Andrea Monzani [Mat 952817]

VGA Viewer – Progetto di Sistemi Operativi

Sommario

1 Introduzione

Il progetto si pone l'obiettivo di valutare le potenzialità e le limitazioni che si trovano andando a sviluppare un'applicazione per visualizzare delle immagini BMP lette da una scheda SD su uno schermo attraverso il pilotaggio di una uscita VGA. Il focus è sulla gestione dei segnali video e del relativo frame buffer, essendo la parte con requisiti più stringenti. Per lo sviluppo è stata utilizzata la discovery board per STM32F407 assieme al STM32CubeIDE. Non è stato necessario acquistare dispositivi esterni oltre all' adattatore VGA per collegare i pin della discovery board.

1.2 Schema progetto

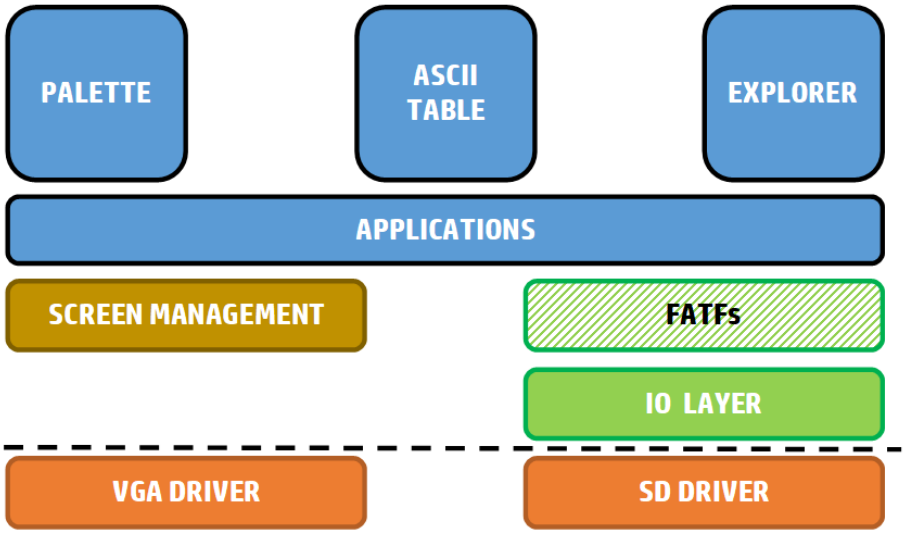


Figura 1 Schema a blocchi del progetto

Il progetto è diviso in tre layer, ognuno dipendente dai sottostanti. A livello più basso dello stack troviamo due *driver* che, una volta implementati, si dovranno interfacciare con i relativi dispositivi hardware. Per la parte di visualizzazione a video, un oggetto di screen management si occuperà di disegnare i vari elementi grafici di cui avremo bisogno. La sezione di I/O per la lettura di un file system si appoggia sulla libreria (già disponibile tramite CubeIDE) FatFs che, tramite implementazione di alcune funzioni predisposte per l'accesso al disco fisico, si interfaccerà con il driver per la lettura dei dati da scheda SD.

Nel layer più alto troviamo tre applicazioni utilizzate per valutare le prestazioni delle varie aree del nostro sistema di cui parleremo successivamente. Partiremo con il descrivere la costruzione del driver VGA, dalla generazione dei segnali al disegno tramite DMA, per poi passare alla descrizione del driver SD. Concluderemo con una breve illustrazione delle applicazioni e un analisi delle prestazioni nelle parti più critiche del sistema. Screen management e I/O layer non verranno trattati essendo semplicemente un'astrazione verso i driver sottostanti.

2 Struttura VGA

Come dispositivo di uscita video dal nostro sistema è stato scelto di utilizzare un monitor VGA con una risoluzione video più grande possibile congruentemente alla disponibilità di risorse e capacità del nostro microprocessore. Analizziamo innanzitutto i segnali che viaggiano in un cavo VGA per capire di cosa abbiamo bisogno

2.1 Connettore VGA

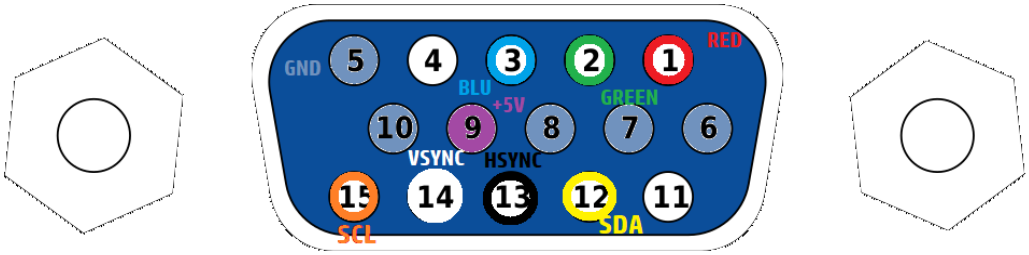


*Figura 2 Mappa dei pin di un connettore VGA, **Wikipedia***

Come si può vedere dall' immagine, la comunicazione VGA viaggia su diversi segnali:

- *Ground pins* (GND, sync and color return channels).
- *Power pin* (utilizzato in alcuni monitor per alimentare le EEPROM EDID).

- *I^2C/IDx data lines*: linee dati utilizzate per la comunicazione con il chip EEPROM.
- Segnali di ***sync orizzontale e verticale.***
- ***RGB***: linee dove viaggia il segnale analogico per ogni colore.

Possiamo raggruppare i segnali in due categorie rispetto alle nostre necessità:

- *Segnali ausiliari*: comprendono tutti quei segnali che non sono direttamente utilizzati per la gestione del video, quali ground, linee dati, e pin +5V, che saranno cablati direttamente alla scheda.
- *Segnali video*: comprendono i segnali di timing e RGB. I segnali di timing verranno collegati (come vedremo successivamente) ad un uscita di un timer; i segnali video sono invece segnali analogici nel range 0-0.7V; attraverseranno di conseguenza un convertitore digitale-analogico esterno.

2.2 Sync signals

Un frame su un monitor è composto da y linee di x pixels e il monitor necessita di due impulsi per capire quando una linea e/o un frame stanno iniziando/finendo. Le linee vengono disegnate da sinistra a destra, mentre i frame dall' alto verso il basso.

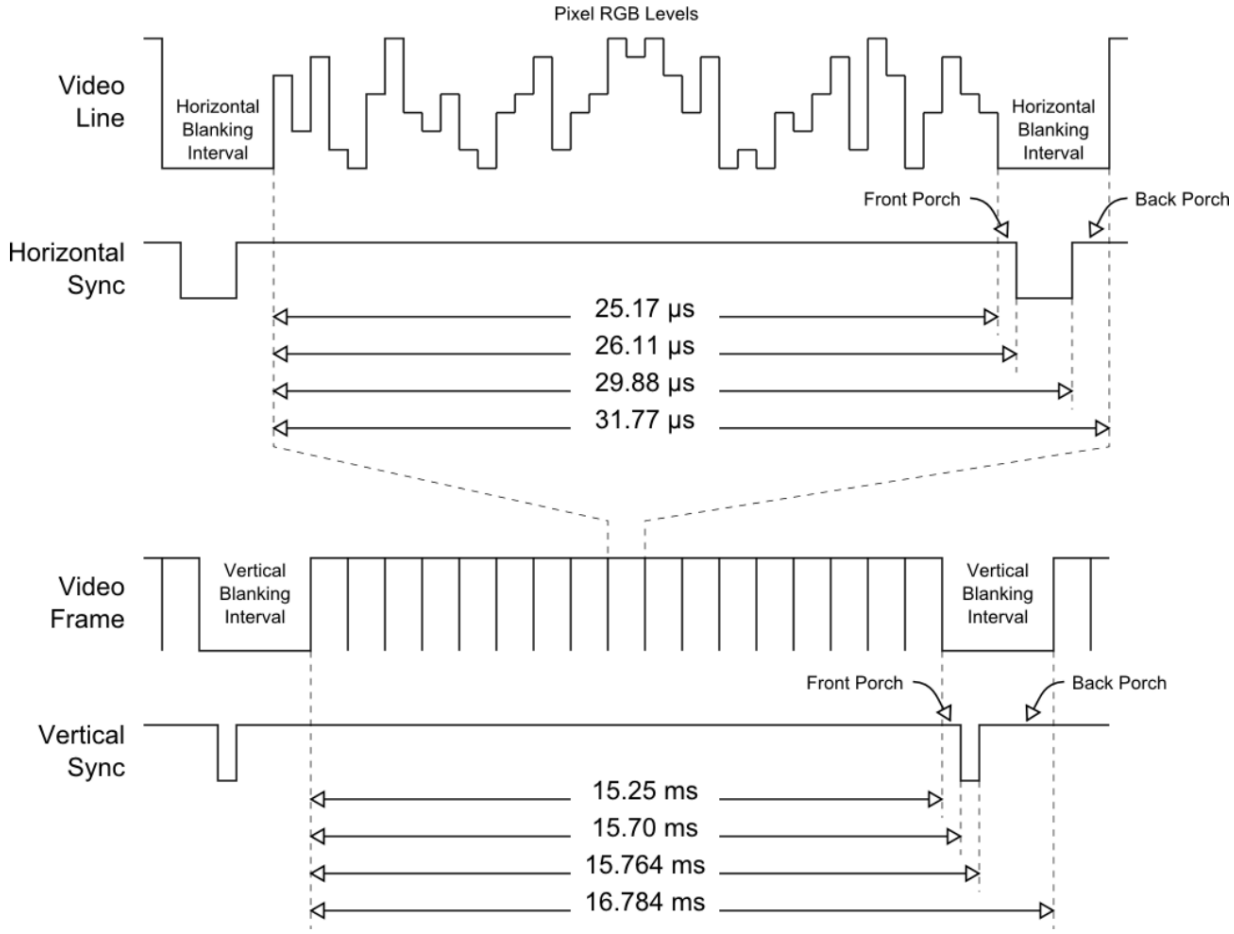


Figura 3 Schema dei segnali di syncing previsti dal monitor

Il segnale di sync orizzontale può essere diviso in quattro parti:

1. ***Area visibile***: parte di tempo in cui i dati inviati sui segnali RGB vengono visualizzati a monitor. La sua durata corrisponde a $line\_pixels * pixel\_time$
2. ***Front porch***
3. ***Sync pulse***
4. ***Back porch***

Osservando la figura, possiamo notare che:

- I segnali sono normalmente alti e l'impulso è negativo
- Il segnale di sync verticale (*vsync*) ha la stessa struttura del sync orizzontale (*hsync*), solo che i tempi sono relativi non più ad un pixel ma ad una intera linea.
- Il vsync viene incrementato dopo la parte di back porch dell'hsync

- La parte di sync e porch viene detta *blanking interval*, e in questo lasso di tempo i segnali devono rimanere a livello zero (*low*)[1]

I segnali di porch venivano per lo più usati nei monitor CRT.
Questo è il nostro punto di partenza. Sappiamo che dobbiamo gestire questi due impulsi e nel mezzo inviare i segnali video, ma come facciamo ad individuare quali risoluzioni supporta il monitor e che tempi utilizzare? Entra in gioco la lettura dell'*EDID*.

