Progettazione di una libreria per la comunicazione con il display SSD1306 attraverso il protocollo I2C

Camilletti Samuele

18/07/2023

Contents

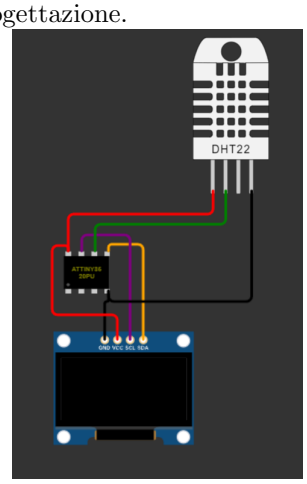
1 Definizione dell'obiettivo

L'obiettivo è di realizzare un software per stabilire una **comunicazione** tra il microcontrollore **Arduino** e il display **SSD1306**, attraverso l'utilizzo del protocollo **I2C**.
Il protocollo I2C (chiamato inizialmente TwoWire nei processori AVR in quanto coperto da brevetto) venne sviluppato dalla Philips nel 1982 ed è una sistema di comunicazione seriale a due fili (bus), tra due o più dispositivi, dove sono presenti almeno un Master e uno Slave.
Il ruolo di **Master** è quello di stabilire una connessione verso lo **Slave**, esplicitando la sua intenzione di leggere o scrivere dati. Lo Slave rappresenta la parte passiva e in base alla richiesta del Master fornisce o acquisisce dati.
Nel circuito proposto il Master è rappresentato dal microcontrollore Arduino e lo Slave dal monitor SSD1306. Affrontando quindi il problema attraverso un approccio iterativo incrementale, l'obiettivo iniziale sarà quello di progettare delle primitive di comunicazione, partendo dall'analisi del circuito proposto da esempio, per poi passare all'analisi dei datasheets dei dispositivi.
In secondo luogo, sarà poi analizzata la parte relativa ai flussi di comandi e di dati necessari al display per trasmettere correttamente le informazioni in output.

2 Analisi

2.1 Studio preliminare del circuito di esempio

Nel circuito proposto (https://wokwi.com/projects/292900020514980360), sono interconnessi tra loro un microcontrollore **AtTiny85**, un sensore DHT e il monitor SSD1306. Nella funzione setup viene mostrata una schermata di avvio sullo schermo e successivamente ad intervalli regolari di 1000 ms (1 secondo) viene stampata sul display SSD1306 la temperatura e l'umidità ottenuti in output dal sensore DHT22.
La gestione del sensore DHT22 non è materia del progetto pertanto viene tralasciata. La comunicazione con il display è gestita da una libreria che gestisce il flusso logico di dati verso il display (Tiny4kOLED) e una libreria più "vicina" all'hardware che implementa le primitive di gestione dell'interfaccia a due fili (TWI.h). Pertanto questa visione a livelli permette già di capire come dovrà essere affrontata la progettazione.



2.2 Analisi del protocollo I2C

2.2.1 Scelta dell'architettura di riferimento

Il protocollo I2C viene implementato seguendo le stesse regole su tutte le architetture, la differenza risiede su come viene implementata internamente la gestione dell'I/O. Pertanto in linea con quanto stabilito dal docente di riferimento, e avendo trattato durante le lezioni del corso soltanto l'architettura **AtMega 328P**, sono stati presi in analisi i datasheets dell'ATmega 328P (https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P˙Datasheet.pdf).

2.2.2 Studio del segnale e della Two Wire Interface

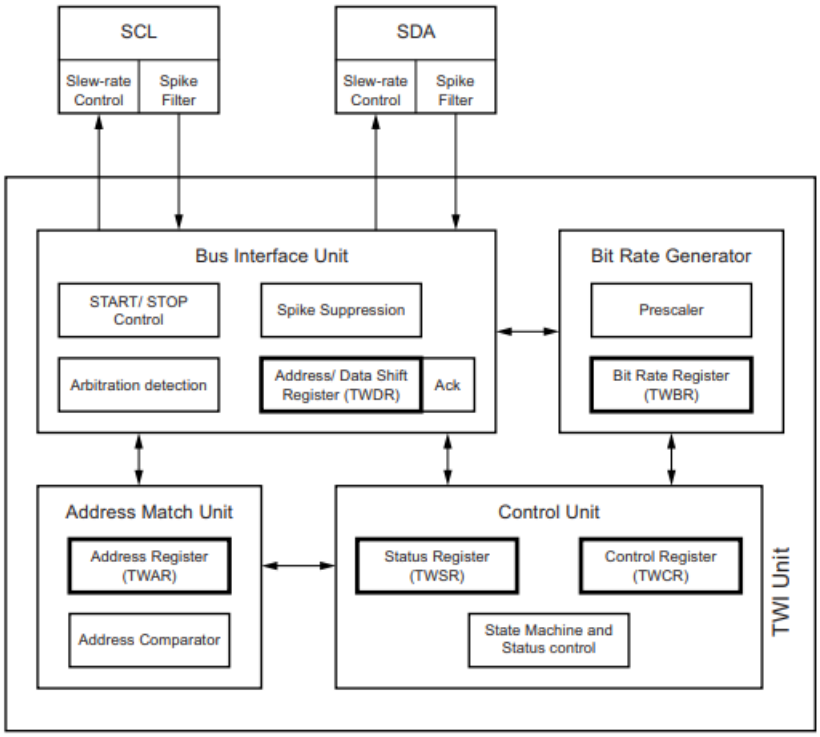
Il protocollo **I2C**, come già descritto, è implementato su un bus costituito da due linee bidirezionali seriali dette SCL e SDA:

- **SCL**: E' utilizzata per la temporizzazione, ovvero per la trasmissione del clock.
- **SDA**: E' utilizzata per il trasferimento dati.

I segnali che transitano sulle linee possono assumere i valori '1' o '0' e le tensioni che li rappresentano sono quelle d'alimentazione e di massa rispettivamente.
Le due linee sono collegate all'alimentazione attraverso una resistenza di **pull-up**. In questo modo le due linee mantengono un valore "alto-debole", facilmente modificabile da un dispositivo.
Un segnale **basso** è generato nel momento in cui viene posto sulla linea uno 0, mentre un segnale **alto** verrà generato all'uscita quando il device "abiliterà" il buffer tri-state, permettendo alla resistenza pull-up di generare un valore alto all'uscita.
Ogni device collegato ai bus è dotato di un indirizzo univoco dai 7 ai 10 bit che permette l'identificazione nel caso di dispositivi multipli collegati al bus.
Tutti i pacchetti data/address che vengono trasferiti sulle linee sono composti da 8 bit + 1 bit di ACK.
In seguito verrà descritta nel dettaglio la struttura di un pacchetto.

Nell'architettura in analisi, l'interfaccia a due fili è gestita da un **modulo di I/O** chiamato **TWI**, cioé Two Wire Interface, ed è una macchina a stati finiti che implementa una gestione della TWI **Interrupt-based**.
L'ATMega328p utilizza i Pinout della Porta C, in particolare il PC4 e il PC5 per implementare rispettivamente il segnale SDA e SCL. Ed è sempre la TWI ad occuparsi di configurare i Pinout in input o in output. La TWI ha internamente a disposizione uno **TWSR** (Status Register) e un **TWCR** (Control Register). Un clock interno programmabile a livello software attraverso il **TWBR** (Bit Rate Register) un prescaler, e un **TWDR** (Data Register) per l'invio/ricezione di dati.
Il modulo TWI dispone anche di un'unita per l'arbitraggio, la quale non verrà analizzata nello sviluppo di questa libreria in quanto prenderemo in analisi il collegamento di una sola periferica di output.

