configurazione vanno in /etc, i program files vanno in /usr/bin o in /usr/sbin a seconda se i programmi sono di sistema o utente)
- **Package:** raggruppare i binari e i file ausiliari per l'installazione su altri sistemi.

Il package step è necessario qualora si voglia installare il software non sul sistema host, ma su un terzo sistema creando un archivio che può essere usato dal package management system per l'installazione. Questo step può avere diverse sfaccettature: si potrebbe pensare di includere un Installer software che si occupa dell'installazione sul sistema target, oppure affidarsi direttamente a un package manager che è già installato sulla macchina target, in tal caso quindi lo step di package consiste solo nel raggruppare i file binari con dei metadata information per il package manager scelto.

Questi metodi, prima di procedere con l'installazione sul sistema target, verificano le dipendenze e le configurazioni del sistema per evitare possibili mismatch che potrebbero comportare problemi al sistema o all'applicativo. Nel caso specifico di Linux, spesso ci si affida a un package management system che fa parte della distribuzione di riferimento, piuttosto che usare un self-contained installation packages; i più comuni sono RPM e dpkg.

I passaggi precedentemente citati sono integrati nel workflow di OpenEmbedded build system affinchè i singoli software packages siano cross-compilati e installati opportunamente all'interno della distribuzione realizzata.

7.2.1 Architettura

OpenEmbedded fa uso di bitbake per l'esecuzione del workflow, la cui configurazione è determinata prevalentemente dai metadata.

Prima di mostrare il workflow è necessario capire come sono organizzati i

metadata e per fare ciò è opportuno introdurre l'architettura di OpenEmbedded. In essa è possibile individuare 3 entità principali: il build system, il build environment e i layer.

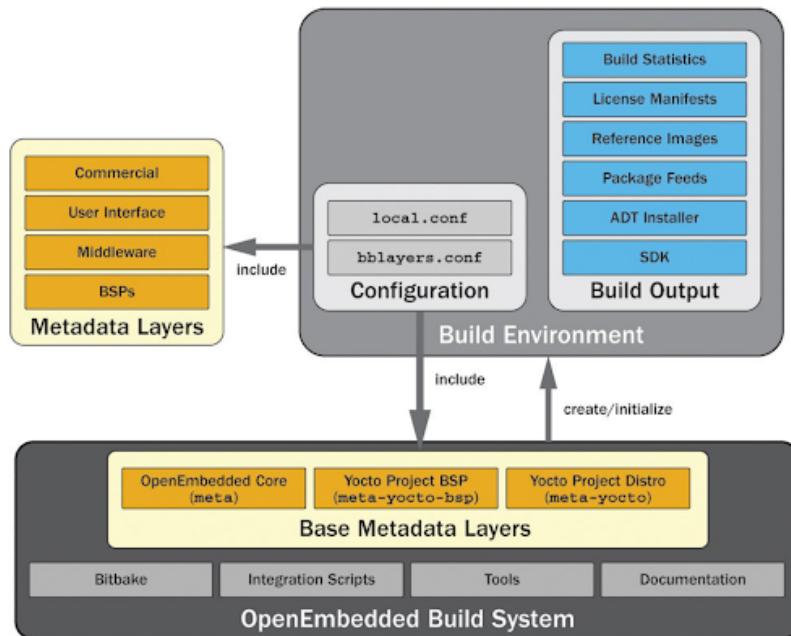


Figura 7.1: OE architecture

Nel seguito saranno approfonditi i componenti dell'architettura.

OE build system

Build System

Il build system è l'insieme dei tools, layers e recipes fondamentali, in esso è contenuto anche bitbake.

Il build system è composto da

- **bitbake build engine**: il motore che permette di eseguire il flusso di processazione guidato dei metadata files (lo possiamo vedere come un make). **Note**: Se è già presente bitbake nel build host bisogna disinstallarlo prima di installare il build system;
- **tools & scripts**: gli strumenti per creare layer e inizializzare build environment (*oe-init-build-env*);
- **Base metadata layers**: un minimal set di metadata layers;

- **OpenEmbedded Core:** l'insieme di metadata files richiesti da BitBake per costruire images;
- **meta-yocto-bsp:** layer che contiene i board support package supportati di default;
- **meta-yocto:** layer contiene le configurazioni e le ricette per la generazione della distro di base.

