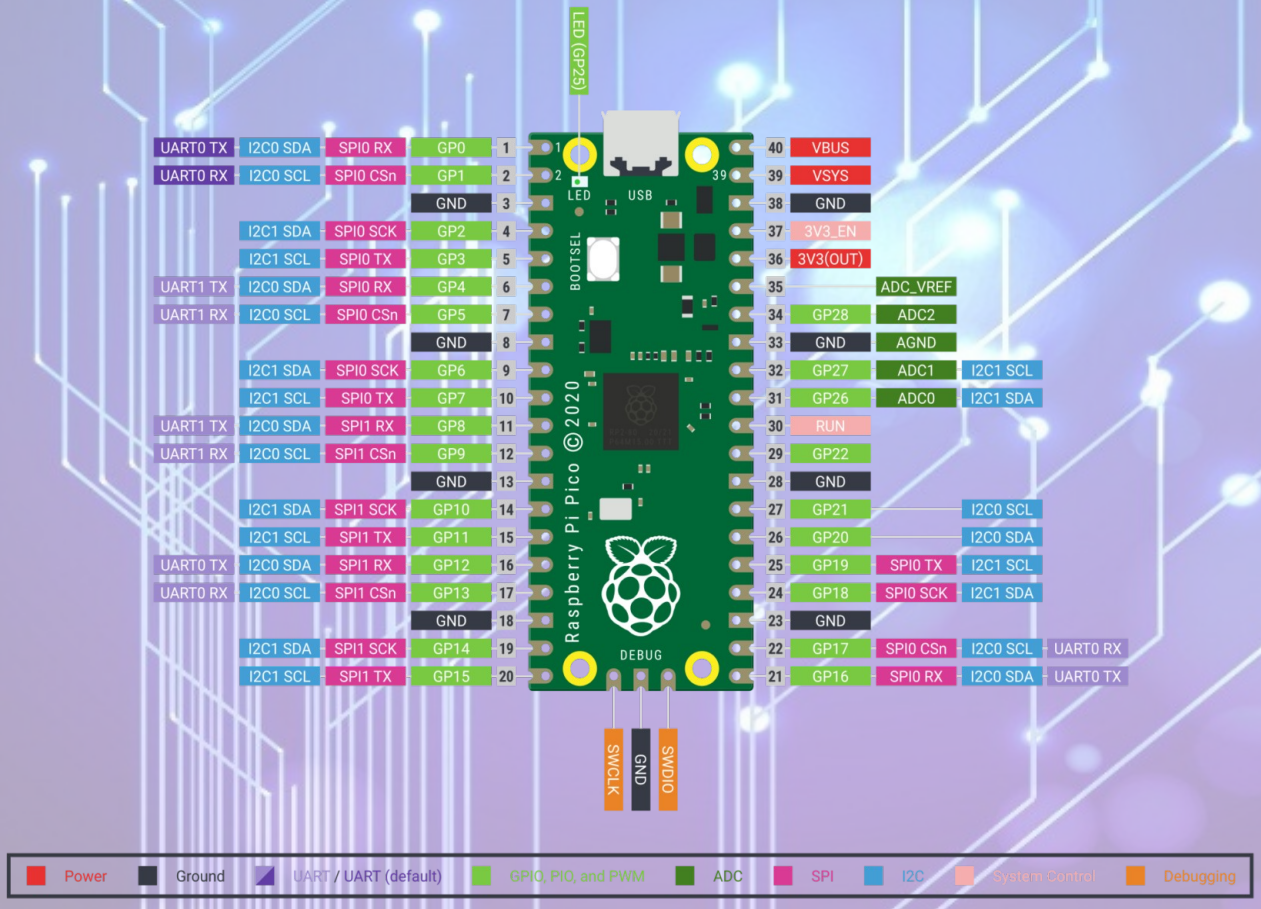
PicoStdLib

Per Nim

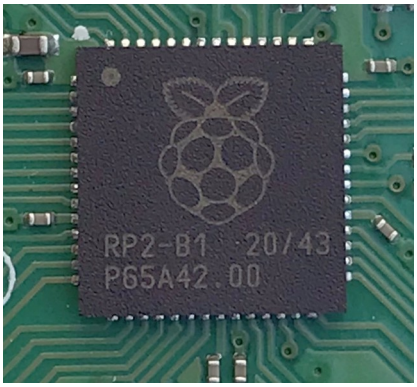


Ver. 1.0.0

Indice generale

1.0 – Presentazione RP2040.

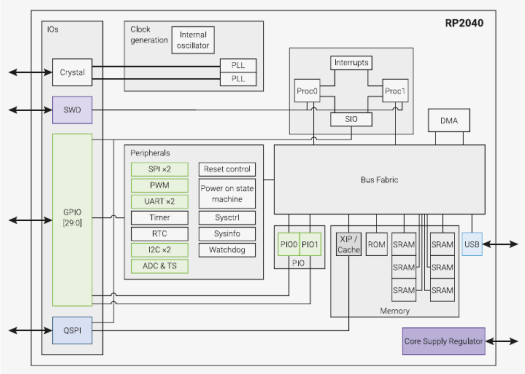
Comincio pcon il presentarti il microcontrollore RP2040 e le sue caratteristiche che lo rendono molto interessante sia per l'hobbista che per il professionista, ed anche perché prima di cominciare a programmarlo vanno conosciute le sue caratteristiche elettroniche, non conoscendole, potresti danneggiare seriamente il dispositivo o non usarlo al meglio. Comincio con il mostrati il protagonista di questa breve guida:



Le caratteristiche principali sono:

- Dual core ARM Cortex-M0+ con velocità di clock di 133Mhz
- 264kB di memoria SRAM on-chip.
- 30 pin GPIO multifunzione.
- ADC (12-bit) a quattro canali con sensore di temperatura interno @ 500KS/sec.
- USB 1.1 integrato con supporto per dispositivi host.
- 16 MB di memoria flash su bus QSPI.
- 16 canali PWM.
- DMA controller.

La struttura interna invece è (schema a blocchi) :



Alcune grandezze elettriche da tenere sempre in considerazione:

- Vout Hihg = 3.63V max
- Imax Tot = 50mA corrente massima erogabile (somma della corrente su tutti i pin).
- Imax per pin = 12mA*
- Rpull-Up = 50-80 KΩ
- Ppll-Down = 50-80 KΩ
- RingADC = 100 KΩ

* Il valore della corrente massima di uscita non è chiaro, dovrebbe essere 12mA ma sembra arrivare fino a 20mA senza problemi. Per rimanere sul sicuro meglio stare sui 12/15mA.!!

2.0 – Installazione.

2.1 – Da Dove Iniziare.

Prima di entrare nella programmazione vera e propria del Rasberry Pico 2040, ce da installare un po' di cose e capire come inizializzare un progetto.

Da installare:

1) Nim che puoi trovare alla pagina: https://nim-lang.org/install.html .
2) Git.
3) L'sdk di Rasberry Pico che trovi in: https://github.com/raspberrypi/pico-sdk .
4) La libreria wrapper per Nim che trovi in https://github.com/EmbeddedNim/picostdlib .
5) Il compilatore C per i microcontrollri di nome " gcc-arm-none".
6) Ed in fine CMake.

Vediamo punto per punto come procedere.
1) L'installazione di Nim è molto facile e sul sito sopra linkato troverai tutte le istruzioni necessarie per poter fare questo passo in maniera veloce ed indolore.
2) Installa Git, o dal repository del tuo sistema operativo, oppure lo scarichi dalla rete e lo installi.
3) A questo punto, creati un directory di lavoro (cosi avrai tutto bello ed in ordine) dove andrai a creare i tuoi programmi (per il Rasberry Pico), e qui clonerai il repositorie del pico-sdk con il comando:

git install https://github.com/raspberrypi/pico-sdk .

4) Ora devi installare la libreria "*picostdlib*" che farà da ponte tra Nim e la libreria prima scaricata in C.

nimble install https://github.com/EmbeddedNim/picostdlib .

5) Senza l'apposito compilatore per dispositivi ARM non si va da nessuna parte! Quindi ti tocca installare quello che almeno sui sistemi Linux viene chiamato **gcc-arm-none** (o altri nomi simili ogni distribuzione lo chiama un po' come vuole) e di solito è un mostro da 1GB!!
6) Sempre dal tuo gestore pacchetti (o recuperalo in rete) **devi** installare CMake.

Se hai installato tutto correttamente puoi creare il tuo primo progetto, e lo puoi fare in due modi. Il primo che però sconsiglio (anche perché dovresti già avere scaricato l'SDK in una directory) è quello di far scaricare ogni volta tutto al inizializzatore, cosa che facevo anche io, finché non mi son trovato con il disco pieno con pochi progetti. Il comando è di certo più semplice:

piconim init -n /*tuo_percorso_progetti_RP2040*/bin/nimbase.h test

La seconda soluzione che ti consiglio vivamente si basa sul avere gia l'SDK sul disco quindi non verrà scaricata ogni volta, e si fa:

piconm init -s /*percorso_sdk_pico*/pico-sdk -n /*tuo_percorso_progetti_RP2040*/bin/nimbase.h test

Adesso puoi entrare nella tua cartella di lavoro e vedrai una nuova cartella di nome "*test*" al suo interno ci sono dei file di configurazione, che normalmente non dovrai mai editare, e due cartelle: "*csource*" ed "*scr*". La prima la vederemo meglio dopo, ma conterrà tra le molte cose il file binario "*uf2*" che installerai sul RP2040 una volta fatta la compilazione, ma ora è più importante concentrasi sulla cartella "*scr*" dove sarà contenuto il file (nel nostro caso) "*test.nim*" e sarà questo file da editare per scrivere i programmi (main).

ATTENZIONE: quando crei il tuo primo progetto, potresti ricevere il seguente errore:

```
Cmake Warning at
/home/utente/Programmazione/PicoNim/pico-sdk/src/rp2_common/tinyusb/
CMakeLists.txt:10 (message):
 TinyUSB submodule has  not been initialized; USB support will be unavailable



 hint: try 'git submodule update --init' from your SDK directory

 (/home/utente/Programmazione/PicoNim/pico-sdk).
```

Niente paura non è obbligatoria, ma comunque è meglio andare nella directory ../pico-sdk e dare il comando **git submodule update –init** (son due trattini prima di init!!) a scanso di possibili, futuri problemi.

Se hai installato anche il "submodule" ora la creazione del progetto dovrebbe avvenire senza ulteriori problemi. Puoi ora compilare il tuo primo progetto con il comando, che dovrà essere dato all'interno della cartella creata per il progetto (nel nostro caso "test"):

piconim build test.nim

Il progetto ora dovrebbe compilare, ma come nel mio caso, mancavano ancora delle dipendenze per riuscire nell'impresa, cose che elenco qui sotto nel caso servissero (anche in attesa di vedere cosa effettivamente serve e cosa no):

1) newlib.
2) Cross-arm-none-newlib.
3) Build-essential (in opensuse **zypper install -t pattern devel_basis**).
4) Gcc-c++ (e dipendenze varie).
5) Nasm (compila anche senza!!).

ATTENZIONE: un'altro problema che potrebbe verificarsi, specie con newlib >= 4.0 è la comparsa dell'errore che dice più o meno:

```
/usr/lib64/gcc/arm-none-eabi/12/ld:
/usr/lib64/gcc/arm-none-eabi/12/../../../../arm-none-eabi/lib/thumb/v6-m/nofp/
libc.a(lib_a-syscalls.o): in function `_exit':
/home/abuild/rpmbuild/BUILD/newlib-4.1.0/build-regular-dir/arm-none-eabi/thumb/
v6-m/nofp/newlib/libc/sys/arm/../../../../../../../../../../newlib/libc/sys/arm/
syscalls.c:493: multiple definition of `_exit';
CMakeFiles/test.dir/home/utente/Programmazione/PicoNim/pico-sdk/src/rp2_common/
pico_runtime/runtime.c.obj:runtime.c:(.text._exit+0x0): first defined here
```

Per fortuna si risolve facilmente andando in **/../pico-sdk/src/rp2_common/pico_runtime/** e all'interno del file "runtime.c" (verso linea 180) troverai una funzione:

void __attribute__((noreturn)) _exit(__unused int status)

che potrebbe anche essere leggermente scritta diversamente a seconda dalla versione dell'sdk che stai usando, ad ogni modo modificala come:

void static inline __attribute__((noreturn)) _exit(__unused int status)

e ora dovrebbe compilare senza problemi!!!

Mi rendo conto che tutto questo può essere un po' macchinoso, ma son cose che prima di iniziare a programmare devono essere risolte e lo saranno una volta e per sempre, quindi ora dovresti essere pronto!!!!

2.2 – Hello World.

Anche per il Rasberry Pico c'è "l' Hello word", che come anticipato viene generato automaticamente da "piconim" quando inizializzi il tuo progetto; è un semplicissimo programma che farà lampeggiare il led che si trova sul dispositivo non preoccuparti se non lo capirai, verrà spiegato bene in seguito per ora lo scopo è verificare che tutto funzioni al meglio e come fare per installare il binario sul RP4020. Questo file sorgente si troverà in:

../test/scr/test.nim

Questo file non dovrà mai essere cancellato o rinominato, perché poi quando andrai a compilare, questo sarà il file principale (main) a cui punta tutta l'infrastruttura e se non lo trova ti darà errore. Non lo puoi cancellare ma lo potrai modificare a tuo piacimento per scrivere il tuo programma. Per ora supponiamo che volgiamo solo far lampeggiare il led, quindi andremo a compilare tutto con il comando:

piconim buid test.nim

Questo comando come già detto va dato dalla posizione: ../test (e non da ../test/src!! questo è molto importante!). Dato il comando ed ora che funziona tutto vedrai correre molte scritte a schermo che dicono cosa sta facendo il compilatore fino a che dirà che tutto è andato bene. A questo punto dovrai andare in:

../test/csource/build

E qui troverai un file di nome *test.uf2* che verrà generato ogni volta e sarà il binario da installare sul Rasberry pico 2040. L'operazione è semplice, basta collegare il dispositivo alla porta usb, tenendo premuto il pulsantino bianco sulla board, a questo punto il sistema lo vedrà come un unità esterna (i nomi dipendono dal sistema operativo in uso) e ti basterà trascinare il tuo file *test.uf2* nel dispositivo e il led comincerà subito a blinkare. Ottimo, hai fatto, compilato ed installato il tuo primo programma sul RP2040!! Ora ce solo da scrivere dei programmi utili, compito che spetta a te. Nel prossimo capitolo vedrai le procedure più comunemente usate ed alcuni moduli (librerie).

3.0 – Programmazione RP2040.

Questo manuale non vuole insegnare le basi della programmazione in Nim, basi che già dovresti avere se affronti un argomento delicato come questo e tanto meno le basi dell'elettronica, ma si concentra sulla libreria *picstdlib* e sulle mie librerie (esterne) per gestire alcuni componenti che potrebbero tornare utili nell'utilizzo del RP2040.

3.1 – Il Primo Programma.

Il primo programma che esaminerai è qualcosa di cui ho già accennato nel **capitolo 1**, ed è quello che si può considerare un "hello world" per in tuo RP2040, infatti farà solo una cosa molto semplice, far lampeggiare il led posto sulla basetta e che fa capo al pin denominato *Gpio25* che è riservato esclusivamente a lui e non compare nemmeno sul "pinout". Portati nella directory del tuo progetto e dentro alla cartella "*scr*" troverai il file principale che avrà lo stesso nome del tuo progetto con estensione "*.nim*".

```
01 import picostdlib/[gpio, time]      → importa gpio e time da picostdlib.

02 DefaultLedPin.init()                → inizializza l'uscita led (Gpio25) a finché possa funzionare correttamente.
03 DefaultLedPin.setDir(Out)           → imponi che questo pin sia usato come uscita.
04 while true:
05  DefaultLedPin.put(High)            → poni ora l'uscita a livello alto (a 3.3V).
06  sleep(250)                         → attendi senza far nulla 250ms e poi prosegui.
07  DefaultLedPin.put(Low)             → poni ora l'uscita a livello basso (0.0V)
08  sleep(250)                         → attendi senza far nulla 250ms e poi prosegui.
```

In questo piccolo programma, ci son già molte cose da osservare, ma che costituiscono la base su cui partire per poi fare cose più complesse. Nella prima riga incontri *import picostdlib/[gpio, time]* che importa due moduli della libreria *picostdlib*. Il primo *gpio* sarà il modulo di cui avrai bisogno per avere accesso a tutte le funzionalità che riguardano le uscite/ingressi digitali del microcontrollore. Il secondo *time* è il modulo per la gestione del tempo. Nella seconda e terza linea incontri *DefaultLedPin* altro non è che una variabile definita all'interno del modulo *gpio* che identifica *Gpio25* ovvero l'uscita dedicata al led di sistema. DefaultLedPin.init() inizializza questo pin al fine di poterlo utilizzare come uscita digitale, le inizializzazioni, come vedrai andranno sempre fatte per ogni singolo pin che si voglia utilizzare. DefaultLedPin.setDir(Out) dice che quel pin sarà usato come uscita (*In* se lo si vuole come ingresso); anche le variabili *In* ed *Out* son definite nel modulo in esame. La quinta riga ovvero DefaultLedPin.put(High) dice semplicemente di mettere a livello alto (*High* = 3.3V) quell'uscita, mentre DefaultLedPin.put(Low) dice di porlo a livello basso (*Low* = 0.0V), anche le variabili *Low* e *High* sono definite nel modulo. In fine rimane sleep(250) che blocca l'esecuzione del programma per il tempo indicato nell'argomento, questo tempo è in millisecondi.

> ATTENZIONE: la procedura sleep() blocca il programma per il tempo indicato, questo vale a dire che il microcontrollore in quel tempo non farà nessun'altra operazione, quindi usala con massima attenzione.

Ma se tu volessi realizzare ad esempio, un progetto con ingressi ed uscite digitali, con pulsanti e led posti direttamente sul microprocessore? Nel prossimo esempio vedrai come si può fare.

3.2 – Ingressi Uscite Digitali.

Questo esempio sarà un po' più complicato del precedente sia dal punto di vista del programma che da quello elettronico, perché sarà richiesta una minima manualità nel costruire circuiti elettronici su breadboard ed una buona comprensione degli schemi elettrici, abilità che comunque saranno fondamentali per lavorare con un microcontrollore, ma niente paura si comincia con qualcosa di "semplice" poi lo espanderò e complicherò man mano che ti verranno spiegati alcuni concetti. Ecco lo schema elettrico a cui faremo capo in questo esempio:

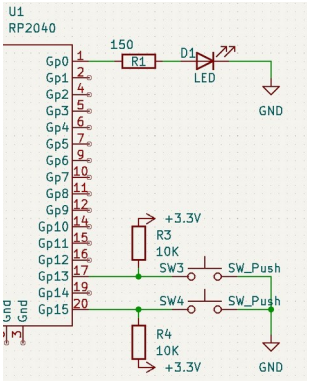


Figura 1

Il programma che andrai a scrivere, si comporterà in maniera molto simile a quello dell'esempio di prima, soltanto che ora alla pressione del pulsante *sw3* il led comincerà a lampeggiare fino a che non verrà premuto il tasto *sw4*. Ecco il programma:

```
01 import picostdlib/[gpio, time]      → importa gpio e time da picostdlib.

02 const led1 = 0.Gpio                 → assegna ad 0.Gpio il nome di led1 (in compilazione).
03 led1.init()                         → inizializza l'uscita led1 (Gpio0) a finchè possa funzionare correttamente.
04 led1.setDir(Out)                    → imponi che questo pin sia usato come uscita.

05 const pulsante1 = 13.Gpio
06 pulsante1.init()
07 pulsante1.setDir(In)                → imponi che questo pin sia usato come ingresso

08 const pulsante2 = 15.Gpio; pulsante2.init(); pulsante2.setDir(In)     → come sopra solo scritto in una sola riga.

09 var flag: bool = false              → crei una variabile di flag di tipo booleiano inizializzata a false.

10 while true:                         → inizia il ciclo infinito.
11   if pulsante1.get() == 0.Value or flag == true:     → se il valore di tensione sul Gpio17 va a zero e falg è = true..
12     led1.put(High)                                   → allora accendi il led posto su Gpio0.
13     sleep(300)                      → aspetta per 300ms.
14     led1.put(Low)                   → ora spegni il led.
15     flag = true                     → setta la variabile flag su true.
16     sleep(700)                      → aspetta per 700ms.
17   if pulsante2.get() == Low:        → se il valore di tensione su Gpio20 va a zero….
18     led1.put(Low)                   → spegni il led (per sicurezza ma non necessario).
19     flag = false                    → setta la variabile flag su fase.
```

Questa volta, andrai a scegliere tu il pin di ingresso/uscita che vuoi utilizzare assegnandogli un nome (che magari sia auto esplicativo) come ad esempio const led1 = 0.Gpio notare l'uso di "*const*" che fa si che venga impostato il tutto in fase di compilazione, per il resto è simile al programma di prima, infatti per prima cosa inizializzerai *led1* (led1.init()) e di seguito gli dirai che lui deve funzionare come uscita (led1.setDir(Out)). Per gli altri assegnamenti è la stessa cosa solo che funzioneranno come ingressi. Nota che *pulsante2* è scritto sulla stessa linea ma i vari comandi son separati da ";" che mi sembra più leggibile. Altra cosa da evidenziare

è come si acquisisce lo stato di un pulsante, ovvero con la funzione *get()* if pulsante2.get() == Low: in questo caso si controlla lo stato di *pulsante2* se è a livello basso. Nel programma ho utilizzato sia *0.Value* che *Low* che sono equivalenti. In questo esempio hai utilizzato due resistenze di pull-up esterne che mantengono il livello di tensione alta sul pin d'ingresso finché il pulsante non viene premuto, che in tal caso lo manda a zero. Questo dover mettere delle resistenze esterne è dovuto al fatto che durante l'inizializzazione non abbiamo specificato di usare il pull-up interno che invece userai nel terzo esempio (ne eventualmente il pull-down che però pare sia impostato di default), ma se ne volgiamo esserne assolutamente sicuri di non usare ne il pull-up ne il pull-down, durante l'inizializzazione è meglio usare la procedura *disablePulls()*.Il funzionamento di questo programma è abbastanza semplice una volta inizializzate le porte di uscita ed ingresso si entra in un ciclo infinito (con *while true:*) il primo *if* controlla la tensione su *Gpio13* (pulsante1) se questa vale zero o se *flag* è posto a *true* entra nel corpo e fa lampeggiare il led; la variabile *flag* è essenziale affinché il led continui a lampeggiare anche dopo il rilascio del pulsante infatti se entra nel corpo viene posto *flag = true* condizione che permette l'ingresso nel corpo (con l'*or*) anche a pulsante rilasciato. Il secondo *if* controlla la tensione su *Gpio15* (pulsante2) che se va a zero, pone l'uscita di *led1* a zero (lo spegne incondizionatamente) e porta la variabile *flag = false*, che impedirà l'ingresso nel blocco di lampeggiamento del led. Nel terzo esempio oltre a quanto prima detto, vedrai anche come semplificare (tramite un template disponibile nella libreria) la dichiarazione dei pin rendendo il lavoro ancora più semplice e veloce.