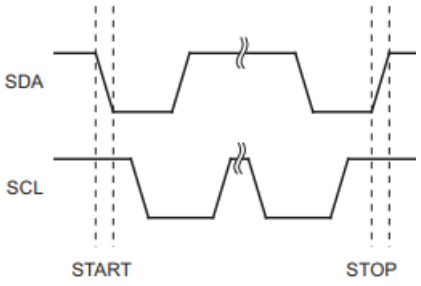
Il **TWI Control Register** è il registro più importante del modulo in quanto ci permette di dare istruzioni sulla prossima azione da compiere, solitamente in base allo **stato** dell'unità, descritto nel **TWI Status Register**.
Il bit TWI Interrupt Flag del registro TWCR, detto **TWINT**, è uno dei bit più importanti, in quanto **scandisce** il flusso operazionale dell'unità. Il bit viene posto ad 1 lato hardware nell'istante in cui l'unità ha eseguito un compito e si aspetta che lato software la TWI venga istruita sulla prossima azione da eseguire nel seguente ciclo di clock. Finché il bit rimarrà settato la linea SCL rimarrà **bassa**.
Quando un interrupt TWI viene "triggerato" la CPU passa al vettore interrupt ed esegue la routine 0x0030. Pertanto a questo punto lato software sarà necessario **configurare** correttamente i registri TWCR o TWDR in base alla prossima azione da compiere e poi **resettare** il bit TWINT ponendolo ad 1 lato software. La TWI riprenderà la sua esecuzione normale, così come la CPU, fino a che non riceverà un nuovo segnale di Interrupt dalla TWI.
I comandi che la TWI può compiere sono la trasmissione di un segnale di **START** (bit TWSTA in TWCR) o un segnale di **STOP** (bit TWSTO in TWCR) o la trasmissione di dati inserendo il byte nel registro TWDR.
Vedremo in seguito come esegue a livello hardware queste operazioni.



Il periodo del clock che verrà trasmesso sulla linea SCL è controllato dal registro TWBR e dai bit del prescaler nel TWSR. E' specificato che la frequenza della CPU dello Slave deve essere almeno 16 volte superiore a quella della linea SCL, condizione che verrà poi gestita nel dettaglio in fase di progettazione.

La comunicazione è costituita da 4 fasi principali ed è ovviamente gestita dal Master, pertanto verrà descritto il funzionamento del protocollo, analizzando nello specifico i passi da compiere nel modulo TWI; passi che verranno poi tradotti in fase di progettazione nel Flowchart e StateChart corrispondenti: Viene considerato che entrambe le due linee abbiano un valore di partenza ALTO:

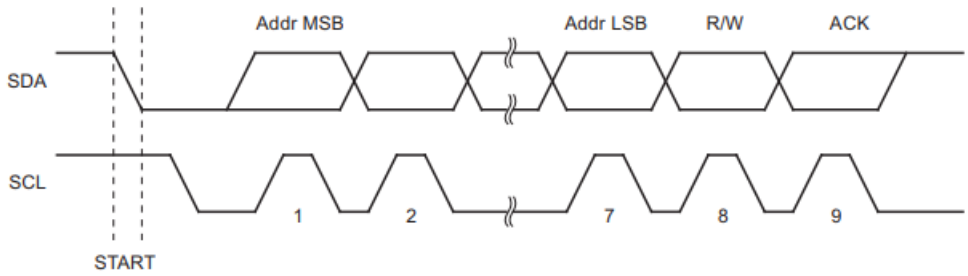
1. **START**: Segnale di avvio della comunicazione Un segnale di START viene generato ponendo la linea SDA bassa mentre la linea SCL è alta. Nel TWCR deve essere posto il bit TWSTA ad 1 e settando ad 1 il bit TWINT il quale corrisponde a resettare la flag dell'interrupt e cioè far "procedere" il modulo. Quando viene trasmesso correttamente il segnale di START, la macchina cambia stato (TWI˙START) e viene generato un Interrupt.



ADRESSING: Invio dell'address corrispondente al device selezionato (slave) Dopo aver inviato un segnale di start allora è possibile stabilire una connessione con il device interessato, inviando un byte composto dal suo indirizzo (7 bit) + 1 bit di R/W per esplicitare l'intenzione di voler leggere o inviare byte.
Se l'ottavo bit è 0, allora il PIN viene lasciato in scrittura mentre se è 1 il pin SDA viene impostato in lettura; questa operazione è eseguita automaticamente lato hardware dalla TWI. Se il device slave è disponibile allora pone la linea SDA a bassa al nono ciclo di clock per segnalare di aver ricevuto correttamente i dati. Questo segnale è detto di ACK.
Pertanto se il bit di ACK è 0 la comunicazione è stato stabilita altrimenti se SDA rimane alto la comunicazione è fallita.

In base al valore dell'ACK la TWI cambia stato (TWI˙SLAVE˙ACK o TWI˙SLAVE˙NACK).

Nel TWI dopo aver settato il registro di I/O TWDR con il byte corrispondente all'indirizzo + R/W (dove il MSB è trasmesso per primo) è necessario nuovamente settare il bit TWINT e pulire il bit corrispondente a TWSTA settato precedentemente. Questa operazione genererà il clock, ad ogni ciclo di clock verrà trasmesso un bit, o meglio il TWDR shifta di un bit a partire dal MSB, il quale che verrà settato sulla linea SDA quando il clock sarà BASSO mentre verrà letto dallo slave quando il clock è ALTO. Al nono ciclo di clock con il quale viene ricevuto/inviato l'ACK indica la conclusione della trasmissione e pertanto viene generato un segnale di Interrupt. Il segnale di Interrupt è solitamente generato da un Overflow in un timer, infatti il timer interno conta 16 clock edges alla diciassetesima che corrisponde al completamento dell'ACK viene generato un interrupt.

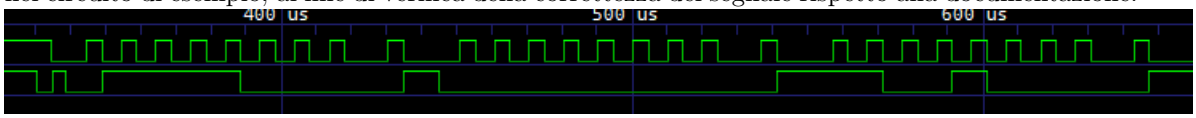


DATA STREAM: Invio flusso di dati Stabilita la comunicazione con lo slave, il funzionamento dell'invio dei dati è analogo a quello dell'indirizzo. Il registro TWDR viene utilizzato anche per i dati e la comunicazione inizia quando TWINT viene settato. Al termine dell'invio di un byte viene generato un Interrupt e la TWI cambia stato in base al valore del bit di ACK (TWI˙DATA˙ACK o TWI˙DATA˙NACK). A questo punto a livello software è possibile scegliere se inviare un altro byte o trasmettere un segnale di STOP.

STOP: Segnale di stop alla comunicazione Un segnale di STOP viene generato ponendo la line SDA alta mentre la linea SCL è alta.
Nel TWI deve essere posto il bit TWSTO ad 1 nel TWCR e settando ad 1 il bit TWINT il quale corrisponde a resettare la flag dell'interrupt e cioè far "procedere" il modulo.

2.2.3 Studio del segnale e della Two Wire Interface

Si è fatto utilizzo dello strumento Wokwi logic per lo studio del segnale trasm esso sui pinout SDA e SCL
nel circuito di esempio; al fine di verifica della correttezza del segnale rispetto alla documentazione.



