



UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di laurea triennale in Ingegneria Elettronica

**Sviluppo di un'interfaccia USB-BDM su RP2040 per la programmazione
di microcontrollori Freescale**

**Development of a USB-BDM interface on RP2040 for Freescale
microcontroller programming**

Relatore:

Prof. **Biagetti Giorgio**

Tesi di Laurea di:

Lorenzo Bartolini

A.A. 2021/2022

Sommario

Introduzione	4
1 – Il protocollo BDM	5
1.1 Generalità	5
1.2 Modalità Active Background	5
1.3 Comandi BDM	5
1.4 Connettore BDM	7
1.5 Pin BKGD	7
1.6 Dettagli del protocollo di comunicazione	7
1.7 Throughput	9
1.8 Comando di sincronizzazione	9
2 – Il protocollo USB	11
2.1 Storia del protocollo	11
2.2 Specifiche	11
2.3 Formato dei dati	12
2.4 Protocollo	17
2.5 Cattura Wireshark	20
3 – Il dispositivo: RP2040 (<i>Raspberry Pi Pico</i>)	25
3.1 RP2040	25
3.2 Modulo PIO	26
3.3 Realizzazione del protocollo BDM	33
3.4 Modulo USB	39
3.5 Libreria <i>TinyUSB</i>	40
4 – Il progetto	44
4.1 Comunicazione con la IDE: <i>Codewarrior</i>	44
4.2 Comando di sincronizzazione	46
4.3 Ricezione comandi via USB e gestione dei pacchetti	51

4.4	Esecuzione del comando USB	56
4.5	Esecuzione del comando BDM	61
4.6	Trasmissione risposta via USB	66
5	- Test e validazione sperimentale	68
5.1	Prova di scrittura sul target	68
5.2	Stima della velocità di scrittura	69
6	- Conclusioni e sviluppi futuri	70
6.1	Conclusioni	70
6.2	Sviluppi futuri	70
	Sitografia	71

Introduzione

L'obiettivo di questa tesi è quello di realizzare un'interfaccia di programmazione/debug USB per i dispositivi della famiglia HCS08 prodotti dalla Freescale Semiconductor. Tale interfaccia di debug è stata pensata come un'alternativa economica e più performante ai dispositivi ufficiali della Freescale.

A tale scopo era nato il progetto open source USBDM. Tuttavia, a causa dell'hardware utilizzato, le prestazioni che si riuscivano ad ottenere non erano ottimali.

L'idea, allora, è stata quella di progettare un'architettura simile a quanto fatto nel progetto USBDM, ma sfruttando le potenzialità di un microcontrollore più moderno, che potesse raggiungere prestazioni migliori e che, al tempo stesso, avesse un prezzo contenuto.

Si è optato per l'RP2040, in quanto dotato di un modulo per l'I/O dedicato (modulo P.I.O., Programmable Input Output). Tale modulo, come si approfondirà in seguito, al contrario di un processore general purpose, è stato pensato con un focus particolare sul determinismo e sul rispetto di temporizzazioni precise. Ciò ha permesso di ottenere delle forme d'onda estremamente accurate, anche a frequenze elevate.

Questo non sarebbe stato possibile se fosse stato utilizzato un qualunque altro microcontrollore, sia anche di ultima generazione, in quanto il segnale generato sarebbe stato condizionato dal non determinismo tipico di un processore general purpose, impegnato a gestire più processi contemporaneamente. Questo è esattamente il motivo per cui nel progetto USBDM, che impiega microcontrollori classici per generare le forme d'onda, non è stato possibile raggiungere prestazioni elevate, in termini di velocità di trasmissione.

1 - Protocollo BDM

1.1 - Generalità

Il protocollo BDM è utilizzato da molti prodotti della Freescale Semiconductor con lo scopo di trasmettere comandi di debug, come ad esempio la lettura/scrittura di locazioni di memoria, la possibilità di inserire breakpoint, fermare momentaneamente l'esecuzione del programma utente, far avanzare il programma un'istruzione alla volta e tanto altro.

Per questo progetto, è stato utilizzato per programmare la memoria del microcontrollore, attraverso una sequenza particolare di comandi, impartiti via USB all'RP2040 dal software di sviluppo Codewarrior.



Figura 1: Freescale Semiconductor - MC13213.

1.2 – Modalità Active Background

Per programmare la memoria del microcontrollore, è necessario che questo si trovi nella modalità *Active Background*, cioè una modalità di debug in cui l'MCU non sta eseguendo il programma caricato in memoria dall'utente. Per entrare in questa modalità è necessario tenere il pin BKGD al livello basso, mentre il microcontrollore sta eseguendo un *power-on reset*.

1.3 – Comandi BDM

I comandi che è possibile impartire tramite protocollo BDM sono di due tipi:

- **Comandi non intrusivi:** possono essere eseguiti anche quando il dispositivo non si trova in modalità *Active Background*. Permettono di leggere/scrivere in locazioni di memoria dell'MCU oppure accedere ai registri di `STATUS` o di `CONTROL`.
- **Comandi modalità *Active Background*:** possono essere eseguiti soltanto quando il dispositivo si trova in modalità *Active Background*. Permettono di leggere/scrivere i registri della CPU e di tenere traccia di un'istruzione alla volta. Comprendono anche i comandi per fare riprendere l'esecuzione del programma utente.

Command Mnemonic	Active Background Mode/ Non-Intrusive	Coding Structure ⁽¹⁾	Description
SYNC	Non-intrusive	n/a ⁽²⁾	Request a timed reference pulse to determine target BDC communication speed
ACK_ENABLE	Non-intrusive	D5/d	Enable handshake. Issues an ACK pulse after the command is executed.
ACK_DISABLE	Non-intrusive	D6/d	Disable handshake. This command does not issue an ACK pulse.
BACKGROUND	Non-intrusive	90/d	Enter active background mode if enabled (ignore if ENBDM bit equals 0)
READ_STATUS	Non-intrusive	E4/SS	Read BDC status from BDCSCR
WRITE_CONTROL	Non-intrusive	C4/CC	Write BDC controls in BDCSCR
READ_BYTE	Non-intrusive	E0/AAAA/d/RD	Read a byte from target memory
READ_BYTE_WS	Non-intrusive	E1/AAAA/d/SS/RD	Read a byte and report status
READ_LAST	Non-intrusive	E8/SS/RD	Re-read byte from address just read and report status
WRITE_BYTE	Non-intrusive	C0/AAAA/WD/d	Write a byte to target memory
WRITE_BYTE_WS	Non-intrusive	C1/AAAA/WD/d/SS	Write a byte and report status
READ_BKPT	Non-intrusive	E2/RBKP	Read BDCBKPT breakpoint register
WRITE_BKPT	Non-intrusive	C2/WBKP	Write BDCBKPT breakpoint register

Command Mnemonic	Active Background Mode/ Non-Intrusive	Coding Structure ⁽¹⁾	Description
WRITE_A	Active Background Mode	48/WD/d	Write accumulator (A)
WRITE_CCR	Active Background Mode	49/WD/d	Write condition code register (CCR)
WRITE_PC	Active Background Mode	4B/WD16/d	Write program counter (PC)
WRITE_HX	Active Background Mode	4C/WD16/d	Write H and X register pair (H:X)
WRITE_SP	Active Background Mode	4F/WD16/d	Write stack pointer (SP)
WRITE_NEXT	Active Background Mode	50/WD/d	Increment H:X by one, then write memory byte located at H:X
WRITE_NEXT_WS	Active Background Mode	51/WD/d/SS	Increment H:X by one, then write memory byte located at H:X. Also report status.

GO	Active Background Mode	08/d	Go to execute the user application program starting at the address currently in the PC
TRACE1	Active Background Mode	10/d	Trace 1 user instruction at the address in the PC, then return to active background mode
TAGGO	Active Background Mode	18/d	Same as GO but enable external tagging (HCS08 devices have no external tagging pin, so TAGGO is just like GO in an HCS08)
READ_A	Active Background Mode	68/d/RD	Read accumulator (A)
READ_CCR	Active Background Mode	69/d/RD	Read condition code register (CCR)
READ_PC	Active Background Mode	6B/d/RD16	Read program counter (PC)
READ_HX	Active Background Mode	6C/d/RD16	Read H and X register pair (H:X)
READ_SP	Active Background Mode	6F/d/RD16	Read stack pointer (SP)
READ_NEXT	Active Background Mode	70/d/RD	Increment H:X by one, then read memory byte located at H:X
READ_NEXT_WS	Active Background Mode	71/d/SS/RD	Increment H:X by one, then read memory byte located at H:X. Report status and data.

Figura 2: lista di comandi BDM dal datasheet di MC13213.

1.4 – Connettore BDM

Tipicamente si utilizza un dispositivo per convertire i comandi che il computer host trasmette al dispositivo target. A seconda del produttore del dispositivo, questo può comunicare con il PC in vari modi (USB, porta seriale, ...), mentre comunica con il dispositivo target attraverso un connettore a sei pin che comprende il pin **BKGD**, **RESET** e **V_{DD}**.

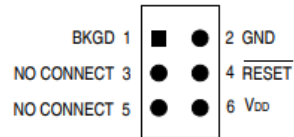


Figura 3: connettore BDM dal datasheet di MC13213.

1.5 – Pin BKGD

Il pin BKGD è utilizzato principalmente per trasmettere comandi (dall'host verso l'MCU) e dati (provenienti sia dall'host che dall'MCU). Il pin BKGD è una porta pseudo-open-drain. La linea è connessa ad un resistore di pull-up interno; non vi è alcun bisogno di connetterne uno esterno. Il protocollo prevede che vengano prodotti degli impulsi ad alta intensità e di breve durata, per riportarla in breve tempo al livello alto; la costante di tempo della linea non gioca nessun ruolo sul valore dei tempi di salita. Il pin può essere pilotato sia dall'host che dal target.

1.6 – Dettagli del protocollo di comunicazione

La comunicazione seriale avviene principalmente attraverso il pin BKGD.

1.6.1 - Inizio comunicazione

Tutte le transazioni vengono inizializzate dall'host, che genera un fronte di discesa, il quale segna l'inizio dello start bit.

1.6.2 - Frequenza di comunicazione

La frequenza di comunicazione è stabilita dal clock del dispositivo target. Per conoscere la frequenza di comunicazione, l'host può trasmettere un comando di sincronizzazione **SYNC** al target. A seguito di ciò, il target genera un impulso di sincronizzazione di risposta, attraverso il quale l'host stima la frequenza di comunicazione. La comunicazione è quindi sincrona per il target (siccome è sincronizzata con il clock interno), ma asincrona per l'host.

1.6.3 - Velocità di comunicazione

I dati vengono trasferiti ad una velocità pari a 1/16 volte la frequenza di riferimento (clock del dispositivo target).

L'interfaccia va in time-out dopo 512 colpi di clock. Quando ciò si verifica, vengono annullati tutti i comandi che erano in esecuzione, ma non viene modificata la memoria dell'MCU in alcun modo, né la modalità operativa del target. I dati vengono trasferiti a partire dal MSB.

1.6.4 - Dall'host al target

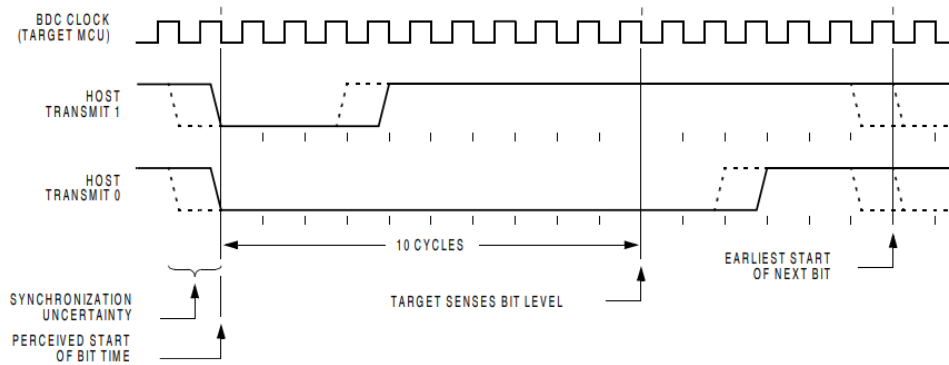


Figura 4: trasmissione di un bit dall'host al target dal datasheet di MC13213.

La trasmissione di 1 bit impiega esattamente sedici colpi di clock.

L'host genera lo start bit, mantenendo la linea bassa per almeno quattro colpi di clock, dopodiché può decidere di farla ritornare alta, nel caso voglia trasmettere un 1, oppure continuare a mantenerla bassa, nel caso voglia trasmettere uno 0. Il dispositivo target campiona la linea dopo circa dieci colpi di clock dallo start bit.

1.6.5 - Dal target all'host (trasmissione di un uno)

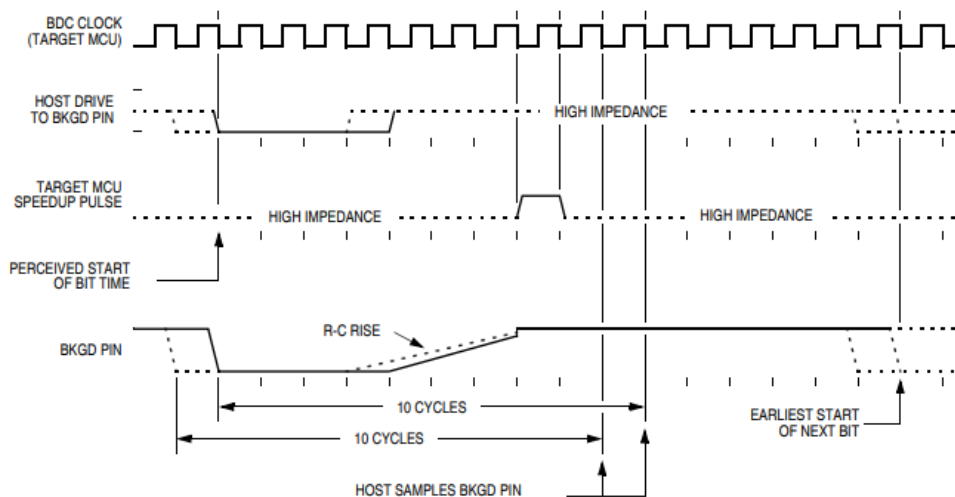


Figura 5: trasmissione di un uno dal target all'host dal datasheet di MC13213.

L'host inizia sempre la trasmissione, generando uno start bit, informando il target che la comunicazione è iniziata. Dopo quattro colpi di clock, l'host smette di pilotare la linea, la quale torna in alta impedenza. A sette colpi di clock dallo start-bit, cioè *prima* che l'host campioni la linea, il dispositivo target genera un impulso di *speed-up*, che porta la linea al livello alto. Questa poi, verrà

campionata subito dopo, a dieci colpi di clock dallo start bit. La trasmissione termina dopo sedici colpi di clock dallo start bit.

1.6.6 - Dal target all'host (trasmissione di uno zero)

L'host inizia sempre la trasmissione generando uno start bit. A questo punto il target comincia *subito* a pilotare la linea, che viene mantenuta bassa per tredici colpi di clock. Questa verrà campionata al decimo colpo di clock, mentre si trova ancora al livello basso. In questo caso il target genera l'impulso che riporta la linea alta *dopo* che l'host ha campionato la linea.

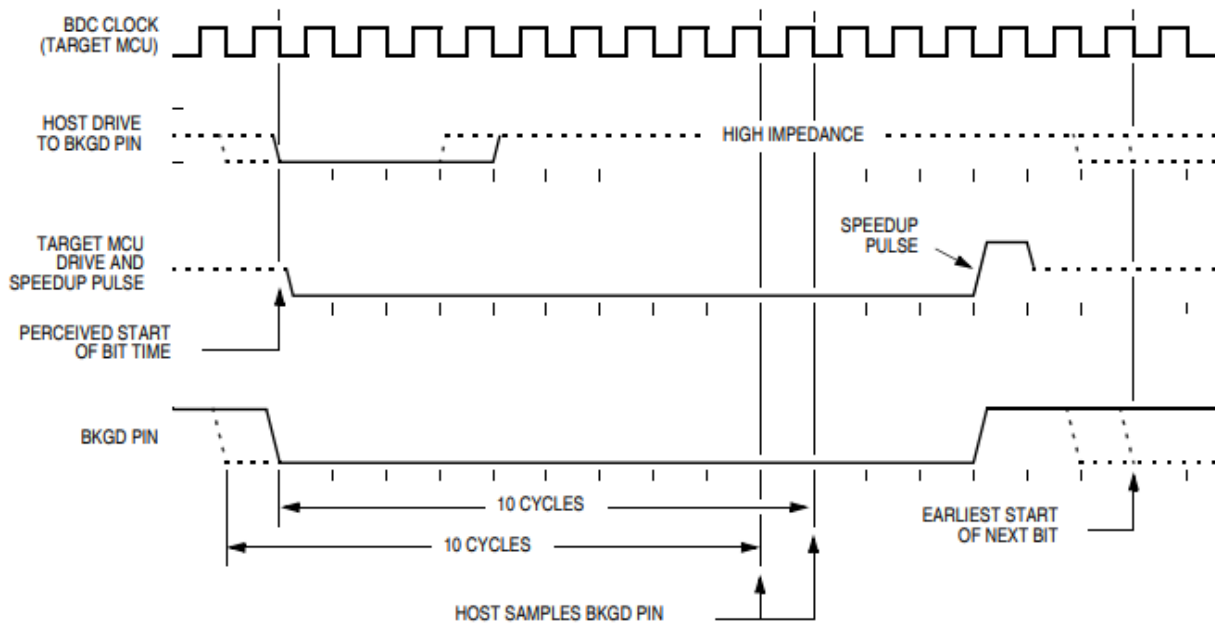


Figura 6: trasmissione di uno zero dal target all'host dal datasheet di MC13213.