3.3 – Pull-Up Interno.

Come anticipato, in questo capitolo, vedrai come modificare il programma di prima per poterlo usare senza le resistenze esterne. Va subito detto che queste resistenze interne al microprocessore hanno un valore che può variare dai 50 ai 80KΩ e quindi non saranno sempre idonee, specie quando si useranno bus come l' I2C o altre applicazioni più critiche. Vediamo di seguito come fare:

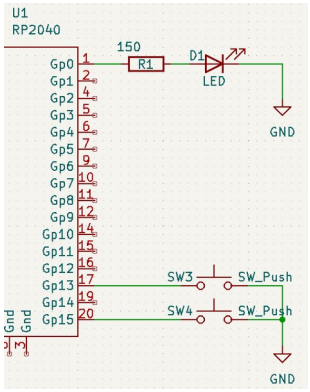


Figura 2

La differenza tra questo schema e quello precedente è solo la mancanza delle resistenze esterne di pull-up, ma il codice sarà apparentemente molto diverso da prima:

```
01 import picostdlib/[gpio, time]

02 setupGpio(led1, 0.Gpio, true)              → impostazione della porta di uscita con setupGpio(), true = uscita.
03 setupGpio(pulsante1, 13.Gpio, false)       → impostazione della porta di uscita con setupGpio(), false = Ingresso.
04 pulsante1.pullUp()                         → dichiarazione di attivazione del pull-up per pulsante1.
05 setupGpio(pulsante2, 15.Gpio, false); pulsante2.pullUp()   → stessa cosa solo scritta su una sola riga.

06 var flag: bool = false

07 while true:
08   if pulsante1.get() == 0.Value or flag == true:
09     led1.put(High)
10     sleep(300)
11     led1.put(Low)
12     flag = true
13     sleep(700)
14   if pulsante2.get() == Low:
15     led1.put(Low)
16     flag = false
```

Il codice ora risulta essere, almeno nel inizializzazione dei pin, assai molto più compatto e semplice, il template *setupGpio()* accetta come primo parametro il nome della variabile in questo caso *pulsante1*, il secondo parametro è la porta Gpio che si intende utilizzare ovvero *0.Gpio*, il terzo è se deve essere posto come uscita (= true) o come ingresso (= false); Pull-Up/Down ancora bisogna dichiararli esplicitamente, ma con l'evolversi della libreria magri si semplificherà pure quello.

3.4 – Pwm.

Cominciamo a complicare un po' le cose, in questo paragrafo vedrai come funziona il PWM (pulse width modulation, in italiano modulazione a larghezza di impulso), ma cosa è questo *pwm*? Come si può dedurre dal nome è una particolare modulazione del segnale, a frequenza fissa, dove però si va a variare il rapporto (in tempo) di quanto il segnale resta alto (3.3V) e per quanto tempo rimane basso (0.0V) rapportato alla frequenza (e quindi al periodo) scelta per il funzionamento. Solitamente si esprime in percentuale e viene definito come:

$$\text{DutyCycle}\% = \frac{\text{Ton}}{\text{Ttot}} * 100$$

Eq 1

Dove *Ton* è il tempo in cui il segnale rimane alto, mentre *Ttot* è il periodo del segnale (1/frequenza). La figura di seguito dovrebbe aiutarti a comprendere meglio il concetto.

Figura 3

Nella figura 3, stai generando un onda quadra ad una frequenza di 500Hz e quindi di periodo di 2ms, ed un duty cycle del 25%, ovvero di circa 500µs (microsecondi) ed applicando **Eq 1**.

$$\text{DutyCycle}\% \rightarrow \frac{500 * 10^{-6}}{2 * 10^{-3}} * 100 = 25\%$$

Eq: 2

Chiarito questo concetto di base, nel prossimo programma vedrai come calcolare ed impostare l'RP 2040 per avere in uscita la stessa forma d'onda vista in **Figura 3**. In più vedrai come questo microcontrollore gestisce il *pwm*, perché questa gestione, è molto flessibile e potente, anche se leggermente più complessa rispetto ad altri sistemi. Ecco il programma:

```
01 import picostdlib/[gpio, pwm]      → importa il modulo pwm.

02 0.Gpio.setFunction(PWM)            → indichi esplicitamente che la porta di uscita 0.Gpio dovrà essere PWM.
03 let pwmOut = toSliceNum(0.Gpio)          → indirizza un generatore pwm al Gpio0 composto da ch A  e ch B.
04 pwmout.setClockDivide(249)         → indica per quanto dividere il clock base che è di 125Mhz (255 volte: uint8).
05 pwmOut.setWrap(1004)               → in base alla frequenza settata, può ulteriormente diminuirla(:uint16).
06 pwmOut.setChanLevel(A, 251)        → setta il canale A al valore di duty cycle corretto.
07 pwmOut.setEnabled(true)            → abilita il canale come uscita e lo rende disponibile.
```

Per poter usare questa modalità, devi importare il modulo *pwm* che fa parte di picostdlib. L'illustrazione di questo programma, coincide con il capire come funziona il sistema che governa questo segnale, poiché non è il classico sistema ma è più complesso, in quanto ci sono più generatori semi-indipendenti, legati però dallo

stesso clock a 125Mhz, questa frequenza è particolare perché è davvero elevata per questa funzione. Qui sotto puoi vedere la struttura di come è concepito questo generatore:

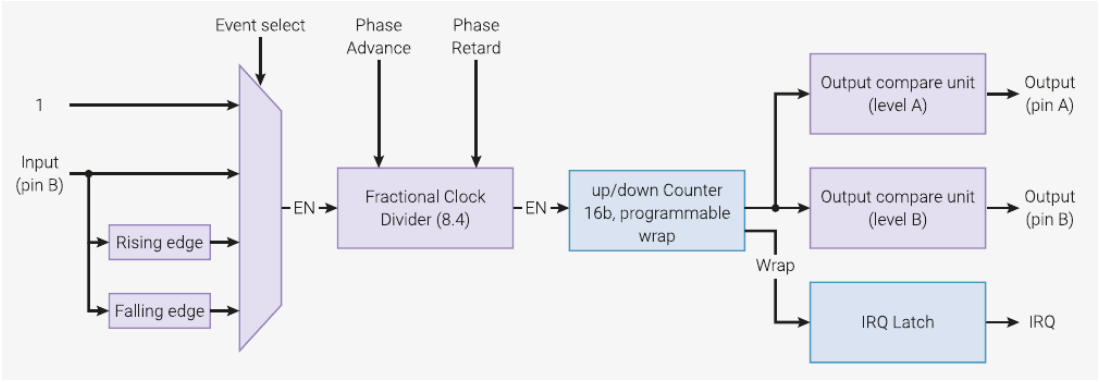


Figura 4

Con questo schema sotto gli occhi, forse sarà più facile seguire la spiegazione del programma che seguirà. Come prima cosa andrai a selezionare un pin di uscita, in realtà potresti selezionarne due, che poi farebbero capo alla stessa linea che fornirebbe due uscite alla stessa frequenza (output *A* ed output *B*) ma, come vedremo, con duty cycle che può essere diverso, di fatto divenendo sincrone. Ho scelto ancora di usare *Gpio0* come uscita esattamente come negli esempi di prima. Nella seconda linea, indichi esplicitamente che l'uscita *Gpio0*, deve essere ti tipo *pwm* e non più di tipo on/off con l'istruzione setFunction(PWM). La terza linea è forse la più bizzarra, ma anche la più importante; infatti l'istruzione toSliceNum() fa in modo di associare l'uscita *0.Gpio* al circuito di generazione del segnale *pwm*, assegnandogli una nuova variabile *pwmOut* in questo caso. Procedendo sull'immagine troverai il *Fractional Clock Divider* questo blocco fa capo a setClockDivide(249) che altro non fa che dividere il segnale di base, quello a 125Mhz, per il numero dato come argomento; ricorda che è un *uint8* quindi il massimo valore è 255 (0xFF), quindi nel nostro esempio avremo che il clock che andremo ad utilizzare sarà di:

$$\text{FrqDivisa} \rightarrow \frac{125 * 10^6}{249} \approx 500\,\text{Khz}$$

Eq 3

La quinta riga, cioè setWrap(1004) fa capo al blocco (sempre di **Figura 4**) *up/down Counter ProgrammableWrap*, questa procedura (da quanto ne ho capito) fa passare in uscita uno stato alto ogni *n* impulsi generati, in questo caso 1 ogni 1004, questo di fatto riduce la frequenza che avrai in uscita, ma andrà anche a modificare il duty cycle effettivo. Il valore massimo accettato è un *uint16* di valore massimo 65535 (0xFFFF). Va evidenziato il fatto, che più alto riesci a mantenere questo numero (l'argomento della funzione) più poi potrai ottenere dei valori di duty cycle precisi. Ora sapendo che la frequenza base è stata portata a 500Khz e facendo passare 1 impulso ogni 1004 finalmente avrai in uscita il valore corretto:

$$\text{FrqUscita} \rightarrow \frac{500 * 10^3}{1004} \approx 500\,\text{Hz}$$

Eq 4

Nella sesta riga trovi la procedura setChanLevel(A, 251) che imposta il canale "A" al valore 251, che equivale ad un duty-cycle di 25% (è il 25% di 1004 che imparerai dopo come calcolarlo). Se tu stessi utilizzando anche l'uscita "B", avrebbe al stessa frequenza e fase del canale "A", ma potresti facilmente variare il duty-cycle in modo indipendente. La seguente formula ti aiuta a capire meglio:

$$\text{DutyCytcle}\,\% \rightarrow \frac{251}{1004} * 100 = 25\,\%$$

Eq 5

La settima linea ovvero setEnabled(true) altro non fa che abilitare tutto quanto abbiamo fatto prima e portarlo effettivamente sull'uscita *Gpio0*. Fino ad ora hai usato dei valori che avevo precedentemente calcolato, ma ora vedrai come calcolare tu stesso questi numeri, che poi potrai sostituirli nel programma precedente per vederne gli effetti. Questa volta si vuole un uscita a 247Khz ed un duty cycle del 58%; ecco un esempio:

Per prima cosa bisogna pensare ad una frequenza di base ma che sia >= a 490Khz (massima frequenza scalabile con 255 divisioni) ma tenendo sempre ben a mente che poi più alto sarà setWrap() più potremmo avere una selezione del duty cycle fine; quindi una frequenza divisa a 49.4Mhz potrebbe andare bene (ma questa la decidi tu e sarà molto importante). Da ***Eq 3*** ricavi il numero da mettere come divisore della frequenza base:

$$\text{setClockDivide}() \rightarrow \frac{\text{FrequenzaMaxSistema}}{\text{FrequenzaScalataVoluta}} \rightarrow \frac{125 * 10^6}{49.4 * 10^6} \approx 2$$

Eq: 6

Dall' **Eq 4** ora puoi ricavare il valore da dare a setWrap:

$$\text{setWrap}() \rightarrow \frac{125 * 10^6}{\text{setClockDivide}() * \text{FreqUscita}} - 1 \rightarrow \frac{125 * 10^6}{2 * 247 * 10^3} - 1 \approx 253$$

Eq: 7

Dall'**Eq 5** ora è semplice ricavare il valore da dare a setChanLevel():

$$\text{setChanLevel}() \rightarrow \frac{\text{DutyCycle}\,\% * \text{setWrap}()}{100} \rightarrow \frac{58 * 253}{100} \approx 146$$

Eq: 8

Quindi i valori da inserire sono: **setClockDivide(2)**; **setWrap(253)**; **setChanLevel(146)**. Ora puoi divertirti a cambiare questi valori adattandoli alla tue esigenze.
I valori restituiti dalle equazioni sopra, difficilmente saranno numeri interi, la cosa migliore dai test sembra prendere il numero ritornato, spogliato dai decimali ma senza arrotondamenti di nessun genere ad esempio 101.34 → 100, ma anche se fosse 101.98 → 101.

Dopo tanta teoria, ora vedrai un piccolo esempio pratico del *pwm* che andrà anche in parte a completare quanto visto e descritto sopra (ci sarebbero ancora molte procedure da esaminare ma si andrebbe oltre l'intento di questo manuale). Lo schema di riferimento del programma sarà il seguente:

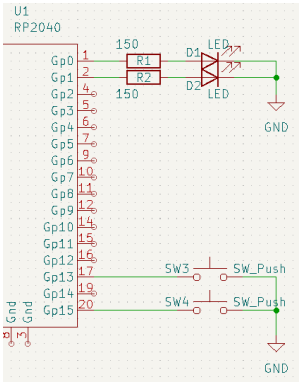


Figura 5

In sostanza aggiungerai un secondo led che collegherai alla porta *Gpio1* che sarà quel secondo canale (canale *B*) che avrà la stessa frequenza del canale *A* che potrà avere un duty cycle diverso ma che sostanzialmente rimarranno sincronizzati. Ecco un immagine che ti aiuterà a capire meglio:

Figura 6

Il prossimo programma, farà si che se premi *pulsante1*, la luminosità del led su *Gpio1* diminuirà mentre aumenterà quella del led posto su G*pio0*. Schiacciando *pulsante2* le cose si invertiranno, questo è dovuto al fatto che i duty cycle saranno complementari ovvero cala fino al 0.47% un e sale al 99.50% l'altro incontrandosi al 50% circa. La frequenza per semplicità è stata mantenuta la stessa della dimostrazione in cornice e di conseguenza tutto il set-up. Ecco il programma:

```
01 import picostdlib/[gpio, time, pwm]

02 setupGpio(pulsante1, 13.Gpio, false);pulsante1.pullUp()
03 setupGpio(pulsante2, 15.Gpio, false); pulsante2.pullUp()

04 0.Gpio.setFunction(PWM)              → imposta 0.Gpio come PWM.
05 1.Gpio.setFunction(PWM)              → imposta 1.Gpio come PWM.

06 let pwmOut = toSliceNum(0.Gpio)
07 pwmout.setClockDivide(2)
08 pwmOut.setWrap(253)
09 pwmOut.setEnabled(true)

10 var
11   valPwmA: uint16 = 1
12   valPwmB: uint16 = 25

13 while true:
14   if pulsante1.get() == 0.Value:
15     valPwmA += 1                     → incrementa valPwmA di uno ad ogni ciclo se il pulsante1 è premuto.
16     if valPwmA > 253:
17       valPwmA = 253
18   valPwmB -= 1                       → decrementa valPwmB di uno ad ogni ciclo se il pulsante1 è premuto.
19   if valPwmB <= 1:
20     valPwmB = 1
21  if pulsante2.get() == Low:
22   valPwmB += 1                       → incrementa valPwmB di uno ad ogni ciclo se il pulsante2 è premuto.
23   if valPwmB > 253:
24     valPwmB = 253
25   valPwmA -= 1                       → decrementa valPwmA di uno ad ogni ciclo se il pulsante2 è premuto.
26   if valPwmA <= 1:
27     valPwmA = 1
28  pwmOut.setChanLevel(A, valPwmA)     → setta il valore del PWM sul canale A  dopo la lettura dei pulsanti.
29  pwmOut.setChanLevel(B, valPwmB)     → setta il valore del PWM sul canale B dopo la lettura dei pulsanti.
30  sleep(20)
```

Su questo programma non mi soffermo molto, perché già dovresti riconoscere molte cose. Tutti gli *if* usati servono solo o per entrare nella specifica funzione del pulsante premuto, oppure per controllare che i valori non escano dai valori massimi e minimi consentiti (1 e 253). La cosa essenziale, è notare l'uso dei due canali con l'ausilio di setChanLevel(A, valPwmA) e setChanLevel(B, valPwmB). La libreria *pwm* di picostlib ha ancora molte sorprese da offrirti, ma per un uso di base e vista la complessità del argomento direi che per questa sezione può bastare.

3.5 – Ingressi Analogici.

Come tutti i microcontrollori, anche l'RP2040 ha degli ingressi analogici, in questo caso ce ne sono tre più uno per il sensore di temperatura interna. Questi dispositivi non possono maneggiare direttamente questi valori, per questo motivo ci sarà un ADC che trasforma un segnale analogico in uno digitale. Gli ADC non sono tutti uguali, ed un parametro importante è quanti bit usano per la conversione, questo ne ha 12, quindi può dare in uscita:

$$\text{Valori} \rightarrow 2^{12} - 1 = 4095$$

Eq 9

In altre parole vuol dire che l'intervallo di tensione tra 0.0V e 3.3V(valore massimo che si può dare) può essere suddiviso in 4095 "fette"; questo a sua volta significa che ogni fettina varrà circa $800\,\mu V$/step (microvolt):

$$\text{VoltStep} \rightarrow \frac{3.3\,V}{4095} \approx 805\,\mu V$$

Eq 10

Lo schema da seguire per questo progetto è il seguente:

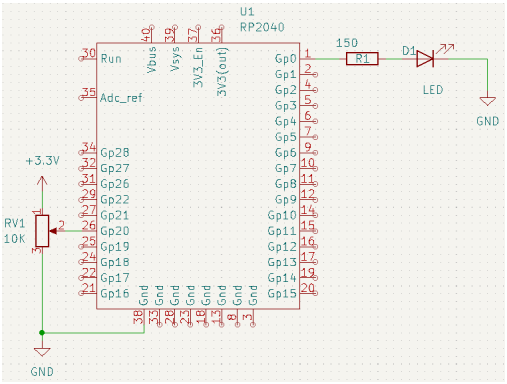


Figura 7

Vedrai che sarà molto simile a quanto visto nella **sezione 2.4**, ma questa volta invece di variare la luminosità del led premendo un pulsante (ovvero utilizzando un ingresso digitale), essa varierà proporzionalmente alla tensione in ingresso su *Gpio26* che sarà data dal cursore del potenziometro da 10KΩ, variando al tensione da circa 0.0V a 3.3V. Per questo programma devi avere a disposizione un emulatore di terminale (cutecom, putty, ecc...) per poter visualizzare i messaggi provenienti dal microcontrollore che ti potrà tornare utile in futuro sia, come in questo caso, per visualizzare i messaggi dell'RP2040 sia per la comunicazione (da/e per il micro) e sia per il debug:

Il microcontrollore, non ti darà il valore reale della tensione presente sul ingresso selezionato, ma un numero che può variare da 0 a 4095 che equivale alla tensione presente; sarai tu a manipolare questo valore per visualizzarlo correttamente nel formato da te desiderato.