2.3 EDID

L' *Extended Display Identification Data* è una struttura dati inviata da un generico dispositivo video digitale ad una scheda video per comunicare le proprie caratteristiche. Il blocco base definisce una struttura da 128 byte che contiene tutte le informazioni per stabilire quali risoluzioni sono supportate. Nelle versioni successive, il blocco è stato allargato a 256 byte, poi surclassato dallo standard E-EDID. Per la lettura, viene usato il bus I^2C come canale di comunicazione.
L' *EDID* contiene i seguenti dati:

- *[Bytes 0-19]* **Header information**
  Header con una parte fissa più le informazioni sul costruttore, serial number, anno e mese di produzione, versione dell'EDID utilizzata.
- *[Bytes 20-24]* **Basic display parameters**
  Caratteristiche sull' input analogico (o digitale ma non è il nostro caso), dimensioni fisiche dello schermo, gamma, bitmap delle feature supportate.

Bit 7 = 0	Analog input. If clear, the following bit definitions apply:
Bits 6–5	Video white and sync levels, relative to blank: `00` = +0.7/−0.3 V `01` = +0.714/−0.286 V `10` = +1.0/−0.4 V `11` = +0.7/0 V (EVC)
Bit 4	Blank-to-black setup (pedestal) expected
Bit 3	Separate sync supported
Bit 2	Composite sync (on HSync) supported
Bit 1	Sync on green supported
Bit 0	VSync pulse must be serrated when composite or sync-on-green is used.

Figura 4 Descrizione dei bit per l' input analogico, ***Wikipedia***

  Per quanto riguarda il progetto, siamo interessati a sapere se il monitor supporta i segnali nel range 0.7/0V, che corrisponde all' uscita del DAC. Notiamo che, anche se non supportato, il range è incluso in tutti gli altri, quindi non dovremmo avere grossi problemi.
  Gli altri parametri sono per lo più residui dei vecchi segnali per i monitor CRT.
- *[Bytes 25-34]* **Coordinate cromatiche**
- *[Bytes 35-37]* **Bitmap per i timing comuni supportati**
  Contiene i flag[2] che indicano quali delle risoluzioni standard sono supportate dal monitor. Ci servirà leggere questi dati per capire se il monitor a cui siamo collegati supporta la risoluzione di 800x600@60Hz
- *[Bytes 38-53]* **Standard timing information**
  Ulteriori bit per descrivere temporizzazioni standard dello schermo
- *[Bytes 54-125]* **18 bytes descriptors**
- *[Byte 126]* **Numero di estensioni che seguono**
- *[Byte 127]* **Checksum**
  La somma di tutti i byte (compreso checksum) deve essere 0

[1] https://electronics.stackexchange.com/a/209494 – **Output blanking**
[2] http://tinyvga.com/vga-timing – **Timing standard informations**

2.4 I^2C

L' *Inter-Integrated Circuit* è un protocollo e una specifica hardware per la comunicazione seriale multi-slave, half-duplex (utilizza un solo canale per i dati), 8 bit oriented con un singolo master.

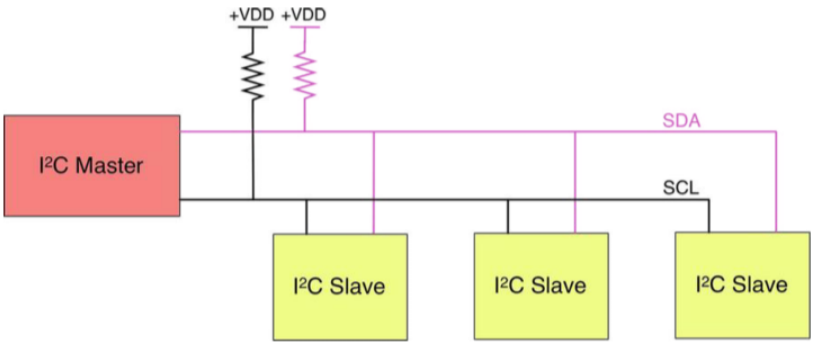


Figura 5 Schema di un bus I^2C, ***Noviello***

Il bus è composto da due linee di comunicazione che sono configurate (da specifica del protocollo) come open drain con resistore di pull-up e vengono chiamate **SDA** (*Serial Data Line*) e **SCL** (*Serial Clock Line*).
L' open drain è un tipo di output dove l'uscita forma un circuito aperto (alta impedenza o hi-Z) quando è impostato ad un livello $high$ e invece è connessa al ground in caso di livello low; in questo modo è possibile condividere il bus con più dispositivi senza causare interferenze: se un device non in open drain tentasse di riportare il livello a $high$ quando un altro dispositivo lo tiene low, si avrebbe un risultato imprevedibile.
Lo standard richiede che ogni slave sia identificato con uno *slave address* univoco per il bus a cui sono connessi (l'id è solitamente a 7 bit, ma esiste anche la versione a 10 bit). La velocità standard di trasmissione per le prime versioni era nel range dei 100-400 KHz, mentre adesso si riescono a raggiungere velocità di qualche MHz.

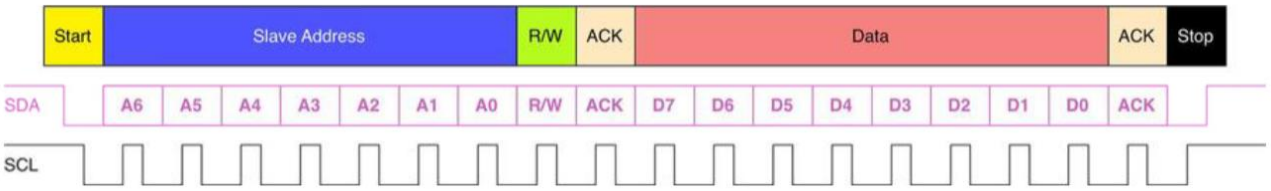


Figura 6 Trasmissione di un messaggio I^2C sul bus, ***Noviello***

Una transazione in I^2C è sempre iniziata e completata dal master e tutti i messaggi vengono divisi in due frame: *address*, dove il master indica il destinatario della sua comunicazione, e *data* contenente il byte letto o scritto.
Il master inizia a comunicare inviando una condizione di START sul bus e termina con una condizione di STOP (notiamo come qui la definizione per l'inizio e la fine di un messaggio siano precisate, garanzia che non avremo più avanti quando parleremo del bus SPI).
Ogni parola trasmessa sul canale deve essere di 8 bit + un bit di ACK (linea data messa low per un acknowledgement positivo); indicando al trasmettitore l'avvenuta lettura del byte. L'impulso del clock per l'ACK viene sempre generato dal master.
Una volta trasmesso il frame per l'address, può iniziare la trasmissione dei dati da parte del master o dello slave secondo il contenuto del bit R/W. Il numero di frame dati è arbitrario e potrebbero essere trasmessi in burst fino a che non viene generata una condizione di STOP.

2.5 Time generation

2.5.1 Struttura base dei timer

General timing

Screen refresh rate	60 Hz
Vertical refresh	37.878787878788 kHz
Pixel freq.	40.0 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is positive.

Scanline part	Pixels	Time [µs]
Visible area	800	20
Front porch	40	1
Sync pulse	128	3.2
Back porch	88	2.2
Whole line	1056	26.4

Vertical timing (frame)

Polarity of vertical sync pulse is positive.

Frame part	Lines	Time [ms]
Visible area	600	15.84
Front porch	1	0.0264
Sync pulse	4	0.1056
Back porch	23	0.6072
Whole frame	628	16.5792

Figura 7 Temporizzazione risoluzione 800x600@60Hz, ***tinyvga.com***

Dopo aver letto l'*EDID* abbiamo a disposizione tutti i dati per poter iniziare a generare i nostri segnali di clock secondo le specifiche standard oppure seguendo le informazioni nei vari descrittori.
Analizziamo prima di tutto l'hsync, ma la struttura rimane poi la stessa anche per il vsync.

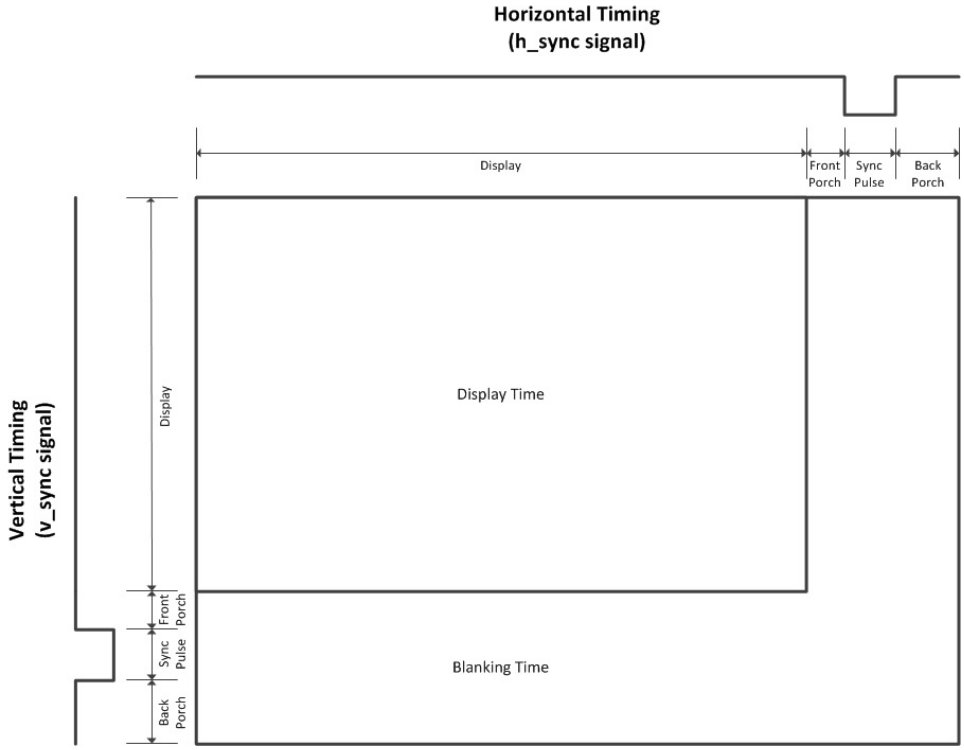


Figura 8 Composizione timing schermo

Come notiamo dalla figura, il sync che dobbiamo generare può essere ricondotto ad un segnale PWM con un certo duty cycle ricavato dai nostri tempi di porch e visible area. I canali di output sui timer dell'STM32F407 supportano questo tipo di modalità, ma la parte attiva del segnale (o anche quella a livello basso se vogliamo) può essere solo all'inizio o alla fine. Dobbiamo quindi "shiftare" l'impulso di modo da avere il back porch all' inizio e il sync alla fine. Collegando due timer in cascata e configurando il secondo per essere aggiornato usando un output del primo, riusciamo a gestire correttamente l'incremento del vsync rispetto alla fine del back porch del sync orizzontale. Con questa soluzione riusciamo a generare due segnali abbastanza precisi per la nostra sincronizzazione.
Il primo problema che si presenta riguarda la gestione del pixel clock: ad ogni tick del sync orizzontale dobbiamo inviare al nostro monitor i segnali RGB (tutti e tre contemporaneamente).