Nell'immagine osserviamo una sequenza di start, l'invio dello SlaveAddress e l'invio di due byte seguiti dai rispettivi ACK.

2.3 Studio libreria standard avr/io.h e twi.h

Per la progettazione di questa libreria sono state prese in considerazione le seguenti librerie standard:

- avr/io.h Sapendo che l'architettura ATMega328p mappa i registri I/O in memoria e pertanto vengono trattati come indirizzi di memoria, definisce delle costanti autodocumentanti per i registri della TWI e inoltre definisce i pinout della scheda Arduino (https://www.nongnu.org/avr-libc/user-manual/group¨avr¨io.html).
- util/twi.h Definisce delle MACRO autoesplicative per gli stati che può assumere il TWSR dopo un segnale di Interrupt (https://www.nongnu.org/avr-libc/user-manual/group¨util¨twi.html).

- avr/interrupt.h Definisce le costanti autodocumentanti per il vettore di interrupt (https://www.nongnu.org/avr-libc/user-manual/group¨avr¨interrupts.html).

3 Metologie di risoluzione

Si è applicata una metodologie di sviluppo del software a **spirale incrementale**. Ciò significa che è stata prima analizzata tutta la documentazione relativa al protocollo two-wire e il relativo modulo I/O, per poi spostare l'attenzione sul dispositivo slave, cioé l'SSD1306. Pertanto nella progettazione che segue verrà realizzata una libreria che implementa le primitive di comunicazione I2C per una MCU ATMega 168p.

3.1 Progettazione I2C Interrupt based

Nella progettazione di un protocollo di comunicazione è necessario definire le fasi della trasmissione dei dati: che come in tutti i protocolli informatici si suddividono in Setup, Connessione, Trasmissione, Acknoledgement e Chiusura.

3.1.1 Gestione della memoria necessaria

Per far sì che il protocollo di comunicazione segua il corretto flusso di esecuzione descritto sopra è stata introdotta una variabile di **stato** che come descritto nello statechart in seguito, va ad ampliare l'insieme di stati già utilizzati dalla TW Interface. (L'insieme di stati è illustrato nello statechart).
Si introduce inoltre un **array** di interi da **8 byte** di lunghezza 32 per implementare il buffer dati, allocato come memoria statica in SRAM. Come descritto in fase di analisi, infatti, la TWI impone che ad ogni interrupt venga preparata per il successivo ciclo di clock: pertanto ad ogni interruzione dovrà essere pronto il prossimo byte di dati da inviare. Il primo byte dell'array costituirà l'indirizzo dello Slave e i successivi 31 byte saranno disponibili all'applicazione.
Inoltre saranno necessari ulteriori due interi: la lunghezza effettiva del buffer, e l'indice del buffer ovvero il numero di byte inviati.

3.1.2 Avvio connessione e Setup

Si sviluppa pertanto una funzione di avvio della connessione (twStart) che acquisisce come parametri **l'indirizzo dello slave e il bit di lettura o scrittura**.
A questo punto, prima ancora di "occupare" la TWI, se la variabile di stato ci indica che è il modulo TWI non è ancora mai stato utilizzato (stato TWI˙OFF) allora sarà necessario **inizializzarlo**. Ipotizzando di non conoscere lo stato delle linee SCL e SDA, è necessario progettare una funzione di Begin() che inizializza le linee SCL e SDA ad HIGH (come definito nella documentazione), i registri del modulo TWI e abiliti l'interrupt globale.
I registri devono essere così configurati:
Nel registro **TWCR** tutti i bit vengono ripuliti, ad eccezione del bit TWEN il quale va posto ad 1 per indicare l'applicazione del protocollo I2C.
Il registro **TWDR** può essere resettato.
E' necessario definire il bitrate: il display supporta due modalità slow mode (100 kHz) e fast mode (400 kHz) di frequenza per la linea del clock. Conoscendo la frequenza della CPU (8 Mhz) settiamo nel registro TWBR il valore esadecimale 0x20 per dividerla di 32 e quindi usare una frequenza di 100kHz sulla linea SCL.

$$SCLfrequency = \frac{CPU\ Clockfrequency}{16 + 2(TWBR) - 4^{TWPS}}$$

Questa funzione sancisce la transizione allo stato TWI˙READY.

Ipotizzando, invece, che la macchina sia ancora **"in uso"**, la variabile di stato deve dapprima assicurarci che solo 1 comunicazione sia attualmente in corso sull'interfaccia Two-Wire, in quanto è necessario che la libreria segua il principio di mutua esclusione.
Pertanto inizierà una fase di busy wait, in cui la CPU dovrà attendere che la TWI finisca di eseguire la comunicazione precedente (long wait I/O).

Pertanto appena la macchina sarà nello stato TWI_READY la funzione chiamante prenderà il controllo della TWI facendola passare ad uno stato **TWI_STARTED**, inizializzerà il buffer in memoria e memorizzerà il proprio indirizzo slave nel primo byte del buffer.

La funzione di avvio non è una vera funzione di avvio del protocollo I2C, in questa progettazione la connessione a livello hardware verrà avviata solo nel momento in cui il buffer dati è pronto. E' da ricordare che in questa versione la libreria supporta solamente la scrittura, la lettura potrà essere implementata in futuri aggiornamenti.

3.1.3 Preparazione buffer dati

La funzione twWrite permette al software chiamante di riempire il **buffer** dati che sarà poi inviato allo slave. Acquisisce un intero di 1 byte in input e restituisce un 0 se viene allocato con successo, mentre 1 se la TWI non è stata avviata (stato TWI_STARTED) o se il buffer dati ha superato la dimensione massima di 32 byte.

3.1.4 "Chiusura connessione" e Trasferimento dati

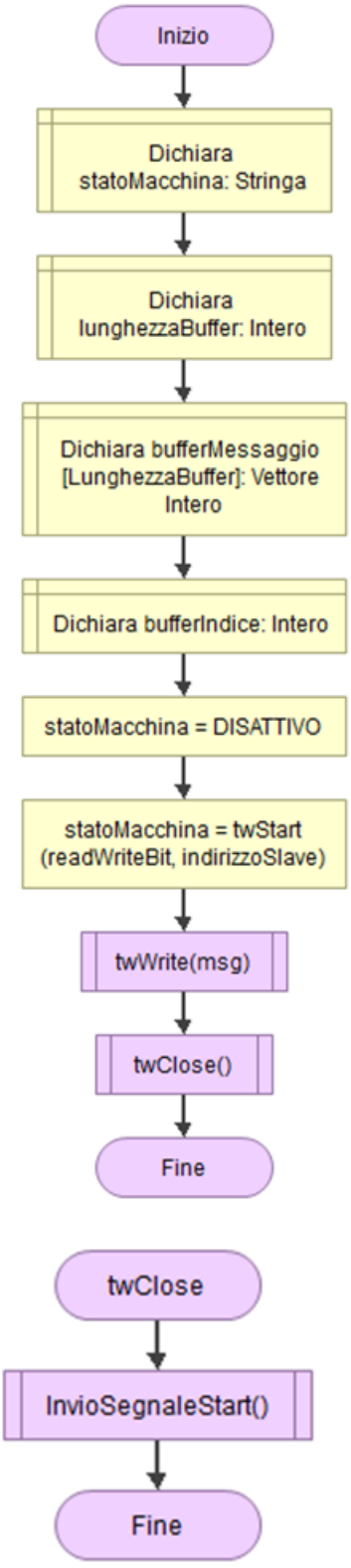
La funzione **twClose** è puramente simbolica, in quanto la CPU si limita ad inviare il segnale di **START** per poi tornare al suo normale flusso computazionale (rete di governo), e ogni qualvolta che la TWI cambierà stato sarà l'ISR (Interrupt Service Routine) ad eseguire la corretta sotto-procedura.
La start condition è caratterizzata dal porre la linea SDA BASSA mentre la linea SCL è ALTA. Questa operazione è eseguita direttamente dall'hardware nel momento in cui nel registro TWCR il bit TWSTA è ad 1 e il bit TWINT viene resettata la flag.