Per praticità d'uso qui riporto i nomi che userai per gli ingressi analogici e a quale pin corrispondono:

- Adc26 → Gpio26 (ADC0 sul pinout del manuale).
- Adc27 → Gpio27 (ADC1 sul pinout del manuale).
- Adc28 → Gpio28 (ADC2 sul pinout del manuale).
- AdcTemp → ingresso riservato.

```
01 import picostdlib/[gpio, time, pwm, stdio, adc]

02 stdioInitAll()                          → inizializza i tipi standard “stdio” ed abilita il supporto UART, USB.
03 adcInit()                               → inizializza l’hardware per il supporto a l ADC.
04 0.Gpio.setFunction(PWM)
05 let pwmOut = toSliceNum(0.Gpio)
06 pwmout.setClockDivide(251)
07 pwmOut.setWrap(2002)
08 pwmOut.setEnabled(true)

09 selectInput(Adc26)                      → seleziona come ingresso analogico Adc26 (Gpio 26).
10 while true:
11   let valRes = adcRead()                → effettua la lettura del valore su Gpio26.
12   let valPwm: uint16 = uint16(float(valRes)*0.488)
13   print("Il Valore Numerico e': " & $(valRes))     → stapa su seriale il valore (0..4095) presente sull’ingresso.
14   print("Il Valore Resitivo e': " & $(uint16(float(valRes)*2.44)) & "Ohm")
15   pwmOut.setChanLevel(A, valPwm)
16   sleep(100)
```

Nella prima riga come di consueto cominciamo importando i moduli che ci servono, avari notato che questa volta ne abbiamo importati due nuovi, ovvero *adc* e *stdio*. Il primo serve per avere le procedure per la gestione del ADC che vedremo di seguito, mentre il secondo meno ovvio, importa i *tipi stdio* e la comunicazione via seriale, utile in questo contesto per visualizzare a terminale le varie informazioni che ci fornisce il microcontrollore in fase di esecuzione. La seconda linea stdioInitAll() di fatto abilita tutti i tipi importati e la comunicazione della seriale. Nella terza linea troverei adcInit() che andrà ad inizializzare tutto l’hardware che occorre affinché l’ADC sia funzionante. Le linee dalla quarta alla ottava son vecchie conoscenze su cui non mi soffermo e ti rimando alla **sezione 3.4**. Nella nona linea andrai a selezionare quale ingresso vorrai leggere con la procedura selectInput(Adc26), in questo caso *Gpio26*. Nella undicesima riga, andrai ad effettuare le lettura vera e propria dell’ingresso usando adcRead() che come già detto fornirà un valore compreso tra 0 e 4095, che in seguito potrà essere manipolato per esempio per ottenere il valore resistivo o di tensione reale (dipende qui dalla tua applicazione). Visto che per la prima volta abbiamo usato la procedura *print()* credo valga la pena spendere due parole sul suo uso. Come avrai già capito questa procedura serve per comunicare con il tuo computer, inviando dei dati sulla porta usb, questi dati sono delle semplici stringhe ed anche eventuali numeri sono in realtà stringhe e come tali devono essere trattati. All’interno di *print()* puoi inserire i tuoi messaggi semplicemente inserendoli tra le virgolette in questo modo *“Prova ”* ma ci sono altri due operatori molto importanti: il primo è *&* che ti permette di concatenare due stringhe in questo modo: *“Prova ” & “di Comunicazione”* otterrai *“Prova di Comunicazione”* (ricordati di inserire gli spazi). Il secondo è *$* che converte ad esempio un numero in una stringa: *num: int = 345; print(“Ora stampo: “ & $num)* otterrai *“Ora Stampo: 345”* ma ci sarà modo per approfondire. Fino ad ora ti ho detto che ci son tre ingressi analogici, in realtà c’è ne un quarto, che è riservato esclusivamente al sensore di temperatura del microcontrollore, in più magari, vorresti leggere più ingressi analogici, ma come puoi vedere in **Figura 8** il ADC è unico:

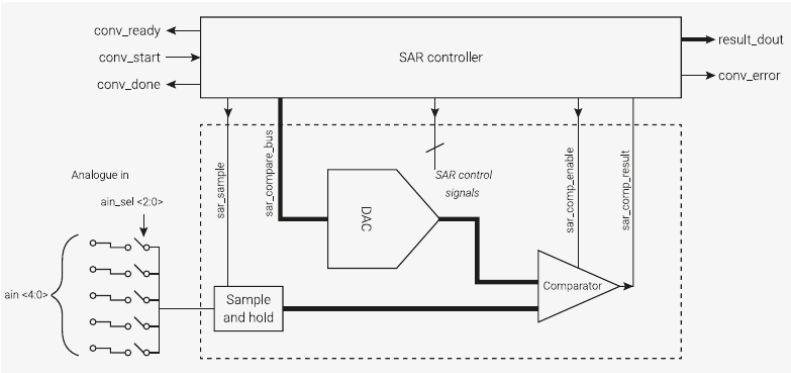


Figura 8

Nel prossimo programma vedrai come leggere la temperatura dal sensore montato sul RP2040 e contemporaneamente leggere il segnale analogico proveniente dal potenziometro (su due canali diversi) inoltre vedrai delle procedure che potrebbero esserti utili nel futuro. Il circuito di riferimento rimarrà quello di **Figura 7**.

```
01 import picostdlib/[gpio, time, pwm, stdio, adc]
02 import std/[math, strformat]                  → importa math e strformat dalla libreria standard di Nim.

03 stdioInitAll()                                → inizializza i tipi standard "stdio" ed abilita il supporto UART, USB.
04 adcInit()                                     → inizializza l'hardware per il supporto a l ADC.

05 0.Gpio.setFunction(PWM)
06 let pwmOut = toSliceNum(0.Gpio)
07 pwmout.setClockDivide(251)
08 pwmOut.setWrap(2002)
09 pwmOut.setEnabled(true)

10 enableTempSensor(true)                        → abilita il sensore di temperatura.

11 let voltPerStep = 3.3 / 4095                  → calcola il valore di tensione per step (Eq 10).
12 while true:
13   selectInput(AdcTemp)                        → ora selezione il canale AdcTemp come ingresso al ADC (da leggere).
14   let convLettVolt = float(adcRead()) * voltPerStep → trasforma il valore letto (numerico) in valore tensione effettivo.
15   let gradi = round(27 - ((convLetVolt - 0.706) / 0.001721), 2)   → converte il valore di tensione in °C.
16   print("----------------------")
17   print(fmt "Temp:  {gradi: 0.2f}")           → stampa i gradi usando le f-string (format string).
18   selectInput(Adc26)                          → ora seleziona il canale Adc26 come ingresso al ADC (da leggere).
19   let valRes = adcRead()
20   let valPwm: uint16 = uint16(float(valRes)*0.488)  → calcola il valore pwm da inserire nella funzione.
21   print(fmt"Il Valore Resitivo e':  {(uint16(float(valRes)*2.44))}  Ohm") → stampa il valore del potenziometro in Ohm.
22   print("----------------------")
23   pwmOut.setChanLevel(A, valPwm)
24   sleep(300)
```

Le novità introdotte in queste programma non son molte, ma comincia a farti intravedere come gestire i segnali provenienti da un sensore (nel nostro caso un potenziometro, ma cambia poco) e come manipolare questi valori per fornire una misura che abbia un senso pratico. Cominciamo dalla seconda linea, dove andrai ad importare i moduli *math* e *strformat*, di cui si useranno le procedure *round()* e la macro *fmt*, che vedrai in seguito. La procedura enableTempSensor(true) della decima linea, abilita il sensore di temperatura per poter funzionare correttamente. La undicesima linea assegna alla variabile *voltPerStep* il valore di 805µV in accoro alla ***Eq 10*** e servirà per la conversione volt → temperatura. Arrivi ora alla tredicesima riga, che assolve allo scopo principale di questo programma, ovvero selezionare l'ingresso da mandare al ADC per essere acquisito, e verrà fatto dalla procedura selectInput(AdcTemp) che dichiara esplicitamente quale ingresso sarà mandato all'ADC per essere letto; analogamente alla linea diciotto, verrà esplicitamente dichiarato che da li in poi dovrà essere acquisito il canale *Adc26*. Potrebbe essere interessante soffermarsi maggiormente sulle linee quattordici e quindici, perché qui viene fatta la conversione di un numero (0..4095) in una temperatura (in gradi Celsius). Il sensore fornirà la temperatura che legge sotto forme di una tensione proporzionale alla temperatura stessa; è noto dal manuale che:

$$\text{Tensione @ } 27\,^{\circ}\text{C} = 706\,\text{mV}$$

In più si sà che ad ogni incremento di 1°C la tensione diminuisce di 1,721mV, ovvero:

$$+1\,^{\circ}\text{C} = -1{,}721\,\text{mV}$$

Noti questi fattori, e nota la relazione di ***Eq 10*** (voltStep), è abbastanza facile ricavare la seguente relazione, che lega il valore letto dal ADC alla temperatura:

$$C° = 27 - \left(\frac{voltStep - 0.706}{0.001721}\right)$$

Eq 11

Che potrebbe essere semplificata come (se non richiesta precisione assoluta, altrimenti 581 → 581.06):

$$C° = 27 - (581 * (voltStep - 0.706))$$

Eq 12

Il valore ricavato ora, può essere stampato a video (o su un display) facendo visualizzare un dato concreto, questo dovrai farlo per ogni tipo di sensore applicando poi le conversioni del caso se necessarie. Questa volta per stampare i risultati si è utilizzato dentro a *print()* l'operatore *fmt* che è molto comodo e ci risparmia il massiccio utilizzo di *&* e *$* perché lo fai lui per noi, in più scrivendo {gradi: 0.2f} ti permette di controllare facilmente quante cifre mostrare (non modifica la variabile ma solo la sua visualizzazione!!), in questo caso due cifre decimali, ma ci son molte più opzioni disponibili. Altra cosa che voglio portare alla tua attenzione è la procedura round(27 - ((convLettraVolt - 0.706) / 0.001721), 2) che come puoi intuire, fa un arrotondamento corretto a due cifre come indicato dal suo secondo argomento. C'è un'altro pin su cui vale la pena soffermarmi, ovvero ***adc_ref***. Normalmente l'ADC per la sua alimentazione usa i 3.3V della stessa alimentazione del Rp2040, ma se vuoi prestazioni migliori del dispositivo, allora devi fornire esternamente su questo pin, una tensione più stabile e pulita; per applicazioni comuni e non critiche questa tensione esterna non è necessaria!

3.6 – Timer.

Come in tutti i programmi in generale, la gestione del tempo è di vitale importanza e forse sui microcontrollori lo è anche di più, perché non hai alle spalle un sistema operativo che provvede a fare certe operazioni tempo-dipendenti. In questo capitolo quindi vedrai alcune procedure messe a disposizione dal PR2040 per assolvere a questo compito. Il prossimo esempio, ricorderà qualcosa già visto nelle precedenti sessioni, e farà capo al già noto schema elettrico:

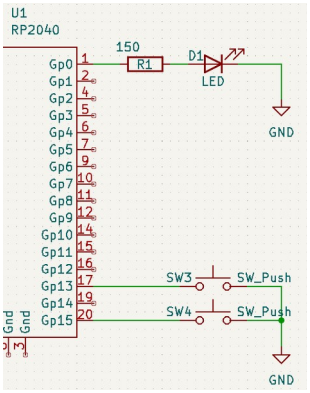


Figura 9

La differenza sostanziale tra quello che hai fatto nella **sezione 3.3** e quello che farai ora, è che per far accendere il led, questa volta dovrai premere il *pulsante1* per almeno **tre** secondi, mentre per far farlo spegnere dovrai tenere premuto *pulsante 2* per almeno **sei** secondi.

```
01 import picostdlib/[gpio, time]

02 setupGpio(led1, 0.Gpio, true)
03 setupGpio(pulsante1, 13.Gpio, false); pulsante1.pullUp()
04 setupGpio(pulsante2, 15.Gpio, false); pulsante2.pullUp()

05 var                                  → le seguente variabili son tutte di tipo “var”.
06  pulsanteRilasciato: bool = true     → determina se il pulsante è stato rilasciato.
07  tempoAttuale: uint32 = 0
08  tempoOn: uint32 = 3000000           → tempo necessario per accendere il led 3s.
09  tempoOff: uint64 = 6000000          → tempo necessario per spegnere il led 6s.

11 sleepMicroseconds(3000000)           → questa volta il tempo sleep è in microsecondi, anziché in millisecondi.

12 while true:
13   if pulsante1.get() == Low:         → se il pulsante1 è premuto entra qui.
14     if pulsanteRilasciato == true:   → se il pulsante rimane premuto non entrare qui, solo se rilasciato.
15       tempoAttuale = timeUs32()      → aggiorna la variabile con il tempo corrente.
16       pulsanteRilasciato = false     → il pulsante non è stato rilasciato quindi non aggiornare più il tempo.
17     if timeUs32()-tempoAttuale > tempoOn:    → se la differenza tra il tempo ora e quello catturato prima è > a 3s.
18       led1.put(High)                 → accende il led.
19       pulsanteRilasciato = true      → il pulsante è stato rilasciato, puoi rientrare a prendere di nuovo tempo
20  if pulsante2.get() == Low:          → se il pulsante 2 è premuto entra qui.
21    if pulsanteRilasciato == true:
22      tempoAttuale = timeUs32()
23      pulsanteRilasciato = false
24    if timeUs64()-tempoAttuale > tempoOff:
25      led1.put(Low)                  → spegne il led.
26      pulsanteRilasciato = true
27  if pulsante1.get() == High and pulsante2.get() == High:    → nessun pulsante premuto entra qui.
    tempoAttuale = timeUs32()                                  → aggiorna il tempo.
```

Questa volta l’argomento è stato decisamente più leggero, ho solo voluto portare alla tua attenzione la procedura sleepMicroseconds(3000000) che è uguale a *sleep()* solo che anziché accettare il tempo in

millisecondi, lo accetta in microsecondi rendendola molto più precisa per applicazioni più critiche, ed alle procedure timeUs32() e timeUs64() che ritornano rispettivamente un numero *uint32* e *uint64* che rappresenta il tempo passato dall'avvio del microcontrollore. Da notare nella penultima riga, l'uso dell'operatore *and* assieme agli *if* che fa si che quella condizione sia vera solamente se entrambi i pulsanti non son schiacciati (ingressi a livello logico alto).

3.7 – Irq.

Un microcontrollore in genere esegue delle operazioni in una sequenza predefinita, e con delle tempistiche ben stabilite; normalmente questo è ciò che si desidera in un programma ma a volte ci possono essere eventi interni o esterni imprevedibili (come tempistica) che devono essere subito presi in considerazione e non possono attendere la conclusione di una routine o altro perché magari lo stesso evento nel frattempo potrebbe essersi già concluso di fatto "perdendolo". Per questi eventi particolari, sono previsti degli ingressi *irq* (*Interrupt ReQuest*) che faranno sospende al microcontrollore qualsiasi cosa stai facendo in quel momento per fargliene fare un'altra ben più importante. Anche in questo caso utilizzerai il solito schema, dove questa volta i pulsanti simuleranno un evento esterno imprevisto:

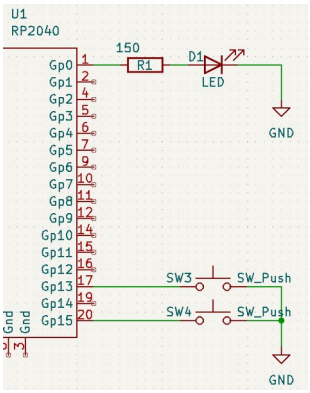


Figura 10

Nel programma che seguirà, ho fatto di tutto per renderlo il meno performante possibile, l'implementazione sarà al quanto discutibile, ma questo è voluto per farti vedere meglio, quando andrai ad eseguirlo, come funziona l'interrup. Ecco il programma:

```
01 import picostdlib/[gpio, time, stdio]
02 import std/[math]

03 setupGpio(led1, 0.Gpio, true)
04 setupGpio(pulsante1, 13.Gpio, false); pulsante1.pullUp()
05 stdioInitAll()

06 proc accendiLed(pin: Gpio, events: set[IrqLevel])  {.cdecl.} =          → procedura chiamata all'evento di interuupt.
07  led1.put(High)
08  print("Led Acceso")

09 enableIrqWithCallback(17.Gpio, {IrqLevel.rise}, true, accendiLed)    → abilitazione del interrupt su Gpio17 mod. rise.

10 while true:
11   print("inizio..")
12   led1.put(Low)                          → pone il led a livello basso ad ogni fine del ciclo for e dopo la chiamata.
13   var lista:seq[int]                     → crea una sequenza vuota di interi.
14   var somma:int
15   for numero in 0..50:
16     lista.add(numero^2)                  → ^ è l'elevamento a potenza (numero²).
17     print("Valore Istantaneo: " & $(numero^2))
18     sleep(250)                           → introduce una pausa di 250ms per farti vedere meglio le cose.
19  somma = lista.sum()                     → somma tutti gli elementi presenti nella sequenza.
20  print("Valore Finale: " & $somma)
```

Le linee da analizzare in questo programma sono in realtà due, la sei e la nove. La sesta linea è la procedura di *callback*, ovvero quella procedura che viene richiamata quanto accade un evento (in questo caso un *irq*), ma non è una normale procedura e va analizzata `proc accendiLed(pin: Gpio, events: set[IrqLevel])  {.cdecl.}`. Puoi darle il nome che vuoi, ma la suo interno deve essere dichiarato il pin `pin: Gpio` il livello di *irq* `events: set[IrqLevel]` questi argomenti vanno scritti sempre in questo modo. Questa procedura deve essere molto

piccola e il più performante possibile, tanto è vero che non ti sarà consentito usare procedure come *sleep()* per il motivo sopra citato; qui non avrebbe senso. La nona linea enableIrqWithCallback(17.Gpio, {IrqLevel.rise}, true, accendiLed) è a tutti gli effetti la dichiarazione di *irq*, e come avrai notato è dichiarata fuori dal loop infinito. In questa procedura va dichiarato esplicitamente l'ingresso che sarà predisposto per questa chiamata, nel nostro caso *Gpio13*, ma sopratutto va dichiarato quando deve "scattare" l'evento in *irqLevel*, e ci son quattro opzioni:

1. low → scatta quando il segnale è basso (0.0V).
2. high → scatta quando il segnale è alto (3.3V).
3. fall → scatta nella transizione del segnale da alto a basso (3.3 → 0.0V).
4. raise → scatta nella transizione del segnale da basso ad alto (0.0V → 3.3V).

Il prossimo parametro che incontri è *true*, che va ad abilitare o meno (*false*) questo sistema di chiamata. In fine trovi il nome dalla procedura da chiamare quando si scatena l'evento. A quanto pare, si può usare questa procedura su un solo ingresso, infatti se abiliti il secondo pulsante, e scrivi una nuova procedura di callback legata ad esso, non funzionerà più nulla, vero che non ho fatto ulteriori prove, ma per ora pare sia così.

3.8 – Watchdog.

Il nome watchdog (cane da guardia in italiano) è davvero molto azzeccato, perché controlla che il normale svolgimento del programma avvenga nei tempi previsti, se ciò non fosse molto probabilmente saresti di fronte ad un bug sul programma o peggio ad un guasto interno (per esempio una memoria guasta). Il watchdog, usa un contattore ed un clock totalmente indipendente dal resto del processore, garantendo il suo funzionamento anche se l'unità centrale avesse dei problemi. Il suo funzionamento è molto semplice: appena abilitato, incomincia il conteggio fino ad arrivare al tempo massimo impostato dal programmatore se non vien resettato prima (durante il runtime) genera un segnale di time-out, che va a resettare il microcontrollore, facendo ripartire il programma da capo, è ovvio che risolve il problema solo se ce stato un malfunzionamento transitorio, nel caso di guasto reale sarà solo un riavviarsi continuo del dispositivo, che però ti segnala il problema. Il primo esempio che ti propongo fa capo ad uno schema molto semplice:

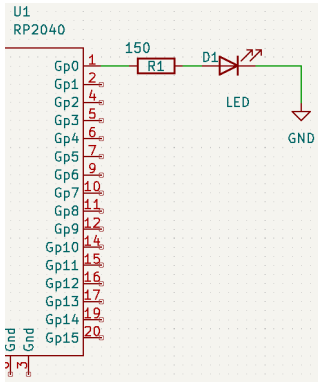


Figura 11

Il codice che andrai a scrivere e compilare, simulerà un malfunzionamento (o un bug) che farà si che l'RP2040 si resetti continuamente, cosa che normalmente non è desiderata, ma spero ti faccia capire il funzionamento di questo sistema e quindi come impostarlo correttamente:

```
01 import picostdlib/[gpio, time, stdio, watchdog]

02 setupGpio(led1, 0.Gpio, true)
03 for _ in 0..4:                    → crea un ciclo di 5 rapidissimi lampeggiamenti iniziale (durata: 1s).
04   led1.put(High)
05   sleep(5)
06   led1.put(Low)
07   sleep(195)

08 watchdogEnable(3000, true)        → abilita il watchdog e lo impasta a 3 secondi (poi resetta).
09 while true:
10   for _ in 0..4:                  → crea un ciclo di 5 lampeggiamenti lenti (durata: 5 secondi).
11     led1.put(High)
12     sleep(50)
13     led1.put(Low)
14     sleep(950)
15     watchdogUpdate()              → azzera il conteggio del watchdog ed evita il reset del micronctrollore.
```

Nella prima linea, tra le varie cose, importi il modulo *watchdog* che fornisce le procedure per utilizzarlo. Alla terza linea, crei un ciclo dove farai lampeggiare molto velocemente il led per cinque volte (per una durata di 1 secondo) questo è stato fatto per rendere subito riconoscibile la partenza e/o la ripartenza del microcontrollore. All'ottava riga con la procedura watchdogEnable(3000, true) andrai ad abilitare il watchdog, e gli imponi il reset del sistema dopo tre secondi (3s) il secondo parametro posto a *true* dice che in caso di funzionamento in debug il watchdog viene sospeso, se entro questo tempo prestabilito non riceve l'azzeramento, può voler dire che il programma si è piantato, e quindi lo fa ripartire nella speranza che il problema cosi si risolva. Alla decima riga (dentro il blocco *while*) crei un secondo ciclo di lampeggiamento del led, ma sta volta molto più lento rispetto a quello di prima, se fai il calcolo questo ciclo dura

(50ms+950ms)*5 = 5s che è un tempo molto più lungo di quello che hai fissato per il reset del RP2040 (ricordo che lo si è fatto apposta!!) ed infatti il sistema ripartirà prima di poter eseguire la quindicesima riga; questa linea se eseguita watchdogUpdate() riporta a zero il conteggio ed il programma continuerebbe il ciclo lento in eterno. Prova a cambiare il tempo di time-out ad esempio watchdogEnable(6000, true) e vedi cosa succede. Piccola nota, il tempo massimo di time-out è di 8.3s. Nel prossimo esempio, che fa capo sempre alla **Figura 11**, voglio mostrarti un sistema che può "prendere una decisione" e cerca di risolvere il problema in caso dell'attivazione del watchdog, utile applicazioni più critiche:

```
01 import picostdlib/[gpio, time, stdio, watchdog]

02 setupGpio(led1, 0.Gpio, true)
03 stdioInitAll()
04 sleep(3000)

05 print("Partenza..")
06 for _ in 0..4:                          → crea un ciclo di 5 rapidissimi lampeggiamenti iniziale (durata: 1s).
07   led1.put(High)
08   sleep(5)
09   led1.put(Low)
10   sleep(195)

11 if watchdogCausedReboot() == true:      → se si solleva il watchdog esegue quanto scritto in questo blocco.
12   print("Entro nel Ciclo da 3s causa reset WD")
13   while true:
14     for _ in 0..2:                      → crea un ciclo di 3 lampeggiamenti lenti (durata: 3 secondi).
15       led1.put(High)
16       sleep(920)
17       led1.put(Low)
18       sleep(80)
19    watchdogUpdate()                     → azzera il conteggio del watchdog ed evita il reset del microcontrollore.

20 else:                                   → se tutto va bene viene eseguito sempre il seguente blocco.
21   watchdogEnable(4000, true)            → abilita il watchdog e lo imposta a 4 secondi (poi resetta).
22   print("Watchdog Abilitato a 4s")
23   while true:
24     print("Entro nel Ciclo da 5s tutto Ok")
25     for _ in 0..4:                      → crea un ciclo di 5 lampeggiamenti lenti (durata: 5 secondi).
26       led1.put(High)
27       sleep(50)
28       led1.put(Low)
29       sleep(950)
30    watchdogUpdate()
```

Questo è solo un semplice esempio di come utilizzare il watchdog, ma tu per esempio potresti cambiare il valore di alcune variabili o mandare via usb un messaggio di allarme che un computer leggerà. Molte cose di questo programma dovrebbero essere note, ma è interessante capire come funziona la procedura watchdogCausedReboot() se avviene per qualche motivo un restet del sistema, cambia il suo valore da *false* → *true* dandoti cosi la possibilità di prendere provvedimenti. Durante i test ho notato che se mando in time-out il watchdog, e poi caricando di nuovo il programma (magari perché modificato) tiene traccia dell'avvenuta anomalia ed entra subito in "allarme" ma resettandolo a runtime al riavvio si comporta come previsto.

3.9 – Multicore.