1.7 – Throughput

Dal momento che per trasmettere un bit si impiegano sedici colpi di clock e che la frequenza tipica per la comunicazione BDM è di 16MHz, il throughput che è possibile ottenere con questo protocollo è di circa 1Mbit/s. Tuttavia, la frequenza del target può essere più bassa di 16MHz, quindi di conseguenza si riduce anche il throughput. Per stabilire qual è la frequenza del target, si utilizza un comando BDM speciale, detto di sincronizzazione.

1.8 – Comando di sincronizzazione

Il comando di sincronizzazione è diverso dagli altri comandi, dal momento che l'host non necessariamente conosce la frequenza di comunicazione da utilizzare fino a quando non ha analizzato la risposta del target.

Per generare il comando di sincronizzazione, l'host deve seguire i seguenti passi:

- Pilotare la linea BKGD al livello basso per 128 colpi di clock alla frequenza più bassa possibile, cioè la frequenza dell'host/16.

- Generare un impulso di *speed-up* al livello alto, che dura un ciclo di clock.
- Rilasciare la linea in alta impedenza.
- Monitorare lo stato della linea e attendere la risposta del target.

Il target, invece, dovrà:

- Attendere che la linea torni al livello alto.
- Aspettare sedici cicli di clock, a partire dal momento che la linea torna al livello alto.
- Pilotare la linea bassa per 128 colpi di clock.
- Generare un impulso di 'speed-up' al livello alto.
- Rilasciare la linea in alta impedenza.

L'host può stimare la frequenza sapendo che la linea viene mantenuta bassa per 128 periodi di clock alla frequenza del dispositivo target. Quindi sarà sufficiente calcolare il tempo in cui la linea viene mantenuta bassa e dividere per 128 per ricavare il periodo del segnale di clock del target.

$$T_{TARGET} = \frac{T_{measured}}{128};$$

$$T_{measured} = ticks * T_{HOST}$$

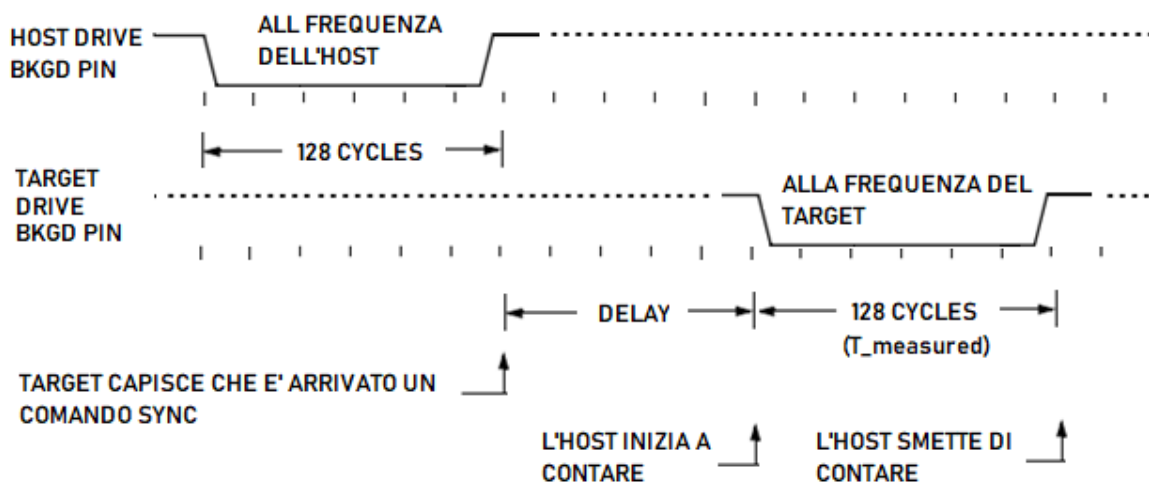


Figura 7: struttura comando sync.

2 - Protocollo USB

2.1 – Storia del protocollo

Il protocollo USB (Universal Serial Bus) è stato sviluppato da Compaq, Intel, Microsoft e NEC, a cui si aggiunsero in seguito Hewlett-Packard, Lucent e Philips. Queste compagnie fondarono la USB “Implementers Forum, Inc” o USB IF, una società non-profit avente lo scopo di definire il nuovo standard e organizzarne i futuri sviluppi.

L’obbiettivo del protocollo UB era quello di definire nuova metodologia di connessione che potesse essere utilizzata in diverse applicazioni e, eventualmente, che andasse a sostituire gli svariati metodi di connessione esistenti fino a quel momento (porta seriale, parallela, MIDI, mouse e tastiera e così via). Era importante che all’utente finale non fosse richiesto di possedere particolari conoscenze per installare un nuovo dispositivo e che il computer potesse facilmente distinguere un dispositivo dall’altro per poter utilizzare il driver software corretto.

2.2 – Specifiche

2.2.1 - Specifiche 1.0

Lo standard USB fece il suo debutto nel 1996 con la versione 1.0. Questa versione prevedeva solo due modalità. La prima, Low Speed, pensata per dispositivi di fascia bassa, che non avevano necessità di trasmettere molti dati, come ad esempio mouse e tastiere. La seconda modalità, Full Speed era originariamente pensata per supportare tutti i restanti dispositivi.

Nome	Velocità
Low Speed	1.5 Mbit/s
Full Speed	12 Mbit/s

2.2.2 - Specifiche 2.0

Agli inizi degli anni 2000 venne rilasciata una specifica aggiornata del protocollo, che ora supportava una nuova modalità, chiamata High speed, in grado di raggiungere velocità più elevate.

Nome	Velocità
High Speed	480 Mbit/s

Queste velocità stabiliscono la velocità a cui viaggiano i dati, ma non rappresentano il throughput effettivo, che sarà sempre inferiore, a causa della presenza del protocollo.

2.2.3 - Specifiche 3.0

La terza versione delle specifiche del protocollo venne rilasciata a novembre 2008, la quale venne introdotta la modalità Super Speed.

Nome	Velocità
Super Speed	5Gbit/s

Lo standard 3.0 aumentò anche la potenza trasferibile fino a 5V e 900mA.

2.2.4 - Specifiche 3.0

La versione 4.0 del protocollo USB venne rilasciata nell'agosto del 2019 e si basa sul protocollo Thunderbolt 3. Permette trasferire dati fino a 40Gbit/s e alimentare dispositivi fino a 240W. Il protocollo Thunderbolt 3 è stato sviluppato da Intel nel 2015 per supportare un'elevata velocità di trasferimento dati e video.

2.3 – Formato dei dati

2.3.1 - Pacchetti

Il pacchetto, nel protocollo USB, è l'elemento più piccolo di una trasmissione. Un pacchetto è composto da diversi campi, il primo dei quali è sempre **SYNC**, mentre l'ultimo è l'**EOP**. A seconda della tipologia del pacchetto, possono comparire uno o più dei seguenti campi:

- **SYNC**: un pacchetto comincia con dei bit di sincronizzazione che contengono informazioni sulla frequenza da utilizzare per la ricezione dei dati.
- **PID**: è l'identificatore di pacchetto (Packet Identifier). Serve a stabilire il tipo di pacchetto.
- **ADDR**: contiene l'indirizzo del dispositivo USB a cui è indirizzato il pacchetto.
- **ENDP**: specifica l'endpoint del dispositivo USB indirizzato da utilizzare.
- **DATA**: è il campo in cui sono presenti i dati (da 0 a 1024 byte).
- **CRC**: Cyclic Redundancy Check. Sono bit dedicati ad un codice di correzione dell'errore. Si utilizza un codice a 5 bit (CRC5) per i pacchetti di tipo token, mentre a 16 bit (CRC16) per quelli di tipo data.
- **EOP**: bit che segnalano la fine del pacchetto.

Esistono 17 diversi tipi di **PID**:

Tipologia	Nome	Valore <3:0>
Token	OUT	0001
	IN	1001
	SOF	0101
	SETUP	1101
Data	DATA0	0011
	DATA1	1011
	DATA2	0111
	MDATA	1111

Handshake	ACK	0010
	NACK	1010
	STALL	1110
	NYET	0110
Special	PRE	1100
	ERR	1000
	SPLIT	0100
	PING	0000

In base al **PID**, quindi, è possibile distinguere quattro tipi di pacchetti:

- **Token:** utilizzato per trasmettere pacchetti di tipo **OUT**, **IN** e **SETUP**. È sempre il primo pacchetto di una transazione. Identifica lo scopo della stessa e l'endpoint desiderato.

SYNC	PID	ADDR	ENDP	CRC5	EOP
	8 bit	7 bit	4 bit	5 bit	

- **Data:** è il tipo di pacchetto da utilizzare quando si vogliono trasferire dati.

SYNC	PID	DATA	CRC16	EOP
	8 bit	0 - 1024 byte	16 bit	

- **Handshake:** utilizzato quando si vuole trasmettere un acknowledge (**ACK**), not acknowledge (**NACK**), per segnalare che non vi è stata nessuna risposta (**NYET**) oppure che l'endpoint è bloccato o la richiesta effettuata non è supportata (**STALL**).

SYNC	PID	EOP
	8 bit	

- **SOF:** sebbene sia classificato come pacchetto token, il SOF ha una struttura diversa. Viene trasmesso una volta ogni millisecondo, nei collegamenti in modalità Full Speed, per segnalare l'inizio di una finestra temporale in cui i dati devono essere trasferiti.

SYNC	PID	Frame number	CRC5	EOP
	8 bit	11 bit	5 bit	

2.3.4 - Transazioni

Una transazione consiste nella trasmissione di due o tre pacchetti USB. Il terzo pacchetto di handshake è assente nei trasferimenti Isocroni. Esistono tre tipi di transazioni:

- **OUT:** comincia con la trasmissione di un pacchetto Token di tipo **OUT**, che segnala l'intenzione da parte dell'host di trasmettere dei dati dal dispositivo. Nel pacchetto successivo, di tipo **DATA0** o **DATA1**, l'host trasmette al dispositivo la sequenza di dati. Infine, viene trasmesso un eventuale pacchetto di Handshake di tipo **ACK**, per segnalare all'host la corretta ricezione dei dati da parte del dispositivo.

Pacchetto Token				Pacchetto Data			Pacchetto Handshake
OUT	ADDR	ENDP	CRC5	DATA0/1	PAYLOAD DATA	CRC16	ACK

- **IN:** comincia con la trasmissione di un pacchetto Token di tipo **IN**, che segnala al dispositivo che l'host ha richiesto di ricevere dei dati. Nel pacchetto successivo, di tipo **DATA0** o **DATA1**, il dispositivo trasmette all'host i dati richiesti. Infine, viene trasmesso un eventuale pacchetto di Handshake di tipo **ACK**, per segnalare al dispositivo la corretta ricezione dei dati da parte dell'host.

Pacchetto Token				Pacchetto Data			Pacchetto Handshake
IN	ADDR	ENDP	CRC5	DATA0/1	PAYLOAD DATA	CRC16	ACK

- **SETUP:** è costituita dalla successione di un pacchetto Token di tipo **SETUP**, seguito da un pacchetto Data, che può essere **DATA0** o **DATA1**, trasmessi dall'host al dispositivo, e infine un eventuale pacchetto di Handshake di tipo **ACK**.

Pacchetto Token				Pacchetto Data			Pacchetto Handshake
SETUP	ADDR	ENDP	CRC5	DATA0	8 BYTE SETUP DATA	CRC16	ACK

In azzurro sono indicati i campi che vengono trasmessi dall'host al dispositivo e in bianco quelli che viaggiano nel senso opposto.

2.3.3 - Trasferimenti

Vi sono quattro diversi modi di trasferire dati su un bus USB, ognuno dei quali ha il suo scopo e le sue peculiarità.

- **Control:** vengono utilizzati per configurare il dispositivo. Per questi trasferimenti è obbligatorio l'utilizzo dell'endpoint 0 sia **OUT** che **IN**.
- **Bulk:** i trasferimenti di tipo Bulk vengono utilizzati per trasferire grandi quantità di dati, in maniera affidabile e che non necessitano di una bassa latenza. È previsto, infatti, l'utilizzo del CRC per segnalare un errore durante il trasferimento. Se un dato viene ricevuto in

maniera errata, il bus procede alla sua ritrasmissione. Vengono utilizzati ad esempio per trasferire dati da e verso memorie di archiviazione di massa.

- **Interrupt:** utilizzati quando si ha necessità di trasmettere e ricevere piccole quantità di dati, come per esempio quando si deve tenere sotto controllo lo stato di una periferica. Utilizzati principalmente nei dispositivi di input, come mouse e tastiere.
- **Isocroni:** vengono utilizzati in contesti in cui la bassa latenza è più importante della correttezza del dato, come ad esempio per trasmettere contenuti multimediali, come audio e video. Questo è dovuto alla mancanza dell'impiego del codice di correzione dell'errore (CRC). Inoltre, il dato non viene ritrasmesso qualora si presenti un errore durante trasferimento.

2.3.4 – Trasferimenti di controllo

Il trasferimento di controllo è suddiviso in tre stadi:

- **Setup stage:** viene realizzato attraverso una transazione di tipo **SETUP**, cioè una sequenza di tre pacchetti. Contiene 8 byte di dati, in cui l'host esplicita la richiesta che intende inoltrare al dispositivo. Il significato degli otto byte presenti in questo stage può essere riassunto nella seguente tabella. Alcuni campi assumono un significato particolare in base al tipo di richiesta.

Offset	Campo	Size	Valore	Descrizione
0	bmRequestType	1	Bit-Map	D7 Data Phase Transfer Direction 0 = Host to Device 1 = Device to Host D6...5 Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0 Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	bRequest	1	Valore	Request
2	wValue	1	Valore	Value
4	wIndex	2	Indice o Offset	Index
6	wLength	2	Conteggio	Number of bytes to transfer if there is a data phase

Le richieste che l'host inoltra al dispositivo sono definite in base al parametro contenuto nel secondo byte del payload, ovvero bRequest. Qui di seguito si riporta l'elenco di alcune richieste possibili e il relativo significato dei parametri addizionali, quali wValue, wIndex e wLength:

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0000b	GET_STATUS (0x00)	Zero	Zero	Two	Device Status
0000 0000b	CLEAR_FEATURE (0x01)	Feature Selector	Zero	Zero	None
0000 0000b	SET_FEATURE (0x03)	Feature Selector	Zero	Zero	None
0000 0000b	SET_ADDRESS (0x05)	Device Address	Zero	Zero	None
1000 0000b	GET_DESCRIPTOR (0x06)	Descriptor Type & Index	Zero or Language ID	Descriptor Length	Descriptor
0000 0000b	SET_DESCRIPTOR (0x07)	Descriptor Type & Index	Zero or Language ID	Descriptor Length	Descriptor
1000 0000b	GET_CONFIGURATION (0x08)	Zero	Zero	1	Configuration Value
0000 0000b	SET_CONFIGURATION (0x09)	Configuration Value	Zero	Zero	None

Altri tipi di richieste sono possibili e dipendono dal dispositivo utilizzato.

- **Data stage (opzionale):** contiene l'eventuale risposta del dispositivo alla richiesta particolare dell'host.
- **Status stage:** è costituita da una transazione contenente un pacchetto **DATA1** di lunghezza zero.

2.4 – Protocollo

2.4.1 - Indirizzi

L'indirizzo del dispositivo USB è a 7 bit, pertanto è possibile indirizzare fino a 127 dispositivi diversi. L'indirizzo zero è riservato, siccome viene assegnato automaticamente ad un nuovo dispositivo prima che questo venga configurato e gli venga assegnato un indirizzo.

2.4.2 – Configurazioni

Un dispositivo può avere una o più configurazioni, anche se solo una attiva per volta. Solitamente i driver standard di Windows selezionano sempre la prima configurazione.

2.4.3 – Interfacce

Un dispositivo può avere una o più interfacce. Ciascuna interfaccia può avere un certo numero di endpoint e rappresenta una particolare unità funzionale di una particolare classe.

2.4.4 - Endpoint

Ciascun dispositivo USB possiede un certo numero di endpoint. Gli endpoint possono essere di due tipi: **OUT** oppure **IN**. Il tipo di endpoint identifica la direzione in cui fluiscono i dati.

- **OUT**: i dati viaggiano dall'host al dispositivo.
- **IN**: i dati viaggiano dal dispositivo all'host.

Ciascun dispositivo può avere fino a 16 endpoint per ogni direzione. L'endpoint 0 è un caso particolare, siccome viene utilizzato soltanto per trasferimenti di controllo.

Ad esempio, un telefono VOIP potrebbe avere due interfacce, la prima con due endpoint, uno per trasferire l'audio in ingresso e l'altro in uscita, mentre la seconda di tipo HID (Human Interface Device) con un singolo endpoint IN per gestire il tastierino numerico. È anche possibile avere due versioni alternative di un'interfaccia, che possono essere scambiate in qualsiasi momento.

2.4.6 - Descrittori

Un dispositivo contiene un certo numero di descrittori che servono a dettagliare quali sono le capacità del dispositivo. Esistono diversi tipi di descrittori.

- **Device descriptor**: è il primo descrittore che viene ricevuto dall'host. Contiene una serie di informazioni che riguardano il dispositivo che è appena stato connesso e deve essere configurato dal sistema operativo.

Offset	Campo	Size	Value	Description
0	bLength	1	Number	Dimensione del descrittore in byte.
1	bDescriptorType	1	Constant	Device Descriptor (type = 1).
2	bcdUSB	2	BCD	È la versione della specifica USB per cui è stato sviluppato il dispositivo.