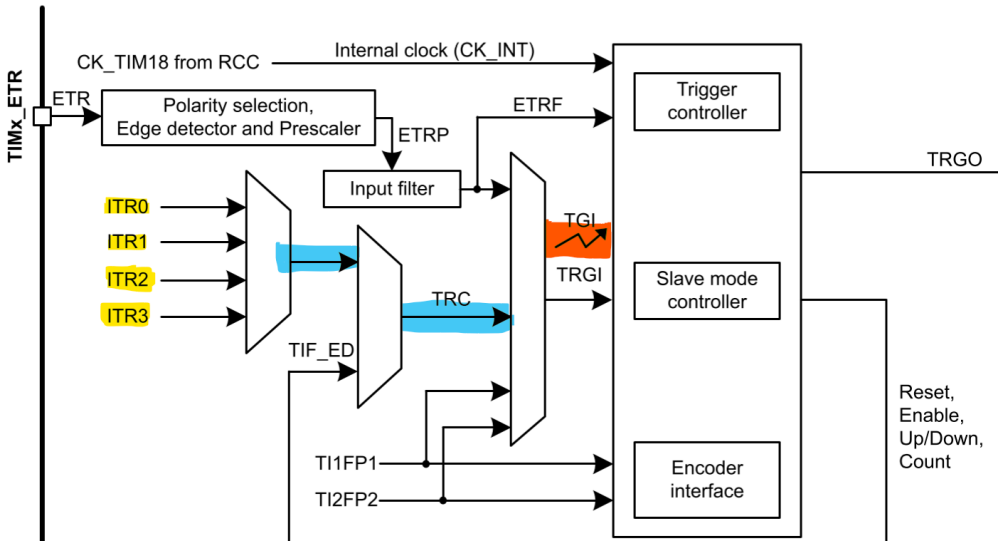


*Figura 9 Schema del TIM1 sull STM32F407, **RM0090***

Osservando lo schema a blocchi di un timer, notiamo come sui trigger esterni sia disponibile un evento di interrupt/DMA: a monte del sync orizzontale possiamo quindi configurare un timer che, ridimensionando correttamente il clock del bus, genera il trigger alla periferica DMA per l'invio dei dati di un pixel; dobbiamo solo capire come attivarlo correttamente all'inizio di una linea.

2.5.2 *Implementazione interrupt via timer*

Il primo problema che riscontriamo riguarda il blanking: durante questo periodo il segnale deve essere per forza portato a livello zero[3]. La soluzione più comoda sarebbe quella di creare un frame buffer che comprenda la parte visible più la parte di blanking inizializzata a zero (e non modificabile); così facendo il DMA potrebbe trasferire automaticamente tutto il buffer e avremmo la garanzia di rispettare questa condizione (a meno di latenze sul BusMatrix di cui parleremo più avanti). L' unico problema è che, per ovviare alla capacità di memoria ridotta dell'STM32F407, non è possibile sprecare "spazio" per la gestione dei bordi.
Dobbiamo quindi ripiegare ad una gestione con interrupt del disegno:

- *Interrupt all'inizio di una linea (±delta per correzione ritardi)*: lanciamo il DMA per visualizzare i pixel appartenenti ad una linea.
- *Interrupt alla fine di una linea*: terminiamo il DMA e forziamo l'output a zero, non avendo la garanzia di aver disegnato i pixel di blanking per via delle latenze sul bus.
- *Interrupt all'inizio e alla fine visible area (frame)*: Gestiamo semplicemente il flag di vsync per abilitare o disabilitare il disegno di altre linee.

Con questa soluzione possiamo gestire le nostre aree di blanking a discapito di una complessità maggiore nel codice e ulteriori ritardi sul BusMatrix (e nell' intero sistema perché abbiamo sempre interrupt di alta priorità che bloccano la CPU) che per le risoluzioni con pixel clock più grandi potrebbero risultare inaccettabili.
I quattro interrupt verranno generati dai nostri canali PWM configurati con uscita disabilitata.

2.6 *Stabilità dell'HSI*

Una volta sviluppata la parte di clock, durante i primi test si è presentato un problema interessante: il segnale che arrivava allo schermo non era stabile per la visualizzazione "verticale" (sul monitor di test usato per lo sviluppo, su un televisore l'uscita invece veniva rilevata e poi disconnessa).
Come è possibile vedere dalla figura, lo schermo sembra flickerare in verticale (il menù dello schermo è spostato per visualizzare meglio il problema), come se il vsync non fosse corretto.
Senza oscilloscopio non è purtroppo facile individuare la sorgente del disturbo, ma nei forum ST[4] si discute molto spesso del fatto che l'*HSI* non è abbastanza accurato per temporizzare dispositivi che richiedono timing precisi.

[3] https://electronics.stackexchange.com/a/209494 – **Output blanking**

[4] https://community.st.com/s/question/0D53W00000LeSuwSAF/stm32f4-use-hsi-for-usb-clock – **HSI accuracy**

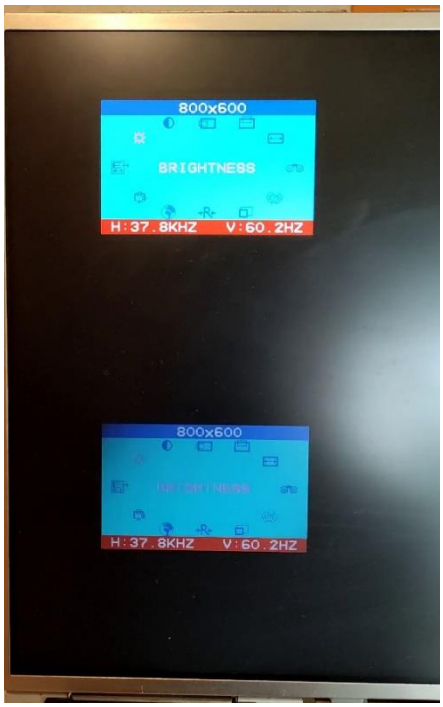


Figura 10 Flickering dello schermo usando l'HSI

Fortunatamente il discovery board possiede un oscillatore esterno (*HSE*) che può essere usato, andando così a risolvere il problema. L' *HSE* si abilita dalla configurazione dell'*RCC* e deve poi essere selezionato dal clock tree. Per poter utilizzare l'HSE come sorgente principale del clock bisogna verificare che le saldature sulla discovery board rispettino quanto dichiarato nel relativo manuale.

2.7 Dimensioni frame buffer

Una volta definito il metodo di generazione dei nostri segnali di sync e verificato che il segnale ricevuto dallo schermo è stabile, possiamo iniziare a descrivere la costruzione e gestione del nostro frame buffer. Il primo grosso limite imposto dalla scheda ST è la capacità di memoria: solo 128KB di RAM utilizzabili. La FLASH non è utilizzabile come memoria per via delle complessità e limitati cicli di lettura/scrittura disponibili, mentre la CCMRAM non è connessa al BusMatrix e la useremo invece, come vedremo più avanti, per contenere tutte le altre sezioni del nostro codice.
Di conseguenza il nostro frame buffer sarà al massimo di risoluzione 400x300 pixels con 1byte per pixel per la codifica del colore (2 bit per il rosso, 3 per blu e verde). Questo buffer è 4 volte più piccolo del classico 800x600 quindi possiamo semplicemente scalare il pixel clock per visualizzarlo senza problemi.
Torniamo sul nostro timing 800x600 originale. Scalandolo a 400x300, la frequenza diventerà 20Mhz e tutti i conteggi dei pixel orizzontali verranno dimezzati (il tempo finale deve rimanere lo stesso). Per il timing verticale, non cambierà nulla ma le linee visibili verranno ripetute 2 a 2, tornando così ad una effettiva risoluzione di 800x600 rispettando i limiti imposti.

2.8 Periferiche di uscita

Per visualizzare il nostro screen buffer a schermo abbiamo bisogno di convertire i dati da digitale ad analogico e per farlo dobbiamo utilizzare un DAC interno o esterno (e quindi inviare ogni singolo bit su un pin GPIO).
Come abbiamo già osservato, il nostro frame buffer ha una risoluzione di 8bit per pixel quindi dobbiamo necessariamente usare un convertitore esterno (il DAC interno converte 8 bit in un singolo segnale analogico, mentre noi ne dobbiamo ricavare 3).
Dei vari convertitori che si possono costruire, si è optato per un partitore resistivo per via della semplicità e delle minime richieste in termini di hardware (ci servono solo delle resistenze).

2.8.1 DAC usando il partitore resistivo

Il partitore resistivo è un circuito passivo dove il livello di output è una frazione dell'input definita dal rapporto di due resistenze in serie.

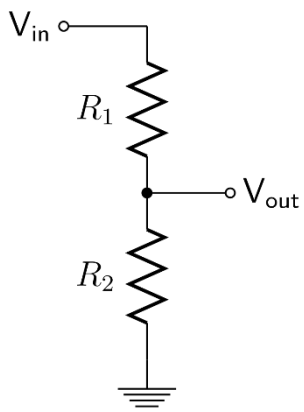


Figura 11 Schema partitore resistivo

La relazione della tensione si ricava analizzando l'intensità di corrente I nel circuito $I = \frac{V_{in}}{R_1 + R_2}$ $V_{out} = IR_2 = V_{in}\frac{R_2}{R_1+R_2}$, e come abbiamo visto, il V_{out} massimo dovrà essere di $0.7V$.