Una delle caratteristiche più interessanti dell'RP2040, è che questo microcontrollore è dotato di due core, ovvero due unità di calcolo indipendenti che quindi possono svolgere due "lavori" contemporaneamente senza bloccarsi o rallentarsi a vicenda, questa caratteristica è nota come parallelismo. Sui computer con i moderni sistemi operativi e linguaggi di programmazione adatti, gestite questi processi è relativamente semplice, ma qui sul RP2040 (che non ce un sistema operativo che ti aiuta) potrebbe essere un po' più complicato, non tanto sul far girare due programmi (o procedure) sui due core, ma quando dovranno, se necessario, scambiarsi dei dati o magri i risultati di un calcolo, che un core deve avere per poter finire il suo lavoro. Nei prossimi esempi vedrai come fare, come di consueto si partirà con delle cose facili. Lo schema che adotterai questa volata è:

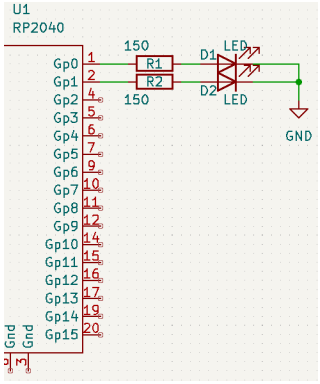


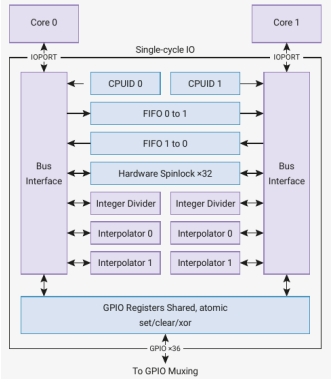
Figura 12

Lo scopo di questo programma sarà come promesso molto semplice, ovvero far lampeggiare i due led, usando per ognuno un core diverso, certo non occorrono due core per fare questa cosa, ma sarà utile per cominciare a vedere come si gestiscono come attivarli e comunque cominciare a prendere confidenza con le procedure base. Ecco di seguito il programma:

```
01 import picostdlib/[gpio, time, stdio, multicore]

02 setupGpio(led1, 0.Gpio, true)
03 setupGpio(led2, 1.Gpio, true)

04 proc lampeggio2() {.cdecl.} =          → procedura che sarà eseguita sul core1.
05   while true:
06     led2.put(High)
07     sleep(150)
08     led2.put(Low)
09     sleep(500)

10 multicoreResetCore1()                  → il core1 va assolutamente resettato prima di utilizzarlo.
11 multicoreLaunchCore1(lampeggio2)       → lancia la procedura "lampeggio2" sul core1.

12 while true:                            → questa parte viene eseguita sul core0.
13   led1.put(High)
14   sleep(300)
15   led1.put(Low)
16   sleep(700)
```

Far eseguire due compiti diversi sui due core non è difficile. Nelle prima riga hai importato il modulo *multicore* che ti fornisce tutte le procedure per la gestione dei due core. Nella quarta linea hai semplicemente creato una normalissima procedura `proc lampeggio2() {.cdecl.}` che potrà essere più o meno complessa (in questo caso molto semplice) questa procedura quando richiamata sarà processata sul core1. La decima riga ovvero `multicoreResetCore1()` deve essere sempre eseguita prima di lanciare la procedura sull'altro core, altrimenti non parte. L'undicesima linea `multicoreLaunchCore1(lampeggio2)` non fa altro che lanciare

realmente la procedura sul core1, da ora i due led lampeggeranno pilotati indipendentemente l'un l'altro. E fin qui è facile, ora si complica un po', e per poter proseguire è meglio vedere la struttura interna del RP2040:



Due cose vanno sottolineate, la prima che entrambi i core hanno accesso al registro *Gpio*, ma la cosa di maggior importanza sono i due registri, FIFO to 1, che avrà il compito di portare i dati dal core0 → al core1, e FIFO to 0 che porterà i dati dal core1 → al core 0 e questo è l'unico modo che avrai di scambiare dati, risultati ecc tra le due unità. Nel prossimo esempio ci sarà un primo scambio di dati tra core0 e core1 per indicare a quest'ultimo di quanti elementi dovrà essere la sequenza che calcolerà; in seguito i due core calcoleranno e creeranno indipendentemente le sequenze e calcoleranno le somme (core0 è volutamente più veloce di core1). Core0 essendo più veloce nel fare l'operazione dovrà necessariamente attendere che core1 finisca il compito e ponga sul registro il valore, che verrà "prelevato" e sommato per poi fornire la somma totale. Ecco il programma:

```
01 import picostdlib/[gpio, time, stdio, multicore]
02 import std/[strformat, math]

03 stdioInitAll()
04 sleep(2000)                                  → attende 2 secondi per essere certi di leggere l'usb.

05 print("Partenza..")                          → segnala la corretta partenza del dispositivo.
06 # ---------- Core1 ----------
07 proc alCore1() {.cdecl.} =                   → procedura che deve girare sul core1.
08   var
09     numElemLC1: uint32
10     listC1 = newSeq[uint32](0)               → crea una sequenza vuota di tipo uint32.
11     sumListC1: uint32                        → variabile che conterrà il valore della somma degli elementi.

12   if multicoreFifoRvalid() == true:          → controlla se ci son dati validi sul FIFO 0to1.
13     numElemLC1 = multicoreFifoPopBlocking()  → se ci sono dati, li legge e li mette in numElemLC1.
14     multicoreFifoDrain()                     → elimina tutti i dati eventualmente rimasti sul FIFO.

15   print("Creo Sequenza Core1 \n")
16   for y in countup(uint32(1), numElemLC1):   → riempe la lista (1..numElemLC1).
17     listC1.add(y)
18   sleep(50)                                  → pausa introdotta per rallentare l'operazione.

19   sumListC1 = sum(listC1)                    → calcola la somma di tutti gli elementi in lista.
20   print(fmt"Somma Lista2: {sumListC1}")

21   if multicoreFifoRvalid() == false:         → se sul registro non ci son dati…
22     multicoreFifoPushBlocking(sumListC1)     → poni sul FIFO 1to0 il valore uint32 calcolato (somma elementi).

23 # ---------- Core0 ----------
24 const
25   lenListC0: uint32 = 100                    → numero elementi lista da creare su core0.
26   lenListC1: uint32 = 200                    → numero elementi lista da creare su core1.

27 multicoreResetCore1()                        → necessario resettare core 1 prima di poterlo usare.
28 multicoreLaunchCore1(alCore1)                → esegue la procedura "alCore1" sul core1.
29 if multicoreFifoRvalid() == false:           → se sul FIFO 0to1 non ci son dati...
30   multicoreFifoPushBlocking(lenListC1)       → allora scrivi sul registro il valore "lenListC1" (uint32).

31 var
32   listC0 = newSeq[uint32](0)                 → crea una sequenza vuota di tipo uint32.
33   sumListC1: uint32                          → variabile per contenere il risultato della somma lista di core0.
34   sumListC0: uint32                          → variabile per contenere il risultato della somma lista di core1.

35 print("Creo Sequenza Core0 \n")
36 for x in countup(uint32(1), lenListC0):      → riempe la lista (1..numElemLC0).
37   listC0.add(uint32(sqrt(float(x))))         → fa la radice quadrata del numero x.
38   sleep(20)                                  → pausa introdotta per rallentare l'operazione.

39 sumListC0 = sum(listC0)                      → calcola la somma di tutti gli elementi in lista.
40 print(fmt"Somma Lista1: {sumListC0}")
41 multicoreFifoDrain()                         → elimina tutti i dati eventualmente rimasti sul FIFO.

42 while true:                                  → crea un ciclo infinito di attesa dati...
43   if multicoreFifoRvalid() == true:          → se sul FIFO 1to0 ci son dati validi...
44     sumListC1=multicoreFifoPopBlocking()     → se ci sono dati, li legge e li mette in sumListC1.
45     print(fmt"La Somma L1+L2 --> {sumListC1+sumListC0}")
46     break                                    → interrompi il ciclo ed esce da esso.

47 print("-- Fine Processo –")                  → stampa di conclusione corretta del programma.
```

```
Partenza..Creo Sequenza Core0
creo Sequenza Core1
Somma Lista1: 625
Somma Lista2: 20100
La Somma L1+L2 --> 20725
-- Fine Processo --
```

Come nel programma di visto precedentemente, la prima cosa che incontri è la procedura proc alCore1() {.cdecl.} che sarà eseguita sul core1 quando richiesto. Al sui interno alla linea dodici troverai multicoreFifoRvalid(), che quando risulterà vera consentirà di entrare in quel blocco, solo se sul FIFO 0to1 ci sarà un dato valido da poter essere letto. La procedura che andrà poi effettivamente a leggere il registro è multicoreFifoPopBlocking() e che poi salverà questo valore (uint32) in una variabile interna alla procedura (linea 13). Alla linea quattordici ce la procedura multicoreFifoDrain() che va a cancellare tutti i dati eventualmente presenti nel FIFO; nel caso (ipotetico) ci fosse stato un ciclo dopo questa procedura, multicoreFifoRvalid() sarebbe risultato falso e non ci sarebbe più stata la lettura del registro. Il ciclo *for* alla linea sedici, crea con il core1 una lista di duecento elementi (numero che hai passato da core0 a core1), e di seguito questi elementi verranno tutti sommati tra loro e salvati nella variabile *sumListC1* di tipo unint32 che verrà passata a core0; la creazione di questa lista che dura circa 10s sarà molto più lenta di quella creata sul core0, questo è voluto per farti notare che il core (o il processo) più veloce deve rimanere in attesa di tutti i dati prima di procedere. Calcolata la somma (di core1), controlli che sul registro non ci siano altri dati, se non ci sono, allora sul FIFO 1to0 si riversa il contenuto di *sumListC1* con la procedura multicoreFifoPushBlocking(sumListC1). La linea ventisette e ventotto le conosci, sono rispettivamente quelle che attivano il core1 e che lanciano la procedura *alCore1*. Anche le linee ventinove e trenta ormai dovresti conoscerle, una controlla se il registro è disponibile per la scrittura, l'altra nel caso lo sia va a scrivere la variabile *lenListC1* (è il numero di elementi che vogliamo nella lista sul core1). Alla linea quarantadue, crei un loop infinito che diventerà vero solamente quando il core1 ha finito i suoi calcoli e pone il risultato sul FIFO, prima di allora il programma non può proseguire perchè altrimenti gli mancherebbe un dato, quindi lo deve attendere. Una volta diventata vera l'espressione if multicoreFifoRvalid() == true: il programma entra nel blocco, somma i risultati dei due core e li stampa a video, l'istruzione *break* interromperà il ciclo per passare al flusso esterno e far stampare che il programma è terminato correttamente.

3.10 – I2C.

Fino ad ora abbiamo interagito solo con le uscite e gli ingressi propri del RP2040, ma ti capiterà molto spesso di dover collegarci dei dispositivi elettronici esterni, come display, memorie o sensori digitali. Per poter comunicare con questi dispositivi, ci son molti sistemi, uno è il bus *i2c,* che è uno standard assi diffuso e che troverai molto di sovente. Il modulo *picostdlib* ti mette a disposizione delle procedure per poterlo usare. In questa sezione non vedrai una vera e propria applicazione o come interagire con un specifico dispositivo cosa di solito complessa, perché normalmente per fare ciò avrai bisogno di moduli esterni che semplificheranno di molto le cose nascondendoti per lo più tutto il processo di lettura e scrittura (vedrai degli esempi nel **capitolo 4**) ma vedrai come funziona a basso livello per avere una visione completa delle cose nel caso volessi scrivere un tuo modulo per un qualche dispositivo. Il bus I2C è costituito da due fili per l'alimentazione (+Vcc e Gnd), e due fili per la la comunicazione: *sda* dove transitano i dati, ed *scl* che è il clock per sincronizzare le tempistiche del segnale. Su questo bus, potrai collegare più dispositivi dove uno sarà il master mentre tutti gli altri saranno degli slave come in **Figura 13**.

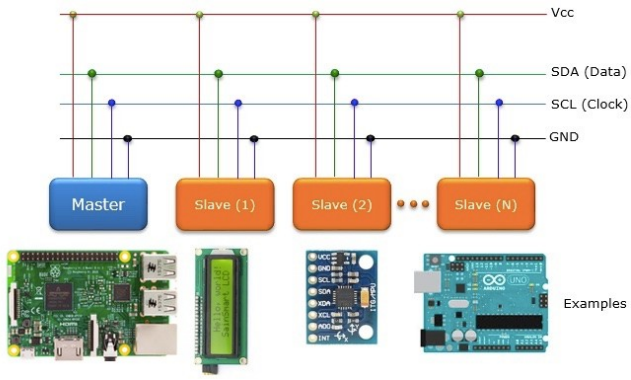


Figura 13

Per mandare i dati corretti al dispositivo corretto, ogni salve avrà un proprio indirizzo (generalmente costituito da un parte impostata dal costruttore e dei bit che setta il progettista). I dati che dovrai mandare invece dipendono dal dispositivo che hai, e dovrai vederlo sul suo datasheet. Nell'esempio qui vedrai solo come impostare i dati da trasmettere nel primo esempio in un modo un po' più complesso ma molto elastico, mente nel secondo farò l'ipotesi che questo dato sia noto a priori o comunque "calcolabile" facilmente in compilazione. Primo esempio:

```
01 import picostdlib/[gpio, time,  i2c]          → importa il modulo i2c.

02 sleep(2000)                                   → pausa di due secondi prima di cominciare.

03 var i2cBlok = i2c1                            → seleziona il blocco su cui usare i2c (i2c1/i2c0).
04 i2cBlok.init(50_000)                          → imposta la frequenza di trasmissione (50khz).
05 18.Gpio.setFunction(I2C)                      → imposta la funzionalità di Gpio 18 (I2c) sda.
06 18.Gpio.pullUp()                              → attiva il pull-Up interno (50-80k).
07 19.Gpio.setFunction(I2C)                      → imposta la funzionalità di Gpio 19 (I2c) scl.
08 19.Gpio.pullUp()                              → attiva il pull-Up interno (50-80k).

09 let frase = "ciao"                            → stringa o set di caratteri da scrivere su i2c.
10 let indirzzoEl1 = frase[0].addr #ex-unsafAddr → indirizzo non tracciato del primo elemento della frase.
11 let noElem = csize_t(frase.len*sizeof(frase[0])) → numero di elementi presenti nella frase.

12 while true:
13   writeBlocking(i2cBlok, 0x50, indirzzoEl1,   → scrittura dei dati su i2c.
                   noElem, true)
14   sleep(5)
```

Prima di iniziare la spiegazione delle singole righe, va detto che per vedere quanto riportato in ***Figura 14***, dovrai collegare un dispositivo i2c e dovrai probabilmente modificare l'indirizzo *0x50* con quello a cui risponde, altrimenti non ci sarà trasmissione.

Figura 14

Nella **Figura 14** in viola puoi vedere l'indirizzo del dispositivo che viene generato automaticamente dalla procedura writeBlocking() più una lettera "*W*" che equivale ad un bit alto ovvero che si sta scrivendo qualcosa sul dispositivo (se stessi leggendo avresti un bit basso "*R*"). in blu invece puoi osservare i quattro byte di cui è composta la parola "ciao" che verranno trasmessi una alla volta. Tra un byte e l'altro ce un bit di acknowledge che qui è rappresentato in verde. Alla prima linea devi importare il modulo *i2c* che ti fornisce tutte le procedure per la gestione del bus. Alla linea tre indichi quale blocco usare per la trasmissione, ce ne sono due: *i2c1* ed *i2c0*. Alla linea quattro indichi al frequenza di trasmissione, in questo caso ho scelto un frequenza veramente molto bassa, questo perché si è attivato il pull-up interno che ha un valore molto alto (50-80 Kohm) per aumentare questa velocità è meglio usare un pull-up esterno che di solito è sui 10k (ma leggi il datasheet del dispositivo in uso per maggiori informazioni). Le linee di seguito dovrebbero esserti piu famigliari, si impostano le uscite come *i2c* e si attiva il pull-up; questa procedura può essere semplificata come nell'esempio che vedrai dopo. Nella linea nove, crei una variabile che contiene la parola "ciao"; in Nim questa stringa può essere vista come un array di caratteri (capirai dopo perche l'ho sottolineata). La linea dieci è molto importante, perché va a prendere l'indirizzo di memoria (non tracciato) del primo elemento del array di caratteri che sarà l'indirizzo a cui punta la funzione di scrittura per iniziare il suo lavoro. Alla linea undici invece si fa un'altra importante azione, ovvero si calcola quanti elementi ha l'array a cui stiamo puntando (l'array è omogeneo). Fatto tutto questo, passi alla linea tredici writeBlocking(i2cBlok, 0x50, indirzzoEl1, noElem, true) dove darai in pasto alla procedura tutti i dati necessari per la scrittura del tuo array in questo caso i caratteri "c,i,a,o" come in **Figura 14**. Nota che alla fine dei quattro byte manca il bit di stop, questo perché ho impostato a *true* l'ultimo parametro della procedura, se vuoi il bit di stop poni questo parametro a *false*. Come già detto questo è il caso più complesso ma anche il più "elastico" perché ti permette di trasmettere anche dati con lunghezza non prestabilita (i dati devono essere sempre di otto bit, se più lunghi vanno spezzati e trasmessi su più byte!!). Nel secondo esempio vedrai come lo stesso programma può semplificarsi, questa volta l'ipotesi è che a priori conosciamo quanti byte trasmettere:

```
01 import picostdlib/[gpio, time,  i2c]                  → importa il modulo i2c.

02 sleep(2000)                                           → pausa di due secondi prima di cominciare.
03 setupI2c(i2c1, 18.Gpio, 19.Gpio, 50_000, true)        → configurazione rapida di i2c.

04 let numero: byte = 75                                 → assegna a numero 75 (nota numero =< 255 8bit).
05 let indirzzoEl1 = numero.addr                 → indirizzo non tracciato di "numero".
06 while true:
07   writeBlocking(i2c1, 0x50, indirzzoEl1, 1, true)     → scrittura dei dati su i2c.
08   sleep(5)
```

Questa volta abbiamo semplificato le cose; balza subito all'occhio la rapida configurazione dell'*i2c* grazie al template *setupI2c()* a cui basta passare come argomenti il blocco, le uscite dell'ic2, la frequenza, e se attivare

(*true*) o meno (*false*) il pull-up. Alla linea quattro hai assegnato alla variabile *numero* il valore di 75, e **per evidenziare che deve avere 8 bit** l'ho definita come *byte*, di seguito verrà catturato il suo indirizzo (non occorre mettere [] perché ora ce solo un valore non è un array!!). Questa volta il calcolo della lunghezza non viene fatto, perché è noto ed in questo caso vale uno (1=1Byte), quindi nella procedura writeBlocking(i2c1, 0x50, indirzzoEl1, 1, true) scriverai direttamente il valore corretto (uno). Nel caso avessi avuto un array per esempio di sette elementi (dove non puoi ne levare ne aggiungere elementi) avresti potuto scrivere writeBlocking(i2c1, 0x50, indirzzoEl1, 7, true). Ecco cosa ottieni con questo esempio, nella traccia dell'oscilloscopio leggerai "K" anziché 75, questo perché gli otto bit che risultano dalla conversione di quel numero (01001011) in ascii equivale alla lettera "K".

Figura 15

Nel prossimo esempio, andrai a scrivere in una ipotetica eerpom *i2c* dei dati, in questo caso due byte, e poi andrai a leggerli. **Non è importante ora capire come funziona questo dispositivo e la sequenza di dati da inviarle, ma è importante capire come si leggono i byte sulla *i2c***. Di seguito il segnale di risposta.

Figura 16

La prima cosa che salta all'occhio è che il primo byte questa volta è di colore giallo (anziché viola) e termina con un "R", questo significa che ora stai dicendo al dispositivo che lo vuoi leggere, di seguito ci sono i due byte di risposta che la tua ipotetica eprom ha inviato sul bus alla tua richiesta, ed alla fine ce una barra bianca, che indica che abbiamo inviato il segnale di *stop*. Ecco il programma:

```
01 import picostdlib/[gpio, time, stdio, i2c]                → importa il modulo i2c.
02 import std/[strformat]
03 stdioInitAll()

04 setupI2c(i2c1, 18.Gpio, 19.Gpio, 40_000, true)    → configurazione rapida di i2c.
05 sleep(2000)                                        → pausa di due secondi prima di cominciare (per i messaggi).

06 let arrayScrittura: array[0..3, byte] = [byte(0), byte(0),      → array di byte con 4 elementi da scrivere sulla eeprom.
                                byte(75), byte(12)]
07 var arrayLettura: array[0..1, byte]                            → array di byte con 2 elementi per la lettura della eeprom.

08 # ---------- Scrittura sulla Eeprom ----------
09 print(fmt"Scrivo  sulla eeprom {arrayScrittura[2]}")
10 let addrScrittura = arrayScrittura[0].addr  → indirizzo non tracciato del primo elemento dell'array.
11 let lenScrittura = csize_t(arrayScrittura.len*sizeof(arrayScrittura[0]))  → calcola il numero di elementi dell'array.
12 writeBlocking(i2c1, 0x50, addrScrittura, lenScrittura,                    → scrive l'intero array sulla eeprom.
                false )
13 sleep(100)                                         → pausa per attendere che la eeprom memorizzi i dati inviati.

14 # ---------- Lettura sulla Eeprom ----------
15 let addrLettura = arrayLettura[0].addr      → indirizzo non tracciato dell'array per memorizzare i dati.
16 writeBlocking(i2c1, 0x50, addrScrittura, 2, true )   → scrive i soli primi byte (indirizzo interno del dato da leggere).
17 discard readBlocking(i2c1, 0x50, addrLettura, 2,     → legge i dati e li salva nell'array (dati dall'indirizzo prima indicato).
                        false)
18 sleep(5)

19 print(fmt"Leggo sulla eeprom {arrayLettura}")
```

Questo programma è un po' più complicato, ma cercherò di spiegartelo almeno a grosse linee ma non preoccuparti se non lo capisci, ora la cosa essenziale da comprendere è la risposta del dispositivo e non la sua implementazione. Alla linea sei, trovi arrayScrittura: array[0..3, byte] = [byte(0), byte(0),byte(75), byte(12)], questo array contiene quattro elementi, di cui, i primi due, essenziali e sono l'indirizzo interno alla eprom dove salverà il byte che devono essere indicati esplicitamente sia in scrittura che in lettura, i seguenti due byte son i valori che vuoi salvare (a partire dall'indirizzo fornito). Il secondo array arrayLettura: array[0..1, byte] ti servirà quando andrai a leggere la eprom per memorizzare i dati da usare poi nel tuo programma. Alla linea dieci calcoli l'indirizzo del primo elemento di *arrayScrittura*, nella successiva calcoli quanti elementi son contenuti nell'array. Ottenuti questi dati, scrivi il tutto sulla eprom con writeBlocking(). Alla linea tredici diamo una pausa affinché l'eprom memorizzi permanentemente i dati inviati. Entriamo ora nel vivo della lettura su *i2c*; per prima cosa anche qui (linea quindici) andrai a trovare l'indirizzo del primo elemento dell'array *arrayLettura* con il metodo che ormai dovresti ben conoscere addrLettura = arrayLettura[0].addr. Ora alla linea sedici scrivi sulla eprom l'indirizzo dove è contenuto il byte che ti interessa in questo caso 0x00 writeBlocking(i2c1, 0x50, addrScrittura, 2, true); nota che l'ultimo parametro è posto a *true*, in pratica gli stai dicendo di NON inserire il bit di stop, ma di continuare. Finalmente possiamo leggere i due byte che ci interessano con l'istruzione discard readBlocking(i2c1, 0x50, addrLettura, 2, false). All'inizio troviamo l'operatore *discard* questo perché la funzione ritorna un valore, che in questo caso non ci interessa e bisogna dirlo esplicitamente al compilatore, altrimenti da errore. Proseguendo in questa procedura, questa volta dopo l'indirizzo dell'array, ho scritto direttamente "*2*" questo lo puoi fare quando sei sicuro del numero esatto del numero di byte da leggere (e che non potrà mai variare come in questo caso) e per concludere, trovi ora "*false*", che contrariamente a prima ora il microcontrollore inserirà il bit di stop (la linea bianca sul tracciato di figura 16).