4	bDeviceClass	1	Class	<p>Definisce la classe del dispositivo, assegnata dall'ente USB IF.</p> <ul style="list-style-type: none"> • 00h significa che ciascuna interfaccia definisce la sua classe. • FFh significa che è definita una sola classe di tipo vendor. <p>Tutti gli altri valori devono corrispondere ad una classe riconosciuta.</p>
5	bDeviceSubClass	1	SubClass	Sottoclasse assegnata dall'ente USB IF.
6	bDeviceProtocol	1	Protocol	Codice del protocollo assegnato dall'ente USB IF.
7	bMaxPacketSize0	1	Number	Dimensione massima dei pacchetti destinati all'endpoint 0 (8, 16,32 o 64 byte).
8	idVendor	2	ID	Vendor ID (assegnato dall'USB-IF). Il codice identificativo del produttore.
10	idProduct	2	ID	Product ID (assegnato dall'USB-IF). Il codice identificativo del prodotto.
12	bcdDevice	2	BCD	Device release number in binary-coded decimal.
14	iManufacturer	1	Index	Indice della stringa che contiene la descrizione del produttore.
15	iProduct	1	Index	Indice della stringa che contiene la descrizione del prodotto.
16	iSerialNumber	1	Index	Indice della stringa che contiene il numero seriale del dispositivo.
17	bNumConfigurations	1	Number	Numero di possibili configurazioni.

- **Configuration descriptor:** l'host richiede informazioni sulla configurazione del dispositivo, la quale viene solitamente seguita anche da descrittori delle interfacce presenti e dei relativi endpoint.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Dimensione del descrittore in byte.
1	bDescriptorType	1	Constant	Configuration Descriptor (0x02)
2	wTotalLength	2	Number	Numero totale di byte in questo descrittore sommato a quelli contenuti nei descrittori che seguiranno.
4	bNumInterfaces	1	Number	Numero di interfacce.
5	bConfigurationValue	1	Number	Valore utilizzato dal comando Set Configuration per selezionare questa configurazione.
6	iConfiguration	1	Index	Indice della stringa che descrive questa configurazione.
7	bmAttributes	1	Bitmap	D7 deve essere settato a 1 D6 Self Powered D5 Remote Wakeup D4...0 devono essere settati a 0

8	bMaxPower	1	mA	Consumo di corrente del dispositivo in milliampere.
---	-----------	---	----	---

- **Interface descriptor:** contiene informazioni riguardo all'interfaccia, quali il numero dell'interfaccia stessa, il numero degli endpoint ad essa associati e la classe a cui appartiene (Vendor, CDC, HID, ...).
- **Endpoint descriptor:** contiene informazioni riguardo all'endpoint, quali l'indirizzo dell'endpoint stesso, il tipo di endpoint e la dimensione massima dei pacchetti che vengono scambiati utilizzando quell'endpoint.
- **String descriptor:** contiene un testo riguardante un particolare aspetto del dispositivo, della configurazione o dell'interfaccia.

2.5 – Cattura Wireshark

Per comunicare con l'RP2040 e impartire comandi di debug, la IDE (Codewarrior) utilizza la connessione USB. Si analizzerà ora la cattura, effettuata tramite il programma Wireshark, dei pacchetti USB scambiati a partire dall'istante in cui si connette il dispositivo RP2040 al computer.

2.5.1 – Configurazione del dispositivo

Appena connesso, il dispositivo USB riceve, dal sistema operativo, una serie di richieste, volte a inizializzarne il funzionamento.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	host	2.1.0	USB	36	GET_DESCRIPTOR Request DEVICE
2	0.001932	2.1.0	host	USB	46	GET_DESCRIPTOR Response DEVICE
3	0.001948	host	2.1.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
4	0.004933	2.1.0	host	USB	37	GET_DESCRIPTOR Response CONFIGURATION
5	0.004953	host	2.1.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
6	0.007936	2.1.0	host	USB	60	GET_DESCRIPTOR Response CONFIGURATION
7	0.007950	host	2.1.0	USB	36	GET_STATUS Request
8	0.010931	2.1.0	host	USB	30	GET_STATUS Response
9	0.010966	host	2.1.0	USB	36	SET_CONFIGURATION Request
10	0.013057	2.1.0	host	USB	28	SET_CONFIGURATION Response

Figura 88: cattura Wireshark dei pacchetti USB scambiati tra il computer host e l'RP2040.

1. Per prima cosa l'host richiede al dispositivo di trasmettere il Device Descriptor, effettuando un trasferimento di controllo all'endpoint 0.

```
Setup Data
  bmRequestType: 0x80
    1... .... = Direction: Device-to-host
    .00. .... = Type: Standard (0x0)
    ...0 0000 = Recipient: Device (0x00)
  bRequest: GET_DESCRIPTOR (6)
  Descriptor Index: 0x00
  bDescriptorType: DEVICE (0x01)
  Language Id: no language specified (0x0000)
  wLength: 18
```

Figura 9: cattura Wireshark della richiesta del descrittore della configurazione.

2. Il dispositivo risponde trasmettendo il Device Descriptor. Al suo interno si trovano i codici che identificano in maniera univoca il dispositivo, cioè il Vendor ID e il Product ID, attraverso cui il sistema operativo può caricare i driver corretti, che ne consentono il funzionamento.

```

  ▾ DEVICE DESCRIPTOR
    bLength: 18
    bDescriptorType: 0x01 (DEVICE)
    bcdUSB: 0x0200
    bDeviceClass: Vendor Specific (0xff)
    bDeviceSubClass: 255
    bDeviceProtocol: 255
    bMaxPacketSize0: 32
    idVendor: MCS (0x16d0)
    idProduct: Unknown (0x0567)
    bcdDevice: 0x004c
    iManufacturer: 1
    iProduct: 2
    iSerialNumber: 3
    bNumConfigurations: 1

```

Figura 100: cattura Wireshark del descrittore del dispositivo, trasmesso dall'RP2040.

- Viene poi richiesto dall'host il Configuration Descriptor.

```

  ▾ Setup Data
    ▾ bmRequestType: 0x80
      1... .... = Direction: Device-to-host
      .00. .... = Type: Standard (0x0)
      ...0 0000 = Recipient: Device (0x00)
    bRequest: GET_DESCRIPTOR (6)
    Descriptor Index: 0x00
    bDescriptorType: CONFIGURATION (0x02)
    Language Id: no language specified (0x0000)
    wLength: 9

```

- Il dispositivo risponde alla richiesta con il Configuration Descriptor, che contiene una serie di informazioni sulla configurazione del dispositivo, tra cui il numero di interfacce disponibili. Per questo progetto è stato sufficiente implementare una sola interfaccia di tipo Vendor.

```

  ▾ CONFIGURATION DESCRIPTOR
    bLength: 9
    bDescriptorType: 0x02 (CONFIGURATION)
    wTotalLength: 32
    bNumInterfaces: 1
    bConfigurationValue: 1
    iConfiguration: 4
    > Configuration bmAttributes: 0xc0 SELF-POWERED NO REMOTE-WAKEUP
    bMaxPower: 250 (500mA)

```

Figura 11: cattura Wireshark del descrittore della configurazione.

- Dopo il Configuration Descriptor, l'host richiede al dispositivo di trasmettere tutti i descrittori rimanenti, ovvero l'Interface Descriptor e gli Endpoint Descriptors.
- Dalla cattura dei dati si può notare che il dispositivo, così come è stato configurato per questo progetto, supporta una sola interfaccia di tipo Vendor e due endpoint, uno per ogni direzione. Il primo è di tipo OUT ed è associato all'indirizzo 0x01, mentre il secondo è di tipo IN ed associato all'indirizzo 0x82.

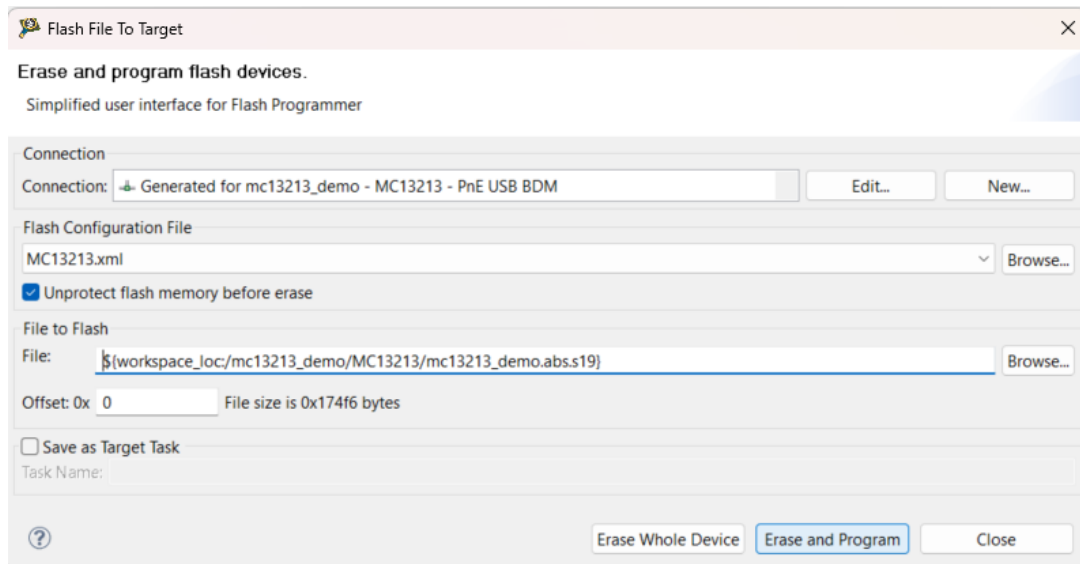


Figura 134: screenshot della IDE Codewarrior. Pop-up per eseguire il comando di scrittura.

Si aprirà allora un pop-up e, dopo aver cliccato su *Erase and Program*, la IDE comincerà a trasmettere via USB una serie di comandi, di cui si approfondirà in seguito la sintassi e il significato. Per ora ci basta sapere che questi comandi vengono ricevuti dall'RP2040, che poi li interpreta ed elabora, per poi inoltrarli attraverso la porta BDM al dispositivo target Freescale.

Di seguito è riportata la cattura di due trasferimenti Bulk di tipo diverso:

No.	Time	Source	Destination	Protoc	Length	Info
1281	9571.867266	host	2.1.1	USB	35	URB_BULK out
1282	9571.867316	2.1.1	host	USB	27	URB_BULK out
1283	9571.867345	host	2.1.2	USB	27	URB_BULK in
1284	9571.867642	2.1.2	host	USB	29	URB_BULK in

Figura 145: cattura Wireshark di un trasferimento bulk.

- Nel primo trasferimento Bulk di tipo OUT, l'host trasmette dei dati (frame no. 1281) sull'endpoint 1 al dispositivo, il quale risponde (frame no. 1282) con un acknowledge.

```

> Frame 1281: 35 bytes on wire (280 bits), 35 bytes captured (280 bit
  USB URB
    [Source: host]
    [Destination: 2.1.1]
    USBPcap pseudoheader length: 27
    IRP ID: 0xffffbc819b119aa0
    IRP USBD_STATUS: USBD_STATUS_SUCCESS (0x00000000)
    URB Function: URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER (0x0009)
  > IRP information: 0x00, Direction: FDO -> PDO
    URB bus id: 2
    Device address: 1
  > Endpoint: 0x01, Direction: OUT
    URB transfer type: URB_BULK (0x03)
    Packet Data Length: 8
    [Response in: 1282]
    [bInterfaceClass: Vendor Specific (0xff)]
    Leftover Capture Data: 082101010000084
  
```

Dati trasmessi dall'host

Figura 15: cattura Wireshark dei dati trasmessi dall'host nel primo trasferimento Bulk OUT.

- Nel secondo trasferimento Bulk di tipo IN, l'host trasmette un pacchetto (frame no. 1282) contenente la richiesta di ricevere dati dall'endpoint 2 del dispositivo, il quale risponde (frame no. 1284) con i dati richiesti.

```

> Frame 1284: 29 bytes on wire (232 bits), 29 bytes captured (232 bits)
  USB URB
    [Source: 2.1.2]
    [Destination: host]
    USBPcap pseudoheader length: 27
    IRP ID: 0xffffbc81a398caa0
    IRP USBD_STATUS: USBD_STATUS_SUCCESS (0x00000000)
    URB Function: URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER (0x0009)
  > IRP information: 0x01, Direction: PDO -> FDO
    URB bus id: 2
    Device address: 1
  > Endpoint: 0x82, Direction: IN
    URB transfer type: URB_BULK (0x03)
    Packet Data Length: 2
    [Request in: 1283]
    [Time from request: 0.000297000 seconds]
    [bInterfaceClass: Vendor Specific (0xff)]
    Leftover Capture Data: 0000
  
```

Dati trasmessi dal dispositivo

Figura 16: cattura Wireshark dei dati trasmessi dal dispositivo nel secondo trasferimento Bulk IN.

In questo specifico caso, la IDE ha trasmesso un comando in cui segnala di voler leggere un byte dalla memoria del target, all'indirizzo 0x0084 (ultimi due byte del primo frame), locazione che corrisponde ad una porzione della RAM. Il dispositivo risponde con il valore letto, ovvero 0x00, preceduto dal codice di assenza di errori 0x00.

3 – Il dispositivo

3.1 – RP2040

L'RP2040 è un microcontrollore a basso costo e dalle alte prestazioni. Ha a disposizione le seguenti funzionalità:

- Processore Dual Cortex M0+ con frequenza di clock fino a 133MHz;
- 264 kb di memoria SRAM suddivisa in 6 banchi;
- 30 GPIO multifunzione;
- Sei pin di IO dedicati per l'interfaccia SPI;
- Hardware dedicato per le più comuni periferiche;
- Periferiche Input Output programmabili (PIO);
- Un ADC a 12 bit e 4 canali, con sensore di temperatura interno a 0.5MSa/s.

3.2 – Modulo PIO: Programmable Input Output

3.2.1 - Introduzione

Il modulo PIO è un'interfaccia programmabile di IO. Supporta diversi tipi di standard, tra cui I2C, I2S, SDIO, SPI, UART, DPI e VGA. Per implementare l'interfaccia BDM, tuttavia, è stato necessario scrivere una libreria a basso livello che implementasse tale protocollo.

Il modulo PIO è programmabile, allo stesso modo di un microprocessore, con l'importante differenza che la sequenza di istruzioni che dovrà eseguire vengono salvate su registri che sono mappati in RAM e non nella memoria Flash. Ciò significa che è possibile modificare in qualsiasi momento la successione di istruzioni che deve eseguire, alterando il contenuto di opportuni registri.

L'RP2040 possiede un totale di 2 moduli PIO identici, che possono lavorare anche in parallelo, ognuno dei quali possiede quattro macchine a stati, dedicate alla manipolazione dei dati in ingresso e in uscita. Una differenza fondamentale tra il PIO e un qualunque altro processore general purpose è il fatto che le macchine a stati del PIO sono altamente specializzate per l'I/O, con un'attenzione particolare al determinismo e in grado di soddisfare le specifiche più stringenti in termini di temporizzazione.

3.2.2 – Macchine a stati

Ciascuna macchina a stati dispone di:

- **FIFO**: due buffer FIFO a 32 bit per la comunicazione con il sistema principale, uno per trasferire i dati in una direzione e l'altro per trasferirli nella direzione opposta.
- **Input Shift Register (ISR)**: registro a scorrimento a 32 bit per i dati in ingresso.
- **Output Shift Register (OSR)**: registro a scorrimento a 32 bit per i dati in uscita.
- **Scratch Register X & Y**: una coppia di registri a 32 bit.
- **Divisore di frequenza**: è possibile modificare la frequenza di clock del PIO, impostando opportunamente il divisore di frequenza.
- **GPIO**: porte per l'I/O.
- **Interfaccia DMA**: per l'accesso diretto alla memoria.
- **IRQ**

Le quattro macchine a stati condividono la stessa memoria per le istruzioni. Il software di sistema ha il compito di caricare i programmi in RAM e di associare ciascuna macchina a stati alle GPIO che dovrà pilotare.

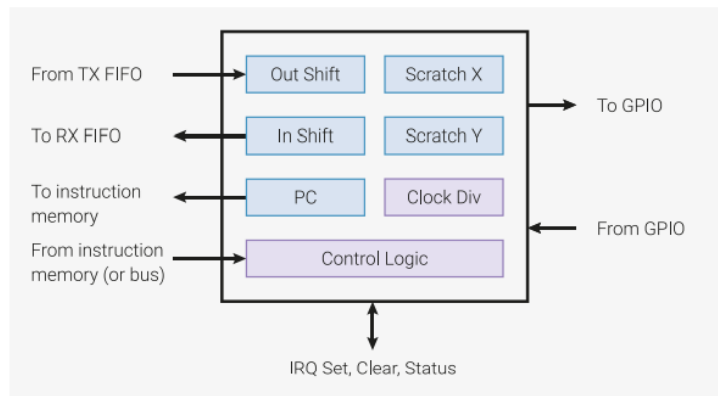


Figura 18: panoramica di una macchina a stati dal datasheet dell'RP2040.

3.2.3 – Registri

Ciascuna macchina a stati possiede un certo numero di registri interni. Questi servono a ricevere dati in ingresso, trasferirli in uscita oppure mantenere alcuni dati temporanei, come contatori o variabili.

3.2.4 – FIFO

Ciascuna macchina a stati possiede una coppia di buffer FIFO con una dimensione pari a quattro.

- **TX FIFO:** è utilizzata per trasferire dati dal sistema principale verso la macchina a stati.
- **RX FIFO** è utilizzata per trasferire dati dalla macchina a stati al sistema principale.