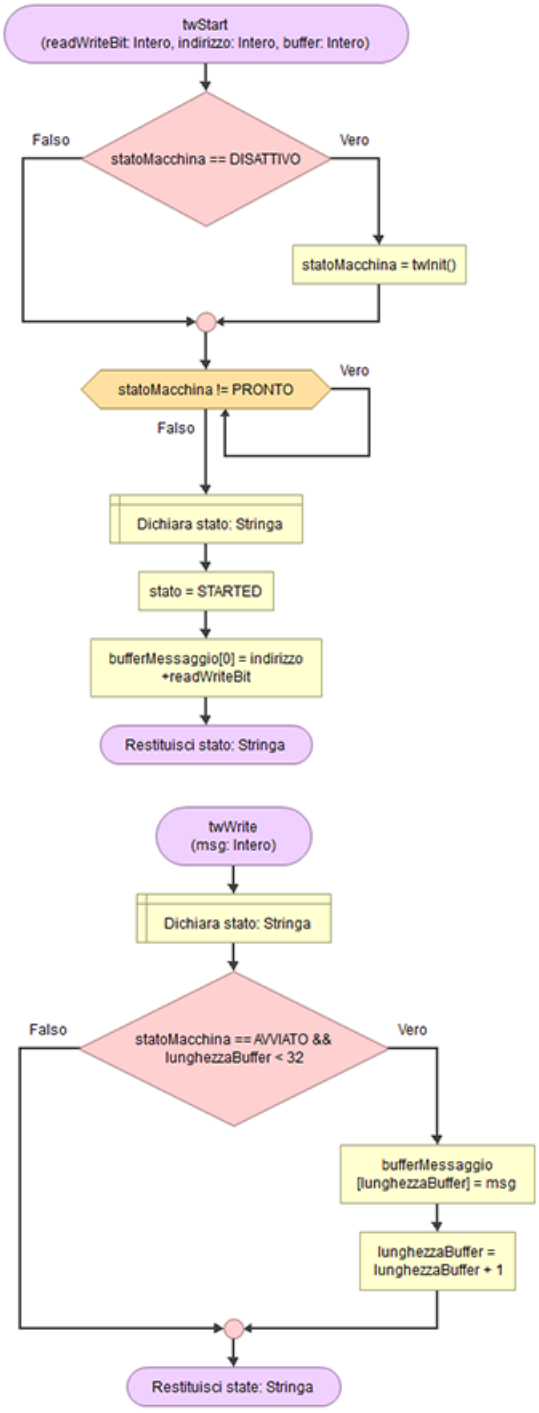
3.1.5 Progettazione della routine di Interrupt TWI

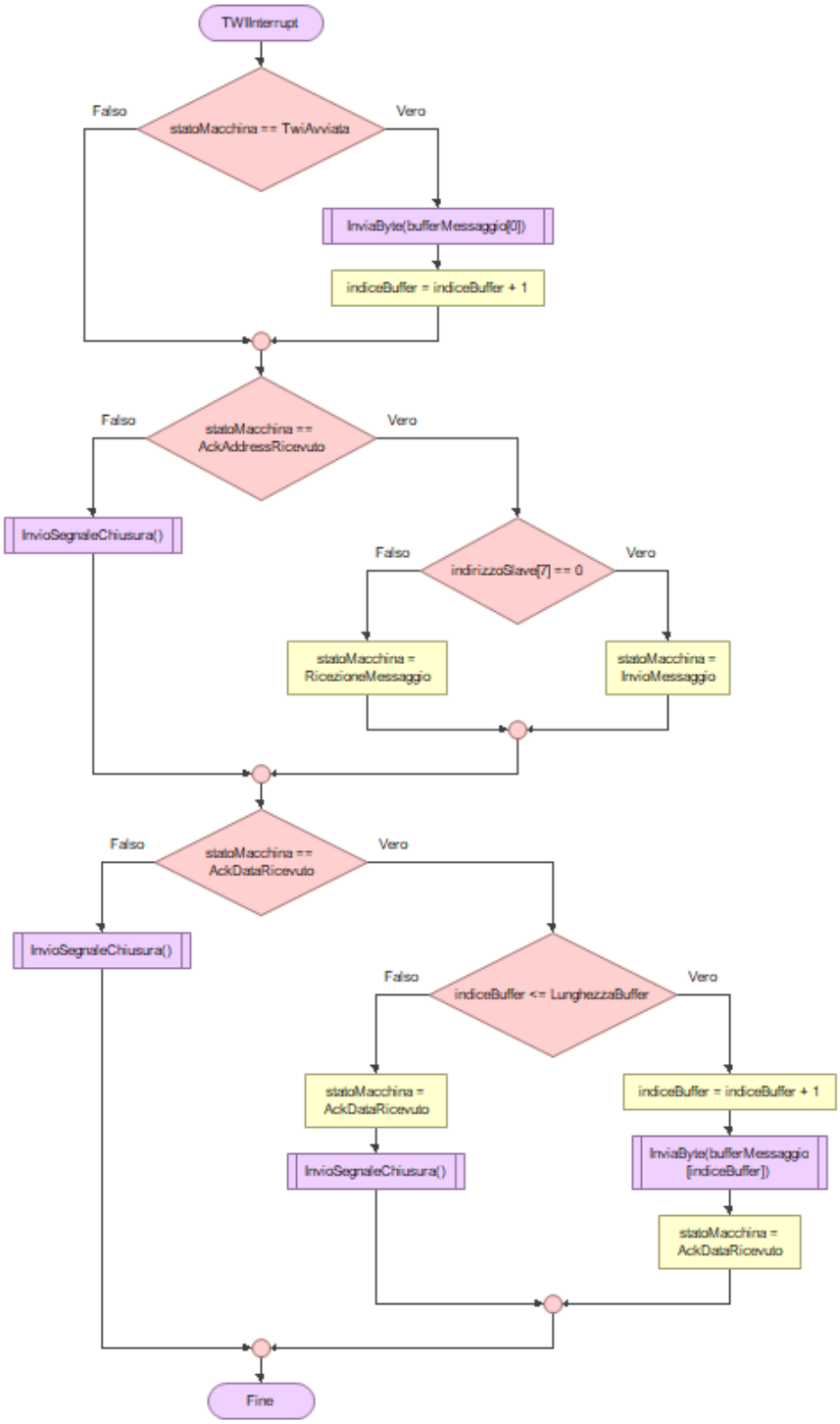
La gestione è ora lasciata alla Interrupt Service Routine.
Ogni qualvolta viene generato un interrupt TWI la routine è definita in modo tale da verificare lo stato del registro TWSR del modulo TWI:

- Se è stato propagato correttamente un segnale di START: allora inserisce nel registro TWDR l'indirizzo Slave + R/W bit e lo invia settando ad 1 il bit TWINT che avvia il trasferimento.
- Se l'ACK è stato ricevuto allora il TWSR acqusirirà un valore corrispondente allo stato TWI_SLAVE_ACK: dove pertanto si procede ad inserire nel TWDR il prossimo byte del buffer ed incrementare l'indice di 1.
- Se l'ACK data viene ricevuto il TWSR acquisirà il valore TWI_DATA_ACK: a questo punto si continua ad incrementare l'indice dell'array buffer fino a che non si raggiunge la dimensione allocata.
- Se il buffer è terminato allora si procede ad inviare il segnale di STOP settando il bit TWSTO nel registro TWCR e settando il TWINT. Lo stato della macchina torna quello iniziale, avendo concluso la sua operazione, ovvero TWI_READY.
- Se invece viene ricevuto un NOT ACK ovvero il TWSR assume i valori TWI_SLAVE_NACK o TWI_DATA_NACK: allora si procede a chiudere la connessione.