Nota:
se nella procedura readBlocking e writeBloking il parametro noStop è:

- noStop = ***false*** → viene messo il bit di stop.
- noStop = ***true*** → viene omesso il bit di stop.

4.0 – Moduli Esterni.

In questo capitolo, voglio presentarti alcune utility e moduli per componenti esterni che ho scritto e che potrebbero esserti utili per i tuoi prototipi, come display, expander I/0, eprom. Va chiarito subito che io non sono un informatico, quindi il codice di queste librerie non sarà il massimo, ma ad ora che scrivo questo manuale, non ci son molte alternative, e se ne usciranno di migliori e più performanti sarò ben felice di apportare i dovuti aggiornamenti. Questi moduli puoi trovarli in:

https://github.com/Martinix75/Raspberry_Pico

I moduli che qui analizzeremo, per poter funzionare correttamente devono essere posti nella stessa cartella del file da compilare, oppure nell'import va specificato il percorso completo della collocazione del file.

4.1 – Picousb.

Ti capiterà molto spesso, di dover mandare dati verso l'RP2040 per interrogarlo o per dargli un qualsiasi input, un modo per farlo, oltre a quanto hai visto nel **capitolo 2**, è quello di mandare i dati via usb (ricordo che son stringhe anche gli eventuali numeri!!). La libreria standard offre la procedura *getCharWithTimeout*, che però legge dal buffer un carattere alla volta. Il modulo *picousb* invece legge per te tutto ciò che hai mandato (una frase intera, un "numero" ecc) che poi userai per i tuoi scopi e ti offre anche dei pratici moduli per le conversioni numeriche. Lo schema che ti occorre per il prossimo esempio è il seguente:

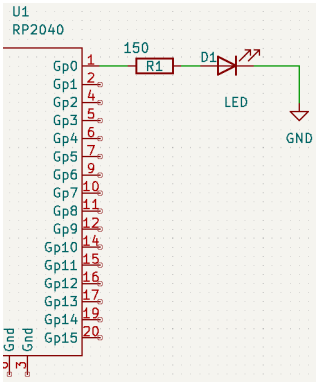


Figura 17

Mentre il programma che andrai a compilare ed ad installare sarà:

```
01 import picostdlib/[gpio, time,  stdio, ]
02 import picousb                          → importa il modulo picousb.

03 stdioInitAll()
04 sleep(2000)
05 print("PicoUsb Ver: " & picousbVer)     → stampa sul terminale usb la versione del modulo.

06 let serialUsb = PicoUsb()               → crea l'oggetto serialUsb.
07 setupGpio(led1, 0.Gpio, true)           → configura la porta di uscita.

08 while true:                             → crea un ciclo infinito.
09   if serialUsb.isReady == true:         → vero se sul buffer usb ci sono dei dati.
10     let stringa = serialUsb.readLine()  → legge il contenuto del buffer e lo memorizza in stringa.
11     let nuLamp = serialUsb.toInt(stringa) → trasforma la stringa appena letta in un numero intero.
12     for _ in countup(1, nuLamp):        → fa lampeggiare il led, quante volte indicato via usb.
13       led1.put(High)
14       sleep(300)
15       led1.put(Low)
16       sleep(700)
```

La libreria puoi trovarla qui:

https://github.com/Martinix75/Raspberry_Pico/tree/main/Utils/picoUsb

L'uso del modulo è semplice, alla linea sei crei l'oggetto PicoUsb() che conterrà tutte le procedure che ti possono essere utili per lavorare con la usb. Alla linea nove isReady(), va a controllare ad ogni ciclo, se son arrivati dei nuovi dati; se ci sono entri nel corpo di *if* e vai effettivamente a leggere questi dati ed a salvarli in una variabile con la procedura serialUsb.readLine() (verrà salvato tutto non solo un carattere come farebbe la funzione di libreria). Se ora ti interessa avere solo la stringa, magari perché il tuo protocollo prevede così, puoi liberamente usarla, ma come in questo caso che devi inserire un numero intero nel costrutto *countup()* quindi lo devi convertire. Per questi casi ho inserito nel modulo due procedure, la prima toInt(stringa) che come avari intuito converte da stringa ad intero, e la seconda toFloat(stringa) che converte in un numero *float,* nel caso non potesse fare la conversione stampa sul terminale "ERROR!! not INT converted!".
Se hai avuto a che fare con odispositivi elettronici, avrai sicuramente visto che con molti di essi puoi comunicare con un protocollo. Questo protocollo permette di inviare molti comandi complessi e di solito la struttura è "*comando#valore1#valore2..*" o qualcosa del genere. Nel prossimo esempio voglio crearne uno molto semplice che ti permette di selezionare un led e farlo lampeggiare per il numero di volte che desideri, lo schema da utilizzare sarà:

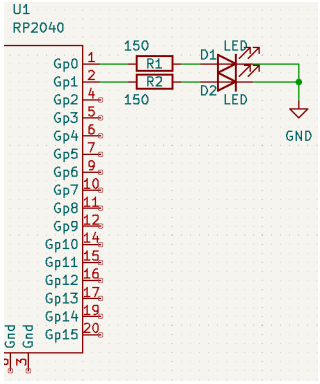


Figura 18

Per prima cosa devi decidere come costruire il protocollo, in questo caso va bene quello prima proposto, quindi sarà qualcosa tipo: "*led1#5*" oppure "*led2#7*". Ed ecco il programma:

```
01 import picostdlib/[gpio, time,  stdio, ]
02 import picousb                                  → importa il modulo picousb.
03 from strutils import split, toLowerAscii       → importa vari procedure utili per l'usb.

04 stdioInitAll()
05 sleep(2000)
06 print("PicoUsb Ver: " & picousbVer)
07 const protocol = "0.1.0"                       → stringa per l'identificazione della versione.

08 let serialUsb = PicoUsb()                      → crea l'oggetto serialUsb.
09 setupGpio(led1, 0.Gpio, true)                  → configura 0.Gpio come uscita.
10 setupGpio(led2, 1.Gpio, true)                  → configura 1.Gpio come uscita.
11 while true:
12 if serialUsb.isReady == true:                  → vero se sul buffer usb ci sono dei dati.
13   let stringa = serialUsb.readLine             → legge il contenuto del buffer e lo memorizza in stringa.
14   let splitStringa = stringa.split('#')        → crea una sequenza dividendo la stringa al carattere #.
15   if splitStringa[0].toLowerAscii() == "led1": → se il l'elemento 0 è = a led1 entra qui..
16     print("Led1 Attivato..")
17     let numeroRipetizioni = serialUsb.toInt(splitStringa[1])  → converte il numero (stringa) in numero intero.
18     for _ in countup(1, numeroRipetizioni):    → fai il numero di ripetizioni inviato via usb.
19       led1.put(High)
20       sleep(200)
21       led1.put(Low)
22       sleep(800)
23     elif splitStringa[0].toLowerAscii() == "led2":  → se il l'elemento 0 è = a led2 entra qui..
24       print("Led1 Attivato..")
25       let numeroRipetizioni = serialUsb.toInt(splitStringa[1])  → converte il numero (stringa) in numero intero.
26       for _ in countup(1, numeroRipetizioni):  → fai il numero di ripetizioni inviato via usb.
27         print("Led2 Attivato..")
28         led2.put(High)
29         sleep(400)
30         led2.put(Low)
31         sleep(600)
32     elif splitStringa[0].toLowerAscii() == "version":  → se il l'elemento 0 è = a version entra qui..
33       print("Protocol Version: " & protocol)  → stampa il numero di versione sulla console usb.
34     else:
35       print("Comando Errato! Ripeti!")        → se nessun comando prima valido, entra qui.
```

Una volta inviato il dato via usb, viene letto alla linea tredici, di seguito con *split* viene creata una *sequenza* divisa dove incontra il carattere "#"; ad esempio:

"led1#45" → @["led1", "45"]

Ora ti basta analizzare i singoli elementi della sequenza per dare degli "ordini", ad esempio alla linea quindici, controlli se il comando è *led1* in questo caso entri in quella porzione di codice che governa quel led. All'interno trovi un altro controllo alla linea diciasette, che come hai visto nell'esempio precedente, converte il numero stringa in un intero (string: "45" → int: 45), adatto per essere inserito in *countup()*. In questo programma ho usato la procedura toLowerAscii() questa fa si che si prevengano possibili errori di scrittura rendendo la stringa sempre con i caratteri minuscoli (questo per evitare tutte le possibili permutazioni tra caratteri piccoli e grandi). In fine ho messo un *else* alla fine, questo è per catturare tutte le stringe che non rientrano nel protocollo e segnalare all'utente che sta sbagliano qualcosa, cosa che ti consiglio sempre di inserire nei tuoi programmi.

4.2 – Scanner I2C.

Prima di proseguire nelle prossime librerie, che faranno largo uso dell'I2C, è bene sapere dell'esistenza di un piccolo scanner I2C, che va a controllare la presenza di tali dispositivi sul PR2040, e che ti fornirà delle indicazioni utili, tipo sul blocco in cui è collegato il dispositivo (I2c0/i2c1), su quali pin è collegato e l'indirizzo a cui risponde. Non ti occorre compilarlo, perché assieme ai sorgenti, ho messo anche il file *uf2* già pronto all'uso solo da installare sul microcontrollore, questa utility la puoi trovare in:

https://github.com/Martinix75/Raspberry_Pico/tree/main/Utils/scannerI2c

l'uso è semplice, una volta installato, ti basta aprire il terminale seriale, inviare un qualsiasi carattere per far iniziare la scansione, quando arriva alla fine, leggerai qualcosa del tipo:

```
====================
Start i2c Scanner ver 1.1.1
====================
Now Sacanning Blok i2c0...
*
*
*
*
*
*
No Devices Found in This Blok!!
Now Sacanning Blok i2c1...
*
*
*
*
*
*
1 Device Found @["0x50 Gpio(18 & 19)"]
```

Che indica che è stato trovato un dispositivo sul blocco ***I2C1*** sui pin ***18*** e ***20*** e il suo indirizzo è ***0x50*** e che la versione in uso dello scanner è la "***1.1.1***".

4.3 – Numeri Casuali.

Nella libreria del RP2040, non ce un generatore di numeri casuali, e tanto meno si può usare la libreria *std/random* di Nim perché si basa sulle librerie sottostanti che si interfacciano al sistema operativo, e qui non hai un sistema operativo! Ecco la necessità di scriverne una che colma questa lacuna e la puoi trovare in:

https://github.com/Martinix75/Raspberry_Pico/tree/main/Libs/random

Questo modulo genera dei numeri pseudo-casuali usando l'algoritmo Lehmer, forse non è il massimo, ma ha il vantaggio di essere relativamente semplice e quindi non richiede grossi calcoli da parte della cpu, rendendo cosi i tuoi programmi molto leggeri a livello computazionale. La versione attuale è la 0.5.5 che ancora non genera numeri casuali float, ma sarà implementata nella 0.6.0. Ecco un semplice esempio:

```
01 import picostdlib/[stdio, gpio, i2c, time]
02 import std/strformat                    → importa il modulo strformat.
03 import random                           → importa il modulo random.

04 stdioInitAll()
05 sleep(2000)
06 print(fmt"Inizio a generare numeri casuali..")

07 randomize()                             → tenta di inizializzare il seme con un numero nuovo.

08 let rnd = random(3)                     → genera un numero float compreso tra 0 ed 1.
09 print(fmt "Ho generato il numero (da 0..1):  {rnd: 0.3f}")        → 0.825

10 let intRnd = randomInt(2,30)            → genera un numero intero compreso tra 2 e 30.
11 print(fmt"ho generato il numero casuale intero tra 2 e 30: {intRnd}")  → 6

12 let charRnd = randomChar()              → genera un carattere casuale tra a e z e tra A e Z.
13 print(fmt"Ho generato il carattere casuale:  {charRnd}")          → t
```

Alla linea tre importi il modulo per la generazione dei numeri pseudo-casuali, alla linea sette, si tenta di generare un numero di seme che dipende da quanto tempo (in millisecondi) il microcontrollore è acceso, se non lo usi viene utilizzato il valore di default, ma questo rischia di darti numeri "casuali" ripetibili. La linea otto con random(3) genera un numero casuale compreso tra 0 e 1 ed utilizza tante cifre significative quante specificate in argomento (max 10). Alla linea dieci vai a generare un numero intero randomInt(2,30), che sarà compreso tra i due numeri indicati (di default è tra 0 e 100). La procedura randomChar() alla linea dodici invece ritorna un carattere minuscolo o maiuscolo.

4.4 – Display LCD1602.

Figura 19

Il prossimo modulo che voglio presentarti, si chiama con molta fantasia, *display1602*, e serve per accedere facilmente a questa tipologia di display (dovrebbe essere della famiglia HD44780). In questo caso lo piloterai via I2C (il modulo non supporta il pilotaggio diretto) con il modulo *PCF8574* che è un expander I/O che conoscerai più avanti, che andrà effettivamente a pilotare il display. La libreria è disponibile in:

https://github.com/Martinix75/Raspberry_Pico/tree/main/Libs/display1602

Sempre all'indirizzo sopra, trovi già molti esempi d'uso, che dovrebbero illustrarti abbastanza bene l'uso di questa libreria, ma qui comunque ti proporrò un esempio base. Per quanto riguarda lo schema, si può adottare questo (le resistenze di pull-up son consigliate, ma funziona anche senza a frequenze basse almeno):

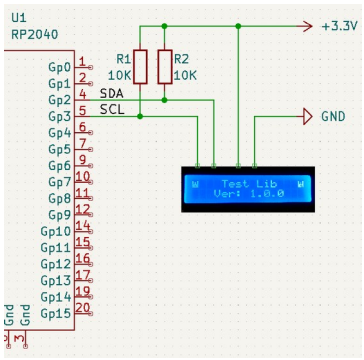


Figura 20

Ecco il programma, che ti illustrerà alcune delle procedure del display:

```
01 import picostdlib/[stdio, gpio, i2c, time]
02 import display1602                                  → importa il modulo display1602.

03 setupI2c(i2c1, 2.Gpio, 3.Gpio,100_000, true)       → inizializza l'2C.

04 let lcd = newDisplay(i2c=i2c1, lcdAdd=0x27, numColum=16, numLines=2)    → inizializza l'oggetto "lcd".

05 while true:
06   lcd.clear()                                      → pulisce l'intero display.
07   lcd.moveto(0,0)                                  → sposta il cursore nella posizione 0,0 (riga 0, colonna 0 home).
08   lcd.putString("Ciao da Nim")                     → scrive la stringa a partire dalla posizione attuale del cursore.
09   lcd.moveTo(7,1)                                  → sposta il cursore alla riga 1, colonna 7.
10   lcd.putChar('#')                                 → scrive un carattere ascii nella posizione attuale del cursore.
11   sleep(2000)
12   lcd.clear()                                      → pulisce l'intero display.
13   lcd.centerString(disp1602Ver)                    → scrive una stringa al centro esatto del display.
14   lcd.moveto(0,1)                                  → sposta il cursore alla riga 0 colonna 1.
15   lcd.shiftString("-->", dir=false, cross=true, effect=1)        → fa muovere la stringa sul display.
16   sleep(2000)
17   lcd.moveto(0,0)
18   lcd.clearLine()                                  → cancella la sola linea indicata.
19   sleep(2000)
20   lcd.backLightOff()                               → spegne la retroilluminazione del display.
21  sleep(2000)
```

Queste sono solo alcune delle procedure che puoi usare su questo display, comincio analizzando la quarta riga, newDisplay(i2c=i2c0, lcdAdd=0x27, numColum=16, numLines=2), qui devi indicare il blocco su cui si trova il display, ed il suo indirizzo, ma la cosa più importante è il numero di colonne ed il numero di righe, nella **Figura19** è mostrato quello più classico con appunto 2 righe e 16 colonne, ma ne esistono anche con 4 righe e 20 colonne e vanno indicate correttamente. Alla linea sei clear() cancella tutto ciò che è presente sul display e ritorno alla posizione 0,0 (la linea sette infatti è inutile, ma l'ho inserita per farti vedere come si torna alla pozione "home"). Alla linea otto trovi la più semplice delle istruzione per scrivere sul display, ovvero putString("Ciao da Nim") questa procedura scriverà ciò che ha nel suo argomento, a cominciare dalla posizione assegnata al cursore (nel tuo caso 0,0). La procedura putChar('#') è simile a quella vista prima, solo che invece di scrivere delle stringhe, scrive un solo carattere sempre a cominciare dalla posizione attuale del cursore. La linea tredici cercherà di posizionare la stringa al centro del display, in questo caso sarà determinante solo la posizione della riga e non quella della posizione sulle colonne. Arrivi ora alla funzione più complessa, ovvero shiftString("-->", dir=false, speed=100, cross=true, effect=1) fondamentalmente fa scorrere la stringa sul display, ma ci son diverse modalità:

- dir: se **false** la stringa parte dal lato destro del display per raggiungere il lato sinistro, se **true** parte da sinistra per giungere a destra.
- speed: impone la velocità di scorrimento della stringa. Contro intuitivamente, più basso è il numero più veloce sarà lo spostamento.
- cross: se **false** la stringa sbucherà dal lato selezionato fino a raggiungere la sua massima estensione e non oltre, se **true** invece percorrerà per intero lo schermo fino al lato opposto.
- effect: se uguale a 1 la stringa non uscirà (dal alto opposto) dal display, mentre se posto uguale a 0, la si vedrà uscire per intero dal display.

La linea diciotto al contrario di *clear()* non andrà a cancellare tutto, ma soltanto la linea dove si trova in quel momento il cursore. Infine alla linea venti andrai a spegnere il led di retroilluminazione. Ci sono ancora altre procedure che però qui non ti mostro, ma che son ben documentate nella libreria stessa e nei vari esempi allegati ad essa, in modo particolare potrebbe piacerti la procedura *customChar()*che ti consente di creare i tuoi caratteri (o disegni, come la corona nella **Figura 19**) con una matrice di pixel 5x8. Nel capitolo dedicato alle librerie troverai le funzioni pubbliche che potrebbero maggiormente interessarti con tutti i commenti.

4.5 – Display SSD1306.