I due buffer possono essere accorpati, nel caso si abbia la necessità di utilizzare un'unica FIFO unidirezionale con dimensione pari a otto.

3.2.5 – Output Shift Register (OSR)

L'OSR è un registro a scorrimento con una capacità di 32 bit. Può trasferire un certo numero di bit alla volta verso una data destinazione, ogni volta che viene chiamata l'istruzione **OUT**. A seguito di ciò, viene incrementato un contatore che tiene traccia di quanti dati sono usciti dal registro.

Per scrivere su questo registro si utilizza l'istruzione **PULL**, che vi trasferisce una parola da 32 bit proveniente dalla **TX FIFO**. Quando i dati vengono portati fuori dal registro a scorrimento, vengono sostituiti con degli zeri.

La funzionalità *Auto pull*, se abilitata, permette di riempire automaticamente l'OSR. Questa si attiva qualora il contatore dei dati in uscita superi una certa soglia e trasferisce i dati provenienti dalla **TX FIFO** su di esso, senza dover chiamare l'istruzione **PULL**. Tale soglia può essere impostata scrivendo su degli opportuni registri.

3.2.6 – Input Shift Register (ISR)

L'ISR è un registro a scorrimento con una capacità di 32 bit. Può trasferire al suo interno un certo numero di bit alla volta provenienti da diverse fonti, ogni volta che viene chiamata l'istruzione **IN**. A seguito di ciò, viene incrementato un contatore che tiene traccia di quanti dati sono entrati nel registro.

Per leggere da questo si utilizza l'istruzione **PUSH**, che trasferisce la parola da 32 bit in esso contenuta, alla **RX FIFO**. Successivamente il contenuto dell'ISR viene azzerato.

La funzionalità *Auto push*, se abilitata, permette di svuotare automaticamente l'ISR. Questa si attiva qualora il contatore dei dati in ingresso superi una certa soglia e trasferisce i dati presenti su di esso alla **RX FIFO**, senza dover chiamare l'istruzione **PUSH**. Tale soglia può essere impostata scrivendo su degli opportuni registri.

3.2.7 – Scratch registers

Ciascuna macchina a stati possiede due *Scratch register* a 32 bit, chiamati *Scratch X* e *Scratch Y*, che possono essere utilizzati come registri di appoggio per le operazioni intermedie oppure come contatori.

3.2.8 – Stalling

La macchina a stati può interrompere momentaneamente l'esecuzione del programma, poiché si è verificata una delle seguenti condizioni.

- Un'istruzione **WAIT** sta attendendo che si realizzi una determinata circostanza.
- È stata eseguita un'istruzione **PULL**, mentre la **TX FIFO** è vuota.
- È stata eseguita un'istruzione **PUSH**, mentre la **RX FIFO** è piena.
- È stata eseguita un'istruzione **OUT**, quando la funzionalità *Auto pull* è abilitata e l'OSR ha raggiunto la soglia prestabilita, mentre la **TX FIFO** è ancora vuota.
- È stata eseguita un'istruzione **IN**, quando la funzionalità *Auto push* è abilitata e l'ISR ha raggiunto la soglia prestabilita, mentre la **RX FIFO** è ancora piena.

3.2.9 – Pin mapping

Il PIO può controllare la *direzione* e il *livello* di tutte le 32 GPIO, oltre che osservarne lo stato.

Ciascuna operazione che osserva (istruzione **IN**) oppure modifica (istruzioni **OUT**, **SET**, *Side-set*) lo stato di un PIN, deve essere associata ad una o più GPIO, attraverso opportuna configurazione del registro PINCTRL presente in ciascuna macchina a stati.

Ogni istruzione ha un mapping indipendente dalle altre, che può anche essere condiviso tra istruzioni diverse. Ad esempio, non è infrequente che sia l'istruzione **OUT** che l'istruzione **IN** vengano associate allo stesso PIN. Le operazioni *Side-set* vengono sempre applicate per ultime.

3.2.10 – Istruzioni

Ogni istruzione è codificata su 16 bit, dove i 3 bit più significativi identificano il tipo di istruzione. Sono disponibili un totale di nove istruzioni.

- **JMP**: imposta il PC al valore specificato nell'*Address* se *Condition* risulta verificata. Per selezionare quale PIN utilizzare, è sufficiente mapparlo, scrivendo sul registro EXECCTRL_JMP_PIN. La condizione !OSRE compara il numero di bit che sono stati portati

fuori dall'ultima **PULL** e una soglia configurabile attraverso il registro `SHIFTCTRL_PULL_THRESH` (stessa soglia di *Auto pull*).

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set				Condition			Address					

Figura 19: codifica dell'istruzione *JUMP* dal datasheet dell'RP2040.

Valore	Condition	Descrizione
000	(Nessuna condizione)	Esegui il salto sempre.
001	!X	Salta se lo <i>Scratch X</i> è zero.
010	X--	Salta se lo <i>Scratch X</i> è diverso da zero e post-decrementa.
011	!Y	Salta se lo <i>Scratch Y</i> è zero.
100	Y--	Salta se lo <i>Scratch Y</i> è diverso da zero e post-decrementa.
101	X! =Y	Salta se lo <i>Scratch X</i> è diverso dallo <i>Scratch Y</i> .
110	PIN	Salta se il valore di uno specifico PIN è alto.
111	!OSRE	Salta se l'Output Shift Register non è vuoto.

- **WAIT**: interrompe momentaneamente l'esecuzione del codice finché non si verifica una certa condizione.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WAIT	0	0	1	Delay/side-set				Pol	Source		Index					

Figura 20: codifica dell'istruzione *WAIT* dal datasheet dell'RP2040.

Valore	Source	Descrizione
00	GPIO	Aspetta che si verifichi una condizione su una data GPIO, selezionata da <i>Index</i> . Si riferisce al valore assoluto delle GPIO.
01	PIN	Aspetta che si verifichi una condizione su un certo PIN, selezionato da <i>Index</i> . Si riferisce al mapping interno al PIO e non quello assoluto.
10	IRQ	Aspetta finché non viene settato un flag di interrupt.
11	Reserved	-

Valore	Pol	Descrizione
0	0	Aspetta che si verifichi uno zero.
1	1	Aspetta che si verifichi un uno.

- **IN**: trasferisce un numero di bit pari a *Bit count* all'interno dell'Input Shift Register. È possibile configurare in quale direzione trasferire i dati. Inoltre, incrementa il conteggio dei bit in ingresso di *Bit count*, saturando a 32. La soglia per la funzione *Auto push*, è configurabile dal

registro SHIFTCTRL_PUSH_THRESH. Il conteggio dei bit in ingresso viene automaticamente resettato, così come il contenuto dell'ISR.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IN	0	1	0	Delay/side-set				Source			Bit count					

Figura 21: codifica dell'istruzione IN dal datasheet dell'RP2040.

Valore	Source	Descrizione
000	PINS	Viene campionato il PIN opportunamente mappato.
001	X	Vengono trasferiti dati dal registro <i>Scratch X</i> all'ISR.
010	Y	Vengono trasferiti dati dal registro <i>Scratch Y</i> all'ISR.
011	NULL	L'ISR è riempito con tutti zeri.
100	Reserved	-
101	Reserved	-
110	ISR	La sorgente dei dati è lo stesso ISR.
111	OSR	Vengono trasferiti dati dall'OSR all'ISR.

- **OUT:** trasferisce un numero di bit pari a *Bit count* al di fuori dell'Output Shift Register (OSR) e scrive tali dati in *Destination*. Inoltre, incrementa il conteggio dei bit in uscita di *Bit count*. La soglia per la funzione *Auto push*, è configurabile dal registro SHIFTCTRL_PULL_THRESH. Il conteggio dei bit in uscita viene automaticamente resettato.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUT	0	1	1	Delay/side-set				Destination			Bit count					

Figura 22: codifica dell'istruzione OUT dal datasheet dell'RP2040.

Valore	Destination	Descrizione
000	PINS	I bit presenti sull'OSR vengono utilizzati per impostare il <i>livello</i> del PIN.
001	X	Vengono trasferiti dati dall'OSR al registro <i>Scratch X</i> .
010	Y	Vengono trasferiti dati dall'OSR al registro <i>Scratch Y</i> .
011	NULL	Tutti i dati vengono scartati.
100	PINDIRS	I bit presenti sull'OSR vengono utilizzati per impostare la <i>direzione</i> del PIN.
101	PC	Vengono trasferiti dati dall'OSR al Program Counter.
110	ISR	Vengono trasferiti dati dall'OSR all'ISR.
111	EXEC	I dati presenti sull'OSR vengono eseguiti come fossero un'istruzione.

- **PUSH:** trasferisce il contenuto dell'ISR nella RX FIFO, come un'unica parola a 32 bit. Azzerà il contenuto dell'ISR.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUSH	1	0	0	Delay/side-set				0	IfF	Blk	0	0	0	0	0	0

Figura 23: codifica dell'istruzione PUSH dal datasheet dell'RP2040.

- **PULL:** carica una parola di 32 bit dalla TX FIFO nell'OSR.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PULL	1	0	0	Delay/side-set				1	IfE	Blk	0	0	0	0	0	0

Figura 24: codifica dell'istruzione PULL dal datasheet dell'RP2040.

- **MOV:** copia dei dati da *Source* a *Destination*.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	1	0	1	Delay/side-set				Destination			Op	Source				

Figura 25: codifica dell'istruzione MOV dal datasheet dell'RP2040.

- **IRQ:** imposta o azzerà il flag IRQ, selezionato da *Index*.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ	1	1	0	Delay/side-set				0	Clr	Wait	Index					

Figura 26: codifica dell'istruzione IRQ dal datasheet dell'RP2040.

- **SET:** trasferisce il valore presente su *Data* in *Destination*.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SET	1	1	1	Delay/side-set				Destination			Data					

Figura 27: codifica dell'istruzione SET dal datasheet dell'RP2040.

Ogni istruzione viene eseguita esattamente in un colpo di clock, con l'unica eccezione dell'istruzione **WAIT**, che interrompe l'esecuzione fino a che non si verifica una data condizione.

È possibile aggiungere dei cicli di ritardo, attraverso la funzione *Delay*.

Il numero massimo di istruzioni che è possibile scrivere sulla memoria del PIO è di 32 istruzioni. Il PC punta alla locazione di memoria che contiene l'istruzione corrente e viene incrementato ad ogni ciclo di clock.

3.2.11 – Side-set

È una funzionalità che permette di cambiare il livello o la direzione di uno o più PIN, mentre viene eseguita una delle istruzioni sopra elencate. Questa funzionalità permette, ad esempio, mentre si riceve un dato oppure si cambia il valore di un registro, contemporaneamente, cambiare il livello o la direzione di un PIN.

3.3 – Realizzazione del protocollo BDM

3.3.1 – Comunicazione con il PIO: panoramica

Prima di analizzare il codice, è essenziale capire come i dati vengono trasferiti dal programma principale, che li riceve tramite USB, al modulo PIO, che li deve trasmettere tramite protocollo BDM. Le informazioni di cui il PIO ha bisogno prima di iniziare un trasferimento BDM sono le seguenti:

- Sequenza di byte da mandare in output, che comprende sempre il comando BDM (primo byte), seguito da una serie di dati il cui significato cambia in funzione del comando stesso.
- Numero di bit da trasmettere.
- Numero di bit da ricevere.

Vediamo qualche esempio:

Comando BDM	Descrizione	Sequenza di byte	# di bit da trasmettere	# di bit da ricevere
GO (0x08)	Uscire dalla modalità Active Background e riprendere l'esecuzione.	0x08	8	0
READ_PC (0x6B)	Leggere il contenuto del Program Counter.	0x6B	8	16
WRITE_SP (0x4F)	Scrivere sullo Stack Pointer i due byte XXXX.	0x4FXXXX	24	0
WRITE_BYTE_WS (0xC1)	Scrivere il byte XX all'indirizzo a 16 bit AAAA e ritornare il contenuto del registro di stato.	0xC1AAAAXX	32	8

L'unico modo che il PIO ha per comunicare con il programma principale sono le due FIFO, ovvero **TX FIFO** (dal programma principale al PIO) e la **RX FIFO** (dal PIO al programma principale).

La sequenza di byte da trasmettere sarà scritta dal programma principale sulla **TX FIFO**, mentre i byte ricevuti verranno letti dalla **RX FIFO**.

Ora si pone il problema di come il PIO può conoscere il *numero* di bit da trasmettere in *uscita*, così come il *numero* di bit da ricevere in *ingresso*.

3.3.2 – Comunicazione con il PIO: impostare il numero di bit da scrivere

Si imposta il numero di bit da scrivere nel registro **SM0_SHIFTCTRL**, in particolare nei bit <25:29>, ovvero quelli relativi al parametro **PULL_THRESH**.

In questo modo si configura una soglia (*pull threshold*), raggiunta la quale il modulo PIO considera terminata la sequenza di bit da trasmettere. In altre parole, impostare questa soglia riduce la dimensione massima dell'OSR da 32 bit al valore desiderato.

3.3.4 – Comunicazione con il PIO: trasferire dati sulla FIFO

```
156 // Data command
157 void do_bdm_command(PIO pio, uint sm, uint data, uint tx_bit, uint rx_bit, uint offset)
158 {
159     // Set number of bit to shift out by changing pull threshold
160     pio_set_pull_threshold(pio, sm, tx_bit);
161
162     // Set number of bit to shift in by encoding a "set x, rx_bit" instruction in pio instr memory
163     uint instr = pio_encode_set(pio_x, rx_bit);
164     pio_add_instr(pio, instr, offset + 1);
165
166     // Put data in tx fifo:
167     put_tx_fifo(pio, sm, data, tx_bit, SHIFT_RIGHT);
168 }
169
```

Figura: 30: screenshot del codice dal file `pio_functions.c`.

```
62
63 void put_tx_fifo(PIO pio, uint sm, uint data, uint bit, bool shift_right)
64 {
65     // If shift_right is disabled, align data to the left.
66     if(!shift_right)
67     {
68         uint shift = REG_WIDTH - bit;
69         data = data << shift;
70     }
71
72     pio_sm_put_blocking(pio, sm, data);
73 }
74
```

Figura: 31: screenshot del codice dal file `pio_functions.c`.

A questo punto non bisogna far altro che scrivere i dati da trasmettere sulla **TX FIFO**, attraverso la funzione `put_tx_fifo()`, allineandoli a sinistra, dal momento che il protocollo BDM prevede che vengano trasferiti a partire dall'MSB. Essa richiama a sua volta la funzione dell'SDK `pio_sm_put_blocking()`, che trasferisce finalmente i dati sulla **TX FIFO**.

I byte ricevuti vengono letti dalla **RX FIFO**, attraverso la funzione dell'SDK `pio_sm_get()`.

```
92
93     do_bdm_command(pio, sm, data, tx_bit_count, rx_bit_count, pio_offset);
94
95     // Wait the end of the operation
96     wait_end_operation(pio, sm);
97
98     uint received_data = 0;
99     // Read data from rx fifo
100     if(!pio_sm_is_rx_fifo_empty(pio, sm))
101     {
102         received_data = pio_sm_get(pio, sm);
103     }
104
105     return received_data;
106 }
```

Figura: 32: screenshot del codice dal file `pio_functions.c`.

Prima di leggere i dati, però, si attende la fine della trasmissione BDM, segnalata al programma principale attraverso la stessa scrittura sulla **RX FIFO**. La funzione `wait_end_operation()`, infatti, non fa altro che attende finché la **RX FIFO** rimane vuota.

```

75
76 void wait_end_operation(PIO pio, uint sm)
77 {
78     // Wait for an operation to complete.
79     // When any operation ends, some data are transferred to rx fifo
80     while(pio_sm_is_rx_fifo_empty(pio, sm));
81 }
82

```

Figura 33: screenshot del codice dal file `pio_functions.c`.

Importante notare che ogni trasferimento BDM termina con una lettura dalla **RX FIFO** dei dati ricevuti, anche quando non ve n'è nessuno, come ad esempio quando si esegue un comando di sola scrittura. In questo caso il PIO trasferisce comunque dei dati sulla **RX FIFO** (bit dummy) per segnalare la fine del trasferimento.

3.3.5 – Codice per il modulo PIO: trasmissione comandi BDM

Vediamo la sequenza di istruzioni che il PIO esegue per implementare il protocollo BDM.

```

≡ bdm-data.pio
1  .program bdm_data
2  .side_set 1 opt
3
4  .wrap_target
5  start:
6  | pull            side 1      ; Pull data from tx fifo(could be 8, 16, 24 or 32 bit). Stall here if tx fifo is empty
7  | set x, 0        ; Set x as loop counter. X indicates number of bit to read
8
9  tx_loop:
10 | set pindirs, 1
11 | out y, 1        side 0 [1]  ; Shift one bit from OSR to scratch y. Pin BKGD goes low for 4 cycles
12 | jmp !y keep_low [1]       ; If scratch y = 0, jump to keep_low
13 | nop            side 1 [7]  ; If 1 is transmitted, pin BKGD goes back to high state after 4 cycles
14 | jmp end_tx_loop
15 keep_low:
16 | nop
17 | nop            [7]       ; If 0 is transmitted, hold pin BKGD low for 9 cycles
18 | end_tx_loop:   ; Pin BKGD goes high for the last 3 cycles.
19 | jmp !osr tx_loop side 1 [1] ; If osr is not empty(osr_count < pull_threshold), go back to tx_loop
20 | set           ; Keep pin high for the last 16th cycle
21

```

Figura 34: screenshot del codice dal file `bdm-data.pio`.