Nel nostro caso, R_1 può essere vista come la resistenza equivalente $R_{//}$ di tre resistori R_{b0}, R_{b1}, R_{b2} per i nostri bit da convertire in un livello analogico. R_{b2} sarà la resistenza per il bit più significativo, di conseguenza $R_{b1} = 2R_{b2}$ *e* $R_{b0} = 2R_{b1} = 4R_{b2}$. Il partitore deve avere anche un impedenza di output equivalente a quella delle linee colore VGA[5] (75 Ω).

Nella figura qui sotto possiamo vedere una simulazione del circuito usando il software online *falstad*, dove notiamo che l'output massimo è di poco sopra i $0.6V$ quando tutti i bit sono a 1 (la discovery board lavora a $2.94V$, con un input di $3.3V$ il livello sarebbe più vicino ai $0.7V$) e la corrente utilizzata rispetta il limite di $\pm 8\,mA$ indicati nel datasheet per i pin GPIO.

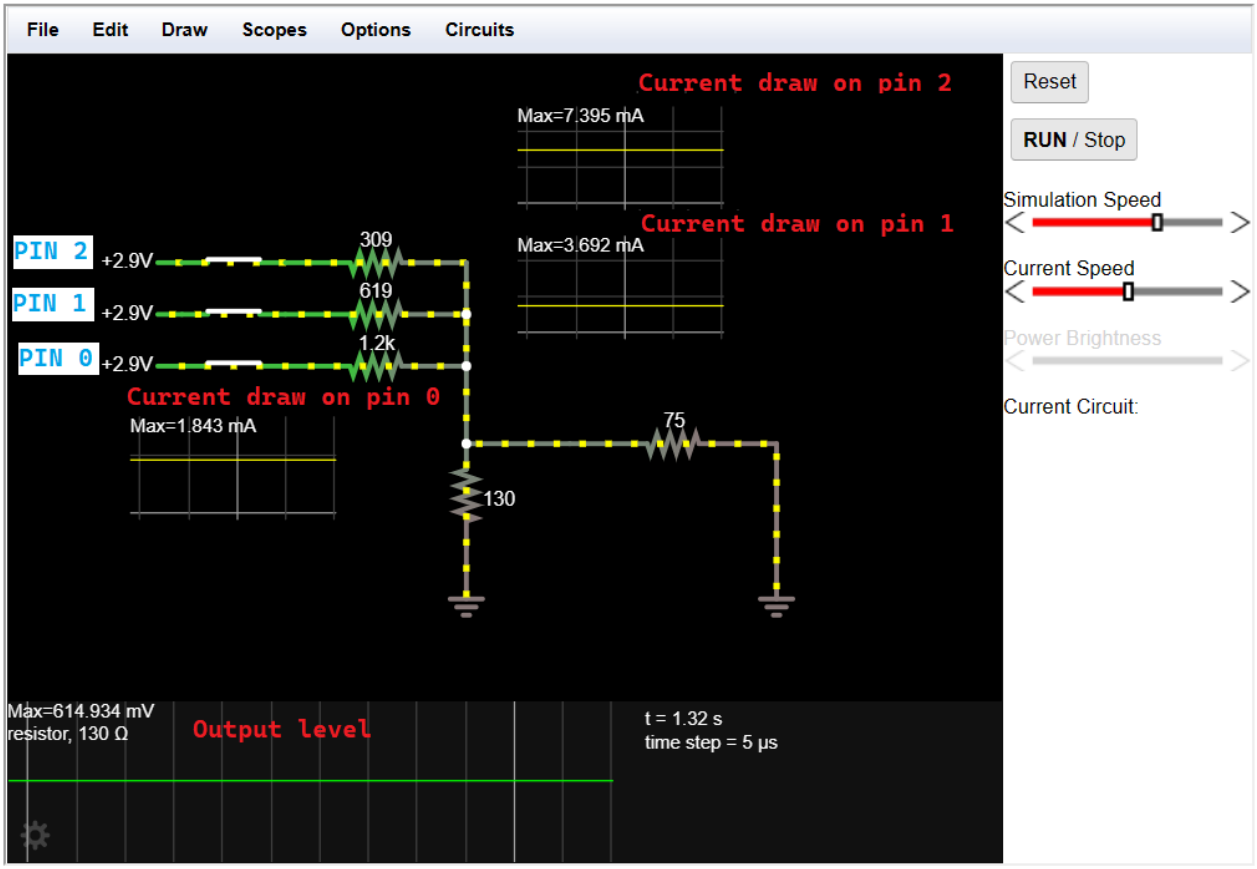


Figura 12 Simulazione DAC

La frequenza del segnale è attorno ai 20Mhz (seguendo il pixel clock) quindi possiamo tranquillamente gestirla impostando la corretta velocità del pin tramite configurazione OSPEED.

2.8.2 Discussione sull' uso del DAC interno/esterno

Per avere un'uscita video a 24bpp, dobbiamo per forza ridurre le dimensioni del buffer a 200x150 in modo da rispettare il limite di memoria; si ha inoltre la necessità di utilizzare un altro DAC (il partitore resistivo diventerebbe troppo grande). L' STM32F407 dispone di un DAC interno ma non è adatto per operazioni grafiche (è invece incentrato sull' audio in questa scheda):

- *Ha solo due canali da 8 o 12 bit* che possono lavorare simultaneamente: dal punto di vista del DMA non è un problema perché possiamo generare transazioni da 16bit e riorganizzare il frame buffer in modo che i

[5] https://electronics.stackexchange.com/a/453444 - **Indicazioni su come calcolare i valori delle resistenze**

pixel letti dalla richiesta siano adiacenti (ad esempio, al posto di avere tre byte configurati come RGB avremo …BGBGBG… in sequenza e dopo un altro buffer solo per i byte del rosso)

- *Introduzione di concorrenza sul BusMatrix*: ipotizzando di usare 2 DMA per le richieste, avremmo il DMA 1 adibito all' invio dei dati al DAC (si trovano sull' APB1) e il DMA 2 per l'accesso ai pin GPIO. Oltre a dover gestire il triggering dei due DMA separatamente, dobbiamo anche considerare che, pur avendo il DMA1 accesso diretto all' APB1, le richieste alla memoria passano tutte dal BusMatrix quindi potremmo aggiungere ulteriori contese (come spiegheremo nel paragrafo successivo). Stesso discorso vale se volessimo usare il solo DMA2
- *Prestazioni limitate*: il datasheet dell'F407 specifica chiaramente (anche in base alle caratteristiche elettriche dell'output) che il massimo rate di conversione per avere un risultato corretto è di 1MS/s. STM fornisce un application note AN4566[6] per aumentare le prestazioni fino a 10MS/s ma richiede l'uso di un OpAmp esterno.

2.9 DMA

2.9.1 Introduzione

Una volta definita tutta l'architettura per disegnare dei pixel, dobbiamo capire come mandare il contenuto del frame buffer alla nostra periferica di output (GPIO come abbiamo già stabilito).

Una generica periferica su STM32 può essere guidata in tre modi: polling, interrupt e DMA (a meno di eccezioni)

- *Polling*: in un loop a livello software, inviamo "blocco per blocco" i nostri dati alla periferica usando la dimensione della destinazione (potremmo inviare byte per byte oppure word per word ad esempio). È facilmente implementabile ma blocchiamo la CPU, e quindi non potremmo svolgere altri task
- *Interrupt*: indichiamo alla periferica di avviare un operazione che ci notificherà poi attraverso un interrupt la fine della transazione (più utile con le periferiche tipo DAC ma ha un grosso overhead visto che la CPU deve attivare tutto il processo di gestione degli interrupt).
- *DMA*: lo spostamento di dati viene effettuato programmando un controller hardware secondo delle regole di scheduling ben definite e senza l'intervento della CPU, che viene poi triggerato e temporizzato da una periferica (o dal bus in alcune configurazioni)

Nel nostro progetto il disegno a schermo deve essere una operazione "asincrona" rispetto al normale flusso di programma; quindi, possiamo escludere un'implementazione via polling. Dobbiamo ripiegare sul DMA.

Il DMA (**Direct Memory Access**) è un controller hardware con (nella configurazione più semplice)

- Due porte master connesse una al AHB per accedere alle periferiche e una al controller della memoria
- Una porta slave connessa al AHB per la programmazione da parte della CPU
- Un numero n di canali programmabili e indipendenti connessi alle varie periferiche
- Possibilità di assegnare diverse priorità ai canali per gestire l'arbitraggio tra richieste multiple
- Flow dei dati sia da memoria a periferica sia da periferica a memoria

Ogni MCU fornisce diversi controller DMA con diverse caratteristiche e connessioni.

[6] https://www.st.com/resource/en/application_note/dm00129215-extending-the-dac-performance-of-stm32-microcontrollers-stmicroelectronics.pdf - **Extending the DAC performance**