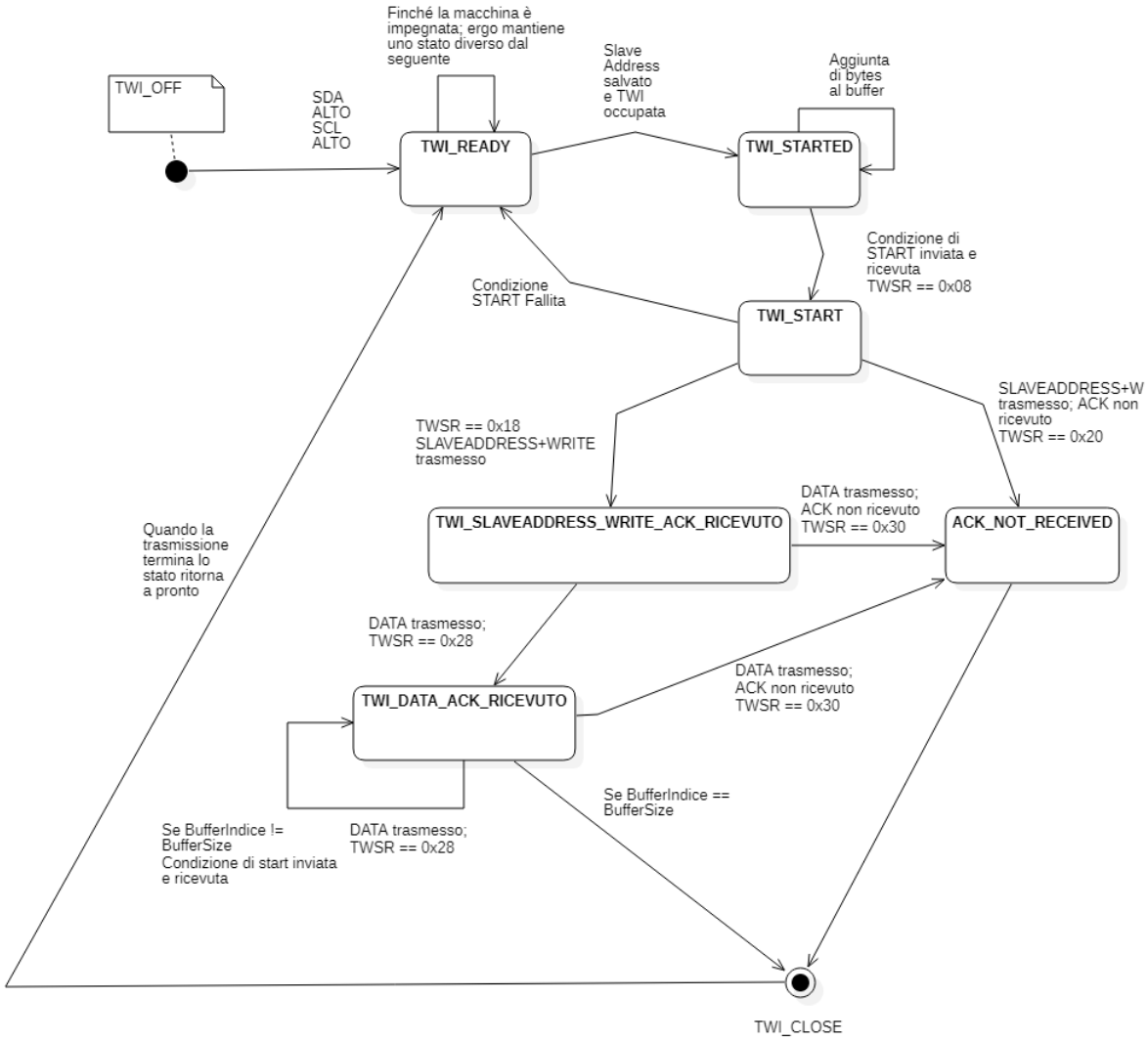
3.2 Flowchart






TWIInterrupt
statoMacchina == TwiAvviata
Falso
Vero
InviaByte(bufferMessaggio[0])
indiceBuffer = indiceBuffer + 1
statoMacchina == AckAddressRicevuto
Falso
Vero
InvioSegnaleChiusura()
indirizzoSlave[7] == 0
Falso
Vero
statoMacchina = RicezioneMessaggio
statoMacchina = InvioMessaggio
statoMacchina == AckDataRicevuto
Falso
Vero
InvioSegnaleChiusura()
indiceBuffer <= LunghezzaBuffer
Falso
Vero
statoMacchina = AckDataRicevuto
InvioSegnaleChiusura()
indiceBuffer = indiceBuffer + 1
InviaByte(bufferMessaggio [indiceBuffer])
statoMacchina = AckDataRicevuto
Fine

3.3 Statechart



3.4 Codice

Il codice è contenuto nei rispettivi file:
TWireMega.h - header
TWireMega.cpp - codice

3.5 Problematiche e ottimizzazioni

Eventuali problematiche sono emerse per via di long wait I/O. Un buffer di 32 byte impedisce ad un software l'invio di sequenze di byte contigue maggiori di 31 byte e pertanto implica la necessità di dover stabilire più connessioni in determinati casi. Un buffer dati più grande renderebbe l'esecuzione più rapida ma allocherebbe maggior spazio in SRAM.

4 Analisi SSD1306

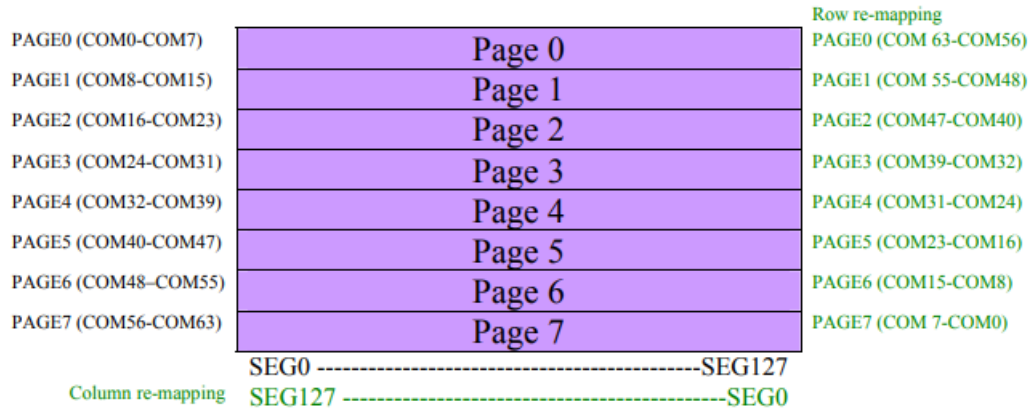
4.1 Definizione dell'obiettivo

L'obiettivo di questa seconda fase dello sviluppo è di sviluppare delle funzioni che, utilizzando le primitive di comunicazione **I2C** definite in precedenza, stabilisca che **tipo** di informazioni e **quando** inviarle al dispositivo Slave SSD1306 affinché mostri le corretti informazioni sul display. Ovviamente, quando si parla di libreria grafica esistono infinite opzioni: dalla più semplice ovvero l'accensione di un pixel, passando per i caratteri e infine per figure etc.. In linea con il circuito di riferimento analizzato all'inizio di questo documento quest'analisi si limiterà ad analizzare come accendere dei pixel per poi passare alla visualizzazione di stringhe (ed "temperatura: x" e "umidità: y").