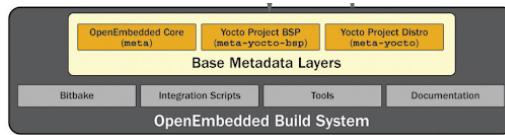


Figura 7.2: OE build system

Tra gli script del build system (cartella scripts) è possibile lanciare quello per costruire un environment: *oe-init-build-env <buildenvname>*.

Build environment

Un build system può essere associato a un numero arbitrario di build environment, ma un build environment può essere associato con un solo build system. Il build environment è possibile vederlo come il container che contiene la configurazione per la generazione della distribuzione Linux che si sta realizzando.

```
yocto@yocto-dev:~/yocto$ tree -L 2 build
x86/
+-- bitbake.lock
+-- cache
|   +-- bb_codeparser.dat
|   +-- bb_persist_data.sqlite3
|   +-- local_file_checksum_cache.dat
+-- conf
|   +-- bblayers.conf
|   +-- local.conf
|   +-- sanity.info
+-- tmp
    +-- abi.version
    +-- buildstats
    +-- cache
    +-- deploy
    |   +-- images
    |   +-- licenses
    |   +-- deb
    |   +-- ipk
    |   +-- rpm
    +-- log
    +-- qa.log
    +-- saved_tmpdir
```

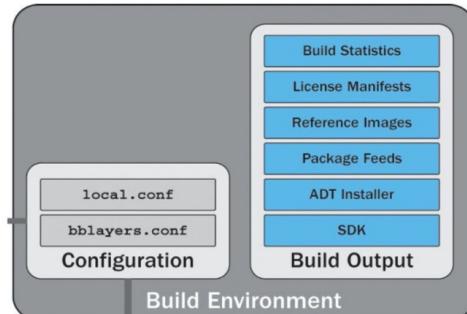


Figura 7.3: Struttura del Build environment

Di seguito sono descritti i file di configurazione presenti nella cartella conf.

- **bblayer.conf**

- **BBLAYERS**: è una variabile che ci permette di indicare i path dei layer che vogliamo includere nell’ ambiente di build;
- **BBPATH**: è una variabile inizializzata con TOPDIR (sarebbe la top level directory del build environment)

```
# LCONF_VERSION: version number for bblayers.conf
# It is increased each time build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "6"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= "
/absolute/path/to/poky/meta \
/absolute/path/to/poky/meta-yocto \
/absolute/path/to//poky/meta-yocto-bsp \
"
BBLAYERS_NON_REMOVABLE ?= "
/absolute/path/to/poky/meta \
/absolute/path/to/poky/meta-yocto \
"
```

Figura 7.4: bblayer.conf

- **local.conf**: contiene le variabili utilizzate per configurare la distro. Tra queste si trova quella necessaria per specificare l’architettura del sistema target e quella per indicare i pacchetti da aggiungere nella distro realizzata.

Tutti gli output di build sono salvati nella cartella **tmp** del build environment.

Tra gli output più importanti del processo di build troviamo i log, la cache necessaria a ottimizzare il processo di compilazione anche quando si cambia architettura del sistema target e ovviamente la distribuzione generata a partire dalle configurazioni contenute nel build environment.

Metadata layer structure

Un build environment include più metadata layer. I metadata layers sono dei contenitori per raggruppare e organizzare in entità logiche i recipes (le ricette), le classi, i configuration files e altri metadata.

Di norma la nomenclatura prevede meta come prefisso nel nome del layer.

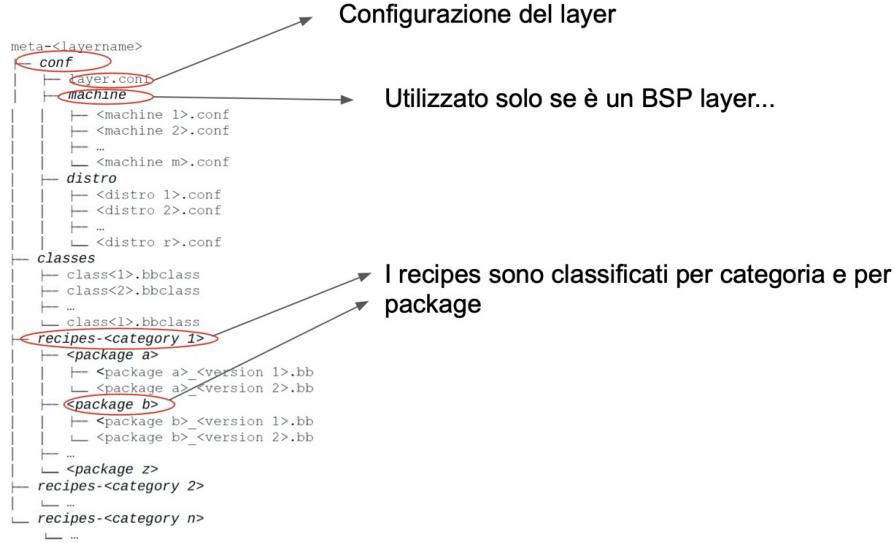


Figura 7.5: layout di un layer

All'interno di un layer è sempre presente una cartella **conf** che contiene le configurazioni del layer e all'interno di essa è sempre presente un file chiamato *layer.conf*.