2.9.2 Scelta del DMA da usare e configurazione

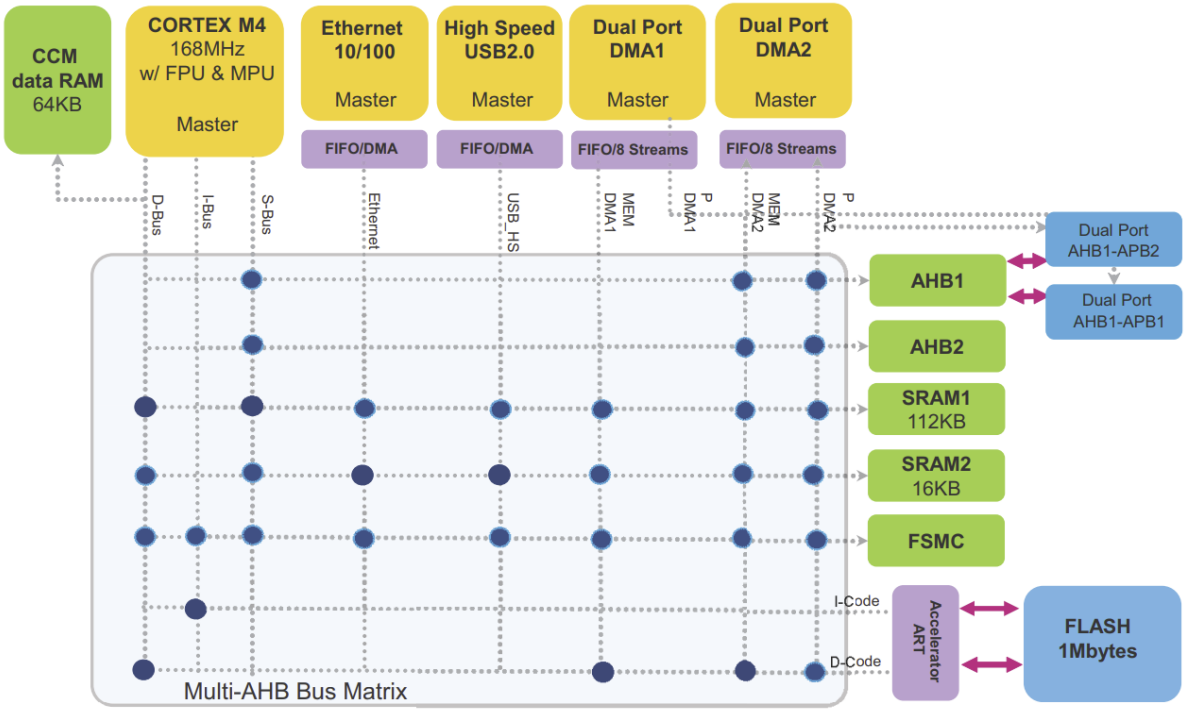


Figura 13 Schema connessioni al BusMatrix, ***AN4031***

Come già specificato, il nostro screen buffer sarà salvato in SRAM e dovrà essere mandato su una periferica esterna (GPIO o DAC); queste si trovano o sul AHB (GPIO) o collegate al APB1 (DAC). Dalla figura notiamo come il DMA1 non è collegato all' AHB, quindi, non può essere utilizzato, ci rimane solo il DMA2 che risulta invece comunicare con l'AHB, dal quale poi può raggiungere i due bus APB1 e APB2 tramite le *Dual Port*.
Una volta individuato il DMA, è necessario selezionare il canale dove arriveranno le richieste della nostra periferica; nel nostro caso il trigger del timer 1 (disponibile tramite stream 0 sul canale 6).
Il transfer mode sarà impostato a memory-to-peripheral: il DMA viene attivato ad ogni richiesta della periferica (trigger del timer), *se abilitato*, e trasferisce un numero di byte pari a quello indicato nel suo registro $SxNDTR$ caricando i dati necessari dalla memoria (vedremo successivamente l'utilizzo delle FIFO per migliorare le prestazioni); in questo modo viene rispettato (in linea teorica, ma scopriremo che non sarà così) il timing richiesto dal nostro pixel clock.

2.9.3 Gestione DMA tramite interrupt del sync orizzontale

Come già analizzato, per via delle limitazioni sulla memoria della nostra scheda, dobbiamo per forza usare un buffer di dimensioni pari all'area visibile; di conseguenza è necessario attivare e disattivare il DMA rispettivamente all' inizio e alla fine di ogni linea dello schermo. Per far ciò dobbiamo utilizzare i gestori dell'interrupt sui due timer di sincronizzazione, che vediamo qui in modo più specifico:

- ***Inizio e fine area visible verticale***: il timer genera due interrupt che vanno solamente a settare un flag per indicare se il display è nell' area di sync verticale. Questo verrà usato poi dal gestore di interrupt del sync orizzontale; forzando il fatto che deve essere l'altro sync a fare tutto il lavoro correttamente e riduciamo il tempo passato nell' interrupt (come vedremo, gli interrupt introducono delle latenze indirettamente anche sul DMA)
- ***Inizio e fine linea visibile orizzontale***: il timer genera sempre due interrupt
    - *Inizio area visibile*: è l'interrupt che deve essere gestito più velocemente (quando la linea finisce abbiamo il margine del porch + sync per fare il lavoro necessario) e quindi va solo ad abilitare il triggering del DMA sul timer (assume che il DMA sia già stato preparato ed abilitato dall' interrupt di fine linea per diminuire la latenza)
    - *Fine area visibile*: l'interrupt forza sempre la cancellazione del DMA e l'output a zero, questo perché il trasferimento potrebbe non essere ancora terminato ma non possiamo più inviare dati. Dopodiché il DMA deve essere preparato per un altro ciclo di lettura se non sta iniziando l' area di sync verticale: viene settato il puntatore alla linea corretta, ricaricato il numero di byte da trasferire

e viene abilitato solo il controller (vedremo che in questo modo i dati vengono precaricati nella FIFO dello stream DMA e al triggering saranno già pronti per essere inviati ai pin GPIO.

2.9.4 BusMatrix, performance e latenze[7]

Il multi-layer BusMatrix è un componente che permette a dei master di eseguire trasferimenti concorrenti finché non si hanno accessi sullo stesso slave; in questo caso l'arbitraggio è fatto attraverso uno schema round-robin.
È quindi indispensabile capire come DMA e BusMatrix interagiscono tra loro e con l'intero sistema per gestire eventuali ritardi e ottimizzare i nostri trasferimenti.
Quando dobbiamo visualizzare un pixel a schermo, le seguenti operazioni sono svolte dal DMA:

- ***Caricare*** il valore da visualizzare dalla memoria SRAM (dimensione del dato dipende dalla modalità di output), quindi attraversare il BusMatrix
- ***Inviare*** il valore alla periferica, passando sempre dal BusMatrix (non abbiamo la possibilità di passare direttamente dalla Dual Port)

Anche assumendo il nostro DMA dello schermo come unico controller attivo, lasciando la CPU interagire con altri dispositivi oppure anche lasciando a FreeRTOS il controllo (ed esempio eseguendo il task di default per lo scheduling), notiamo che la visualizzazione diventa un'immagine distorta, come se alcuni pixel fossero "in ritardo" rispetto agli altri.

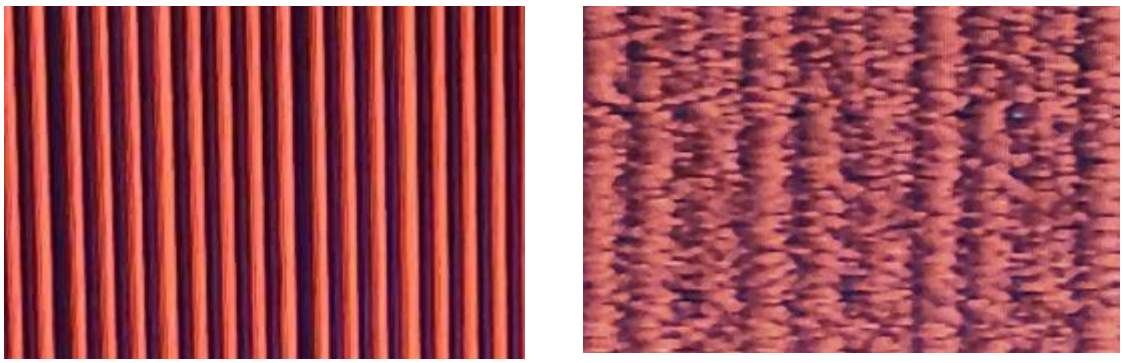


Figura 14 Barre rosse senza contesa sul BusMatrix (sinistra), indroducendo contesa sul BusMatrix (destra)

La prima causa è da cercare nella regione di memoria dove risiedono stack e heap, ovvero la SRAM: ad ogni accesso di variabile che non è contenuta in un registro stiamo andando a bloccare il BusMatrix! Dobbiamo quindi, a discapito di avere meno spazio a disposizione, spostare tutto nella CCMRAM, ovvero la ***Core Coupled Memory***, che risulta essere collegata direttamente al CortexM senza passare dal BusMatrix (risultato ottenibile modificando il file di linking del progetto).
Abbiamo anche una limitazione relativa all' architettura e quindi possiamo solo tentare di diminuirne gli effetti: nell' application note relativo al DMA si fa notare che, per ridurre le latenze, la CPU prende possesso dell'BusMatrix (presumibilmente l'intero bus, non una singola periferica) ogni volta che operazioni di load/store vengono eseguite oppure ad ogni gestione dell'interrupt. Quindi, ad esempio, quando FreeRTOS tenta di schedulare un task richiamando una syscall che genera un interrupt oppure quando interagiamo a lungo con una periferica, il bus risulta bloccato e il DMA non riesce ad inviare correttamente i dati.
Da qui capiamo che:

- ***FreeRTOS deve essere usato il meno possibile*** quando il DMA è attivo, altrimenti lo scheduler introduce latenze per l'accesso al bus. Di conseguenza non possiamo neanche usare la funzione osDelay().
- ***Mai accedere ad una periferica in polling*** quando lo schermo è attivo: se possibile usare un interrupt per minimizzare il tempo in cui il BusMatrix risulta bloccato.

Abbiamo inoltre altri fattori che dobbiamo considerare, come ad esempio il delay del DMA stesso nel trasferire i dati dalla memoria alla periferica. Quanto tempo impiega questa operazione? La risposta arriva sempre dell'application note relativa al DMA.

[7] https://www.st.com/resource/en/application_note/dm00046011-using-the-stm32f2-stm32f4-and-stm32f7-series-dma-controller-stmicroelectronics.pdf - **Specifiche sulle latenze del BusMatrix, Application Note AN4031**

Description	Latency
t_{MA}: DMA memory port arbitration	1 AHB cycle
t_{MAC}: memory address computation	1 AHB cycle
t_{BMA}: bus matrix arbitration (when no concurrency)(1)	1 AHB cycle(2)
t_{SRAM}: SRAM read or write access	1 AHB cycle

Description	Through bus matrix		DMA's direct paths
	To AHB peripherals	To APB peripherals	
t_{PA}: DMA peripheral port arbitration	1 AHB cycle	1 AHB cycle	1 AHB cycle
t_{PAC}: peripheral address computation	1 AHB cycle	1 AHB cycle	1 AHB cycle
t_{BMA}: bus matrix arbitration (when no concurrency)(1)	1 AHB cycle	1 AHB cycle	N/A
t_{EDT}: effective data transfer	1 AHB cycle(2) (3)	2 APB cycles	2 APB cycle
t_{BS}: bus synchronization	N/A	1 AHB cycle	1 AHB cycle

*Figura 15 Latenza del DMA, **AN4031***

Come possiamo vedere dalle immagini, nel migliore dei casi il trasferimento per la memoria e l'accesso alla periferica occupano quattro cicli dell'AHB ciascuno. Da qui ne ricaviamo che per avere una temporizzazione giusta e non perdere dati, il nostro AHB deve viaggiare ad una velocità di almeno quattro volte superiore a quella del pixel clock (nel migliore dei casi dove non ci sono contese nel BusMatrix).
Per ridurre il numero di accessi alla memoria, ogni stream DMA dispone di un buffer FIFO grande quattro word dove possono essere precaricati concorrentemente i dati in modo da poter soddisfare una richiesta il prima possibile. Nel nostro caso possiamo ottimizzare in questo modo:

- ***Caricamento di word dalla memoria***: abilitando la coda FIFO, abbiamo la possibilità di fare dell'unpacking di dati quando scriviamo sulla periferica; quindi, invece di caricare byte per byte dalla memoria e scriverlo sulla GPIO, possiamo caricare word intere nella nostra FIFO che poi verranno inviate un byte alla volta al bus ad ogni richiesta del DMA. In questo modo con un solo accesso alla SRAM carichiamo 4 byte (possibile perchè la larghezza del bus corrisponde o è più grande alla larghezza della nostra richiesta).
- ***Abilitazione del DMA con un certo anticipo rispetto all' arrivo delle richieste***: quando il DMA viene abilitato, l'hardware inizia a leggere i dati e riempire la coda FIFO anche se non arrivano richieste di trasferimento. Questo preloading è molto utile per quando dobbiamo disegnare i primi pixel; infatti, l'interrupt di fine linea deve preparare e abilitare il DMA in modo da essere pronto a servire immediatamente le richieste che arriveranno.