4.2 Studio SSD1306

L'SSD1306 è un **display OLED** dotato di un controllore interno e un display grafico a diodi luminosi disposti attraverso una matrice "a punti". E' disponibile in due versioni: 128x64 o 128x32. In questa analisi verrà trattata esclusivamente la versione **128x64**, la quale è dotata di 128 pixel orizzontali per 64 verticali.
L'altezza di 64 pixel è divisa in 8 **"pagine"** (da Page 0 a Page 7) alte ciascuna 8 pixel. Mentre l'asse orizzontale è suddiviso in 128 **segmenti** (SEG0 a SEG127). Semplificando, per ogni pagina orizzontale, sono presenti 128 segmenti verticali da 8 pixel, che tradotto dal punto di vista software sono 128 Bytes, dove ogni Byte è un segmento verticale. Considerando che le pagine sono per l'appunto 8 la dimensione totale del display è di 1024 Bytes (8192 bits), che difatti, corrispondono alla memoria interna del dispositivo, detta **GDDRAM**. Nella memoria in questione un bit a 0 corrisponde ad un pixel spento mentre un bit ad 1 corrisponde a un pixel acceso.

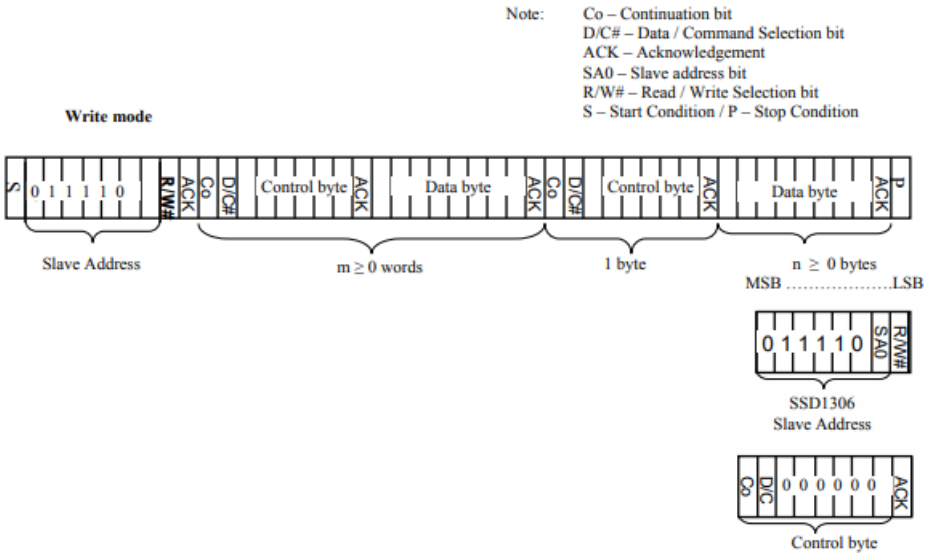
Figure 8-13 : GDDRAM pages structure of SSD1306



Il display in questione dispone inoltre di un clock interno e una interfaccia per diverse MCU tra cui l'8080, la 6800 e in particolare, quella che verrà utilizzata in questa trattazione l'interfacciamento I2C. Nell'interfacciamento di tipo Seriale I2C è necessario seguire questo protocollo: Il primo byte è costituito dallo **Slave Address** (7 bit) che nel caso di questo dispositivo corrisponde alla cifra in esadecimale 3C (b0111100) mentre l'ottavo bit specifica la volontà di leggere o scrivere. Se vogliamo leggere sarà lo Slave ad inviarci le informazioni contenute nella GDDRAM. Se invece vogliamo scrivere (0 all'ottavo bit) il byte successivo da inviare deve specificare che tipo di informazioni stiamo inviando allo Slave. I primi due bit più significativi del byte rappresentano rispettivamente il bit **Co** e il bit **D/C** e devono essere seguiti da 6 bit a zero.

- Il bit Co specifica se invieremo uno **stream** di dati o un singolo byte
- Il bit D/C specifica se i byte successivi saranno interpretati come **comando** o come **data**. Se il bit sarà a 0 allora i byte saranno interpretati come istruzioni che saranno eseguite direttamente dal controllore interno al display. Se invece il bit sarà settato ad 1 allora tutte le informazioni che seguono saranno caricate in GDDRAM. Vedremo anche in seguito come

ad ogni byte il processore interno incrementerà automaticamente il puntatore alla locazione di GDDRAM successiva per l'allocazione del prossimo byte.



4.2.1 Comandi del display

Il display dispone di ulteriori comandi necessari per la configurazione del display descritti nella documentazione (https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf); dei quali ne vedremo alcuni per la progettazione della sequenza di START del display.

4.2.2 Modalità di indirizzamento

Una modalità di indirizzamento definisce come il controllore del device SSD1306 gestisce il proprio puntatore alla memoria principale. Nel display SSD1306 esistono 3 modi di indirizzamento per la GDDRAM: page addressing mode, horizontal addressing mode e vertical addressing mode. Le modalità di indirizzamento si rivelano dei modi efficienti di inviare dati al dispositivo, fondamentali in fase di progettazione.
Esistono due tipi di puntatore alla GDDRAM: il **column address** che specifica la colonna (0-127) e il page address che specifica la riga (0-7).
Pertanto per ogni column address e page address ci riferiamo ad un byte che è graficamente rappresentato come una colonna verticale di 8 pixel. Quando un byte viene scritto in memoria il display SSD1306 dispone di due mappature disponibili una che parte dal LSB e va al MSB del byte e un'altra mappatura che va dal MSB al LSB.

Page addressing mode:
In questa modalità ad ogni lettura/scrittura in RAM il column address è incrementato di 1. Quando viene raggiunta la 127 colonna della pagina. Il puntatore viene resettato alla locazione 0 della colonna e il page address non cambia. Pertanto il puntatore riscrive sulla stessa page.
Horizontal addressing mode:
La modalità segue lo stesso principio di funzionamento della page addressing mode con l'unica differenza che raggiunta la 127 colonna, al prossimo byte viene incrementato il page address, con il risultato che il puntatore va visivamente "a capo". Se il puntatore raggiunge l'ultima colonna dell'ultima riga ricomincia dal primo byte.
Vertical addressing mode:
In questa modalità ad ogni lettura/scrittura in RAM è soltanto il page address ad essere incrementato per ogni byte trasmesso.

4.2.3 Flusso dati / Accensione pixel

Dopo aver accesso il Display, definito la modalità di alimentazione, e definita la modalità di indirizzamento in memoria (con specifico indirizzo di partenza), per accendere dei pixel è necessario specificare la volontà di inviare Data (come spiegato sopra) e inviare un flusso di Byte.