Figura 21

In questa sezione, ti presento un modulo per un'altra tipologia di display, ovvero gli *oled* display, che oltre ad essere molto luminosi ed ad alto contrasto, fondamentalmente sono dei display grafici, dove i singoli caratteri non son contenuti in una matrice predefinita e posizionata, ma in sostanza sono una matrice di pixel che occupano tutta la superficie del dispositivo, quindi accendendo o spegnendo un determinato pixel in una determinata posizione, si possono creare caratteri, forme geometriche disegni ecc vedi **Figura 21** (ce ne sono di varie forme e dimensioni). Questi dispositivi sono enormemente più complessi da pilotare rispetto a quelli precedenti (che già non son banali di loro), ma questo modulo (in realtà son più moduli) mascherano la complessità e ti lasciano solo il piacere della programmazione. Ecco lo schema elettrico (anche qui le resistenza di pull-up son consigliate, ma a bassa frequenza si possono omettere, bastano quelle interne al microcontrollore):

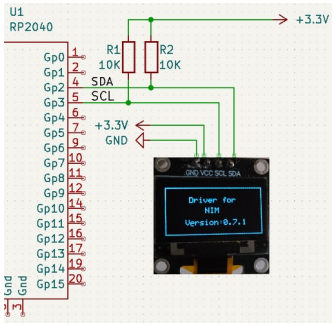


Figura 22

Nel seguente programma ti mostrerò solo le funzioni base, che dovrebbero bastarti per l'uso normale, ma esiste anche il modulo aggiuntivo *utilssd1306* che aggiungono funzionalità. Di base con questo modulo si dovrebbero poter caricare anche disegni, ma nella versione 0.7.x ancora non è stato implementato. Quello che invece è stato fatto, è darti la possibilità di caricare caratteri custom che puoi creare abbastanza semplicemente e non solo potrai crearli come vuoi, anche usando una matrice diversa dalla 5x8 usata fino ad ora (da testare). I caratteri saranno su un file esterno *font5x8.nim*. In questa versione ce la possibilità di scegliere tra *std* che vengono caricati di default, e *test* che li ho creati io (in parte) per vedere se funzionavano. Ma ecco il programma che ti illustra queste procedure principali:

```
01 import picostdlib/[gpio, i2c, time]
02 import ssd1306                                  → carica il modulo ssd1306 (a sua volta carica framebuffer).
                                                     e font 5x8.
03 sleep(2000)

04 setupI2c(blokk=i2c1,psda=2.Gpio, pscl=3.Gpio, freq=100_000)
05 let oled = newSsd1306I2C(i2c=i2c1,             → crea l'oggetto oled per gestire il display.
                            lcdAdd=0x3C, width=128, height=64)
06 oled.clear(0)
07 oled.rect( x=5 ,y=5, width=18, height=15, color=1, → crea un rettangolo vuoto a partire dalla coordinate date.
              fill=false)
08 oled.circle(xCenter=114, yCenter=15, radius=12,  → crea un cerchio con centro e raggio dati.
               color=1)
09 oled.text(text="Driver for", x=35 ,y=17, color=1)  → scrive una stringa nella posizione voluta.
10 oled.text(text="NIM", x=55 ,y=30 , color=1,
             charType="test")
11 oled.text(text="Version:" & ssd1306Ver , 30 ,45 ,1, → scrive una stringa nella posizione voluta.
             charType="test")
12 oled.vline(x=5, y=40, height=18, color=1)        → traccia una linea verticale a partire dalle coordinate date.
13 oled.hline(x=5, y=35 , width=18, color=1)        → traccia una linea orizzontale a partire dalle coordinate date.
14 oled.line(xStr=7, yStr=37, xEnd=30, yEnd=53,     → traccia una linea arbitraria.
             color=1)
15 oled.show()                                      → ora mostra sul display quanto tracciato sopra (solo in memoria).
16 sleep(3000)
17 oled.invert(1)                                   → inverti i colori (nero → bianco; bianco → nero).
18 sleep(3000)
19 oled.powerOff()                                  → spegne il display.
20 sleep(2000)
21 oled.powerOn()                                   → riaccende il display.
22 oled.text(text="Version:" & ssd1306Ver , 30 ,45 , → fa scomparire la stringa prima scritta (color = 0).
             color=0, charType="test")
23 oled.show()                                      → aggiorna il display.
```

Come ti avevo anticipato, nonostante la complessità delle librerie, si perché anche se non lo vedi ce bisogno di tre diversi moduli affinché tutto funzioni (*framebuffer, fot5x8, ssd1306*), l'uso è semplice. Alla riga cinque crei l'oggetto *oled* con cui poi richiamare tutte le procedure che ti serviranno. Alla linea sei chiami `clear(0)` che porta a zero tutti i pixel del display di fatto lo cancella per intero. Alla linea sette andrai a tracciare in memoria (e solo in memoria per ora) un rettangolo con `rect( x=5 ,y=5, width=18, height=15, color=1, fill=false)` di seguito traccerai un cerchio e poi del testo su cui non mi soffermo perché mi sembra chiaro come funzioni, cosi come *vline, hline* ed *line*. La linea quindici è più interessante, perché finalmente farà stampare sul display, tutte le cose prima scritte solo in memoria. La linea diciassette inverte quanto stampato fino ad ora sul display, portando i pixel spenti accesi e viceversa. Le Procedure *powerOff()* e *powerOn()* accendono e spengono il display. La ventiduesima linea è quasi identica alla undicesima, ma con *color=0*, questa farà si che si cancelli la sola stringa in questione lasciando visibili tutte le altre.

ATTENZIONE!! ogni volta che vuoi stampare (o eliminare) qualcosa dal display DEVI usare la procedura *show()* altrimenti non verrà visualizzata. Prova ad esempio ad eliminare la riga 23 del programma sopra e vedrai che la stringa questa volta non scomparirà anche se "ordinato" di farlo.

ATTENZIONE!! framebuffer non controlla se il testo e/o le forme geometriche rientrano nei limiti consentiti del display, se tu dovessi impostare coordinate errate sul display NON verrà visualizzato nulla (rimane nero).

Per operazioni più complesse sulle stringhe come centratura, scorrimento, ecc usa il modulo *utilss1306* che lo puoi trovare assieme a questo modulo nel repositorie prima indicato, che però qui non ti faro vedere l'uso anche perché ci sono degli esempi già pronti.

4.6 – PCF8574 Expander I/O.

Ti accorgerai ben presto che i le uscite di un microntrollore non son mai sufficienti, per quante ne abbia son sempre poche. Ecco che perché esistono gli I/O expander. In pratica, in questo caso via I2C, aggiungono uscite o ingressi utilizzabili. Il Pcf8574, è chiamato quasi-bidirezionale, perché o lo usi come buffer di uscita o ingresso, ma non in maniera ibrida. Questo modello può gestire 8bit (1 Byte) ma ce ne sono altri per esempio che ne gestiscono 16 (2 Byte) che svilupperò più avanti. Lo schema base è il seguente:

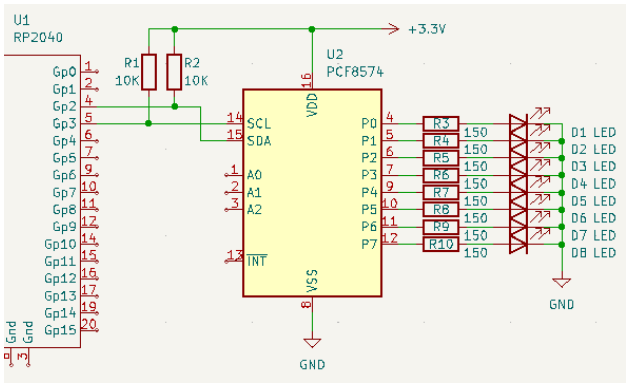


Figura 23

In questa configurazione, per quanto detto prima, userai tutti i bit come uscita e piloteranno dei led. Gli ingressi *A0, A1, A2* servono a cambiare gli indirizzi e non son collegati perché internamente posti a massa e di conseguenza varranno *A1=A2=A3=0* se desideri cambiare indirizzo basta che poni a *+VCC* uno o più di loro. Come di consueto le resistenze di pull-up son opzionali, perché si userà quelle interne (bassa velocità). Ecco il programma illustrativo:

```
01 import picostdlib/[gpio, i2c, time]
02 import pcf8574                                  → importa il modulo pcf8574.

03 setupI2c(i2c1, 2.Gpio, 3.Gpio, 50_000, true)

04 let expander = newExpander(blokk = i2c1, expAdd = 0x20) → crea l'oggetto expander che governa il componente.
05 expander.writeByte(data=0xAA) #10101010=170    → scrive il byte 0xAA sul registro di uscita del pcf8574.
06 sleep(2500)
07 expander.writeBit(p0, on)                      → scrive un singolo bit (alto)  sulla pin p0.
08 sleep(1500)
09 expander.writeBit(p2, on)                      → scrive un singolo bit (alto)  sulla pin p2.
10 sleep(2500)
11 expander.writeByte(data=0x55) #01010101=85     → scrive il byte 0x55 sul registro di uscita del pcf8574.
12 sleep(2000)
13 expander.setLow()                              → setta tutti i bit di uscita a livello basso (tutto spento).
14 sleep(2000)
15 expander.setHigh()                             → setta tutti i bit di uscita a livello alto (tutto acceso).
```

Alla linea quattro vai a creare l'oggetto *expander* che avrà accesso alle sue procedure di gestione. Con la procedura writeByte(data=0xAA) vai a scrivere sul registro del PCF8574 quel valore, che in questo caso sarà "*10101010b*" (immaginati i led accesi dove ce 1 e spenti dove ce 0). Con la linea sette, vai ad accendere (o spegnere) un solo bit ed indicherai il nome ed il valore che desideri, la nomenclatura scelta è:

- p0 = bit0 → pin4 p1 = bit1 → pin5
- p2 = bit2 → pin6 p3 = bit3 → pin7
- p4 = bit4 → pin9 p5 = bit5 → pin10
- p6 = bit6 → pin11 p7= bit7 → pin11
- on = led accesso; off = led spento.

Se hai eseguito il programma, avari visto che una volta scritto il byte, se vai a modificare un singolo bit, il valore del byte in precedenza scritto non varierà modificato ma cambia solo il bit scelto, in altre parole scrivendo un bit unico, non si va a perturbare gli altri. Le linee tredici e quattordici, rispettivamente setLow() e setHigh() pongono o tutti i bit bassi oppure tutti alti. Il prossimo passo, è vedere come funziona l'altra modalità, ovvero come ingresso; per fare questo devi modificare lo schema nel seguente modo:

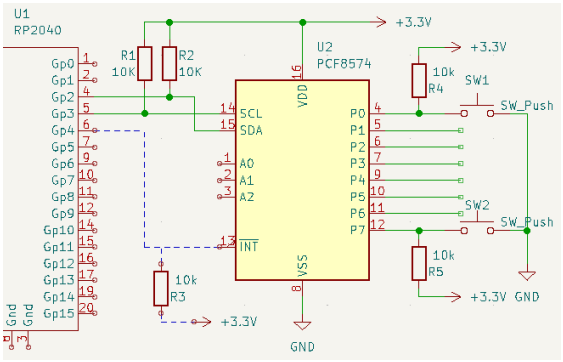


Figura 24

Per semplicità ho usato solo due pulsanti posti nel bit0 (*lsb → 00000001b = 1*) e nel bit7(*msb → 1000000b = 128*) altrimenti avrei appesantito troppo lo schema, ma è ovvio che tu puoi usarli tutti (solo in ingresso!). Di certo avrai notato in **Figura 24** un filo blu tratteggiato, questo è il segnale di interrupt (vedi **sezione 3.7**) che il *PCF8574*, manda al microcontrollore quando ha un cambiamento di stato su uno o più dei suoi pin. Essendoci un pull-up, il segnale normalmente sarà alto, ma premendo il pulsante andrà basso per un istante, attenzione però, anche quando rilascerai il pulsante (cambiando di fatto lo stato) genererà di nuovo il segnale e questo lo dovrai gestire tu nel programma. Nel esempio che ti propongo di seguito, non userò questo segnale; ecco il programma:

```
01 import picostdlib/[gpio, i2c, time, stdio]
02 import pcf8574                                  → importa il modulo pcf8574.

03 stdioInitAll()
setupI2c(i2c1, 2.Gpio, 3.Gpio, 50_000, true)

04 let expander = newExpander(blokk = i2c1, expAdd = 0x20) → crea l'oggetto expander che governa il componente.
05 while true:
06   let bit0 = expander.readBit(p0)               → legge il solo pin p0 (lsb).
07   let bit7 = expander.readBit(p7)               → legge il solo pin p7 (msb).
08   if bit0 == false and bit7 == false:
09     print("Hai premouto il tasto 1 e 2 assieme!")
10   elif bit0 == false:
11     print("Hai premouto il tasto 1 (bit0)")
12   elif bit7 == false:
13     print("Hai premouto il tasto 2 (bit7)")
14 sleep(500)
```

Le uniche due righe da commentare qui sono la sei e la sette che fanno grossomodo la stessa cosa, ovvero leggono il valore presente in quel momento su quel ben determinato pin di ingresso. La Procedura ritorna *false* se si trova a zero, mentre ritorna *true* se si trova a livello alto. Nel prossimo esempio invece leggerai l'intero byte, poi lo gestirai tu capendo quali bit leggere o meno. Ecco un esempio, forse un po' più complesso del precedente per via di alcuni accorgimenti che devi adottare:

```
01 import picostdlib/[gpio, i2c, time, stdio]
02 import pcf8574                                  → importa il modulo pcf8574.

03 stdioInitAll()
04 setupI2c(i2c1, 2.Gpio, 3.Gpio, 50_000, true)

05 let expander = newExpander(blokk = i2c1, expAdd = 0x20) → crea l'oggetto expander che governa il componente.

06 var readBuffer = [byte(0)]                     → crea l'array di byte dove verrà scritto il valore del reg.
07 while true:
08   expander.readByte(readBuffer)                → legge il registro del PCF8547 e lo salva in readBuffer.
09   print("Sul Regostro del pcf8574 ce: " & $readBuffer)
10   var valBuffer = not readBuffer[0]            → fa un not logico per invertire il valore dei bit.
11   print("Registro invertito: " & $valBuffer)
12  if valBuffer == 1:
13    print("Hai premouto il tasto 1 (bit0)")
14  elif valBuffer == 128:
15    print("Hai premouto il tasto 2 (bit7)")
16  elif valBuffer == 129:
17    print("Hai premouto il tasto 1 e 2 assieme!")
18  sleep(500)
```

Anche in questo caso poche cose nuove, ma importanti; alla linea sei, crei un array di *byte* inizializzato a zero, in questo array verrà poi salvato il valore che andrai a leggere dal PCF ovvero ci sarà il byte che rappresenta il tuo ingresso in quel istante. La linea otto va ad interrogare il componente dove gli passa l'array che la procedura andrà a scrivere, e fin qui nulla di complicato o di diverso dal solito, la linea dieci invece va spiegata bene. Siccome nel PCF ci dovrebbe essere un pull-up interno, tutti gli ingressi (se non premi un pulsante) saranno alti, quindi il valore del byte sarà 255, mentre si vuole che sia a 0 (è più facile vedere i bit azionati cosi), per fare ciò basta fare un *not* logico:

```
lettura → byte = 255 → not 255 = 0        (equivale a 00000000b)   → nessun pulsante premuto.
```

Ora se vai a premere ad esempio il pulsante sul bit0, sul registro leggerai 254 (255-1) ma grazie al trucco di prima leggerai il valore che hai su quel bit:

```
lettura → byte = 254 → not 254 = 1        (equivale a 00000001b)   → pulsante 1 premuto.
lettura → byte = 127 → not 127 = 128      (equivale a 10000000b)   → pulsante 2 premuto.
lettura → byte = 126 → not 126 = 129      (equivale a 10000001b)   → pulsante 1 e 2 premuti.
```

Che è molto più semplice da decretare quale o quali pulsanti siano stati premuti, ma non è obbligatorio farlo, rimane una tua scelta!

4.7 – AD 5245 Potenziometro Digitale.

In elettronica si fa largo uso dei potenziometri, che sono sostanzialmente delle resistenze variabili, e possono regolare un guadagno, una tensione o altro, ma è un'operazione che viene fatta a mano, magari ascoltando il volume di uscita di uno stereo o leggendo su un multimetro una tensione, ma se volessimo cambiare questi parametri in base per esempio a dei valori provenienti da un sensore e quindi regolarlo al volo sarebbe un problema. Ecco che i potenziometri digitali possono aiutarti, perché leggendo un ingresso con un microcontrollore (vedi **sezione 3.5**), e facendo gli opportuni calcoli, si può variare il suo valore continuamente senza il tuo intervento. Di potenziometri digitali ce ne sono di moltissimi tipi e complessità. Quello da me scelto l'AD5245, che è relativamente semplice, offre 256 posizioni di regolazione, diversi valori Ohmnici da poter scegliere (5, 10, 50 100KΩ). Il datascheet è scritto molto bene e troverai tutte le informazioni sui dettagli usati per l'implementazione, ma forse la cosa più importante da sapere è come calcolare la resistenza minima che puoi avere con questo dispositivo:

Essendo un dispositivo digitale, la resistenza che avrai in uscita non può assumere valori continui, ma sarà discontinuo con salti ben determinati e calcolabili. Ecco come fare con un potenziometro da 5KΩ:

$$R0 = \frac{Rab}{256} + (2 * Rw) \rightarrow \frac{5000}{256} + (2 * 50) \approx 120\Omega$$

dove:

- Rab = resistenza del potenziometro (5KΩ in questo caso)
- 256 = posizioni massime per questo tipo di potenziometro.
- Rw = Resistenza del terminale che è di 50Ω (che son 2 **a** (o **b**) e **w** come vedrai in **Figura 25**).

Lo schema elettrico che seguirai per il prossimo esempio fa capo alla **Figura 25** qui sotto, dove porrai il tuo multimetro su Ohm, e collegherai i puntali sui terminali ***W*** ed ***A***, perché andrai a misurare la sua resistenza e la variazione di essa, poi spetta a te trovare un applicazione con cui sfruttare al meglio questo dispositivo.

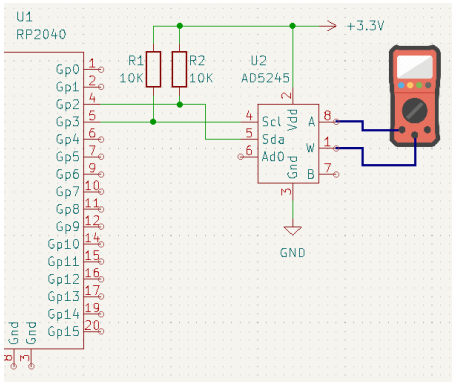


Figura 25

Nel programma che ti propongo di seguito, come puoi osservare in **Figura 25,** utilizzo solo l'uscita ***W*** e ***A***, e le istruzioni per essa, ma tu potresti usare ***W*** e ***B*** niente paura esistono anche quelle e son del tutto identiche a quelle che vedrai di seguito. Ecco il programma:

```
01 import picostdlib/[gpio, i2c, time, stdio]
02 from strutils import split                    → importa la procedura split dal modulo strutils.
03 import picousb                                → importa il modulo picousb.
04 import ad5245                                 → importa il modulo ad5245 per gestire il potenziometro.

05 stdioInitAll()

06 setupI2c(blokk=i2c1, psda= .Gpio, pscl=3.Gpio, freq=50_000, true)
07 let digitPot = newAd5245(ic2= i2c1, address=0x2C, resValue=5000)→ crea l'oggetto digiPot (gestione pot.).
08 let usb = PicoUsb()                                              → crea l'oggetto usb (gestione usb).
09 sleep(2000)

10 var
11    usbVal: string
12    splitList: seq[string]

13 while  true:
14   if usb.isReady == true:
15     usbVal = usb.readLine()                  → leggei dati sulla porta usb.
16     splitList = split(usbVal, '#')           → spezza la stringa dove trova il carattere '#'.
17     case splitList[0]                        → valuta il primo elemento della sequenza...
18     of "setvalue":                           → se = setvalue entra qui...
19       digitPot.setValue(uint8(usb.toInt(splitList[1])))   → setta il potenz. con valori tra 0=min , e 255=max.
20     of "setreswa":                           → se = setreswa entra qui...
21       digitPot.setResWA(usb.toInt(splitList[1]))          → setta il potenz. Con il valore resistivo indicato.
22       print("Il valore resistivo e': " & $digitPot.getResWA())   → ritorna il valore resistivo presente.
23       print("Il valore settato e': " & $digitPot.getValue())     → ritorna il valore impostato 0..255).
```

Dalla linea uno alla linea quattro ci sono le solite importazione dei moduli che ti serviranno per il programma, cosi dalla linea sei alla nove ci son solo le inizializzazioni che ormai conosci molto bene. Qualcosa di più interessante inizia alla linea quattordici dove il programma attende dei dati dalla usb; se ce ne sono allora si attiva il protocollo di comunicazione, in questo caso sarà:

- "setvalue*#un numero tra 0 e 255*" → valori in "posizioni" del potenziometro.
- "setreswa*#un numero tra 0 e 5000*" → valore resistivo effettivo (calcolato).

Alla procedura setValue() passi un numero tra 0 e 255 che setta la "posizione" interna del potenziometro che però è un numero slegato dal valore reale di quel potenziometro in quella posizione. (ricordi il calcolo sopra che ti da l'idea del valore a partire da questo numero?) può essere utile in certe circostanze magari per affinare un valore ricavato con il prossimo metodo. La procedura setResWA() o la sua equivalente setResWB() passa il valore in Ohm che desideri impostare. Una volta impostato il valore Ohmnico e magari non si è del tutto sodisfatti, ecco che è essenziale sapere a che "numero" è stato impostato il potenziometro e questo lo puoi ottenere con getValue() che ritorna il solito valore compreso tra 0 e 255 che potrai finemente modificare con la procedura vista prima. Puoi sapere anche il valore resistivo che hai in uscita con getResWA() o getResWB() (ricorda che è calcolato non sempre è preciso). Nota: qui ho usato *usb.toInt()* anzichè *parseInt()* perché è più idoneo, infatti se usci CR/LF con il primo tutto funziona con il secondo no (funziona solo se imposti NONE) questo perche vengono introdotti caratteri che non può convertire.

Ci sono due funzioni distinte "***setResWa*** e ***setResWb***" e "***getResWa*** e ***getResWb***", perché se immagini un potenziometro, ad un certo valore (non centrale) il cursore sarà più spostato verso un estremo o l'altro, quindi i valori sono complementari e bisogna tenerne conto.

Nel prossimo esempio, vedrai come poter utilizzare questo componente come partitore di tensione, che ritengo ti possa tornare utile nei tuoi progetti.

ATTENZIONE!! La massima tensione che puoi applicare al potenziometro non deve mai superare la tensione di alimentazione dello stesso, **ed in ogni caso mai superare i 5.5V**, pena la distruzione del componente!!

Lo schema che adotterai è il seguente e come quello in **Figura 25**, avrà solo scopo illustrativo senza una reale applicazione:

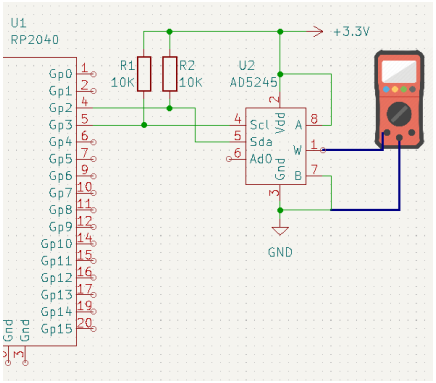


Figura 26

Ed ecco il codice, vedrai che impostare la tensione di uscita desiderata è molto semplice, anche in questo caso è calcolata e le tolleranza intere al componente si faranno sentire specie ai valori alti, ma con la possibilità di cambiare i valori anche di poco, puoi avvicinarti di molto alla tensione desiderata, ora manualmente, ma nessuno ti impedisce in futuro di creare un anello di retro reazione che lo faccia per tè.

```
01 import picostdlib/[gpio, i2c, time, stdio]
02 from strutils import split                    → importa la procedura split dal modulo strutils.
03 import picousb                                → importa il modulo picousb.
04 import ad5245                                 → importa il modulo ad5245 per gestire il potenziometro.

05 stdioInitAll()

06 setupI2c(blokk=i2c1, psda =2Gpio, pscl=3.Gpio, freq=50_000, true)   → crea l'oggetto digiPot (gestione pot.).
07 let digitPot = newAd5245(blokk=i2c1, address=0x2C, resValue=5000)   → crea l'oggetto usb (gestione usb).
08 let usb = PicoUsb()
09 sleep(2000)

10 var
11   usbVal: string
12   splitList: seq[string]

13 while  true:
14   if usb.isReady == true:
15     usbVal = usb.readLine()
16     splitList = split(usbVal, '#')
17     case splitList[0]
18     of "setvalue":
19       digitPot.setValue(uint8(usb.toInt(splitList[1])))
20    of "setvoltage":
21      print("Vset--> " & $splitList[1])
22      digitPot.setVoltage(usb.toFloat(splitList[1]), voltA=3.3)   → setta la tensione di uscita desiderata.
23      print("Il valore settato e': " & $digitPot.getValue())
24   else:
25      print("Errore!!")
```

Rispetto a quanto visto prima qui cambia ben poco, solo la procedura setVoltage(tensione, voltA=3.3) che andrà a calcolare il valore da scrivere sul registro per ottenere il valore da te desiderato. Ricordati di fornire sempre la tensione corretta della tensione applicata al potenziometro. In questo listato alla fine ho aggiunto *else* cosa che dovrebbe essere sempre fatta perché ti può salvare da tutti gli errori che potresti fare scrivendo sulla seriale.