In quest'ultimo sono sempre presenti le seguenti variabili:

- **BBPATH**: update della variabile globale di environment; permette di aggiungere la directory del layer (il path) al BBPATH;
- **BBFILES**: update della variabile globale di environment; permette di aggiungere i recipe files al BBFILES;
- **BBFILE PRIORITY**: permette di impostare la priority del layer (da 1 a 10). Se due layer hanno la stessa priorità allora per determinare la priorità si considera l'ordine di assegnazione alla BBLAYERS variable nel file bblayers.conf (nell'ambiente di build);
- **LAYERDEPENDENS**: permette di specificare le dipendenze del layer da altri layers (è possibile specificare anche la versione del layer).

```
# Add the layer's directory to BBPATH
BBPATH =. "${LAYERDIR}:"

# Add the layer's recipe files to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"

# Add the name of the layer to the layer collections
BBFILE_COLLECTIONS += "layername"

# Set the recipe file search pattern
BBFILE_PATTERN_layername = "^${LAYERDIR}/"

# Set the priority of this layer
BBFILE_PRIORITY_layername = "5"

# Set version of this layer
# (should only be incremented if changes break compatibility)

LAYERVERSION_layername = "2"

# Specify other layers this layer depends on. This is a white space-
# delimited list of layer names. If this layer depends on a particular
# version of another layer, it can be specified by adding the version
# with a colon to the layer name: e.g., anotherlayer:3.
LAYERDEPENDS_layername = "core"
```

Figura 7.6: layout di un layer

7.2.2 Workflow

Il workflow di OpenEmbedded integra le varie build dei singoli software packages con il processo di creazione di una distribuzione Linux completa. OpenEmbedded fa uso di bitbake per l'esecuzione del workflow, la quale configurazione è determinata dai metadata.

Di seguito sono descritti brevemente i passi del workflow.