Nota: le parentesi quadre, servono a segnalare un numero di cicli di ritardo da eseguire prima di passare all'istruzione successiva. Siccome ogni istruzione impiega già un ciclo di clock per essere eseguita, se di fianco ad essa è presente la dicitura `[N]`, significa che si passerà ad eseguire l'istruzione successiva dopo $1+N$ cicli di clock.

1. La prima istruzione eseguita dal PIO è un'istruzione **PULL** (riga 6), che trasferisce la sequenza di byte da trasmettere dalla **TX FIFO** all'OSR.
2. La seconda istruzione è in realtà un *placeholder* (riga 7), dal momento che nella fase precedente questa era stata sostituita con un'istruzione **SET**, che imposta il numero di bit da

9. Dopodiché si cambia la funzione del PIN BKGD, da output a input (riga 35), aspettando ulteriori cinque colpi di clock.
10. Si campiona, allora, la linea, utilizzando l'istruzione **IN** (riga 36), trasferendo il bit letto all'ISR.
11. Si controlla se vi sono ancora dati da leggere e, qualora ve ne fossero, si torna al passo 8, altrimenti si prosegue con l'istruzione **PUSH** (riga 37), che trasferisce i dati letti dall'ISR alla **RX FIFO**. Si salta ora, direttamente al passo 13.
12. Questo passaggio viene eseguito solamente se non vi sono dati da ricevere. Per segnalare la fine della trasmissione BDM, si trasferiscono dei bit dummy alla **RX FIFO**, richiamando l'istruzione **IN** (riga 42), seguita dal parametro *null*, 32.
13. La trasmissione BDM termina copiando il contenuto del registro *Scratch Y*, dove era stato salvato il numero dei bit da ricevere, di nuovo nel registro *Scratch X*. Questa operazione è effettuata poiché, nel caso si voglia ripetere un comando BDM che preveda lo stesso numero di bit letti di quello appena concluso, il registro *Scratch X* risulterebbe già inizializzato con il corretto numero di bit da ricevere. Si tenga conto che questa ottimizzazione non è poi stata implementata.

3.4 – Modulo USB

L'RP2040 contiene un modulo USB che può operare nelle seguenti modalità:

- Dispositivo Full Speed (12Mbit/s).
- Host che può comunicare sia con dispositivi Low Speed (1.5Mbit/s), che con dispositivi Full Speed.

Nella modalità *Device*, ovvero quella che è stata utilizzata per il progetto, il modulo USB risulta compatibile con lo standard USB 2.0, supporta fino a 32 endpoint (dall'endpoint 0 al 15 in entrambe le direzioni) e trasferimenti di tipo *Control*, *Bulk*, *Isocroni* e *Interrupt*.

3.5 – Modulo USB: libreria TinyUSB

Sebbene sia possibile programmare a basso livello il modulo USB, così come fatto per il modulo PIO, per semplicità si è scelto di utilizzare questa libreria per gestire la comunicazione USB ad alto livello.

3.5.1 – Introduzione

TinyUSB è una libreria multiplatforma, disponibile anche per l'RP2040, che implementa lo stack software a basso livello, necessario per la comunicazione USB.

Supporta le seguenti classi USB:

- Audio Class 2.0 (UAC2).
- Bluetooth Host Controller Interface (BTH HCI).
- Communication Device Class (CDC).
- Device Firmware Update (DFU): DFU mode (WIP) and Runtime.
- Human Interface Device (HID): Generic (In & Out), Keyboard, Mouse, Gamepad etc ...
- Mass Storage Class (MSC).
- Musical Instrument Digital Interface (MIDI).
- Network with RNDIS, Ethernet Control Model (ECM), Network Control Model (NCM).
- Test and Measurement Class (USBTMC).
- Video class 1.5 (UVC): non ancora supportato.
- Vendor-specific class, con supporto a endpoint generici di input e output.
- WebUSB con classe vendor-specific.

3.5.2 – Includere la libreria nel progetto

Siccome la libreria è già inclusa nell'SDK dell'RP2040, l'unica cosa che è necessario fare per abilitarla è modificare il file CMakeLists.txt, presente nella cartella principale di un progetto per l'RP2040, nel seguente modo:

```
48 # Link to pico_stdlib (gpio, time, etc. functions)
49 # In addition to pico_stdlib required for common PicoSDK functionality, add dependency on tinyusb_device
50 # for TinyUSB device support and tinyusb_board for the additional board support library
51 target_link_libraries(
52     ${PROJECT_NAME} PUBLIC
53     pico_stdlib
54     hardware_clocks
55     hardware_pio
56     tinyusb_device
57     tinyusb_board
58 )
59
```

Include le funzioni utilizzate dallo stack TinyUSB quando è configurato come dispositivo USB.

Figura: 36: screenshot del codice CMakeLists.txt

3.5.3 – Classi e descrittori

Per configurare la libreria, è necessario includere e modificare i seguenti file.

- Il file *tusb_config.c*, nel quale vengono selezionate le classi che devono essere abilitate per il dispositivo. Qui è opportuno definire anche alcuni parametri fondamentali, come la dimensione massima degli endpoint. La mancanza di queste definizioni verrà segnalata dalla libreria stessa, tramite un *warning*, una volta abilitata la classe.

```
89 //-----
90 // DEVICE CONFIGURATION
91 //-----
92
93 #ifndef CFG_TUD_ENDPOINT0_SIZE
94 #define CFG_TUD_ENDPOINT0_SIZE 32
95 #endif
96
97 //----- CLASS -----//
98 #define CFG_TUD_HID 0
99 #define CFG_TUD_CDC 0
100 #define CFG_TUD_MSC 0
101 #define CFG_TUD_MIDI 0
102 #define CFG_TUD_VENDOR 1
103
104 #define CFG_TUD_VENDOR_RX_BUFSIZE (256)
105 #define CFG_TUD_VENDOR_TX_BUFSIZE (256)
106
107 #define BDM_OUT_EP_MAXSIZE (64) //!< USBDM - BDM out
108 #define BDM_IN_EP_MAXSIZE BDM_OUT_EP_MAXSIZE //!< USBDM - BDM in
```

Figura: 37: screenshot del codice dal file *usb_descriptors.h*.

In questo progetto l'unica classe attiva, è la classe Vendor-specific. Dal file di configurazione è stato scelto di utilizzare una singola interfaccia (riga 102), con la quale sono inclusi due endpoint, uno IN e l'altro OUT. La dimensione massima degli endpoint è pari a 64 byte. Attraverso questi due endpoint avviene la comunicazione tra la IDE e l'RP2040.

- Il secondo file da includere è *usb_descriptors.c*, nel quale dovranno essere definiti tutti i descrittori: del dispositivo, della configurazione, dell'interfaccia, degli endpoint e delle stringhe. Di seguito è mostrato il descrittore del dispositivo.

```

C usb_descriptors.c > desc_configuration
11  tusb_desc_device_t const desc_device =
12  {
13      .bLength           = sizeof(tusb_desc_device_t),
14      .bDescriptorType   = TUSB_DESC_DEVICE,
15      .bcdUSB            = CONST_NATIVE_TO_LE16(USB_BCD),
16      .bDeviceClass      = TUSB_CLASS_VENDOR_SPECIFIC,
17      .bDeviceSubClass   = TUSB_CLASS_VENDOR_SPECIFIC,
18      .bDeviceProtocol   = TUSB_CLASS_VENDOR_SPECIFIC,
19      .bMaxPacketSize0   = CFG_TUD_ENDPOINT0_SIZE,
20
21      .idVendor          = CONST_NATIVE_TO_LE16(USB_VID),
22      .idProduct         = CONST_NATIVE_TO_LE16(USB_PID),
23      .bcdDevice         = CONST_NATIVE_TO_LE16(VERSION_ID),
24
25      .iManufacturer     = s_manufacturer_index,
26      .iProduct          = s_product_index,
27      .iSerialNumber     = s_serial_index,
28
29      .bNumConfigurations = NUMBER_OF_CONFIGURATIONS
30  };
>1

```

Figura: 38: screenshot del codice dal file `usb_descriptors.c`.

Inoltre, dovranno essere implementate delle funzioni, chiamate dallo stack USB al momento opportuno, ad esempio durante la fase di configurazione del dispositivo, che ritornino il descrittore desiderato.

```

31
32  // Invoked when received GET DEVICE DESCRIPTOR
33  // Application return pointer to descriptor
34  uint8_t const * tud_descriptor_device_cb(void)
35  {
36      return (uint8_t const *) &desc_device;
37  }

```

Figura: 39: screenshot del codice dal file `usb_descriptors.c`.

La funzione **`tud_descriptor_device_cb()`** ritorna il puntatore al descrittore del dispositivo, cioè la struttura definita in precedenza.

3.5.4 – Inizializzazione e task TinyUSB

L'ultimo passaggio necessario per poter far funzionare lo stack TinyUSB è quello di inserire all'interno del codice principale, due funzioni fondamentali.

- La funzione **`tusb_init()`**, che inizializza lo stack TinyUSB.
- La funzione **`tud_task()`**, che dovrà essere richiamata periodicamente e che si occuperà di invocare a sua volta tutte le callback e di gestire le funzionalità dello stack.

```

34  /*----- MAIN -----*/
35  int main(void)
36  {
37      board_init();
38      tusb_init();
39
40      // Set clock and connect BDM
41      device_init();
42
43      while (1)
44      {
45          tud_task(); // tinyusb device task
46
47          usbdm_task();
48
49          led_blinking_task();
50      }
51
52      return 0;
53  }

```

Figura: 40: screenshot del codice dal file main.c.

3.5.5 – API per la classe Vendor

Per poter comunicare via USB, la libreria mette a disposizione delle API, cioè delle funzioni ad alto livello. Nel progetto corrente sono state utilizzate le seguenti, specifiche per la classe Vendor.

- ***tud_vendor_read()***: permette di ricevere dati dall'endpoint IN.
- ***tud_vendor_write()***: permette di trasmettere dati sull'endpoint OUT.
- ***tud_vendor_available()***: permette di sapere se sono stati ricevuti nuovi dati sull'endpoint IN.
- ***tud_vendor_write_available()***: permette di sapere se è possibile trasmettere nuovi dati sull'endpoint OUT.

4 – Il progetto

4.1 – Comunicazione con la IDE: *Codewarrior*

4.1.2 – Introduzione

Codewarrior è una IDE per lo sviluppo di software per sistemi embedded della Freescale.

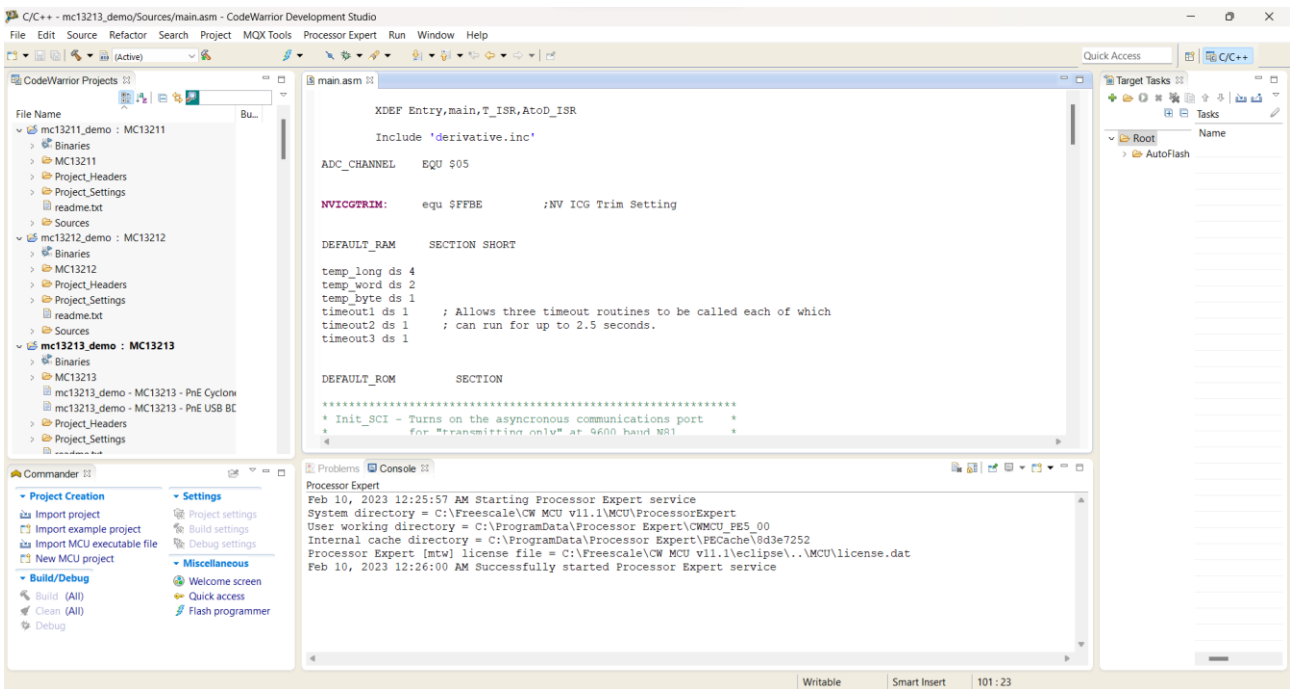


Figura 41: screenshot della IDE Codewarrior.

Una volta sviluppato un programma, è possibile scriverlo nella memoria del dispositivo target, selezionando, dalla barra superiore dell'interfaccia, il pulsante *Flash File To Target*, come si ha avuto già modo di vedere nel capitolo 2. Il Codewarrior mette a disposizione diverse modalità di connessione al dispositivo target.

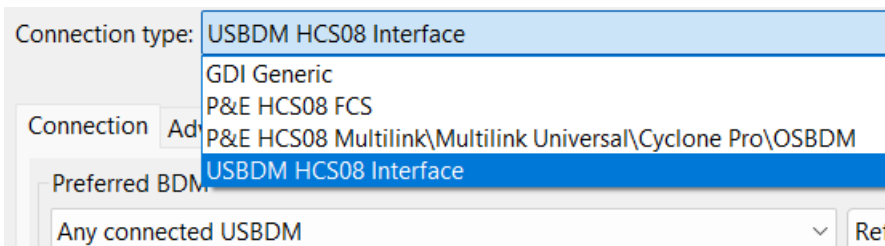


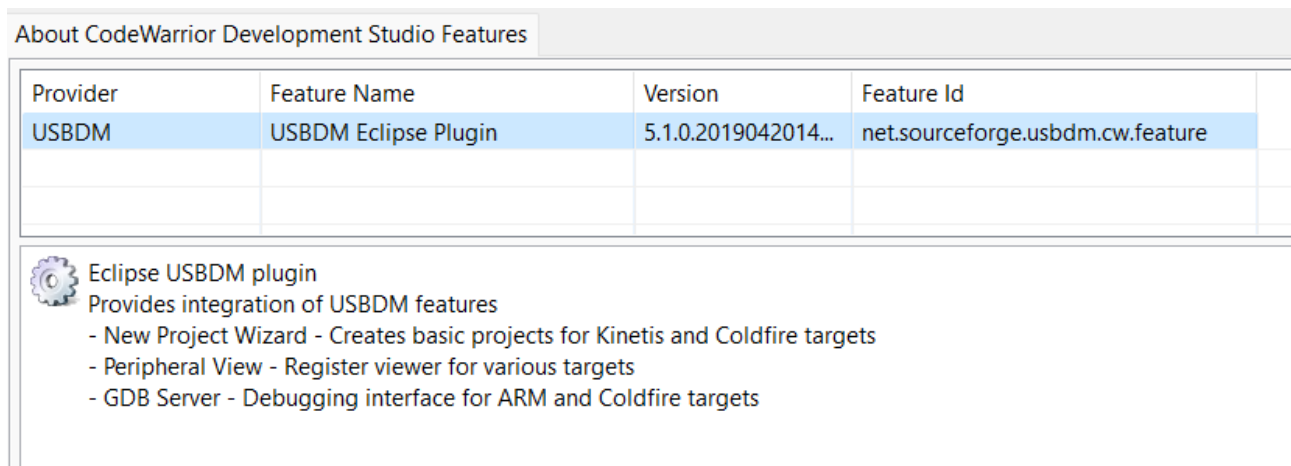
Figura 42: screenshot della IDE Codewarrior, seleziona la modalità di connessione al target.

La terza opzione è la modalità di comunicazione di default prevista dalla Freescale. In questa modalità la IDE comunica con il dispositivo target attraverso dei connettori anch'essi prodotti dalla Freescale via BDM, come ad esempio il *Cyclone Pro* oppure il *Multilink Universal*. Questi dispositivi sono connessi al PC via USB e ricevono i comandi per la scrittura del programma dalla IDE.


4.1.2 – Plugin USBDM per Codewarrior

In alternativa a ciò, è possibile selezionare l'opzione USBDM Interface. In questo caso la sigla HCS08 si riferisce alla famiglia di microcontrollori Freescale a cui appartiene il target utilizzato in questo progetto, l'MC13213. Questa opzione permette di utilizzare, al posto dei connettori ufficiali, altri microcontrollori, cioè tutti quelli supportati dal progetto USBDM.

E' importante sottolineare che questa alternativa non è nativamente disponibile all'interno dell'ambiente Codewarrior, ma è resa disponibile dopo aver installato un plugin sviluppato dallo stesso autore del progetto open source USBDM. Nel seguito si utilizzerà proprio questa opzione, con l'unica differenza, rispetto a quanto previsto dall'autore del progetto originale, che la comunicazione tra la IDE e il target non avviene per mezzo di uno dei microcontrollori supportati dal progetto USBDM, ma attraverso l'RP2040 appositamente per questa ragione.