4.8 – Eeprom.

Questo capitolo per ora non compare nel indice (capitolo fantasma), perché ad oggi sto ancora scrivendo e con non poche difficoltà, il modulo per utilizzare un eprom esterna (l'RP2040 non ne ha una interna), ma appena sarà disponibile sarà tutto aggiornato e descritto.
Spero che questo manuale ti sia servito per cominciare ad utilizzare ed apprezzare questo bel microcontrollore, che poi usato con Nim diventa davvero divertente e gradevole da programmare!

Descrizione dei Muduli.

stdio.nim

```
import system/ansi_c

{.push header: "<stdio.h>".}

proc stdioInitAll*{.importc: "stdio_init_all".}
        Initialize all of the present standard stdio types that are linked into the
        binary.
        Call this method once you have set up your clocks to enable the stdio
        support for UART, USB and semihosting based on the presence of the
        respective libraries in the binary.

proc stdioInitUsb*: bool{.importC: "stdio_usb_init".}
        Explicitly initialize USB stdio and add it to the current set of stdin
        drivers.

proc usbConnected*: bool {.importC: "stdio_usb_connected".}
        Returns true if USB uart is connected.


proc getCharWithTimeout*(timeout: uint32): char {.importC: "getchar_timeout_us".}
        Return a character from stdin if there is one available within a timeout.
        ===========  =======
        **timeout**   the timeout in microseconds, or 0 to not wait for a character if none available.
        ===========  =======
        **Returns:** the character from 0-255 or PICO_ERROR_TIMEOUT if timeout occurs
{.pop.}


proc blockUntilUsbConnected*() =
        Blocks until the usb is connected, useful if reliant on USB interface.
  while not usbConnected(): discard

proc print*(s: cstring) {.inline.} = cPrintf(s)
        write output directly to the console (or serial console)

proc print*(s: string) =
  print(cstring s)
  print(cstring "\n")

proc defaultTxWaitBlocking*(){.importC: "uart_default_tx_wait_blocking", header: "hardware/uart.h".}
        Wait for the default UART'S TX fifo to be drained.
```

gpio.nim

```
type
  GpioFunction* {.size: sizeof(uint32).} = enum
```
GPIO function definitions for use with function select.
Each GPIO can have one function selected at a time. Likewise,
each peripheral input (e.g. UART0 RX) should only be selected on one
GPIO at a time. If the same peripheral input is connected to multiple
GPIOs, the peripheral sees the logical OR of these GPIO inputs.
XIP, SPI, UART, I2C, PWM, SIO, PIO0, PIO1, GPCK, USB, NULL

```
  Gpio* = distinct range[0.uint32 .. 35.uint32]
```
Gpio pins available to the RP2040. Not all pins may be available on some
microcontroller boards.
```
  Value* = distinct uint32
```
Gpio function value. See datasheet.

```
proc `==`*(a, b: Value): bool {.borrow.}
proc `==`*(a, b: Gpio): bool {.borrow.}
proc `$`*(a: Gpio): string {.borrow.}


const
  High* = 1.Value
```
Alias that is useful for put() procedure, or reading inputs.
```
  Low* = 0.Value
```
Alias that is useful for put() procedure, or reading inputs.
```
  In* = false
```
Alias that is useful for setDir() procedure
```
  Out* = true
```
Alias that is useful for setDir() procedure
```
  DefaultLedPin* = 25.Gpio
```
constant variable for the on-board LED

```
{.push header: "hardware/gpio.h".}

proc setFunction*(gpio: Gpio, fun: GpioFunction){.importC: "gpio_set_function".}
```
Select GPIO function.
 ========== ======
gpio Gpio number
fn GpioFunction: XIP, SPI, UART, I2C, PWM, SIO, PIO0, PIO1, GPCK, USB, NULL

```
proc getFunction*(gpio: Gpio): GpioFunction {.importC: "gpio_get_function".}
```
Returns a Gpio function
========== ======
gpio Gpio number
========== ======
Returns: GpioFunction: XIP, SPI, UART, I2C, PWM, SIO, PIO0, PIO1, GPCK, USB, NULL

```
proc pullDown*(gpio: Gpio) {.importC: "gpio_pull_down".}
```
Set specified Gpio to be pulled down.
========== ======
gpio Gpio number

```
proc pullUp*(gpio: Gpio) {.importC: "gpio_pull_up".}
```
Set specified Gpio to be pulled up.
========== ======

```
        **gpio**    Gpio number
proc disablePulls*(gpio: Gpio) {.importC: "gpio_disable_pulls".}
        Disable pulls on specified Gpio.
        ========= ======
        **gpio**    Gpio number

proc setOutever*(gpio: Gpio, value: Value) {.importC: "gpio_set_outover".}
        Set Gpio output override.

proc setInover*(gpio: Gpio, value: Value) {.importC: "gpio_set_inover".}
        Set Gpio input override.

proc setOever*(gpio: Gpio, value: Value) {.importC: "gpio_set_oever".}

proc init*(gpio: Gpio){.importC: "gpio_init".}
        Initialise a Gpio for (enabled I/O and set func to Gpio_FUNC_SIO)
        Clear the output enable (i.e. set to input) Clear any output value.
        ========= ======
        **gpio**    Gpio number

proc initMask*(gpioMask: Gpio) {.importC: "gpio_init_mask".}
        Initialise multiple Gpios (enabled I/O and set func to Gpio_FUNC_SIO).
        Clear the output enable (i.e. set to input) Clear any output value.
        ================ ======
        **gpioMask**      Mask with 1 bit per Gpio number to initialize

proc get*(gpio: Gpio): Value {.importC: "gpio_get".}
        Get state of a single specified Gpio.
        **Returns:** Current state of the Gpio. Low (0.Value) or High (1.Value)

proc getAll*: uint32 {.importC: "gpio_get_all".}
        Get raw value of all Gpios.
        **Returns:** uint32 of raw Gpio values, as bits 0-29

proc put*(gpio: Gpio, value: Value){.importC: "gpio_put".}
        Drive a single Gpio high/low.
        ======================================= ======
        **gpio**                        Gpio number
        **High**, **Low**, **true**, **false**   High or true sets output, otherwise clears Gpio

proc setDir*(gpio: Gpio, isOut: bool) {.importC: "gpio_set_dir".}
        Set a single Gpio direction.
        ====================================== ======
        **gpio**                       Gpio number
        **In**, **Out**, **true**, **false**   true or Output for output; In or false for input

type
  IrqLevel* {.pure, importc: "enum gpio_irq_level", size: sizeof(cuint).} = enum
        GPIO Interrupt level definitions.
        An interrupt can be generated for every GPIO pin in 4 scenarios:
        =========== ======
        **low**      the GPIO pin is a logical 0
        **high**     the GPIO pin is a logical 1
        **fall**     the GPIO has transitioned from a logical 1 to a logical 0
        **rise**     the GPIO has transitioned from a logical 0 to a logical 1
         =========== ======
        The level interrupts are not latched. This means that if the pin is a
        logical 1 and the level high interrupt is active, it will become
        inactive as soon as the pin changes to a logical 0. The edge interrupts
```

```
        are stored in the INTR register and can be cleared by writing to the
        INTR register.
        low, high, fall, rise


IrqCallback* {.importC: "gpio_irq_callback_t".} = proc(gpio: Gpio, evt: set[IrqLevel]){.cDecl.}

proc enableIrq*(gpio: Gpio, events: set[IrqLevel], enabled: bool){.importC: "gpio_set_irq_enabled".}
        Enable or disable interrupts for specified GPIO.
        ==============  =======
        **gpio**        Gpio number to be monitored for event
        **event**       Which events will cause an interrupt
        **enabled**     Enable or disable flag for turning on and off the interupt


proc enableIrqWithCallback*(gpio: Gpio, events: set[IrqLevel], enabled: bool, event: IrqCallback){.
    importC: "gpio_set_irq_enabled_with_callback".}

{.pop.}

proc put*(gpio: Gpio, value: bool) =
  gpio.put(
    if value:
      High
    else:
      Low)

template setupGpio*(name: untyped, pin: Gpio, dir: bool) =
        Makes a `const 'name' = pin; init(name); name.setDir(dir)
  const name = pin
  init(name)
  setDir(name, dir)

proc init*( _ : typedesc[Gpio], pin: range[0 .. 35], dir = Out): Gpio =
        perform the typical assignment, init(), and setDir() steps all in one proc.
        **pin** : *int* (between 0 and 35) - the pin number corresponding the the Gpio pin
        **dir** : *bool* [optional, defaults to Out] - *Out* or *In*
  result = pin.Gpio
  result.init()
  result.setDir(dir)
```

time.nim

```
type
  AbsoluteTime* {.importc: "absolute_time_t", header: "pico/types.h".} = object
          the absolute time (now) of the hardware timer
    time*{.importC: "_private_us_since_boot".}: uint64


  DateTime* = object
    year*: 0u16..4095u16
    month*: 1u8..12u8
    day*: 1u8..31u8
    dotw*: 0u8..6u8
    hour*: 0u8..23u8
    min*, sec*: 0u8..59u8

{.push header:"pico/time.h".}
type
  AlarmCallback* {.importC: "alarm_callback_t".} = proc(id: uint32, data: pointer){.cDecl.}
          User alarm callback.
  AlarmId* = distinct uint32
          The identifier for an alarm.


proc sleep*(ms: uint32){.importc: "sleep_ms".}
          Wait for the given number of milliseconds before returning.
          Note: This procedure attempts to perform a lower power sleep (using WFE) as much as possible.
          ========= =======
          **ms**     the number of milliseconds to sleep

proc sleepMicroseconds*(us: uint64){.importc: "sleep_us".}
          Wait for the given number of microseconds before returning.
          Note: This procedure attempts to perform a lower power sleep (using WFE) as much as possible.
          ========= =======
          **us**     the number of microseconds to sleep

proc getTime*: AbsoluteTime {.importC:"get_absolute_time".}
          Return a representation of the current time.
          Returns an opaque high fidelity representation of the current time
          sampled during the call.
          **Returns:** the absolute time (now) of the hardware timer

proc addAlarm*(time: AbsoluteTime, callBack: AlarmCallback, data: pointer, fireIfPast: bool): AlarmId{.importc:
"add_alarm_at".}
          Add an alarm callback to be called at a specific time.
          Generally the callback is called as soon as possible after the time
          specified from an IRQ handler on the core of the default alarm pool
          (generally core 0). If the callback is in the past or happens before the
          alarm setup could be completed, then this method will optionally call the
          callback itself and then return a return code to indicate that the target
          time has passed.
          =============== ======
          **time**         the timestamp when (after which) the callback should fire
          **callBack**     the callback function
          **data**         user data to pass to the callback function
          **fireIfPast**    if true, this method will call the callback itself before returning 0 if the timestamp happens
          before or during this method call
          =========
          **Returns:**
```

```
=========  ======
**> 0**    the alarm id
**0**      the target timestamp was during or before this procedure call
**-1**     if there were no alarm slots available
=========  ======
```

proc addAlarm*(ms: uint32, callBack: AlarmCallback, data: pointer, fireIfPast: bool): AlarmId{.importc: "add_alarm_in_ms".}

Add an alarm callback to be called after a delay specified in milliseconds.
Generally the callback is called as soon as possible after the time
specified from an IRQ handler on the core of the default alarm pool
(generally core 0). If the callback is in the past or happens before the
alarm setup could be completed, then this method will optionally call the
callback itself and then return a return code to indicate that the target
time has passed.

```
==============  ======
**ms**          the delay (from now) in milliseconds when (after which) the callback should fire
**callBack**    the callback function
**data**        user data to pass to the callback function
**fireIfPast**  if true, this method will call the callback itself before returning 0 if the timestamp happens
before or during this method call
=========  ======
**> 0**    the alarm id
**0**      the target timestamp was during or before this procedure call
**-1**     if there were no alarm slots available
=========  ======
```

proc addAlarm*(us: uint64, callBack: AlarmCallback, data: pointer, fireIfPast: bool): AlarmId{.importc: "add_alarm_in_us".}

Add an alarm callback to be called at a specific time.
Generally the callback is called as soon as possible after the time
specified from an IRQ handler on the core of the default alarm pool
(generally core 0). If the callback is in the past or happens before the
alarm setup could be completed, then this method will optionally call the
callback itself and then return a return code to indicate that the target
time has passed.

```
===============  ======
**us**          the delay (from now) in microseconds when (after which) the callback should fire
**callBack**    the callback function
**data**        user data to pass to the callback function
**fireIfPast**  if true, this method will call the callback itself before returning 0 if the timestamp happens
before or during this method call
=========  ======
**> 0**    the alarm id
**0**      the target timestamp was during or before this procedure call
**-1**     if there were no alarm slots available
=========  ======
```

proc cancel*(alarm: AlarmId): bool {.importC: "cancel_alarm".}

Cancel an alarm from the default alarm pool.

```
=============  ======
**alarm**      the alarm id
=============  ======
**Returns:**   true if the alarm was cancelled, false if it didn't exist
```
{.pop.}

{.push header: "hardware/timer.h".}
proc timeUs32*(): uint32 {.importC: "time_us_32".}

Return time from the start of the microcontroller to microseconds (32-bit)

```
proc timeUs64*(): uint64 {.importC: "time_us_64".}
```

Return time from the start of the microcontroller to microseconds (64-bit)

```
{.pop.}
```

adc.nim

```
import gpio

type
  AdcInput* {.pure, size: sizeof(cuint).} = enum
        Aliases for selectInput() procedure
        ADC input. 0...3 are GPIOs 26...29 respectively. Input 4 is the onboard temperature sensor.
  Adc26 = 0, Adc27 = 1, Adc28 = 2, Adc29 = 3, AdcTemp = 4

const ThreePointThreeConv* = 3.3f / (1 shl 12)
        Useful for reading inputs from a 3.3v source

proc adcInit*{.importC:"adc_init".}
        Initialise the ADC hardware

proc adcRead*: uint16 {.importC:"adc_read".}
        Performs a single ADC conversion, waits for the result, and then returns it.
        **Returns:** Result of the conversion.

proc initAdc*(gpio: Gpio){.importC: "adc_gpio_init".}
        Prepare a GPIO for use with ADC, by disabling all digital functions.
        ========= =======
        **gpio**    The GPIO number to use. Allowable GPIO numbers are 26 to 29 inclusive.
        ========= =======

proc selectInput*(input: AdcInput){.importc: "adc_select_input".}
        ADC input select.
        ========= =======
        **input**   ADC input. 0...3 are GPIOs 26...29 respectively. Input 4 is the onboard temperature sensor.
        ========= =======

proc enableTempSensor*(enable: bool){.importc: "adc_set_temp_sensor_enabled".}
        Enable the onboard temperature sensor.
        =========== =======
        **enable**    Set true to power on the onboard temperature sensor, false to power off.
        =========== =======
```

i2c.nim

```
type
  I2Hw* {.importC: "i2c_hw_t", header: "hardware/structs/i2c.h".} = object
    con*, tar*, sar*: uint32
    pad0: uint32
    dataCmd*, ssSclHcnt*, ssSclLcnt*, fsSclHcnt*, fsSclLcnt*: uint32
    pad1: array[2, uint32]
    intrStat*, intrMask*, rawIntrStat*, rxTl*, txTl*, clrIntr*: uint32
    clrRxUnder*, clrRxOver*, clTxOver*, clRdReq*, clrTxAbrt*, clrRxDone*: uint32
    clrActivity*, clrStopDet*, clrStartDet*, clrGenCall*: uint32
    enable*, status*: uint32
    txFlr*, rxFlr*: uint32
    sdaHold*: uint32
    txAbortSource*: uint32
    slvDataNackOnly*: uint32
    dmaCr*, dmaTdlr*, dmaRdlr*: uint32
    sdaSetup*: uint32
    ackGeneralCall*, enableStatus*, fkSpkLen*: uint32
    pad2: uint32
    clrRestartDet*: uint32
{.push header: "hardware/i2c.h".}
type
  I2cInst* {.importc: "i2c_inst_t".} = object
    hw*: ptr I2Hw
    restartOnNext*: bool

  I2cAddress* = distinct range[0'u8 .. 127'u8]

var i2c0* {.importC: "i2c0_inst".}: I2cInst
var i2c1* {.importC: "i2c1_inst".}: I2cInst

proc init*(i2c: var I2cInst, baudrate: cuint) {.importC: "i2c_init".}
proc deinit*(i2c: var I2cInst) {.importc: "i2c_deinit".}
proc setBaudrate*(i2c: var I2cInst, baudRate: cuint): cuint {.importc: "i2c_set_baudrate".}
proc setSlaveMode*(i2c: var I2cInst, slave: bool, address: uint8) {.importc: "i2c_set_slave_mode".}
proc writeBlocking*(i2c: var I2cInst, address: uint8, data: pointer, len: csize_t, noStop: bool){.importc:
"i2c_write_blocking".}
proc readBlocking*(i2c: var I2cInst, address: uint8, dest: pointer, size: csize_t, noStop: bool): cint {.importC:
"i2c_read_blocking".}

{.pop.}

import picostdlib/[gpio]
template setupI2c*(blokk: I2cInst, psda, pscl: Gpio, freq: int, pull = true) =
        sugar setup for i2c:
        blokk= block i2c0 / i2c1 (see pinout)
        sda/pscl = the pins tou want use (ex: 2.Gpio, 3.Gpio) I do not recommend the use of 0.Gpio, 1.Gpio
        freq = is the working frequency of the i2c device (see device manual; ex: 100000)
        pull = use or not to use pullup (default = ture)



proc writeBlocking*(i2c: var I2cInst, address: I2cAddress, data: openArray[uint8], noStop: bool = false) =
        Write bytes to I2C bus.
        If `noStop` is `true`, master retains control of the bus at the end of
        the transfer.
  writeBlocking(i2c, address.uint8, data[0].unsafeAddr, data.len.uint, noStop)
```

```
proc readBlocking*(i2c: var I2cInst, address: I2cAddress, numBytes: Natural, noStop: bool = false): seq[uint8] =
        Read `numBytes` bytes from I2C bus and return a seq containing the bytes
        that were read. In case of error return a 0-length seq. If `noStop` is
        `true`, master retains control of the bus at the end of the transfer.
  result.setLen(numBytes)
  let n = readBlocking(i2c, address.uint8, result[0].addr, numBytes.uint, noStop)
  result.setLen(max(0, n))

proc readBlocking*[N: Natural](i2c: var I2cInst, address: I2cAddress, dest: var array[N, uint8], noStop: bool = false):
int =
        Fill the array `dest` with bytes read from I2C bus. Return the number of
        bytes that were read successfully. Negative values are error codes (refer
        to Pico SDK documentation). In case of error return a 0-length seq. If
        `noStop` is `true`, master retains control of the bus at the end of the
        transfer.
  result = readBlocking(i2c, address.uint8, dest[0].addr, dest.len.uint, noStop)
```

clock.nim

```
type
  ClockIndex* {.pure, importC: "enum clock_index".} = enum
    ## hardware clock identifiers
    gpOut0
    gpOut1
    gpOut2
    gpOut3
    ciRef
    sys
    peri
    usb
    adc
    rtc
{.pop.}

type
  Fc0SrcValue* {.pure, size: sizeof(cuint).} = enum
    null
    pllSysClksrcPrimary
    pllUsbClksrcPrimary
    roscClksrc
    roscClksrcPh
    xoscClksrc
    clksrcGpin0
    clksrcGpin1
    clkRef
    clkSys
    clkPeri
    clkUsb
    clkAdc
    clkRtc
  AuxSrcValue* {.pure, size: sizeof(cuint).} = enum
    clksrcPllSys
    clksrcPllUsb
    roscClksrc
    xoscClksrc
    clkSrcGpin0
    clkSrcGpin1
  ResusCallback* = proc(){.noConv.}

const
  Fc0SrcOffset* = 0x00000094u32
  Fc0SrcBits* = 0x000000ffu32
  Fc0SrcReset* = 0x00000000u32
  Fc0SrcMsb* = 7u32
  Fc0SrcLsb* = 0u32
  Fc0SrcAccess* = "RW"
  Fc0SrcValueNull* = 0x00u32

  CtrlAuxsrcReset* = 0u32
  CtrlAuxsrcBits* = 0xe0u32
  CtrlAuxsrcMsb* = 7u32
  CtrlAuxsrcLsb* = 0u32
  CtrlAuxsrcAccess* = "RW"
  CtrlAuxsrcValueClksrcPllSys* = 0u32

  CtrlSrcReset* = 0u32
```

```
  CtrlSrcBits* = 1u32
  CtrlSrcMsb* = 0u32
  CtrlSrcLsb* = 0u32
  CtrlSrcAccess* = "RW"
  CtrlSrcValueClkRef* = 0u32
  CtrlSrcValueClksrcClkSysAux* = 1u32

  Khz* = 1000
  Mhz* = 1000000



{.push header: "hardware/clocks.h".}
proc getHz*(clkInd: ClockIndex): uint32 {.importc: "clock_get_hz".}
        Get the current frequency of the specified clock.
        Returns:** Clock frequency in Hz

proc setReportedHz*(clkInd: ClockIndex, hz: cuint) {.importC: "clock_set_reported_hz".}
         Set the "current frequency" of the clock as reported by clock_get_hz
        without actually changing the clock.
        **hz**        frequency in hz to set the new reporting value of the clock

proc frequencyCountKhz*(src: cuint): uint32 {.importC: "frequency_count_khz".}
        Measure a clocks frequency using the Frequency counter.
        Uses the inbuilt frequency counter to measure the specified clocks
        frequency. Currently, this function is accurate to +-1KHz. See the
        datasheet for more details.

proc frequencyCountKhz*(src: Fc0SrcValue): uint32 {.importC: "frequency_count_khz".}
        Measure a clocks frequency using the Frequency counter.
        Uses the inbuilt frequency counter to measure the specified clocks
        frequency. Currently, this function is accurate to +-1KHz. See the
        datasheet for more details.

proc enableResus*(callBack: ResusCallback) {.importc: "clocks_enable_resus".}
        Enable the resus function. Restarts clk_sys if it is accidentally stopped.
        The resuscitate function will restart the system clock if it falls below a
        ertain speed (or stops). This could happen if the clock source the system
        clock is running from stops. For example if a PLL is stopped.
        ============  =======
        **callBack**   a function pointer provided by the user to call if a resus event happens.

proc clockConfigure*(clkInd: ClockIndex, src, auxSrc, srcFreq, freq: uint32): bool {.importC: "clock_configure".}
        Configure the specified clock.
        See the tables in the adc module description for details on the possible
        values for clock sources.
```