5 Metodologie di risoluzione

Si deve pertanto progettare una libreria che implementi una funzione **l'accensione** e la configurazione del display. Una funzione che utilizzi i metodi di indirizzamento per cambiare la locazione di memoria puntata dal puntatore, e una funzione per la trasmissione del flusso dati, in particolare la rappresentazione di caratteri ASCII a schermo.

5.1 Progettazione libreria SSD1306 SSDGraphic

E' necessario nuovamente definire l'insieme di stati che il display può assumere:

- **DISPLAY˙OFF**: Indica che il display non è stato ancora inizializzato e pertanto l'unica funzione eseguibile deve essere quella che trasmette la sequenza di comandi per l'avvio (beginDisplay).
- **DISPLAY˙ON**: Indica che il display è stato inizializzato e pertanto è possibile trasmettere dati.

5.1.1 Sequenza di Avvio

La descrizione dettagliata dei comandi utilizzati può essere trovata nella documentazione allegata https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf. In questo documento saranno esplicitati i comandi trasmessi con eventuali motivazioni.

La funzione di startup (beginDisplay) segue in linea di principio il flowchart descritto a pagina 64 della documentazione ufficiale.
Dapprima ci si assicura che il pannello OLED sia **spento**, trasmettendo il comando **AEh** che disattiva i circuiti relativi al display e ponendo pertanto lo Slave in stato di "sleep".
A questo punto seguendo la **tabella 10-2 a pagina 38.** della documentazione si configura il display affinché gestisca la RAM come nella configurazione **(a)**.
Pertanto si imposta il Multiplex Ratio come descritto nella documentazione attraverso il comando **A8h** e impostandolo a **3Fh** ovvero 63 in decimale (cioè 64 partendo da 0).
Si imposta poi il display offset con il comando D3h e facendolo partire da 0, cioè la prima page.
Successivamente si definisce la linea, ovvero il pixel di partenza del puntatore alla GDDRAM da cui il display inizierà a scrivere Dati. Essendo presenti 64 pixel verticali, impostiamo che parta dal primo (ovvero lo 0) attraverso il comando **40h**.
A questo punto si imposta il Segment Re-map che inverte il column address rispetto ai pin di "segmento" (cioè delle colonne) della RAM. Senza questo comando le informazioni che trasferiamo al device sarebbe visualizzate al contrario.

Successivamente si definisce la configurazione dei PIN della SRAM (COM) in linea con quanto già descritto ovvero attraverso il comando DAh e 12h.
Si definisce il contrast color, ovvero il livello di luminosità dei pixel con il comando 81h e CFh (Non credo funzioni sul simulatore).
A questo punto si abilita l'intero display con il comando **A4h** (cioè che legge il contenuto dalla SRAM) e si attiva la configurazione normale con il comando A6h (1 = pixel acceso, 0 = pixel spento).
Infine si imposta la frequenza del clock interno del display come definito nella documentazione e abilito la charge pump regulator.

Come ultimo passaggio si definisce la modalità di indirizzamento del display: la più efficiente ed in linea con le esigenze di questa progettazione è la Horizontal Address Mode che ci permette di "scrivere" sul display come fosse un foglio di carta a 8 righe.
Il comando **20h** permette di selezionare la modalità selezione indirizzamento e inviando come byte successivo **00h** si imposta la modalità orizzontale.
Si accende pertanto il display con il comando **AFh**.

5.1.2 Gestione allocazione memoria e vettori mappati in ROM

La sequenza di Start, di indirizzamento orizzontale e la configurazione dei pixel dei caratteri ASCII sono dei dati **costanti** e che pertanto occuperebbero un ingente spazio di memoria in

SRAM. La soluzione implementata in questa libreria prevede la memorizzazione di questi 3 array nella memoria di programma (per una occupazione stimata di circa 600 bytes) e pertanto le relative funzioni si occuperanno di leggere, sfruttando le funzioni di libreria standard Arduino **pgmspace.h**, i dati dalla memoria ed inviarli nel formato corretto al dispositivo.
Per la documentazione delle costanti è stato utilizzato un approccio ibrido:

- Costanti auto-documentanti: ovvero che utilizzano un alias indicativo. Tra cui il DataStream Byte o il Control Byte
- Costanti documentate: quasi tutti i comandi del dispositivo sono stati documentati attraverso dei commenti nel codice.

5.1.3 Testing ed accensione pixel

Per verificare che la sequenza di startup abbia configurato correttamente il dispositivo è bene verificare che inviando un flusso dati, si accendano dei pixel sullo schermo. A questo punto inviando dei Byte casuali (in formato Data) è facile verificare che il display inizierà a scrivere i pixel con valore a 1 partendo dalla page 0 e proseguendo in maniera orizzontale senza mai cambiare indirizzo della Page fino alla fine della linea, per poi andare a capo e così via.

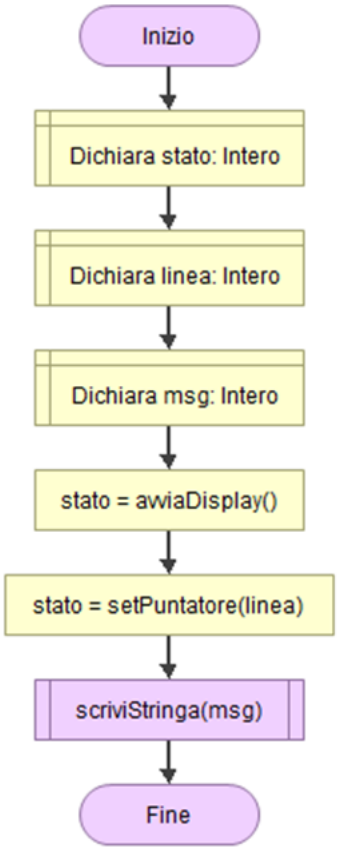
5.1.4 Posizionamento del puntatore

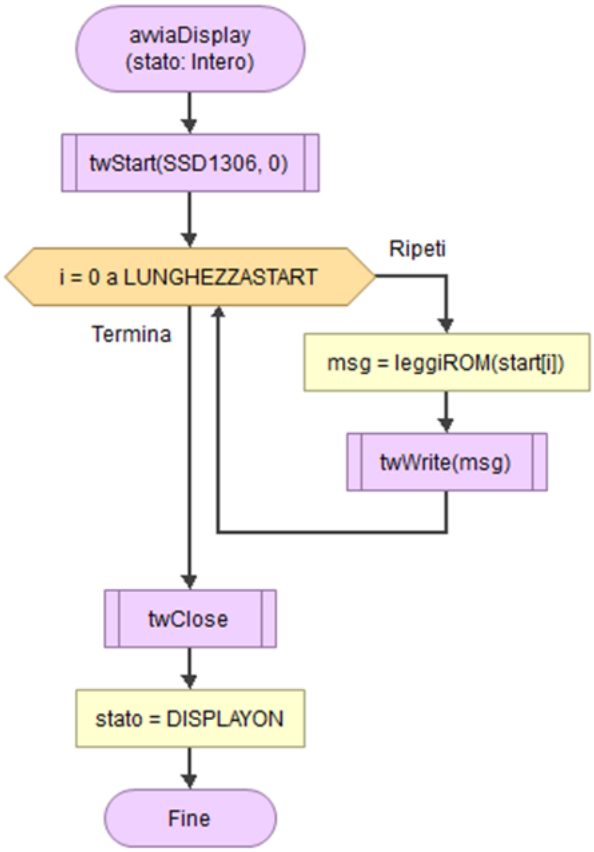
Generalmente servirebbe un buffer di **1024 Bytes** per scrivere l'intero display ad ogni singolo refresh. Ma in un architettura di questo tipo resta una soluzione difficilmente implementabile e poco efficiente. Pertanto si procede ora a realizzare una funzione che permetta di spostare in maniera **dinamica** il puntatore in GDDRAM e sfruttando le caratteristiche della Horizontal Address Mode; in modo tale da inviare soltanto i Byte per rappresentare solo le informazioni necessarie.
Pertanto si acquisisce in input la linea (rispettivamente la coordinata y) la quale dovrà essere compresa rispettivamente tra **0-7** e si invia utilizzando le funzioni di libreria I2C il comando **21h** che deve essere seguito dall'indirizzo della colonna di partenza e dalla colonna di terminazione e successivamente (0-127) il comando **22h** che deve essere seguito dalla page di partenza e dalla page massima su cui il puntatore può essere incrementato. Definito il puntatore i successivi byte inviati come Data saranno caricati nelle locazioni definite seguendo i principi della Horizontal Address Mode. Infine si chiude la connessione.