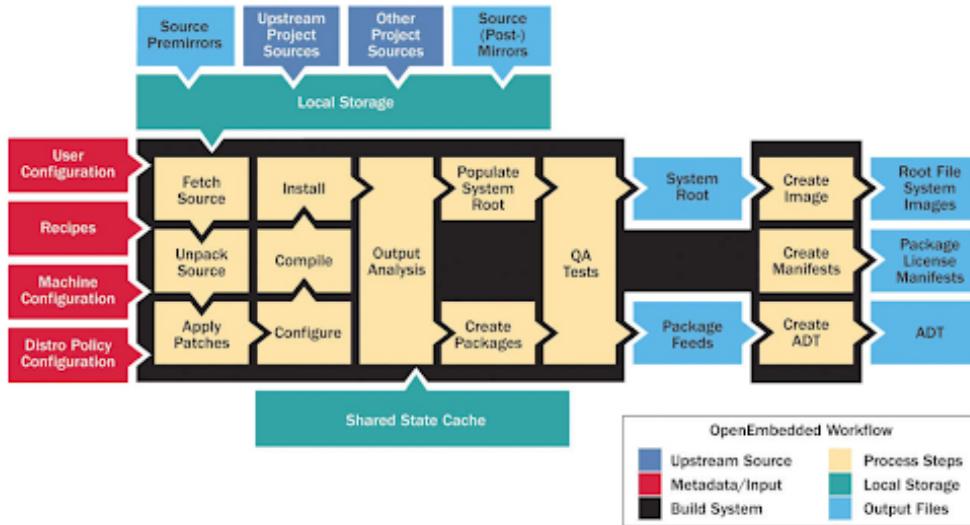


Figura 7.7: Workflow

- **Source fetching:** i recipes indicano la locazione dei sorgenti in modo tale che bitbake possa fare il loro retrieve e conservarli localmente. L'ordine di retrieve è il seguente:
 - locale: cerca i sorgenti tra quelli già scaricati;
 - Pre Mirrors: cerca sulla lista di server preconfigurati;
 - Upstream repositories: i recipes specificano la locazione dei sorgenti includendo le loro URI nella SRC_URI variable;
 - Mirrors: cerca i sorgenti su una lista secondaria di server.
- **Source Unpacking and patching:** una volta che i sorgenti sono scaricati, essi sono estratti nel local build environment. Spesso i source packages non sono adatti per la cross building, e quindi vengono applicate delle apposite patch di integrazione che rendono il sorgente cross-compilabile;
- **Configure, compile, and install:** configurazione e cross-compilazione.
- **Output analysis and packaging:** il software generato e installato nell'environment, nello step precedente, viene categorizzato in base alle sue funzionalità: runtime files, debug files, development files, documentation, locals.
Ciò permette di dividere i file in vari package fisici. Questi sono generati tenendo in considerazione il package management system scelto

per il sistema target, specificato nella variabile PACKAGE_CLASSES nel local.conf del build environment.

- **Image creation:** creazione dell'immagine, gestita da core image class. La variabile IMAGE_INSTALL contiene lista di package aggiuntivi da includere nell'image generata.

7.3 Jumpstarting

7.3.1 Requisiti

Per poter utilizzare yocto è necessario disporre dei seguenti requisiti:

- una distribuzione linux **compatibile**, fondamentale per evitare problemi durante il processo di compilazione;
- 50 Gb di spazio liberi, in modo tale da poter sfruttare al meglio il sistema di caching utilizzato durante la compilazione;
- i pacchetti che possono essere installati usando il comando seguente.

```
sudo apt-get install gawk wget git-core diffstat unzip  
texinfo gcc-multilib build-essential chrpath socat  
libsdl1.2-dev xterm
```

Inoltre è necessario fare il download di una release di yocto, scaricabile dal [repository ufficiale](#).

La scelta della release è molto importante in quanto gli sviluppatori garantiscono la compatibilità dei propri layer soltanto con determinate versioni e cambiando versione i risultati potrebbero essere inattesi.

La versione più recente attualmente supportata da tutti i layer necessari per compilare la distro personalizzata con yocto è la *3.0* il cui codename è **zeus**.

Per consultare la lista dei codename è possibile visualizzare il [wiki ufficiale](#). Per scaricare la release scelta è possibile utilizzare il seguente comando:

```
export CODENAME="zeus"  
mkdir ~/yocto && cd ~/yocto  
git clone -b ${CODENAME} git://git.yoctoproject.org/poky poky-  
${CODENAME}
```

Non resta che inizializzare il build environment e per farlo è necessario eseguire il seguente comando:

```
source oe-init-build-env zynq
```

dopo aver eseguito il comando, sarà possibile utilizzare gli strumenti presenti in yocto e progettare la propria distro di Linux.

7.3.2 Layer Xilinx

Per poter compilare una distro compatibile con l'hardware della zybo è necessario scaricare e aggiungere i layer della **Xilinx** che forniscono il BSP (Board Support Package) per la compilazione.

Per scaricare i meta è necessario posizionarsi nella repo di poky precedentemente scaricata e clonare la repo ufficiale specificando nel branch il codename della versione di yocto che si sta utilizzando.

Per effettuare queste operazioni è possibile utilizzare i seguenti comandi:

```
cd ~/yocto/poky-${CODENAME}
git clone -b ${CODENAME} https://github.com/Xilinx/meta-xilinx
.git
```

All'interno del repository appena scaricato sono presenti le seguenti cartelle:

- **meta-xilinx-bsp**: layer che fornisce il supporto per compilare distribuzioni per prodotti della Xilinx come le zybo;
- **meta-xilinx-contrib**: layer che contiene ulteriori recipie di supporto a quello precedente;
- **meta-xilinx-standalone**: layer che ha lo scopo di aumentare la funzionalità Yocto per fornire una toolchain Baremetal e le basi per la creazione dei componenti embeddedsw legati all'hardware, come il supporto per la generazione del devicetree.

7.3.3 Aggiungere sorgenti personalizzati

La struttura a layer di Yocto prevede che per aggiungere del software personalizzato venga inserito il layer contenente la recipie per la compilazione del software.