Provider	Feature Name	Version	Feature Id
USBDM	USBDM Eclipse Plugin	5.1.0.2019042014...	net.sourceforge.usbdm.cw.feature

 Eclipse USBDM plugin
Provides integration of USBDM features

- New Project Wizard - Creates basic projects for Kinetis and Coldfire targets
- Peripheral View - Register viewer for various targets
- GDB Server - Debugging interface for ARM and Coldfire targets

Figura: 43: screenshot della IDE Codewarrior, lista dei plugin installati.

4.1.3 – Comando di scrittura

Dopo aver configurato la connessione e aver cliccato su *Erase&Program*, come visto nel capitolo 2, Codewarrior segue tre passaggi per scrivere sulla memoria del target:

- **Unprotect:** toglie la protezione dalla memoria FLASH del microcontrollore, in modo che questa possa essere modificata da sorgenti esterne, come ad esempio attraverso comandi BDM. Questo passaggio può essere evitato, disabilitando una spunta prima di cliccare su *Erase&Program*.
- **Erase:** cancella la memoria del microcontrollore.
- **Program:** scrive sulla memoria Flash del dispositivo il programma desiderato.

Per eseguire queste operazioni, Codewarrior trasmette via USB una sequenza di comandi, che vengono interpretati dall'RP2040, il quale comunica con il microcontrollore Freescale attraverso l'interfaccia BDM.

4.2 – Comando di sincronizzazione

4.2.1 – Connettere il target

Le prerogative affinché l'RP2040 possa trasmettere comandi BDM sono le seguenti.

- Il target si trovi nella modalità *Active Background*.
- L'RP2040 conosca la frequenza di comunicazione del target.

Per soddisfare il primo punto, è necessario far eseguire al target un *power-on* reset, mentre il PIN BKGD dell'interfaccia BDM viene tenuto al livello basso, mentre per il secondo è necessario che l'RP2040 trasmetta un comando di sincronizzazione (**SYNC**).

```
80 //-----+
81 // USBDM task
82 //-----+
83 void usbdm_task(void)
84 {
85     // Check if board button is pressed
86     uint32_t const btn = board_button_read();
87
88     if (btn)
89     {
90         // Hold BKGD pin low for 5 seconds
91         bdm_connect();
92
93         // SYNC command
94         bdm_cmd_sync();
95     }
96
```

Figura: 44: screenshot del codice dal file

Analizziamo ora il task USBDM, ovvero quella funzione che viene chiamata periodicamente dal programma principale.

Quando viene premuto l'unico pulsante della board, viene tenuto il pin BKGD al livello basso per cinque secondi. A seguito di ciò si richiede all'utente di spegnere e riaccendere il dispositivo target, così da permettergli di entrare nella modalità *Active Background*. Trascorsi i cinque secondi, viene lanciato il comando di sincronizzazione, ovvero quello che serve a richiedere al target la frequenza di comunicazione BDM. Se tutto è andato a buon fine, l'host è ora connesso al target e si può passare alla fase successiva.

Rimane da vedere come è stato realizzato il comando di sincronizzazione e come fa l'host a calcolare la frequenza del target.

4.2.2 – Calcolo della frequenza

Il comando di sincronizzazione comincia quando l'host setta il PIN BKGD al livello basso e lo mantiene tale per 128 cicli di clock, alla frequenza più bassa possibile. Dopodiché, il target risponde portando la linea bassa a sua volta, sempre per 128 colpi di clock, questa volta alla sua

frequenza di clock. In questo modo, se l'host misura il tempo che la linea impiega a tornare alta quando è il target a pilotarla, può calcolare la frequenza di clock del dispositivo target.

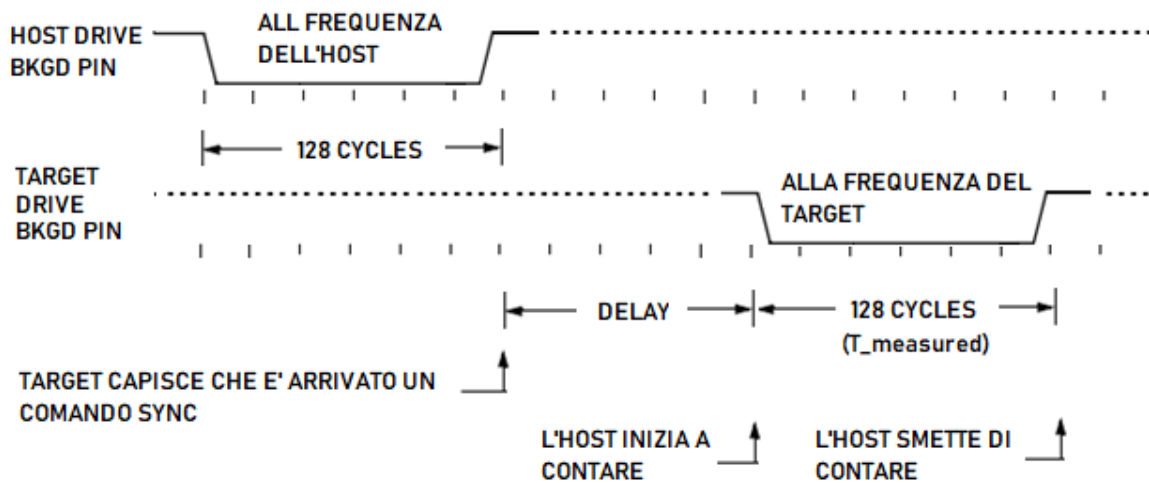


Figura: 45: comando di sincronizzazione BDM.

Per eseguire questo comando si è scelto di impostare la frequenza del modulo PIO a 2MHz. Quindi il periodo è pari al suo reciproco.

$$T_{HOST} = \frac{1}{F_{HOST}} = \frac{1}{2 \text{ MHz}} = 0.5 \mu\text{s};$$

Per misurare il tempo che la linea impiega a tornare al livello alto, il PIO incrementa un contatore interno ogni due cicli istruzione. Questo significa che il contatore viene incrementato ogni microsecondo.

$$T_{tick} = 2 \times T_{HOST} = 1 \mu\text{s};$$

Quindi per misurare il tempo in cui la linea rimane bassa, è sufficiente moltiplicare il valore precedente per il numero di incrementi del contatore.

$$T_{measured} = ticks \times T_{tick};$$

Dividendo tale valore per 128, cioè il numero di cicli in cui la linea rimane bassa, si ottiene il periodo, in microsecondi, del clock del target.

$$T_{TARGET} = \frac{T_{measured}}{128};$$

Per calcolare la frequenza in Hz, è sufficiente calcolare il reciproco del valore precedente e moltiplicarlo per 10^6 .

$$F_{TARGET} = \frac{1}{T_{TARGET}} \times 10^6;$$

Analizziamo ora il codice che realizza queste operazioni.

4.2.3 – Codice per il modulo PIO: comando sync

```
≡ bdm-sync.pio
1  .program bdm_sync
2  .side_set 1 opt
3
4      in null, 32                ; Set isr to all zeros
5      mov y, !isr                ; Copy isr to scratch y bit-reversed. Now scratch y is all ones.
6
7      set pindirs, 1             ; set pin direction to output
8
9      set x, 30                  side 0    ; set loop counter to 30(in order to loop for 31 times)
10 wait_128:
11     nop                        [1]
12     jmp x-- wait_128           [1]    ; 1 + (31 * 4) + 3 = 128 cycles
13     nop                        [2]    ; Hold pin low for 128 cycles
14     nop                        side 1   ; Speed-up pulse
15     set pindirs, 0             ; Removes all drive to the BKGD pin so it reverts to high impedance
16     wait 0 pin 0                ; Wait for the pin to go low(input mapping is used)
17
18 loop:
19     jmp pin_end_timer           ; If pin is high, jump to end_timer
20     jmp y-- loop                ; If y is non-zero, decrement y and restart the loop
21
22 timeout:
23     push                        ; If y reaches zero -> timeout. TODO: lower the timeout(now is ~18min)
24
25 end_timer:
26     mov isr, !y                 ; Copy y to isr bit-reversed and push data.
27     push
28
```

Figura: 46: screenshot del codice dal file bdm-sync.pio.

1. Il primo passaggio da fare è inizializzare il contatore. Dal momento che il modulo PIO non dispone di un'istruzione per *incrementare* le variabili, ma soltanto per *decrementarle*, il contatore dovrà partire da un valore diverso da zero e essere decrementato fino a zero. Per fare ciò (riga 4) si inizializza il registro ISR con tutti zeri. Dopodiché (riga 5) si trasferisce il contenuto dell'ISR nel registro *Scratch Y* in modalità *bit-reversed*. In questo modo si è inizializzato il registro *Scratch Y* con tutti 1. Siccome il registro ha una dimensione pari a 32, il numero da cui parte il contatore è 4.294.967.296.
2. A questo punto si procede a generare il comando di sincronizzazione. Si inizializza il registro *Scratch X* al valore di 30, mentre, tramite operazione *Side-set*, si porta il PIN BKGD al livello basso.
3. Le due istruzioni successive (riga 11 e riga 12) vengono ripetute per 31 volte. Infatti, l'istruzione alla riga 12 decrementa il *loop counter* salvato su *Scratch X* e ritorna alla riga 11, se questo non è arrivato a zero.
4. Siccome le istruzioni alla riga 11 e 12 aggiungono un ulteriore ciclo di ritardo ciascuna, il numero totale di cicli in cui la linea rimane bassa, arrivati a riga 14, è pari a $31 \cdot 4 = 124$, a cui si deve aggiungere un ciclo per l'istruzione di riga 9 e tre cicli per quella di riga 13, che in totale corrispondono a 128 cicli di clock.
5. La linea può tornare alta attraverso l'operazione *Side-set* eseguita a riga 14.
6. A questo punto si attende la risposta del target. Per fare ciò, si utilizza l'istruzione **WAIT** (riga 16), che attende che la linea venga portata bassa dal target, in modo da iniziare la misurazione.

7. Alla riga 19 si controlla se il PIN BKGD è stato riportato al livello alto dal target oppure è ancora basso. Nel primo caso si passa al punto 8, mentre nel secondo caso, si decrementa il contatore su *Scratch Y* e si ritorna alla riga 19 per un nuovo controllo.
8. Quando la linea è tornata al livello alto (*end_timer*), si copia il contatore dallo *Scratch Y* sull'ISR (riga 26), in modalità *bit-reversed* e poi si trasferisce questo valore dall'ISR alla **RX FIFO**, tramite l'istruzione **PUSH** (riga 27). Avendo copiato il contatore in modalità *bit-reversed*, il valore che il programma principale si ritrova sulla **RX FIFO**, è esattamente il numero di *incrementi* desiderato.

4.2.4 – Codice per il programma principale: calcolo della frequenza

Veniamo ora alla parte di codice in cui si calcola la frequenza del target, a partire dal valore ricavato a partire dalla procedura precedente.

```

111 void bdm_cmd_sync(void)
112 {
113     if(!is_sm_init)
114     {
115         // Get first free state machine in PIO 0
116         uint sm = pio_claim_unused_sm(pio, true);
117
118         is_sm_init = true;
119     }
120
121     // Overwrite pio instruction memory with "bdm-sync.pio" program, execute it and return ticks
122     ticks = (uint16_t)sync(pio, sm, SYNC_FREQ);
123
124     // 1 "tick" corresponds to 2 pio instruction cycles, since state machine increment "tick" every 2 instruction cycles
125     // T_measured = ticks * T_tick, where T_tick = 2 * T_pio = 2 * (1/F_pio) = 2 * (1/2MHZ) = 1us.
126     // T_measured corresponds to 128 MCU clock cycles, so T_MCU = T_measured / 128
127     // Finally F_MCU = 1/T_MCU = (1/T_measured) * 128
128     float T_measured_us = (float)(ticks * 2 * SYNC_PERIOD);
129     // F_MCU is in HZ
130     float F_MCU = (1/T_measured_us) * 128 * MHZ;
131
132     pio_freq = F_MCU;
133
134     // Pio memory has been overwritten, so it needs to be re-initialized
135     is_bdm_data_init = false;
136     is_freq_known = true;
137 }
138

```

Figura: 47: screenshot del codice dal file *pio_functions.c*.

Nota: il PIO viene impostato per lavorare alla frequenza di 2MHz dalla funzione *sync()*, a cui viene passato il parametro **SYNC_FREQ**, che vale 2000000 (Hz), mentre **SYNC_PERIOD** è l'inverso del valore precedente in microsecondi e vale 0.5.

Questa funzione è quella richiamata all'interno dell'USBDM task, visto nel paragrafo 4.2.1. Alla riga 122, viene richiamata la funzione **sync()**, che è quella responsabile della scrittura sulla memoria del PIO del programma analizzato nel paragrafo precedente, che ritorna il numero di incrementi (*ticks*) del contatore nei 128 cicli di clock in cui la linea è mantenuta bassa dal target.

Nelle righe successive (da 128 a 130), si calcola la frequenza del dispositivo target, tramite la procedura descritta nel paragrafo 4.2.2.

```

171 // Sync
172 uint sync(PIO pio, uint sm, float pio_freq)
173 {
174     // Clear memory and fifos and add program
175     uint offset = pio_program_init(pio, sm, &bdm_sync_program);
176
177     // Calculate the PIO clock divider
178     float div = get_pio_clk_div(pio_freq);
179
180     // Initialize the program using the helper function in our .pio file
181     bdm_sync_program_init(pio, sm, offset, DATA_PIN, div);
182
183     // Start running bdm-sync PIO program in the state machine
184     pio_sm_set_enabled(pio, sm, true);
185
186     // Wait for the sm to push data in rx fifo
187     uint ticks = pio_sm_get_blocking(pio, sm);
188
189     return ticks;
190 }

```

Figura: 48: screenshot del codice dal file pio_functions.c.

Questa è la funzione che scrive sulla memoria del PIO il programma che serve per eseguire il comando di sincronizzazione.

4.3 – Ricezione dei comandi USB e gestione dei pacchetti

4.3.1 – USBDM task

```
C main.c > usbdm_task(void)
83 void usbdm_task(void)
84
85 // Check if board button is pressed
86 uint32_t const btn = board_button_read();
87
88 if (btn)
89 {
90     // Hold BKGD pin low for 5 seconds
91     bdm_connect();
92
93     // SYNC command
94     bdm_cmd_sync();
95 }
96
97 USBDM_ErrorCode command_status;
98
99 if (tud_vendor_available())
100 {
101     // Receive command from EP1 OUT
102     command_status = receive_USB_command();
103
104     if ((uint8_t)command_status==BDM_RC_OK)
105     {
106         | blink_interval_ms = BLINK_COMMAND_OK;
107     }
108     else
109     {
110         | blink_interval_ms = BLINK_ALWAYS_OFF;
111     }
112 }
113
114 return;
115
```

Figura: 49: screenshot del codice dal file main.c.

La funzione **usbdm_task()** è quella che viene richiamata periodicamente dal programma principale.

A riga 99 la funzione **tud_vendor_available()**, ovvero una delle API della libreria TinyUSB, vedi paragrafo 3.5.5, controlla se sono stati trasferiti nuovi dati dalla IDE sull'endpoint OUT. Nel caso ve ne fossero di nuovi, viene invocata la funzione **receive_USB_command()**, che si occuperà di gestire i dati in arrivo e costituisce l'*entry point* di tutta l'applicazione.

4.3.2 – Codifica dei comandi USB

Prima di analizzare la funzione, è importante sapere come vengono codificati i comandi USB dalla IDE.

Ciascun pacchetto ha una lunghezza massima di 64 byte, cioè pari alla dimensione massima dell'endpoint. Per questo motivo, i comandi USB più lunghi vengono divisi in due o più pacchetti. I dati ricevuti in ciascun pacchetto devono essere opportunamente concatenati per comporre l'intero comando USB. Un comando USB è lungo al massimo 254 byte.

Numero del pacchetto	Primo byte	Secondo byte	Byte successivi
1	Dimensione dell'intero comando	Codice esadecimale del comando USB	Data
2	0x00	Data	Data
Dal 3 in poi	Data	Data	Data

1. Il primo pacchetto inizia con la dimensione dell'intero comando. Quando questo supera la lunghezza del pacchetto ricevuto, significa che il comando verrà suddiviso in più pacchetti. Il secondo byte di questo pacchetto contiene il codice del comando USB da eseguire. I byte seguenti contengono dei dati, il cui valore dipende dal tipo di comando trasmesso.
2. Il primo byte del secondo pacchetto contiene sempre uno zero. Ciò è stato fatto per distinguerlo dal primo pacchetto. I dati ricevuti devono essere concatenati a quelli ricevuti nel primo pacchetto, scartando il primo byte, che contiene uno zero e non costituisce nessuna informazione.
3. Dal terzo pacchetto in poi, vengono trasmessi solamente dei dati, quindi è sufficiente concatenarli ai dati ricevuti in precedenza così come sono.

La ricezione del comando termina, quando il numero totale di byte ricevuti eguaglia quelli dichiarati nel primo byte del primo pacchetto. Questo controllo viene effettuato dopo la ricezione di ogni pacchetto.

4.3.3 – Codice per il programma principale: ricezione comandi USB

```

15  static uint8_t command_buffer[MAX_COMMAND_SIZE];
16  static uint8_t command_size = 0;
17  static uint8_t offset = 0;
18  static uint8_t saved_byte = 0;
19
20  // Signal the presence of first pkt
21  static bool first_pkt_received = false;

```

Figura: 50: screenshot del codice dal file *usbdm.c*.