pwm.nim

```
import gpio
export gpio

type
  ClockDivideMode* {.pure, importc: "enum pwm_clkdiv_mode".} = enum
        PWM Divider mode settings.
        **Modes:**
        ================  ======
        **freeRunning**    Free-running counting at rate dictated by fractional divider.
        **high**           Fractional divider is gated by the PWM B pin.
        **rising**         Fractional divider advances with each rising edge of the PWM B pin.
        **falling**        Fractional divider advances with each falling edge of the PWM B pin.
        ================  ======

    freeRunning, high, rising, falling
  PwmConfig* {.byref, importC: "pwm_config".} = object
        Configuration object for PWM  tasks
    csr, divide, top: uint32

  PwmChannel* {.pure, importC: "enum pwm_chan", size: sizeof(cuint).} = enum
        Alias for channel parameter in the setChanLevel() procedure
    A, B

  SliceNum* = distinct cuint


proc toSliceNum*(gpio: Gpio): SliceNum {.importC: "pwm_gpio_to_slice_num".}
        Determine the PWM slice that is attached to the specified GPIO.
        =========  ======
        **gpio**     Gpio number
        =========  ======
        **Returns** The PWM slice number that controls the specified GPIO.

proc setWrap*(sliceNum: SliceNum, wrap: uint16) {.importC: "pwm_set_wrap".}
        Set the highest value the counter will reach before returning to 0. Also known as TOP.
        The counter wrap value is double-buffered in hardware. This means that,
        when the PWM is running, a write to the counter wrap value does not take
        effect until after the next time the PWM slice wraps (or, in phase-correct
        mode, the next time the slice reaches 0). If the PWM is not running, the
        write is latched in immediately.
        ==============  ======
        **sliceNum**     PWM slice number
        **wrap**         Value to set wrap to
        ==============  ======

proc setChanLevel*(sliceNum: SliceNum, chan: PwmChannel, level: uint16){.importC: "pwm_set_chan_level".}
        Set the value of the PWM counter compare value, for either channel A or channel B
        The counter compare register is double-buffered in hardware. This means
        that, when the PWM is running, a write to the counter compare values does
        not take effect until the next time the PWM slice wraps (or, in
        phase-correct mode, the next time the slice reaches 0). If the PWM is not
        running, the write is latched in immediately.
        ==============  ======
        **sliceNum**     PWM slice number
        **chan**         Which channel to update. 0 for A, 1 for B.
        **level**        new level for the selected output
        ==============  ======
```

proc setBothLevels*(sliceNum: SliceNum, levelA, levelB: uint16){.importC: "pwm_set_both_levels".}

Set the value of the PWM counter compare values, A and B
The counter compare register is double-buffered in hardware. This means
that, when the PWM is running, a write to the counter compare values does
not take effect until the next time the PWM slice wraps (or, in
phase-correct mode, the next time the slice reaches 0). If the PWM is not
running, the write is latched in immediately.

sliceNum	PWM slice number
levelA	Value to set compare A to. When the counter reaches this value the A output is deasserted
levelB	Value to set compare B to. When the counter reaches this value the B output is deasserted

proc setLevel*(gpio: Gpio, level: uint16){.importC: "pwm_set_gpio_level".}

Helper procedure to set the PWM level for the slice and channel associated with a GPIO.
Look up the correct slice (0 to 7) and channel (A or B) for a given GPIO,
and update the corresponding counter-compare field.
This PWM slice should already have been configured and set running. Also
be careful of multiple GPIOs mapping to the same slice and channel
(if GPIOs have a difference of 16).
The counter compare register is double-buffered in hardware. This means
that, when the PWM is running, a write to the counter compare values does
not take effect until the next time the PWM slice wraps (or, in
phase-correct mode, the next time the slice reaches 0). If the PWM is not
running, the write is latched in immediately.

gpio	Gpio to set level of
level	PWM level for this GPIO

proc getCounter*(sliceNum: SliceNum): uint16 {.importC: "pwm_get_counter".}

Get current value of PWM counter

sliceNum	PWM slice number

Returns: Current value of PWM counter

proc setCounter*(sliceNum: SliceNum, level: uint16) {.importC: "pwm_set_counter".}

Set the value of the PWM counter

sliceNum	PWM slice number
level	Value to set the PWM counter to

proc advanceCount*(sliceNum: SliceNum){.importC: "pwm_advance_count".}

Advance the phase of a running the counter by 1 count.
This procedure will return once the increment is complete.

sliceNum	PWM slice number

proc retardCount*(sliceNum: SliceNum){.importC: "pwm_retard_count".}

Retard the phase of a running counter by 1 count
This procedure will return once the retardation is complete.

sliceNum	PWM slice number

proc setClockDivide*(sliceNum: SliceNum, integer, divide: byte){.importC: "pwm_set_clkdiv_int_frac".}

Set the clock divider. Counter increment will be on sysclock divided by
this value, taking in to account the gating.
============= =======
sliceNum PWM slice number
integer 8 bit integer part of the clock divider
divide 4 bit fractional part of the clock divider
============= =======

proc setClockDivide*(sliceNum: SliceNum, divider: float){.importC: "pwm_set_clkdiv".}
Set the clock divider. Counter increment will be on sysclock divided by
this value, taking in to account the gating.
============= =======
sliceNum PWM slice number
divider Floating point clock divider, 1.float <= value < 256.float
============= =======

proc setClockDivide*(pwmConfig: PwmConfig, divider: cfloat(1) .. cfloat(256)){.importC:"pwm_config_set_clkdiv".}
Set clock divider in a PWM configuration.
If the divide mode is free-running, the PWM counter runs at clk_sys / div.
Otherwise, the divider reduces the rate of events seen on the B pin input
(level or edge) before passing them on to the PWM counter.
============= =======
pwmConfig PWM configuration object to modify
divider Floating point clock divider, 1.float <= value < 256.float
============= =======

proc setClockDivide*(pwmConfig: PwmConfig, divider: cuint){.importC: "pwm_config_set_clkdiv_int".}
Set PWM clock divider in a PWM configuration.
If the divide mode is free-running, the PWM counter runs at clk_sys / div.
Otherwise, the divider reduces the rate of events seen on the B pin input
(level or edge) before passing them on to the PWM counter.
============= =======
pwmConfig PWM configuration object to modify
divider Integer value to reduce counting rate by. Must be greater than or equal to 1.
============= =======

proc setWrap*(pwmConfig: PwmConfig, wrap: uint16){.importc: "pwm_config_set_wrap".}
Set PWM counter wrap value in a PWM configuration.
Set the highest value the counter will reach before returning to 0. Also
known as TOP.
============= =======
pwmConfig PWM configuration object to modify
wrap Value to set wrap to
============= =======

proc init*(sliceNum: SliceNum, pwmConfig: PwmConfig, start: bool){.importC: "pwm_init".}
Initialise a PWM with settings from a configuration object.
Use the getDefaultConfig() procedure to initialise a config objecture,
make changes as needed using the pwm_config_* procedures, then call this
procedure to set up the PWM.
============= =======
sliceNum PWM slice number
pwmConfig PWM configuration object to modify
start If true the PWM will be started running once configured. If false you will need to start manually
using pwm_set_enabled() or pwm_set_mask_enabled()
============= =======

proc getDefaultConfig*: PwmConfig {.importc: "pwm_get_default_config".}
Get a set of default values for PWM configuration.
PWM config is free running at system clock speed, no phase correction,
wrapping at 0xffff, with standard polarities for channels A and B.
Returns: Set of default values.

proc setOutputPolarity*(sliceNum: SliceNum, a, b: bool){.importC: "pwm_set_output_polarity".}
Set PWM output polarity.
============= ======
sliceNum PWM slice number
a true to invert output A
b true to invert output B
============= ======

proc setClockDivideMode*(sliceNum: SliceNum, mode: ClockDivideMode){.importc: "pwm_set_clkdiv_mode".}
Set PWM divider mode.
============= ======
sliceNum PWM slice number
mode Required divider mode
============= ======

proc setClockDivideMode*(pwmConfig: PwmConfig, mode: ClockDivideMode){.importc: "pwm_config_set_clkdiv_mode".}
Set PWM counting mode in a PWM configuration.
Configure which event gates the operation of the fractional divider. The
default is always-on (free-running PWM). Can also be configured to count on
high level, rising edge or falling edge of the B pin input.
============= ======
pwmConfig PWM configuration object to modify
mode Required divider mode
============= ======

proc setPhaseCorrect*(sliceNum: SliceNum, phaseCorrect: bool){.importC: "pwm_set_phase_correct".}
Set PWM phase correct on/off.
Setting phase control to true means that instead of wrapping back to zero
when the wrap point is reached, the PWM starts counting back down. The
output frequency is halved when phase-correct mode is enabled.

proc setPhaseCorrect*(pwmcfg: PwmConfig, phaseCorrect: bool){.importC: "pwm_config_set_phase_correct".}
Set phase correction in a PWM configuration.
Setting phase control to true means that instead of wrapping back to zero
when the wrap point is reached, the PWM starts counting back down. The
output frequency is halved when phase-correct mode is enabled.

proc setEnabled*(sliceNum: SliceNum, enabled: bool){.importC: "pwm_set_enabled".}
Enable/Disable PWM.
============= ======
sliceNum PWM slice number
enabled true to enable the specified PWM, false to disable
============= ======

proc setEnabled*(sliceNum: SliceNum, mask: set[0..7]){.importc: "pwm_set_mask_enabled".}
Enable/Disable multiple PWM slices simultaneously.
============= ======
sliceNum PWM slice number
mask Bitmap of PWMs to enable/disable. Bits 0 to 7 enable slices 0-7 respectively
============= ======

proc setIrqEnabled*(sliceNum: SliceNum, enabled: bool){.importc: "pwm_set_irq_enabled".}

Used to enable a single PWM instance interrupt
============= =======
sliceNum PWM slice number
enabled true to enable, false to disable
============= =======

proc setIrqEnabled*(sliceMask: set[0..31], enabled: bool){.importC: "pwm_set_irq_mask_enabled".}
Enable multiple PWM instance interrupts at once.
============= =======
sliceMask Bitmask of all the blocks to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.
enabled true to enable, false to disable
============= =======

proc clear*(sliceNim: SliceNum){.importC: "pwm_clear_irq".}
Clear single PWM channel interrupt.
============= =======
sliceNum PWM slice number
============= =======

proc getStatus*: uint32 {.importc: "pwm_get_irq_status_mask".}
Get PWM interrupt status, raw.
Returns: The PWM channel that controls the specified Gpio

proc forceIrq*(sliceNum: SliceNum){.importc: "pwm_force_irq".}
Force PWM interrupt.
============= =======
sliceNum PWM slice number
============= =======

pll.nim

```
type
  Pll*{.importC: "pll_hw_t", header: "hardware/structs/pll.h".} = object
  cs, pwr, fbdiv_int, prim: uint32

{.push header: "hardware/pll.h".}
proc init*(pll: Pll, refDiv, vcoFreq, postDiv1, postDiv2: cuint){.importC: "pll_init".}
proc deinit*(pll: Pll){.importC: "pll_deinit".}
const
  PllSys* {.importC: "pll_sys".} = Pll()
  PllUsb* {.importC: "pll_usb".} = Pll()
{.pop.}
```

multicore.nim

```
{.push header: "pico/multicore.h".}
type
  ThreadFunc* = proc() {.cDecl.}
proc multicoreLaunchCore1*(p: ThreadFunc) {.importC: "multicore_launch_core1".}

proc multicoreResetCore1*() {.importC: "multicore_reset_core1".}
proc multicoreFifoRvalid*(): bool {.importC: "multicore_fifo_rvalid".}
proc multicoreFifoWready*(): bool {.importC: "multicore_fifo_wready".}
proc multicoreFifoPushBlocking*(data: uint32) {.importC: "multicore_fifo_push_blocking".}
proc multicoreFifoPopBlocking*(): uint32 {.importC: "multicore_fifo_pop_blocking".}
proc multicoreFifoDrain*() {.importC: "multicore_fifo_drain".}
proc multicoreFifoClearIrq*() {.importC: "multicore_fifo_clear_irq".}
proc multicoreFifoGetStatus*(): uint32 {.importC: "multicore_fifo_get_status".}
{.pop.}
```

irq.nim

```
const
  TimerIrq0* = 0.cuint
  TimerIrq1* = 1.cuint
  TimerIrq2* = 2.cuint
  TimerIrq3* = 3.cuint
  PwmIrqWrap* = 4.cuint
          more required for full compatibility

type
  IrqHandler* {.importC: "irq_handler_t".} = proc(){.cDecl.}

proc setExclusiveHandler*(num: cuint, handler: IrqHandler){.importC: "irq_set_exclusive_handler".}
proc setEnabled*(num: cuint, enabled: bool) {.importc: "irq_set_enabled".}
```

watchdog.nim

```
proc watchdogEnable*(delayms: uint32, pauseondebug: bool) {.importC: "watchdog_enable".}

proc watchdogUpdate*() {.importC: "watchdog_update".}

proc watchdogCausedReboot*(): bool {.importC: "watchdog_caused_reboot".}

proc watchdogReboot*(pc: uint32, sp: uint32, delayms: uint32) {.importC: "watchdog_reboot".}
```

picousb.nim

```
proc isReady*(self: PicoUsb): bool = #procedure for checking the buffer status.
  ## Checking the buffer status (if there are characters).
  ##
  ## ==========
  ## **Returns:**
  ## true = there are characters, false = there are no characters.

proc readLine*(self: PicoUsb; time: uint32 = 100): string = #proc for read the string in usb
  ## Read the string in the usb buffer.
  ##
  ## **Parameters**
  ## time = time expected before the timeout (milliseconds).
  ## ==========
  ##
  ## **Returns**
  ## string

proc toInt*(self: PicoUsb; usbString: string): int = #proc for the conversion from string to INT.
  ## Convert string in to int.
  ##
  ## **Parameters**
  ## usbString = string read in usb.
  ## =========
  ##
  ## **Results:**
  ## int

proc toFloat*(self:PicoUsb; usbString: string; nround=2): float = #proc for tthe conversion from string to INT.
  ## Convert string in to float.
  ##
  ## **Parameters**
  ## usbString = string read in usb.
  ## =========
  ##
  ## **Results:**
  ## float
```

random.nim

```
proc randomize*() = #Randomizes the variable with the bootstrap time
  ## Randomizes the variable with the bootstrap time.

proc random*(precision = 10): float = #make a random float number between 0..1 (set precision)
  ## Generates a random (float) number between 0 and 1
  ##
  ## **Parameters:**
  ## - precision = indicates how many numbers to use (10 = 0.1234567891)

proc randomInt*(min = 0, max = 100.0): int = #make a random integer numer between "min" and "max"
  ## Generates an entire random number, between a maximum and a minimum.
  ##
  ## **Parameters:**
  ## - min = minimum integer of generated.
  ## - max = maximum integer of generated.


proc randomChar*(): char = #make a random char (a..z, A..Z)
  ## Generates a random character (a..z, A..Z).
```

disolay1602.nim

```
proc newDisplay*(i2c: I2cInst, lcdAdd:uint8, numLines:uint8, numColum:uint8): Lcd
proc clearLine*(self: Lcd)
proc clear*(self: Lcd)
proc moveTo*(self: Lcd, columx, rowx: uint8)
proc putChar*(self: Lcd, charx: char)
proc putString*(self: Lcd, strg: string)
proc centerString*(self: Lcd, strg: string)
proc shiftChar*(self: Lcd, charx: char, speed: uint16=400, dir=true)
proc shiftString*(self: Lcd, strg: string, speed: uint16=400, dir=true, cross=false, effect: uint8=0)
proc customChar*[T](self: Lcd, location: uint8, charmap: array[0..7, T])
proc displayOn*(self: Lcd)
proc displayOff*(self: Lcd)
proc backLightOn*(self: Lcd)
proc backLightOff*(self: Lcd)
proc hideCursor*(self: Lcd)
```

ssd1306.nim

```
proc newSsd1306I2C*(i2c: I2CInst; lcdAdd: uint8; width, height: int; externalVcc=false): SSD1306I2C
proc powerOff*(self: SSD1306I2C)
proc powerOn*(self: SSD1306I2C)
proc contrast*(self: SSD1306I2C, contrast: uint8)
proc invert*(self: SSD1306I2C, invert: uint8)
proc show*(self: SSD1306I2C)
```

framebuffer.nim

```
proc clear*(self: Framebuffer; color=0)
proc rect*(self: Framebuffer; x ,y, width, height, color: int, fill=false)
proc line*(self: Framebuffer, xStr, yStr, xEnd, yEnd, color: int)
proc pixel(self: Framebuffer; x, y: int; color=none(int)): Option[uint8]
proc hline*(self: Framebuffer; x, y, width, color: int)
proc vline*(self: Framebuffer; x, y, height, color: int)
proc circle*(self: Framebuffer; xCenter, yCenter, radius, color: int)
proc text*(self: Framebuffer; text: string; x, y, color: int; charType="std"; size=1, direct=true)
```

pcf8574.nim

```
proc writeByte*(self: Pcf8574, data: byte ) = #proc to write the byte
  ## Write a whole byte (8 outings) on the PCF8574 register.
  ##
  ## **Parameters:**
  ## - *data:* it is the Byte you want to write on the register.

proc readByte*(self: Pcf8574, data: var array[1,uint8]) =
  ## Read the entry byte present that instant.
  ##
  ## **Parameters:**
  ## - *data:* byte array where the value received by the PFC8574 is stored.

proc writeBit*(self: Pcf8574, pin: uint8, value: bool) =
  ## Write a single byte on the exit (P0, P1 ..) desired.
  ##
  ## **Parameters:**
  ## - *pin:* it is the pin on which you want to write the new value (p0..p7).
  ## - *value:* *on* = set exit high, *low* = set exit low.

proc readBit*(self: Pcf8574, pin: uint8): bool =
  ## Reads the value of a single bit in the PFC8574 register.
  ##
  ## **Parameters:**
  ## - *pin:* read the single pin indicated (p0..p7).
  ## **Return:**
  ## bool

proc setLow*(self: Pcf8574) = #set buffer 0x00 all 0
  ## It puts all the low outputs.

proc setHigh*(self: Pcf8574) = #set buffer 0xff all 1
  ## It puts all the high outputs.

proc newExpander*(blokk: I2cInst, expAdd: uint8, buffer: uint8 = 0x00): Pcf8574 =
  result = Pcf8574(blockk: blokk, expAdd: expAdd, buffer: buffer)
```

ad5245.nim

```
proc newAd5245*(i2c: I2cInst, address: uint8, resValue: int): Ad5245 = #initialize the type Ad5245
  ## Ad5245 initialize
  ##
  ## **Parameters:**
  ## - *i2c* = name of the block where the display connected (i2c0 or i2c1).
  ## - *address* = hardware address of the display.
  ## - *resValue* = value in Ohm of the potentiometer.

proc setInstruction*(self: Ad5245, instruction: uint8) = #proc from write instructions see manale Ad5245

proc setValue*(self: Ad5245, data: var uint8) = #set the value 0 = RESmin, 255 = RESmax
  ## Set the numerical value (0-255) of the potentiometer.
  ## **Parameters:**
  ## - *data* = set the numerical value that the potentiometer can take (it is not the Hominic value)




proc setResWA*(self: Ad5245, ohmValue: var int) = #proc to set the value of the nominal resistance between B and W
  ## Set the resistance (in Ohm) between the W terminal and A.
  ##
  ## **Parameters:**
  ## - *homValue* = value in Ohm of the resistance to be set between W and A.

proc setResWB*(self: Ad5245, ohmValue: var int) = #proc to set the value of the nominal resistance between A and W
  ## Set the resistance (in Ohm) between the W terminal and B.
  ##
  ## **Parameters:**
  ## - *homValue* = value in Ohm of the resistance to be set between W and B.

proc setVoltage*(self: Ad5245, vOut: var float, voltA: float, voltB: float = 0.0) =
  ## Set the desidered voltage on the pin W (works as a voltage divider).
  ##
  ## **Parameters:**
  ## - *vOut* = desired output voltage.
  ## - *voltA* = input voltage on pin A (<= Power supply voltage of the AD5245).
  ## - *voltB* = voltage present on PIN B (usually = 0V).

proc getResWA*(self: Ad5245): int = #proc for read value write in ad5245 in Ohm
  ## Return the value in Ohm set between W and A.
  ##
  ## **Return:**
  ## - *int (Hom).*

proc getResWB*(self: Ad5245): int = #proc for read value write in ad5245 in Ohm
  ## Return the value in Ohm set between W and B.
  ##
  ## **Return:**
  ## - *int (Ohm).*

proc getValue*(self: Ad5245): uint8 = #return the value write in register.
  ## return the numerical value writen in the  register.
  ##
  ## **Return:**
  ## - *uint8 (0-255).*
```

Indice alfabetico