Nel Listing 7.1 è mostrata, come esempio, la struttura di una semplice ricetta.

```
1 LICENSE = "LGPLv2.1 & GPLv2 & GPLv3 & GPLv3 & Unknown"
2 LIC_FILES_CHKSUM = "file://COPYING.LIB;md5=2
d5025d4aa3495befef8f17206a5b0a1 \
3 file://COPYING;md5=59530bdf33659b29e73d4adb9f9f6552 \
```

```
4 file://COPYING3.LIB;md5=6a6a8e020838b23406c81b19c1d46df6 \
5 ...
6 SRC_URI="git://gcc.gnu.org/git/gcc.git"
```

Listing 7.1: recipe di un pacchetto aggiunto manualmente

Utilizzando lo script *bitbake-layers* è possibile creare e aggiungere un layer in modo del tutto automatico.

```
bitbake-layers create-layer layer-name
bitbake-layers add-layer layer-name
cd layer-name
```

Questi comandi creano la cartella del layer, la inizializzano con alcuni file di configurazione di default e aggiungono il layer al file bblayers.conf della build che si sta configurando.

Il passo successivo consiste nel creare una cartella contenente le nuove recipes e aggiungere un recipe utilizzando lo script di yocto specificando il repository da cui clonare i sorgenti o l'archivio che li contiene.

```
mkdir -p recipes-name/package-name && cd recipes-name/package-
name
recipetool create -o package-name.bb git:/package-repo.git
```

Fatto questo verrà creato un file con una struttura simile a quella vista in Listing 7.1 in modo del tutto automatico, senza dover aggiungere o modificare nulla.

Per indicare a Yocto che è necessario aggiungere questo pacchetto nella compilazione e nella distro è necessario modificare il file local.conf aggiungendo:

```
1 IMAGE_INSTALL_APPEND=" package-name"
```

Per eseguire la nuova recipe appena aggiunta è possibile eseguire il comando:

```
bitbake package-name
```

Questo comando avviera solo il processo di compilazione dei sorgenti, per poter aggiungere i binari al fs sarà necessario generare l'immagine come mostrato successivamente.

7.3.4 Configurazione e compilazione

Prima di lanciare bitbake, che si occupera di compilare tutti gli elementi necessari per creare la nostra distro, è necessario modificare i file di configurazione della build.

Come prima cosa è necessario aggiungere i layer di **Xilinx** modificando il file in */yocto/poky-zeus/zynq/conf/bblayers.conf* aggiungendo a BBLAYERS i path dei tre layers della **Xilinx**, come fatto nel Listing 7.2.

```

2 POKY_BBLAYERS_CONF_VERSION = "2"
3 BBPATH = "${TOPDIR}"
4
5 BBFILES ?= ""
6
7 BBLAYERS ?= " \
8 /home/se/yocto/poky-zues/meta \
9 /home/se/yocto/poky-zues/meta-poky \
10 /home/se/yocto/poky-zues/meta-yocto-bsp \
11 /home/se/yocto/poky-zues/meta-xilinx/meta-xilinx-bsp \
12 /home/se/yocto/poky-zues/meta-xilinx/meta-xilinx-contrib \
13 /home/se/yocto/poky-zues/meta-xilinx/meta-xilinx-standalone \
14 "
```

Listing 7.2: bblayers.conf configurato per aggiungere i layer di Xilinx

Successivamente è necessario specificare la terget machine per cui si sta realizzando la distro, modificando la variabile \$MACHINE presente nel file */yocto/poky-zeus/zynq/conf/bblayers.conf*, come mostrato nel Listing 7.3

```

15 #MACHINE ??= "qemux86-64"
16 MACHINE ??= "zybo-zynq7"
```

Listing 7.3: local.conf configurato per compilare la distro per la zybo

Infine è possibile avviare il processo di compilazione utilizzando il seguente comando:

```
bitbake core-image-minimal
```

7.3.5 Caricamento sulla board

Terminato il processo di compilazione, i file generati saranno presenti nella cartella *tmp/deploy/images/zybo-zynq7* della build. Nel nostro caso il path sarà */home/se/yocto/poky-zues/zybo/tmp/deploy/images/zybo-zynq7*. In questa cartella ci sono molti file e collegamenti, quelli che ci interessano sono:

- **boot.bin**: file contenete il fsbl;
- **u-boot.img**: contiene u-boot.bin (il binario compilato di u-boot loader) con un header addizionale usato per determinare come e dove caricare ed eseguire U-Boot durante la fase di boot ROM;

- **uEnv.txt**: definisce la configurazione di avvio che verrà letta e usata da u-boot;
- **zynq-zybo.dtb**: devicetree necessario al kernel linux per conoscere l'hardware su cui è in esecuzione;
- **core-image-minimal-zynq7.tar.gz**: root filesystem generato a partire da poky.