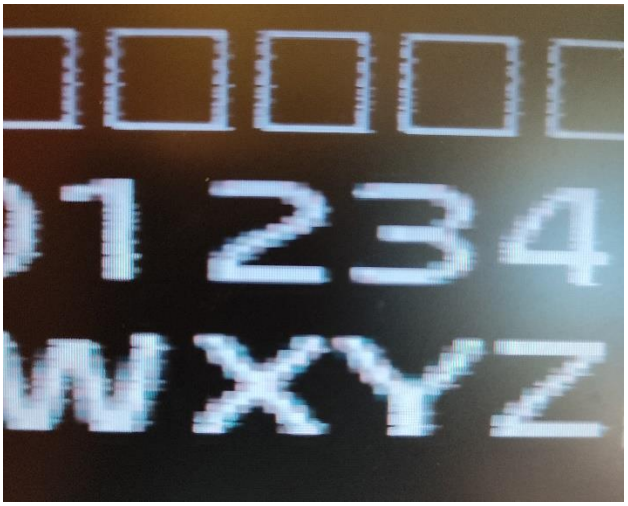


Figura 16 Visualizzazione quando il bus gira solo 3 volte più veloce del pixel clock. Possiamo notare come molti dei pixel dei caratteri risultino "in ritardo". Nella foto non è visibile, ma l'immagine risulta anche allungata orizzontalmente

2.9.5 *Risultato finale*

Una volta completato il DMA, siamo pronti a visualizzare delle immagini (che caricheremo dalla nostra scheda SD).

Figura 17 Risultato finale visualizzazione immagine a schermo

Come è possibile notare dalla figura, la risoluzione in termini di pixel e bit per pixel non aiuta con la qualità ma l'output è stabile per la maggior parte del tempo (rispetto all' area visibile). Osservando l'immagine dal vivo si possono notare alcuni artefatti probabilmente dovuti all' interrupt per il clock di sistema (che è attivato ogni millisecondo).
Nella schermata principale è stato invece inserito un bordo rosso di 1 pixel che, senza regolazioni da parte dello schermo, dovrebbe essere totalmente visibile per 3 lati, mentre il corrispondente a fine linea è fuori dal tempo che la risoluzione impone. Abbassando il threshold per il riempimento della FIFO del nostro DMA, sembra diminuisca il ritardo ma si crea una sorta di bordo allungato sempre sull' ultimo pixel, pur forzando l'output ad un livello low quando termina la linea. I risultati su altri schermi possono essere diversi.

Figura 18 Immagine di un tramonto (FullHD) disegnata a schermo dall' STM32F407

3 CRC[8]

Il CRC, acronimo di *Cyclic Redundancy Check,* è uno dei tanti codici usati per il rilevamento di inconsistenza di un messaggio basato sulla costruzione di un valore chiamato *checksum* (che sarà funzione dei dati in uscita). Può essere di varie dimensioni, indipendentemente da come viene calcolato (nel nostro caso avremo bisogno di un checksum con dimensioni 7 bit e uno con dimensioni di 16 bit), che vengono solitamente esplicitate nel nome. Parleremo quindi più in generale di un certo CRC-n dove n è la dimensione del checksum.

Assumendo quindi che le due parti della comunicazione usino la stessa funzione di CRC, in linea generale abbiamo che chi invia il messaggio, calcola il CRC e lo accoda alla trasmissione, mentre chi riceve calcola il CRC dei dati utente e confronta il valore con quello aggiunto al messaggio. Se checksum inviato e calcolato corrispondono, allora abbiamo buone probabilità che nulla sia stato alterato durante la trasmissione.

3.1 Algoritmo

Il calcolo di un CRC è basato sull' uso di codici ciclici: l'input, visto come un polinomio viene diviso da un *polinomio generatore* e il resto di questa operazione diventa il nostro checksum. I coefficienti sono calcolati secondo l'aritmetica di un campo finito.

3.2 Implementazione di CRC – Shift Registers

Nella quasi totalità dei casi, i dati in input del nostro CRC non possono essere contenuti in un tipo nativo gestito dal processore e di conseguenza non possiamo usare il classico operatore di modulo "input % generatore". L' algoritmo deve essere eseguito step-by-step impiegando dei registri di shift (presenti soprattutto a livello hardware, ma il concetto viene poi anche reso visibile lato software). Uno shift register è un registro di dimensione fissata e il suo contenuto può essere spostato di un bit alla volta verso destra (rimosso il LSB) o verso sinistra (rimosso MSB). Il processo di divisione in modulo rivisitato si riassume quindi in questi step:

1. Inizializzazione del registro al valore 0.
2. Shiftiamo lo stream di input di un bit verso sinistra nel registro. Se il MSB uscente è 0 non accade nulla, altrimenti applichiamo lo XOR con il polinomio generatore (divisione).
3. Ripetiamo il passo precedente fino a che lo stream di bit non è stato consumato. Alla fine, troveremo nel registro il nostro CRC.

3.3 Implementazione di CRC – Codice

A livello di codice, solitamente vengono precalcolati tutti i 256 possibili valori per poi essere adeguatamente "aggiunti" quando viene consumato il buffer di cui calcolare il checksum.

```
static unsafe void GenerateCrcTable<T>(T generatorPoly) where T : IBinaryInteger<T> {
    T[] table = new T[256];     // Allocation of the table
    // Calculates the shift necessary to put our byte value in the 8 MSB of the "shift register"
    // It is the same to put the byte in the LSB and perform the loop below for the entire
    // size of the type T. The first "size - 8" bits will be 0 and the loop will result in just
    // a right shift
    const int shift = (sizeof(T) * 8) - 8;
    // We calculate the MSB mask to check if the "next" shift will pop out a 1 or a 0
    T MSBMask =  T.Create((0x80) << ((sizeof(T) - 1) * 8));
    for (int i = 0; i < 256; i++) { // Loop over the possible byte values
        T crc = T.Create(i) << shift; // Shift in the MSBits
        // Loop 8 times to consume all the bits of our value
        for (int bit = 0; bit < 8; bit++) {
            // if the outgoing bit is a 1, we perform or XOR division
            if ((crc & MSBMask) != T.Zero) {
                crc <<= 1;
                crc ^= generatorPoly;
            } else {
                crc <<= 1;
            }
        }
        table[i] = crc;
    }
}
```

Figura 19 Codice esempio per la generazione di una tabella di CRC

[8] http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html - **Descrizione CRC**

Il codice per il calcolo della mappa di valori mantiene sempre la stessa struttura (a meno di particolari modifiche con CRC di dimensioni non "native" come il CRC-7 usato dai comandi SD)

- *Allocazione* della tabella, che sarà di dimensioni $256 * sizeof(CRC)$ bytes (dato a cui dobbiamo prestare attenzione in caso di limitata memoria disponibile)
- $\forall\, b \in [0, 255]$
    - Se il bit in uscita del byte da b è 1, aggiorniamo il crc con lo XOR dei dati
    - Altrimenti (bit in uscita = 0) non facciamo nulla
    - Ripetiamo ciclo 8 volte

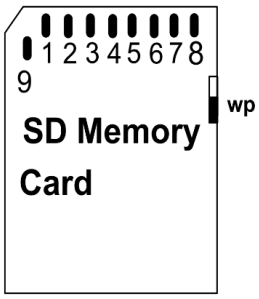
La combinazione di un certo crc con un byte risulta molto semplice:

- *Allineamento*: CRC e byte in input vengono allineati alla stessa dimensione di parola se necessario
- *XOR e recupero nuovo CRC*: ai due valori allineati viene applicato un operatore di xor e viene recuperato il nuovo checksum dalla tabella
- *Correzione MSB*: se è stato necessario eseguire un allineamento, viene fatto un ulteriore xor tra il CRC in input escluso del MSB con il nuovo CRC

4 SD Card

Le SD (acronimo di Secure Digital) sono una delle più utilizzate (ancora oggi) schede di memoria non volatile disponibili su mercato. Ne esistono di diverse versioni (anche in base al periodo di produzione) con diversi protocolli di interfacciamento ma per il progetto si è scelto di adottare la comunicazione attraverso lo standard SPI per una questione di semplicità del protocollo e di disponibilità della documentazione (SD mette a disposizione una versione semplificata dello standard gratuitamente).

4.1 Descrizione fisica



Pin #	SPI Mode		
	Name	Type[1]	Description
1	CS	I[3]	Chip Select (neg true)
2	DI	I	Data In
3	VSS	S	Supply voltage ground
4	VDD	S	Supply voltage
5	SCLK	I	Clock
6	VSS2	S	Supply voltage ground
7	DO	O/PP	Data Out
8	RSV		
9	RSV		

Figura 20 Descrizione pin di una scheda SD, ***Physical Layer Simplified Specification***

Una scheda SD (nella configurazione standard a cui facciamo riferimento) si presenta con nove pin di "interfaccia" e un selettore per la protezione dalla scrittura. Oltre ai classici pin di alimentazione troviamo due pin dei dati (input e output) più uno di *card selection* usato per identificare una specifica scheda attaccata su un bus condiviso. In alcuni slot SD è disponibile un segnale di *Card Detection* (CD) per stabilire quando una card viene collegata o scollegata.

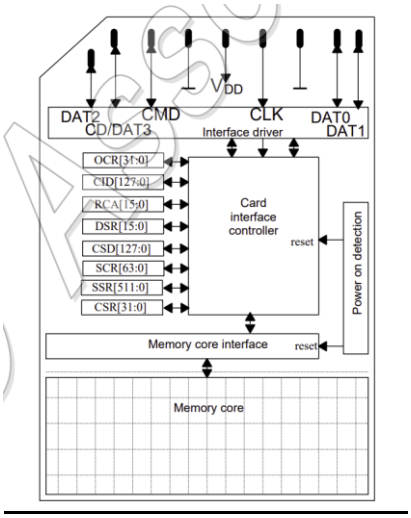


Figura 21 Schema di una scheda SD, ***Physical Layer Simplified Specification***

Ogni scheda SD contiene al suo interno un set di registri che ne descrivono diversi parametri / caratteristiche di cui parleremo successivamente. Quello che per noi è importante è sapere che la card, oltre ad i vari protocolli di comunicazioni "standard", può supportare anche lo schema SPI. Prima di poter comunicare però è necessario accendere la card con una procedura specifica, detta *power cycle*.