1. Si definiscono, innanzitutto, le seguenti strutture.
 - Buffer di dimensione 254 byte, in cui verrà salvato l'intero comando USB (*command_buffer*).
 - Variabile in cui salvare la dimensione dell'intero comando da ricevere (*command_size*).
 - Variabile in cui salvare il conteggio dei byte ricevuti fino ad un dato momento, da utilizzare per sapere in che punto del *command_buffer* inserire i nuovi dati, al fine di costruire il comando completo (*offset*).
 - Variabile in cui salvare l'ultimo byte del primo pacchetto, il cui scopo sarà chiarito in seguito (*saved_byte*).

- Flag per segnalare l'avvenuta ricezione del primo pacchetto (*first_pkt_received*). Questo perché il primo pacchetto e quelli dal terzo in poi sono indistinguibili, siccome iniziano sempre con un byte diverso da zero.

```

86 USBDM_ErrorCode receive_USB_command(void)
87 {
88     uint8_t temp_buffer[BDM_IN_EP_MAXSIZE];
89
90     uint8_t byte_count = tud_vendor_read(temp_buffer, BDM_IN_EP_MAXSIZE);
91
92     // Get first byte
93     uint8_t first_byte = temp_buffer[0];
94

```

Figura: 51: screenshot del codice dal file *usbdm.c*.

2. Passiamo ora all'interno della funzione. Prima di tutto si definisce un buffer temporaneo, di lunghezza pari a 64, ovvero la dimensione massima dell'endpoint.
3. Utilizzando l'API della libreria TinyUSB, *tud_vendor_read()*, si salvano i dati ricevuti via USB in *temp_buffer*.
4. Dopodiché si scrive il primo byte in una variabile, per controllarne il valore.

```

95     // 1st pkt or additional data
96     if (first_byte!=0)
97     {
98         if (!first_pkt_received)
99         {
100             // Save entire command size
101             command_size = first_byte;
102
103             // Save last byte of the first pkt
104             saved_byte = command_buffer[byte_count-1];
105
106             first_pkt_received = true;
107         }
108
109         if (command_size > MAX_COMMAND_SIZE)
110         {
111             command_size = MAX_COMMAND_SIZE;
112         }
113
114         // Save data in command buffer
115         memcpy(command_buffer + offset, temp_buffer, byte_count);
116     }

```

Figura: 52: screenshot del codice dal file *usbdm.c*.

5. Se il primo byte è diverso da zero, allora si tratta del primo pacchetto oppure di un pacchetto dal terzo in poi.

6. Se il flag *first_pkt_received* non è settato, allora significa che si tratta proprio del primo pacchetto. Quindi si procede a salvare la dimensione dell'intero comando in *command_size* e il valore dell'ultimo byte in *saved_byte*. Dopo aver settato il flag *first_pkt_received* a *true*, si passa al punto successivo.
7. Si trasferisce il contenuto di *temp_buffer* in *command_buffer*, iniziando a scrivere a partire da *offset*. Inizialmente *offset* è impostato a 0, ma alla fine della ricezione di ogni pacchetto, si incrementa del numero di byte ricevuti, così, al successivo pacchetto, si inizia a scrivere dalla fine del pacchetto precedente.

```
117 //2nd packet
118 else
119 {
120 // Save data in command buffer
121 memcpy(command_buffer + (offset-1), temp_buffer, byte_count);
122
123 // Overwrite the first byte of the second packet, which contains zero, with the last byte of the first packet previously saved.
124 command_buffer[offset-1] = saved_byte;
125
126 // Do not consider the first 0x00 byte in 2nd pkt in data count
127 byte_count--;
128 }
129
```

Figura: 53: screenshot del codice dal file *usbdm.c*.

8. Se il primo byte, invece, è uguale a zero, allora si sta ricevendo il secondo pacchetto. I dati, in questo caso, vengono salvati in maniera leggermente diversa rispetto da quanto fatto per tutti gli altri pacchetti. Questo siccome è necessario eliminare il primo byte del secondo pacchetto, che contiene uno zero. Si salvano, dunque, i nuovi dati ricevuti, a partire dalla locazione di memoria in cui è contenuto l'ultimo byte del primo pacchetto (*offset-1*) e non quella successiva (*offset*). Così facendo, l'ultimo byte del primo pacchetto viene sovrascritto dallo zero.
9. Rimane ora da eliminare lo zero e ripristinare il dato sovrascritto da esso. Si ricorre, allora alla variabile *saved_byte*, in cui era stato salvato il valore dell'ultimo byte del primo pacchetto.
10. Da notare che la dimensione dell'intero comando, trasmessa nel primo byte del primo pacchetto, non include lo zero appena scartato. Per tale motivo si decrementa *byte_count* di una unità.

```

130     offset += byte_count;
131
132     // All data has been received
133     if(offset == command_size)
134     {
135         // Execute the command
136         // NOTE: after executing a command, command_exec return the number of bytes to send back to host;
137         uint8_t return_size = command_exec(command_buffer);
138
139         send_USB_response(command_buffer, return_size);
140
141         // Reset
142         first_pkt_received = false;
143         offset = 0;
144
145         // Return command status
146         return command_buffer[0];
147     }
148     else
149     {
150         return BDM_RC_BUSY;
151     }
152 }

```

Figura: 54: screenshot del codice dal file `usbdm.c`.

11. Si incrementa `offset` di `byte_count`.
12. Una volta terminata la ricezione del comando, è possibile passare alla fase successiva, che consiste nell'eseguire il comando ed elaborare la risposta da trasmettere alla IDE via USB.

4.4 – Esecuzione dei comandi USB

4.4.1 – Lista di comandi

La lista completa dei comandi USB disponibili è la seguente, molti dei quali invocano a loro volta comandi BDM per completare la richiesta.

Numero	Codice	Nome	Descrizione
0	0x0	CMD_USBDM_GET_COMMAND_STATUS	Ritorna il codice di errore dell'ultimo comando USB.
1	0x1	CMD_USBDM_SET_TARGET	Seleziona con quale target connettersi.
2	0x2	CMD_USBDM_SET_VDD	Imposta la tensione di alimentazione del target, qualora sia alimentato tramite interfaccia BDM. In questo progetto il target si auto-alimenta, quindi il valore impostato dall'utente viene ignorato.
3	0x3	CMD_USBDM_DEBUG	<i>Non implementata.</i>
4	0x4	CMD_USBDM_GET_BDM_STATUS	Ritorna una struttura dati, utile alla IDE per sapere come è configurato il target.
5	0x5	CMD_USBDM_GET_CAPABILITIES	Ritorna una struttura dati, contenente informazioni riguardo alle funzionalità supportate dall'interfaccia USBDM.
6	0x6	CMD_USBDM_SET_OPTIONS	Imposta una struttura dati che contiene informazioni che riguardano l'alimentazione del target e la modalità di configurazione della velocità di comunicazione BDM.
7	0x7	NO FUNCTION	ILLEGAL
8	0x8	CMD_USBDM_SET_CONTROL_PINS	<i>Non implementata.</i>
9	0x9	NO FUNCTION	ILLEGAL
10	0xA	NO FUNCTION	ILLEGAL
11	0xB	NO FUNCTION	ILLEGAL
12	0xC	NO FUNCTION	ILLEGAL
13	0xD	NO FUNCTION	ILLEGAL
14	0xE	NO FUNCTION	ILLEGAL
15	0xF	CMD_USBDM_CONNECT	Si connette al dispositivo target ed esegue il comando di sincronizzazione.
16	0x10	CMD_USBDM_SET_SPEED	Imposta la velocità di comunicazione BDM. In questo progetto, la velocità di comunicazione BDM è ottenuta attraverso il comando di sincronizzazione. Quindi, anche se

			impostata dall'utente, questa viene ignorata.
17	0x11	CMD_USBDM_GET_SPEED	Ritorna la velocità di comunicazione BDM, misurata attraverso il comando di sincronizzazione. L'unità di misura utilizzata è il numero di cicli di clock alla frequenza di 60MHz presenti 128 cicli di clock alla frequenza del target.
18	0x12	NO FUNCTION	ILLEGAL
19	0x13	NO FUNCTION	ILLEGAL
20	0x14	CMD_USBDM_READ_STATUS_REG	Esegue il comando BDM per leggere il contenuto del registro di stato del dispositivo target e ne ritorna il contenuto.
21	0x15	CMD_USBDM_WRITE_CONTROL_REG	Esegue il comando BDM per scrivere sul registro di controllo del dispositivo target.
22	0x16	CMD_USBDM_TARGET_RESET	Resetta il target, scrivendo, tramite comando BDM, sul registro di reset del dispositivo.
23	0x17	CMD_USBDM_TARGET_STEP	<i>Non implementata.</i>
24	0x18	CMD_USBDM_TARGET_GO	Esegue il comando BDM per far uscire il dispositivo target dalla modalità <i>Active Background</i> .
25	0x19	CMD_USBDM_TARGET_HALT	Esegue il comando BDM per far entrare il dispositivo target dalla modalità <i>Active Background</i> .
26	0x1A	CMD_USBDM_WRITE_REG	Esegue il comando BDM per scrivere su un registro del dispositivo target (PC, SP, HX, A, CCR).
27	0x1B	CMD_USBDM_READ_REG	Esegue il comando BDM per leggere da un registro del dispositivo target (PC, SP, HX, A, CCR) e ritorna il valore letto.
28	0x1C	NO FUNCTION	ILLEGAL
29	0x1D	NO FUNCTION	ILLEGAL
30	0x1E	CMD_USBDM_WRITE_BKPT	<i>Non implementata.</i>
31	0x1F	CMD_USBDM_READ_BKPT	<i>Non implementata.</i>
32	0x20	CMD_USBDM_WRITE_MEM	Scrive dei byte a partire da una certa locazione di memoria, attraverso comandi BDM.
33	0x21	CMD_USBDM_READ_MEM	Legge dei byte a partire da una certa locazione di memoria, attraverso comandi BDM. Ritorna tutti i byte letti.

4.4.2 – Codice per il programma principale: esecuzione di un comando USB

```
C cmd_proc.c > ...
99  uint8_t command_exec(uint8_t* command_buffer)
100  {
101  uint8_t command_size = command_buffer[0];
102  BDMCommands command = command_buffer[1];
103
104  // Default response size (maybe will be changed inside a function)
105  response_size = 1;
106
107  switch((uint8_t)command)
108  {
109  case CMD_USBDM_GET_COMMAND_STATUS: //0
110  {
111  break;
112  }
113  case CMD_USBDM_SET_TARGET: //1
114  {
115  command_status = _cmd_usbdm_set_target(command_buffer);
116  break;
117  }
118  case CMD_USBDM_SET_VDD: //2
119  {
120  command_status = _cmd_usbdm_set_vdd(command_buffer);
121  break;
122  }
123  case CMD_USBDM_GET_BDM_STATUS: //4
124  {
125  command_status = _cmd_usbdm_get_bdm_status(command_buffer);
126  break;
127  }
128  case CMD_USBDM_GET_CAPABILITIES: //5
129  {
130  command_status = _cmd_usbdm_get_capabilities(command_buffer);
131  break;

```

Figura: 55: screenshot del codice dal file `cmd_proc.c`.

La funzione **`command_exec()`** viene chiamata al termine della ricezione del comando USB. Essa ha a disposizione il `command_buffer`, dove è memorizzato il comando USB da eseguire.

Tramite uno *switch-case*, si seleziona la funzione che serve ad eseguire il comando desiderato, sulla base del valore del secondo byte del `command_buffer`, che contiene il codice del comando USB.

Ogni funzione chiamata segue sempre la stessa procedura.

- Legge i dati dal `command_buffer` per interpretare la richiesta.
- Soddisfa la richiesta.
- Una volta terminata la sequenza di operazioni, salva la risposta da trasmettere alla IDE nel `command_buffer`. Tipicamente la risposta è composta da un primo byte che contiene il codice di errore del comando (`BDM_RC_OK = 0x00`, se tutto è andato buon fine), seguito da una serie di dati, il cui formato cambia a seconda del comando eseguito.

Si analizzerà ora uno tra i comandi che più di frequente vengono richiesti dalla IDE, ovvero quello che riguarda la scrittura sulla memoria del target.

4.4.3 – Esempio: comando USB WRITE MEM

Il formato dei dati presenti sul *command_buffer* è il seguente.

Command buffer per il comando <i>write mem</i>								
0	1	2	3	4	5	6	7	8 in poi
Dimensione totale comando	0x20	Modalità di scrittura.	Numero di byte da scrivere.	Indirizzo a 32 bit.				Dati da scrivere.

```

556  ///! HCS08/RS08 - Write block of bytes to memory
557  ///!
558  ///! @note
559  ///!  command_buffer          \n
560  ///!  - [2]      = element size/mode          \n
561  ///!  - [3]      = # of bytes                 \n
562  ///!  - [4..7]   = address [MSB ignored]      \n
563  ///!  - [8..N]  = data to write
564  ///!
565  ///! @return
566  ///!  == \ref BDM_RC_OK => success          \n
567  ///!  != \ref BDM_RC_OK => error           \n
568  ///!
569  uint8_t _cmd_usbdm_write_mem(uint8_t* command_buffer)
570  {
571      uint8_t mode      = command_buffer[2];
572      uint8_t count     = command_buffer[3];
573      uint8_t addr_h    = command_buffer[6];
574      uint8_t addr_l    = command_buffer[7];
575      uint16_t addr     = (uint16_t)((addr_h<<8) | addr_l);
576      uint8_t *data_ptr = command_buffer+8;
577

```

Figura: 56: screenshot del codice dal file *cmd_proc.c*.

1. Esistono due tipi di modalità di scrittura, quella classica e quella veloce. Al momento risulta implementata soltanto quella classica. Sulla variabile *count* è memorizzato il numero di byte da scrivere sulla memoria, mentre in *addr* l'indirizzo dalla quale cominciare la scrittura. Per finire si salva in *data_ptr* il puntatore alla locazione di memoria dell'RP2040 nella quale sono salvati i dati da scrivere nel target.

```

578     if (mode & MS_FAST)
579     {
580         // Not supported (yet)
581         return BDM_RC_FEATURE_NOT_SUPPORTED;
582     }
583     else
584     {
585         while (count > 0)
586         {
587             bdm_cmd_write_byte((uint8_t)(addr>>8), (uint8_t)(addr&0xFF), *data_ptr);
588
589             count--;          // decrement count of bytes
590             data_ptr++;      // increment buffer pointer
591             addr++;          // increment memory address
592         }
593     }
594
595     return BDM_RC_OK;
596 }

```

Figura: 57: screenshot del codice dal file cmd_proc.c.

2. Il comando BDM per scrivere dati sulla memoria del target è il comando **WRITE BYTE**, che scrive un byte *per volta* all'indirizzo di memoria desiderato. Siccome il comando USB richiede di scrivere un certo numero di byte a partire da una data locazione, occorre richiamare il comando BDM **WRITE BYTE** tante volte quanti sono i byte da scrivere. Dopo ogni chiamata, è necessario incrementare il valore del puntatore dei dati, così come il valore dell'indirizzo in cui scrivere e decrementare il contatore dei byte rimanenti. Per eseguire il comando BDM, si utilizza la funzione ***bdm_cmd_write_byte()***, che prende in ingresso l'indirizzo della memoria e il byte da scrivere.

4.5 – Esecuzione dei comandi BDM

4.5.1 – Codice per il programma principale: eseguire un comando BDM

I passaggi da seguire per eseguire un comando BDM sono i seguenti.

- Durante l'esecuzione di un comando USB, può essere invocata una o più tra le funzioni che implementano i comandi BDM.
- Tali funzioni scrivono su *data_buffer* i dati da trasmettere via BDM e invocano la funzione ***bdm_command_exec()***.
- Questa scrive sulla memoria del PIO il codice per eseguire i comandi BDM, costruisce una parola a 32 bit contenente l'intero comando BDM, utilizzando la funzione ***make_data()***, e la scrive sulla **TX FIFO** del modulo PIO.
- A questo punto inizia il trasferimento BDM attraverso il modulo PIO.
- Si attende la risposta del target e, qualora il comando preveda la ricezione di dati, questi vengono salvati direttamente sul *command_buffer*, da cui verranno letti per essere trasferiti via USB indietro alla IDE.

```
28 // Data buffer
29 ///! @note : Format
30 ///!      - [0]    = # of bytes to tx (up to 4)
31 ///!      - [1]    = # of bytes to rx (up to 2)
32 ///!      - [2]    = BDM command
33 ///!      - [3]    = 1st byte parameter (opt)
34 ///!      - [4]    = 2nd byte parameter (opt)
35 | ///!      - [5]    = 3rd byte parameter (opt)
36 static uint8_t data_buffer[MAX_BDM_COMMAND_SIZE];
37
```

Figura: 58: screenshot del codice dal file *bdm.c*.

Il *data_buffer* ha la seguente struttura. Il significato assunto dagli ultimi tre byte cambia a seconda del comando.

Data_buffer					
TX_BYTE_COUNT	RX_BYTE_COUNT	COMMAND	FIRST_BYTE	SECOND_BYTE	THIRD_BYTE

```

37 // Build word data from data_buffer
38 uint _make_data()
39 {
40     uint data = 0;
41     uint8_t byte_count = data_buffer[TX_BYTE_COUNT];
42
43     for (int i=COMMAND; i<byte_count+COMMAND; i++)
44     {
45         data = data_buffer[i] | (data<<BYTE);
46     }
47
48     return data;
49 }

```

Figura: 59: screenshot del codice dal file bdm.c.