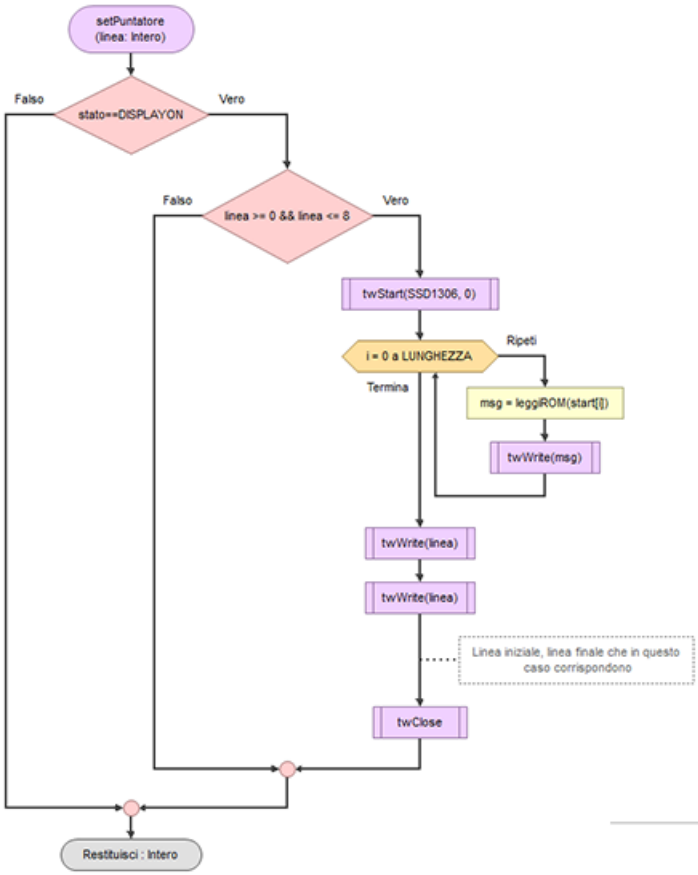
5.1.5 Scrittura di stringhe

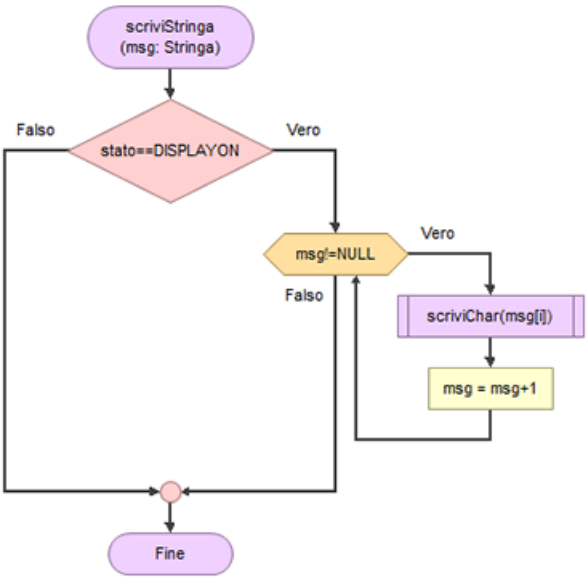
Si definisce una funzione **printString** che acquisita una stringa in input scorre la stringa richiamando carattere per carattere una funzione **printChar**, che acquisiendo il carattere in input stampa a schermo i pixel corrispondenti ad un carattere ASCII. La funzione printChar dato un carattere ASCII calcola la corrispondente **posizione** del carattere nell'array di byte memorizzato in Flash, necessaria ad accendere i pixel nello schermo. Considerando che per ogni carattere sono necessari 6 Byte (Font 6x8) e che nell'array mappato in memoria non sono presenti i primi 32 caratteri non stampabili, si sottrae 32 e si moltiplica per 6, ovvero la "larghezza" di ogni carattere. A questo punto si leggono i successivi 6 byte e si caricano nel buffer I2C. Infine si chiude la connessione.

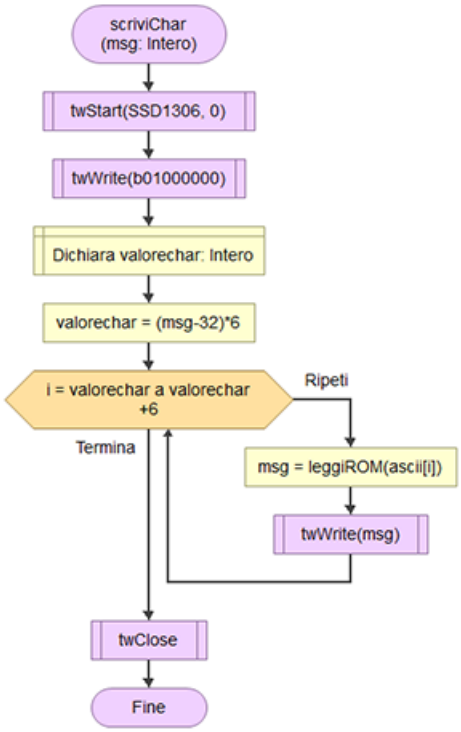
5.2 Flowchart



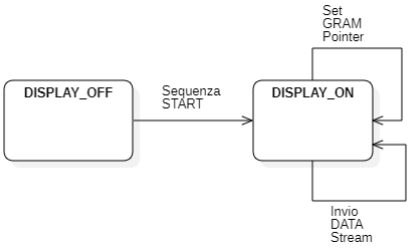



setPuntatore
(linea: Intero)
stato==DISPLAYON
Falso
Vero
linea >= 0 && linea <= 8
Falso
Vero
twStart(SSD1306, 0)
i = 0 a LUNGHEZZA
Ripeti
Termina
msg = leggiROM(start[i])
twWrite(msg)
twWrite(linea)
twWrite(linea)
Linea iniziale, linea finale che in questo caso corrispondono
twClose
Restituisci : Intero


scriviStringa
(msg: Stringa)
stato==DISPLAYON
Falso
Vero
msg!=NULL
Vero
Falso
scriviChar(msg[i])
msg = msg+1
Fine


scriviChar
(msg: Intero)
twStart(SSD1306, 0)
twWrite(b01000000)
Dichiara valorechar: Intero
valorechar = (msg-32)*6
i = valorechar a valorechar +6
Ripeti
Termina
msg = leggiROM(ascii[i])
twWrite(msg)
twClose
Fine

5.3 Statechart



5.4 Codice

Il codice è contenuto nei rispettivi file:
SDDGraphicMega.h - header
SDDGraphicMega.cpp - codice