4.2 Power cycle

Per specifiche della SD Association, una scheda SD deve avere un circuito per riconoscere l'accensione e che permetta al controller di disporsi in uno stato valido. Non esistono particolari segnali di reset ma la procedura di accensione deve essere rispettata, per poi poter inizializzare la comunicazione SPI.

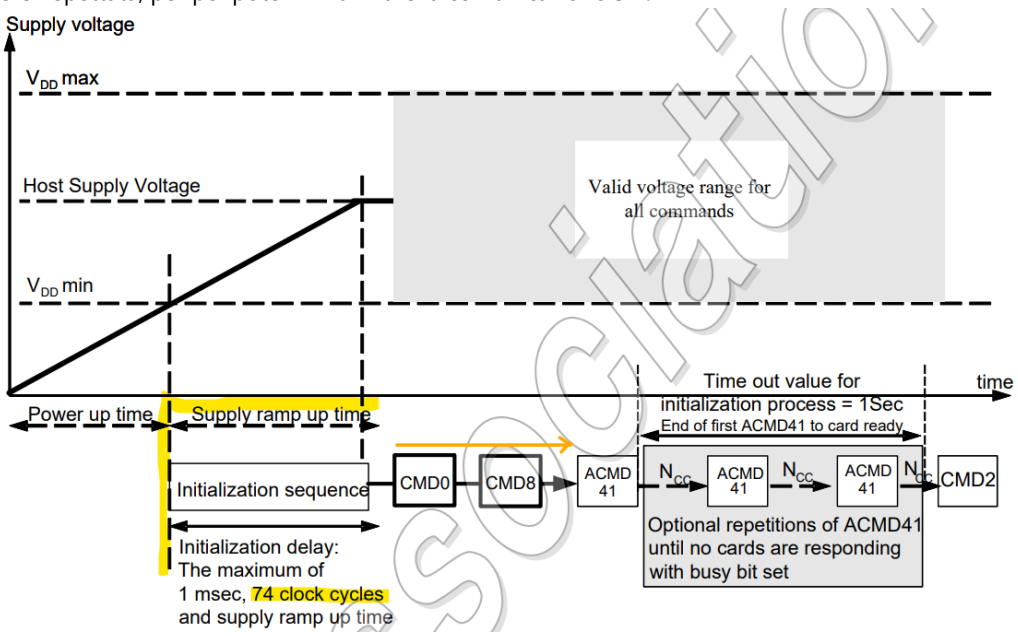


Figura 22 Specifiche lato SD per i tempi di accensione, ***Physical Layer Simplified Specification***

Come possiamo vedere dalle specifiche che riguardano il lato SD, esistono tempi ben definiti per i quali il controller deve essere pronto a ricevere i comandi.

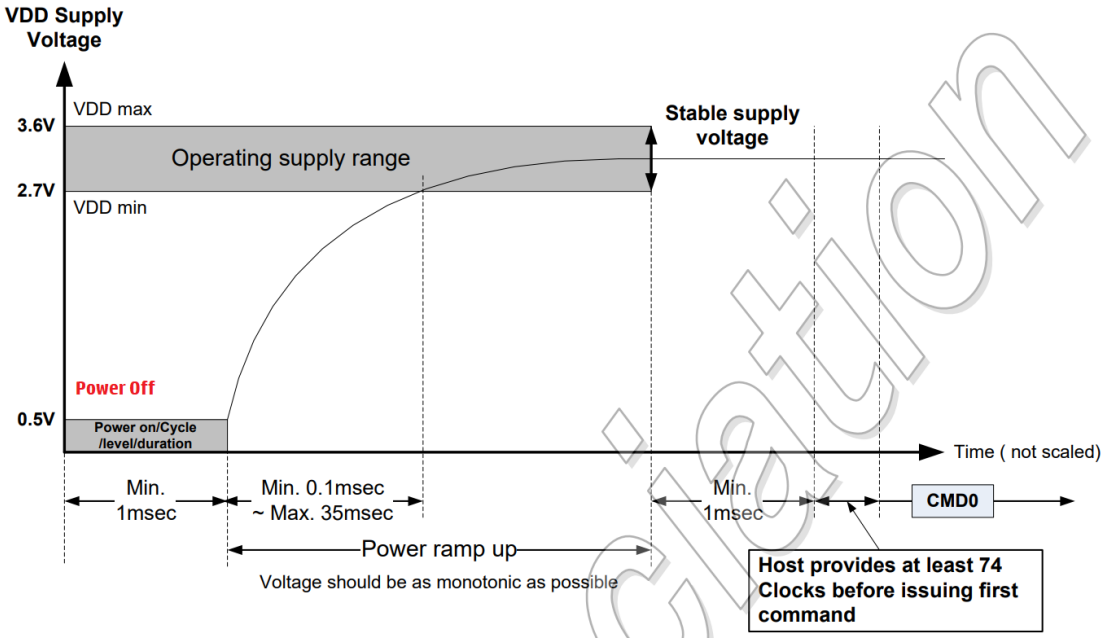


Figura 23 Schema di accensione da seguire per l'host, ***Physical Layer Simplified Specification***

La specifica dichiara precisamente anche tutto il ciclo che l'host deve applicare per un power cycle completo:

- ***Power Off***: Il voltaggio in input sulla SD deve essere mantenuto < di 0.5V per almeno 1ms. È consigliato tenere anche le altre linee a *low* o disconnesse.

- ***Power Ramp Up***: Il voltaggio in input sulla SD, più monotonamente possibile, deve raggiungere il range operativo nei limiti consentiti. Una volta nell' intervallo del VDD, dobbiamo aspettare almeno $1\ ms$ per la stabilizzazione. Secondo le specifiche del nostro MCU, un pin a velocità minima non rientra nel termine imposto, ma non sembra essere limitante per le schede SD con cui sono stati effettuati i test.

OSPEEDRy [1:0] bit value(1)	Symbol	Parameter	Conditions	Min	Typ	Max	Unit
00	$f_{max(IO)out}$	Maximum frequency(3)	C_L = 50 pF, V_{DD} > 2.70 V	-	-	4	MHz
			C_L = 50 pF, V_{DD} > 1.8 V	-	-	2	
			C_L = 10 pF, V_{DD} > 2.70 V	-	-	8	
			C_L = 10 pF, V_{DD} > 1.8 V	-	-	4	
	$t_{f(IO)out}$/ $t_{r(IO)out}$	Output high to low level fall time and output low to high level rise time	C_L = 50 pF, V_{DD} = 1.8 V to 3.6 V	-	-	100	ns

Figura 24 Tempi rampa voltaggio su un pin configurato a velocità minima, ***STM32F407 Datasheet***

- **Clock cycles**: l'host deve fornire almeno 74 cicli di clock prima di inviare il primo comando.

Una volta completato il power cycle, possiamo inizializzare la scheda con una specifica sequenza di comandi.

4.3 *Inizializzazione*

L' inizializzazione di una scheda SD, sia in interfaccia SPI che non, deve seguire un preciso schema, sempre stabilito dalla SD Association.

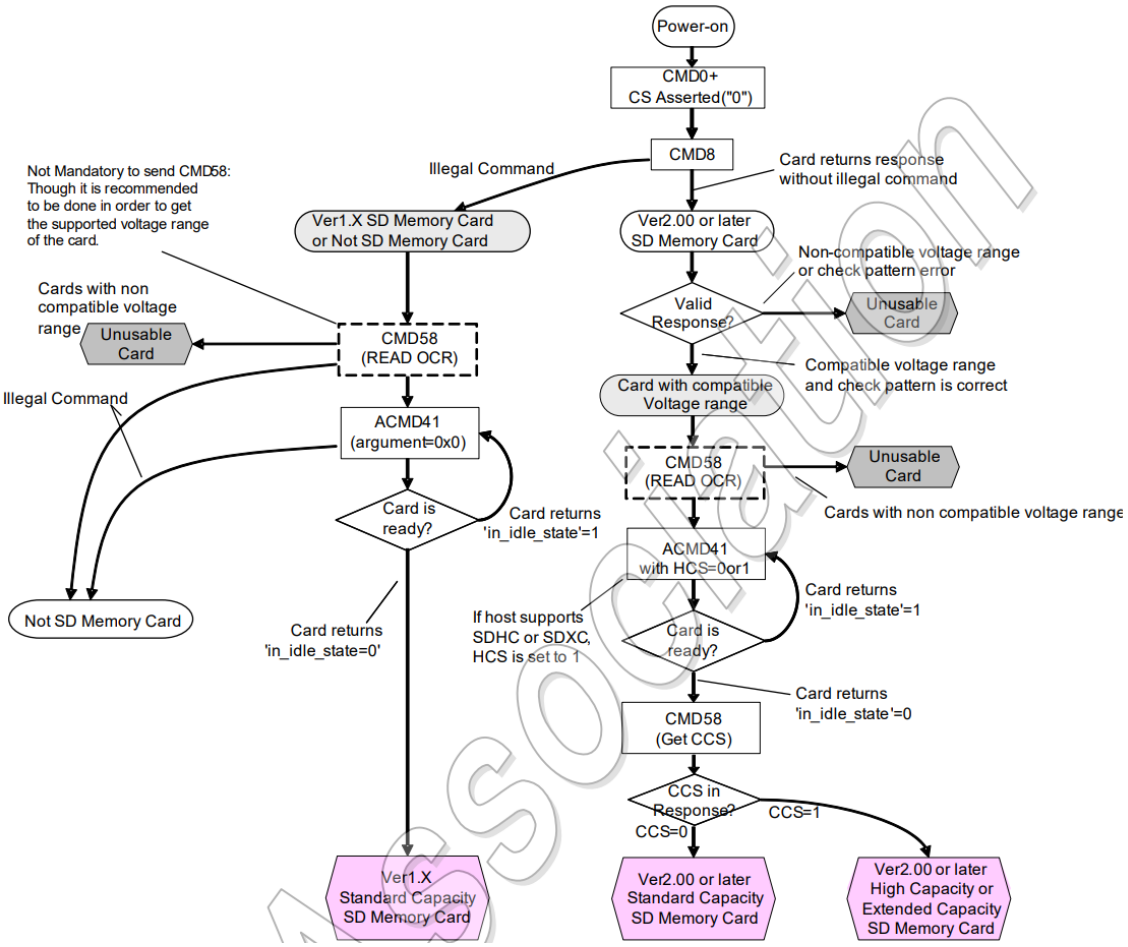


Figura 25 Initialization flow per una scheda SD in modalità SPI, ***Physical Layer Simplified Specification***

Per entrare in SPI mode, l'host deve inviare un **CMD0** con il CS messo a low e *deve ricevere* una risposta dalla scheda, altrimenti la modalità SPI non è supportata (o non abbiamo connesso un dispositivo al bus, ricordiamo che nel power cycle non inviamo nessun comando).