La funzione **make_data()** concatena in un'unica parola a 32 bit i dati presenti su *data_buffer*.

```

63 uint bdm_command_exec(void)
64 {
65     if(!is_sm_init)
66     {
67         // Get first free state machine in PIO 0
68         uint sm = pio_claim_unused_sm(pio, true);
69
70         is_sm_init = true;
71     }
72
73     // Check if frequency is known
74     if (!is_freq_known)
75     {
76         bdm_cmd_sync();
77     }
78
79     // Check if the program is in the pio memory
80     if(!is_bdm_data_init)
81     {
82         // Overwrite pio instruction memory with "bdm-data.pio" program and return offset
83         pio_offset = bdm_init(pio, sm, pio_freq);
84
85         is_bdm_data_init = true;
86     }
87

```

Figura: 60: screenshot del codice dal file bdm.c.

```

88     // Make data based on bytes in data_buffer
89     uint data = _make_data();
90     uint8_t tx_bit_count = data_buffer[TX_BYTE_COUNT]*BYTE;
91     uint8_t rx_bit_count = data_buffer[RX_BYTE_COUNT]*BYTE;
92
93     do_bdm_command(pio, sm, data, tx_bit_count, rx_bit_count, pio_offset);
94
95     // Wait the end of the operation
96     wait_end_operation(pio, sm);
97
98     uint received_data = 0;
99     // Read data from rx fifo
100    if(!pio_sm_is_rx_fifo_empty(pio, sm))
101    {
102        |   received_data = pio_sm_get(pio, sm);
103    }
104
105    return received_data;
106 }
107

```

Figura: 61: screenshot del codice dal file bdm.c.

A riga 89 viene richiamata la funzione **make_data()**, che ritorna la parola da scrivere sulla **TX FIFO**. Dopo aver recuperato il numero di byte da trasmettere da *data_buffer* e averli convertiti nel numero di *bit* da leggere e scrivere, si richiama la funzione **do_bdm_command()**, che era già stata commentata nel paragrafo dedicato all'implementazione del protocollo BDM (paragrafo 3.3.4).

La funzione ritorna una parola a 32 bit che contiene i dati letti attraverso l'interfaccia BDM.

4.5.2 – Esempio: comando BDM WRITE BYTE

```

369    // Write an 8 bit data word to 16 bit register
370    void bdm_cmd_write_byte(uint8_t addr_h, uint8_t addr_l, uint8_t data)
371    {
372        |   data_buffer[TX_BYTE_COUNT] = 4; // Transmit 4 bytes
373        |   data_buffer[RX_BYTE_COUNT] = 0;
374        |   data_buffer[COMMAND] = WRITE_BYTE;
375        |   data_buffer[FIRST_PARAMETER] = addr_h; // Address H
376        |   data_buffer[SECOND_PARAMETER] = addr_l; // Address L
377        |   data_buffer[THIRD_PARAMETER] = data;
378
379        |   bdm_command_exec();
380
381        |   return;
382    }

```

Figura: 62: screenshot del codice dal file bdm.c.

Questa è la funzione che implementa il comando BDM **WRITE BYTE**. Viene richiamata dal comando USB, ogni volta che è necessario leggere un byte dalla memoria del target, come nell'esempio del paragrafo 4.4.3. Il comando BDM **WRITE BYTE** ha la seguente sintassi.

Comando BDM WRITE BYTE			
Codice comando BDM	MSB dell'indirizzo	LSB dell'indirizzo	Byte da scrivere
0xE0	0xAA	0xAA	0xDD

Vengono salvate su `data_buffer` le seguenti informazioni:

- Numero di byte da trasmettere: siccome il comando è lungo 32 bit, si scrive di voler trasmettere 4 byte.
- Numero di byte da ricevere: questo comando BDM non riceve nessun dato, perciò viene scritto zero.
- Il codice esadecimale associato al comando BDM **WRITE BYTE**: 0xC0.
- Come primo parametro i bit più significativi dell'indirizzo a 16 bit su cui andare a scrivere.
- Come secondo parametro i bit meno significativi dello stesso indirizzo.
- Come terzo parametro, il byte da scrivere nella memoria del target.

4.5.3 – Esempio: comando BDM READ BYTE

```

397 // Read an 8 bit data word to 16 bit register
398 void bdm_cmd_read_byte(uint8_t addr_h, uint8_t addr_l, uint8_t* data_ptr)
399 {
400     data_buffer[TX_BYTE_COUNT] = 3;
401     data_buffer[RX_BYTE_COUNT] = 1;
402     data_buffer[COMMAND] = READ_BYTE;
403     data_buffer[FIRST_PARAMETER] = addr_h; // Address H
404     data_buffer[SECOND_PARAMETER] = addr_l; // Address L
405
406     uint8_t read_byte = (uint8_t)bdm_command_exec();
407
408 // Save read value into the command_buffer
409     data_ptr[0] = read_byte;
410
411     return;
412 }

```

Figura: 63: screenshot del codice dal file `bdm.c`.

Questa è la funzione che implementa il comando BDM **READ BYTE**. Viene richiamata dal comando USB, ogni volta che è necessario leggere un byte dalla memoria del target. Il comando BDM **READ BYTE** ha la seguente sintassi.

Comando BDM READ BYTE		
Codice comando BDM	MSB dell'indirizzo	LSB dell'indirizzo
0xE0	0xAA	0xAA

Vengono salvate su `data_buffer` le seguenti informazioni:

- Numero di byte da trasmettere: siccome il comando è lungo 24 bit, si scrive di voler trasmettere 3 byte.

- Numero di byte da ricevere: questo comando BDM riceve come risposta dal target il byte letto, quindi si scrive 1 byte.
- Il codice esadecimale associato al comando BDM **READ BYTE**: 0xE0.
- Come primo parametro i bit più significativi dell'indirizzo a 16 bit su cui andare a leggere.
- Come secondo parametro i bit meno significativi dello stesso indirizzo.

A differenza dell'esempio precedente, questo comando scrive i dati ricevuti sul *command_buffer*, dalla quale verranno letti per poi essere trasmessi via USB alla IDE. Nel paragrafo seguente si approfondirà quest'ultimo aspetto.

4.6 – Trasmissione risposta via USB

4.6.1 – Codice per il programma principale: trasmissione risposta USB

Al termine della ricezione di un qualsiasi comando USB, è necessario trasmettere alla IDE una risposta, che è composta sempre da due parti.

- Il primo byte contiene il codice di errore del comando. Il codice per segnalare che il comando è stato riconosciuto e non vi sono stati errori è *BDM_RC_OK*, che corrisponde al valore zero.
- Il secondo byte in poi costituisce l'effettiva risposta del dispositivo. Viene trasmessa soltanto successivamente a quei comandi che la prevedono, come per esempio i comandi di lettura dalla memoria del target.

Per poter essere trasmessi alla IDE, è necessario salvare i byte su *command_buffer*, la stessa struttura dati su cui era stato salvato il comando ricevuto tramite USB.

```
132 // All data has been received
133 if(offset == command_size)
134 {
135     // Execute the command
136     // NOTE: after executing a command, command_exec return the number of bytes to send back to host;
137     uint8_t return_size = command_exec(command_buffer);
138
139     send_USB_response(command_buffer, return_size);
140
141     // Reset
142     first_pkt_received = false;
143     offset = 0;
144
145     // Return command status
146     return command_buffer[0];
147 }
148 else
149 {
150     return BDM_RC_BUSY;
151 }
152 }
```

Figura 64: screenshot del codice dal file *usbdm.c*.

Dopo aver chiamato *command_exec()*, l'ultimo passaggio della funzione *receive_USB_command()*, di cui si riporta un estratto, è quello di invocare la funzione *send_USB_response()*. Questa prende in ingresso il buffer su cui sono salvati i dati, cioè *command_buffer*, e il numero di byte da trasmettere.

```
38 | * @note : Format
39 | *     - [0] = response
40 | *     - [1..N] = parameters
41 | */
42 | void send_USB_response(uint8_t *buffer, uint8_t byte_count)
43 | {
44 |     if (tud_vendor_write_available())
45 |     {
46 |         tud_vendor_write(buffer, byte_count);
47 |     }
48 | }
49 |
```

Figura: 65: screenshot del codice dal file usbdm.c.

Per trasmettere la risposta si utilizzano due funzioni della libreria TinyUSB. La prima controlla che ci sia spazio per nuovi dati, mentre la seconda li trasferisce all'endpoint.

5 – Test e validazione sperimentale

5.1 – Prova di scrittura sul target

Per verificare che il dispositivo progettato funzioni correttamente, è stato eseguito un test che consiste nei seguenti passaggi.

- È stato recuperato un codice di esempio in Assembly per l'MC13213.
- Il codice è stato modificato in modo che, durante l'esecuzione del programma, venissero accesi i quattro LED della board, così da segnalare l'avvenuta programmazione del programma nella memoria del microcontrollore.

```
sta PTDDD          ; Port D is an output

main_loop:
clr PTDD           ; Turn on all led
bra main_loop
```

Figura: 66: screenshot della IDE Codewarrior. Porzione del codice di test da scrivere nella memoria del target.

- Dopo aver connesso l'interfaccia di debug (RP2040), alla porta BDM del microcontrollore e alla porta USB del PC, si è eseguito un *power-on* reset della board, si è premuto il tasto sull'RP2040, così da far entrare il target in modalità *Active Background* e stimare la frequenza di comunicazione, così come descritto all'inizio del capitolo 4.
- Attraverso la IDE Codewarrior è stato avviato il comando di cancellazione e programmazione della memoria (*Erase&Program*), confermando di voler cancellarne il contenuto attraverso un *mass erase*.

```
Flash Programmer Console
Performing target initialization ...
Device MC13213_FLASH
Flash Operation.
Unprotecting .....
Unprotect Command Succeeded.
Device MC13213_FLASH
cmdwin::fl::image -f "C:\\Freescale\\CW MCU v11.1\\MCU\\CodeWarrior_Examples\\HC_Examples\\mc13213_demo\\MC13213\\mc13213_demo.abs.s19" -t "Auto
cmdwin::fl::erase image
-----
Auto-detection is successful.
File is of type Motorola S-Record Format.

Device MC13213_FLASH
Erasing .....
Erase Command Succeeded.
Device MC13213_FLASH
cmdwin::fl::write
Beginning Operation ...
-----
Flash Operation. ...
Auto-detection is successful.
File is of type Motorola S-Record Format.

Device MC13213_FLASH
Programming .....
Device MC13213_FLASH
Program Command Succeeded |
Flash Operation. done
```

Figura: 67: screenshot della IDE Codewarrior. Console della IDE, in cui vengono riportati i passaggi per scrivere sulla memoria del target.

- Dopo aver disconnesso l'interfaccia di debug e aver acceso la board, si è osservato il risultato atteso: tutti e quattro i LED della board erano accesi, così come era stato ipotizzato. Ciò significa che l'operazione di scrittura sulla memoria è andata a buon fine.

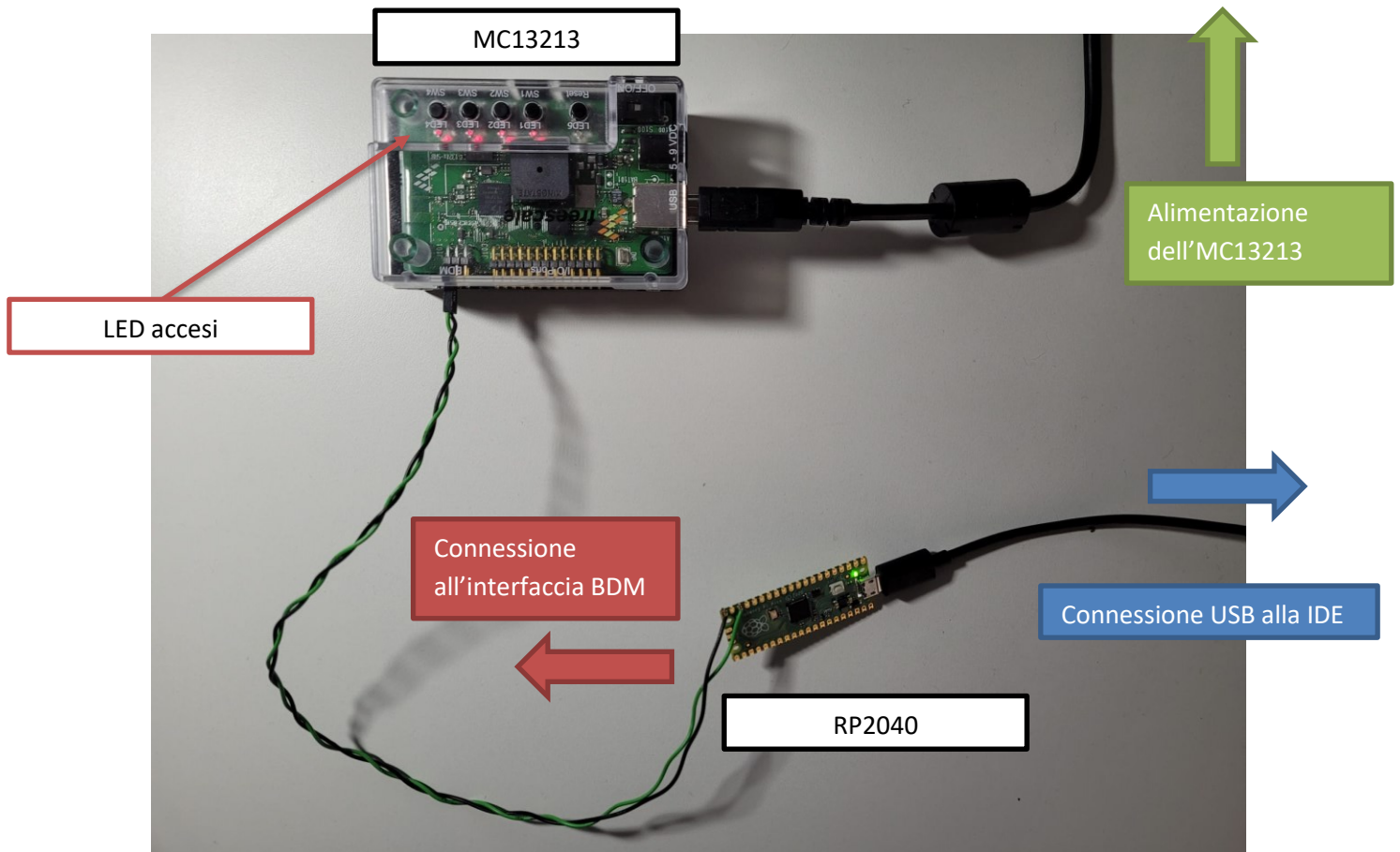


Figura: 68: Foto dell'interfaccia di debug (RP2040), connessa al dispositivo target (MC13213), in cui si dimostra che la scrittura sulla memoria del microcontrollore è andata a buon fine.

5.2 – Stima velocità di scrittura

E' stato possibile stimare la velocità di scrittura sulla flash, inserendo all'interno del codice da programmare nella memoria del microcontrollore, una direttiva per il compilatore, in grado di allocare un certo numero di byte in memoria. Dopodiché si è cronometrato il processo di scrittura descritto nel paragrafo precedente e si è utilizzato tale dato per stimare la velocità di scrittura.

Avendo scritto 20000 word, cioè 40000 byte in memoria e avendo impiegato circa 10.94 secondi, la velocità stimata di scrittura è di 29.24 kbit/s.

C'è da osservare che tale velocità è limitata dalla frequenza stessa del microcontrollore, la quale viene impostata al valore di 4.2MHz in modalità *Active Background*.

6 – Conclusioni e sviluppi futuri

6.1 – Conclusioni

In questa tesi si sono approfondite tematiche riguardanti la comunicazione USB device-side e la comunicazione con un microcontrollore attraverso un'interfaccia di debug (BDM). Si è potuto mettere in pratica quanto studiato, attraverso la realizzazione del progetto stesso, che si è, oltretutto, concluso con successo, come hanno dimostrato i test effettuati descritti nel capitolo precedente. Si è riusciti a scrivere un programma, sviluppato all'interno dell'ambiente di sviluppo Codewarrior, sulla memoria del target, garantendo una velocità di scrittura soddisfacente.

Il progetto concluso è visionabile sulla pagina Github, consultabile al seguente indirizzo: <https://github.com/lorenzobartolini00/USBDM-Pi>.

6.2 – Sviluppi futuri

Questa interfaccia di debug, sviluppata con l'RP2040 è stata progettata e testata solamente per la famiglia di microcontrollori HCS08. In futuro si potrebbe pensare di estenderne la compatibilità anche ad altri microcontrollori Freescale. Inoltre, si potrebbero approfondire le funzionalità di debug, cioè quelle in grado di inserire breakpoint ed eseguire le istruzioni passo-passo, che sono state già parzialmente implementate, ma non sono testate in maniera estensiva per poterne verificare il corretto funzionamento all'interno della IDE.

Sitografia

1. Sito del progetto USBDM:
https://usbdm.sourceforge.io/USBDM_V4.12/html/index.html
2. Sito della NXP (Freescale): <https://www.nxp.com/>
3. Datasheet dell'MC13213 (in cui è descritto anche il protocollo BDM):
<https://www.nxp.com/part/MC13213#/>
4. USB made simple – a series of articles on USB: <https://www.usbmadesimple.co.uk/>
5. Sito web dell'USB IF per specifiche 2.0: <https://www.usb.org/document-library/usb-20-specification>
6. Documentazione libreria TinyUSB: <https://docs.tinyusb.org/en/latest/>
7. Sito web dell'RP2040: <https://www.raspberrypi.com/>
8. Datasheet dell'RP2040: <https://www.raspberrypi.com/products/rp2040/specifications/>
9. Documentazione SDK per l'RP2040: <https://raspberrypi.github.io/pico-sdk-doxygen/index.html>
10. Codice dell'SDK dell'RP2040: <https://github.com/raspberrypi/pico-sdk>