Prima di procedere con la copia è necessario modificare il file **uEnv.txt**, in particolare è necessario rimuovere la voce **run loadfpga &&** dalla variabile **uenvcmd**, come mostrato di seguito:

```
uenvcmd= run bootkernel
```

Per poter eseguire linux sulla board è necessario partizionare la scheda microSD come mostrato in [sezione 4.6](#) e posizionare il jumper **JP5** sulla voce microSD per informare l'FSBL di interfacciarsi con l'SD.

Una volta creata la partizione è necessario copiare i file rispettando il seguente schema:

- Partizione di **BOOT**:
 - boot.bin
 - u-boot.img
 - uImage
 - uEnv.txt
 - zynq-zybo.dtb
- Partizione del **root filesystem**: estrarre il contenuto di core-image-minimal-zynq7.tar.gz nella partizione.

```
tar -C path-della-partizione -xzvf core-image-minimal-zynq7.tar.gz
```

7.4 Aggiungere ulteriori pacchetti utilizzando layer esistenti

Prima di creare una nuova ricetta per aggiungere del nuovo software alla propria build è utile cercare se esiste in qualche layer la ricetta già creata.

Capitolo 7. YOCTO

Per fare ciò è possibile visitare il repository di layer di OpenEmbedded al seguente <https://layers.openembedded.org/>.

The screenshot shows a web interface for managing layers. At the top, there are navigation links: Branch: master, Layers, Recipes (which is selected), Machines, Classes, and Distros. There are also buttons for Submit layer and Log in. Below this is a search bar with the word 'gcc' typed into it and a 'search' button. The main area is a table with the following columns: Recipe name, Version, Description, and Layer. The data in the table is as follows:

Recipe name	Version	Description	Layer
gcc	10.1.0	GNU cc and gcc C compilers	openembedded-core
gcc	linaro-7.2	GNU cc and gcc C compilers	meta-linaro-toolchain
gcc	arm-8.2	GNU cc and gcc C compilers	meta-linaro-toolchain
gcc	arm-8.3	GNU cc and gcc C compilers	meta-linaro-toolchain
gcc	7.3.0	GNU cc and gcc C compilers	meta-luness-backports-2-8
gcc	8.3.0	GNU cc and gcc C compilers	meta-debian
gcc	8.3.0	GNU cc and gcc C compilers	meta-tegra
gcc	7.3.0	GNU cc and gcc C compilers	meta-tegra
gcc	arc	GNU cc and gcc C compilers	meta-synopsys
gcc	9.3.0	GNU cc and gcc C compilers	meta-artesyn

Figura 7.8: Ricerca di ricette su layers.openembedded.org

Utilizzando il sito è possibile individuare il nome delle ricette e dei layer da includere per installare il pacchetto.

Ad esempio per aggiungere gcc alla build sarà necessario scaricare e aggiungere il layer di openembedded-core, come mostrato in Figura 7.9 e aggiungere il nome della ricetta alla variabile come mostrato in Figura 7.10.

```
BBLAYERS ?= " \
/home/se/yocto/poky-zeus/meta \
/home/se/yocto/poky-zeus/meta-poky \
/home/se/yocto/poky-zeus/meta-yocto-bsp \
/home/se/yocto/poky-zeus/meta-gcc \
/home/se/yocto/poky-zeus/meta-openembedded/meta-python \
/home/se/yocto/poky-zeus/meta-openembedded/meta-networking \
/home/se/yocto/poky-zeus/meta-openembedded/meta-oe \
/home/se/yocto/poky-zeus/meta-xilinx/meta-xilinx-bsp \
/home/se/yocto/poky-zeus/meta-xilinx-tools \
/home/se/yocto/poky-zeus/meta-xilinx/meta-xilinx-contrib \
/home/se/yocto/poky-zeus/meta-xilinx/meta-xilinx-standalone \
"
```

Figura 7.9: bblayers.conf per installare

Capitolo 7. YOCTO

```
# CONF_VERSION is increased each time build/conf/ changes incompatibly and is used to
# track the version of this file when it was generated. This can safely be ignored if
# this doesn't mean anything to you.
CONF_VERSION = "1"

#IMAGE_INSTALL += " kernel-module-uio_pci_generic"
KERNEL_MODULE_AUTOLOAD += "uio_pci_generic"
IMAGE_INSTALL_append = " packagegroup-core-buildessential kernel-modules gcc ssh sshd "
```

Figura 7.10: local.conf con per installare gcc

Inoltre come si evince in [Figura 7.10](#) nella build utilizzata sono stati aggiunti anche i pacchetti per build-essential e ssh.