Una volta entrati in modalità SPI, dobbiamo inviare un **CMD8** per verificare le condizioni di funzionamento; in base alla risposta possiamo anche stabilire la versione di SD card con cui stiamo parlando (o eventualmente escludere che stiamo comunicando con dispositivo supportato). Nel comando è specificato il range di voltaggio e la risposta, in caso di scheda v2.00+, è una replica dei dati appena inviati per verificare che non ci siano interferenze o problemi nella comunicazione.
Opzionalmente, possiamo anche inviare un **CMD58** per leggere il registro OCR della scheda e verificare bit per bit il voltaggio supportato. Alcuni field non avranno significato non avendo ancora inizializzato la memory card.
L' inizializzazione vera e propria avviene mandando in loop l'**ACMD41** e attendendo che il bit *Idle* della risposta sia messo a 0. Una volta terminato il procedimento, possiamo richiedere nuovamente l'OCR per stabilire il CCS (*Card Capacity Status*) se necessario, per poter poi derivare la modalità di indirizzamento della memoria interna.
Nell' immagine dell'initialization flow non è specificato, ma prima di mandare l'**ACMD41** è possibile inviare un **CMD59** per abilitare/disabilitare il controllo del CRC sui comandi. Discuteremo dei registri successivamente.
Secondo ChaN, tutto il power cycle deve essere effettuato con una frequenza SPI nel range di 100-400KHz, mentre poi deve essere portata al massimo consentito dalla scheda.

4.4 SPI

L' SPI (*Serial Peripheral Interface*) è, assieme all' I^2C, uno dei protocolli standard più usati per la comunicazione seriale, sincrona e full duplex tra un controller master (nel nostro caso il microcontrollore) e degli slave (scheda SD).

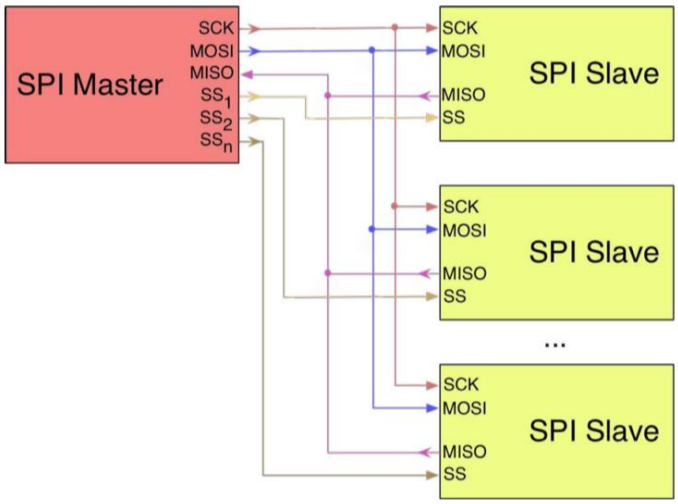


Figura 26 Schema di un bus SPI, ***Noviello***

Il bus è formato da quattro segnali

- ***Clock*** (SCK): sincronizza la comunicazione tra master e slave ed è generato dal master. La velocità può essere di decine di MHz e dispositivi diversi sullo stesso bus possono usare diversi clock.
- ***Master Output Slave Input*** (MOSI): canale usato dal master per l'invio dei dati ad uno slave.
- ***Master Input Slave Output*** (MISO): canale usato dallo slave per l'invio dei dati ad un master.
- ***Slave Select*** (SSx): quando viene messo a *low*, seleziona il dispositivo avviando effettivamente la comunicazione. Gli altri apparati che avranno invece un segnale alto in input ignorano i segnali in arrivo.

La trasmissione inizia con la generazione del segnale di clock e la selezione tramite *SSx*. Sia per master e per slave, il byte da inviare è salvato in un registro interno all' hardware; ad ogni ciclo il MSB viene quindi rimosso ed inviato nel corrispettivo LSB dall' altra parte della comunicazione. La trasmissione termina quando il clock viene disabilitato e liberato il SSx.
Notiamo come per ricevere dei byte il master deve "inviare" comunque lo stesso numero di dati attraverso l'uso di un *"dummy byte"* (nel nostro caso sarà 0xFF): in questo modo il registro del master si "svuota" e lascia spazio per i dati dello slave.

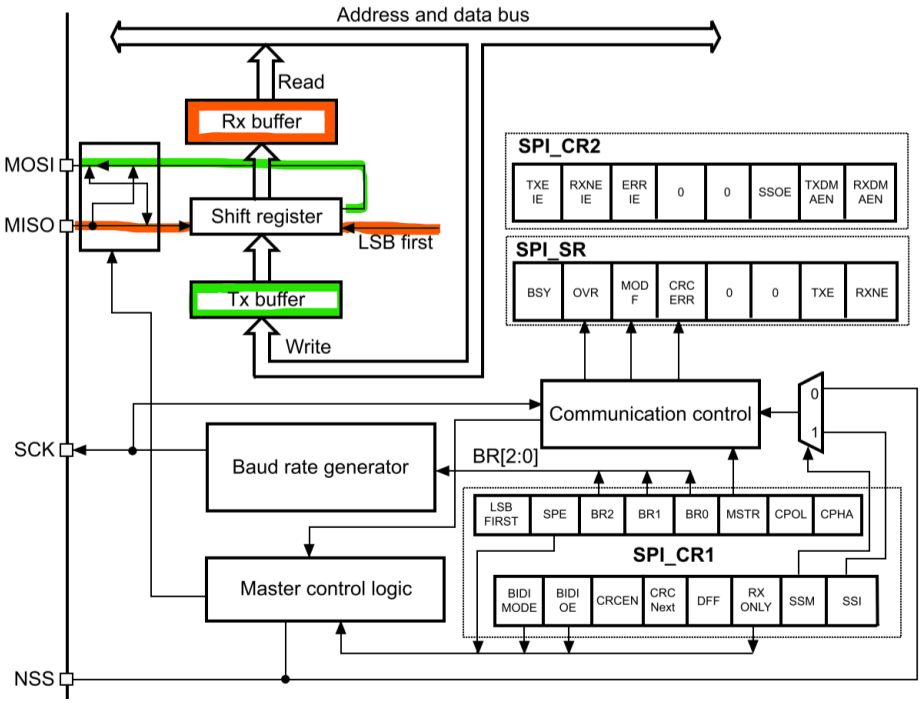


Figura 27 Schema hardware di un dispositivo SPI, ***RM0090***

4.5 Invio dei comandi e risposte

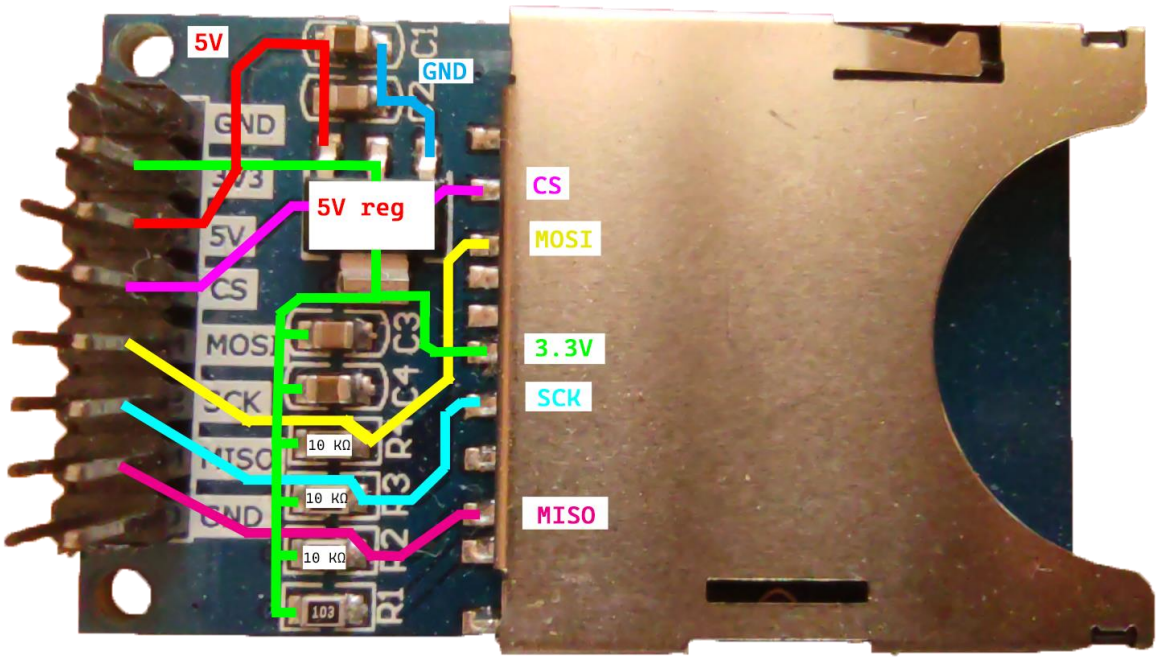


Figura 28 Connessioni connettore SD

Come abbiamo già visto, in un trasferimento I^2C il master avvia la comunicazione applicando sul bus una condizione di START; in SPI non abbiamo invece nulla che indica l'inizio della trasmissione. Notiamo inoltre che nel connettore le linee di dati e di clock solo in una configurazione di pull-up; quindi, anche senza client attaccato, leggeremo sempre il nostro *dummy byte*: 0xFF (stesso discorso vale dal punto di vista dello slave). Da qui capiamo che per indicare l'inizio di una trasmissione, almeno un bit del primo byte trasmesso/ricevuto deve essere 0.

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'1'	x	x	x	'1'
Description	start bit	transmission bit	command index	argument	CRC7	end bit

Figura 29 Composizione di un comando inviato alla SD tramite SPI, ***Physical Layer Simplified Specification***

Tutti i comandi che inviamo hanno la struttura presentata in figura: il command index dipende dal comando stesso (CMD0, index = 0; CMD1, index = 1; ecc) e l'argomento varia in base alla funzione che stiamo richiamando.
Ugualmente, le risposte sono nella quasi totalità dei casi di tipo *R1* oppure hanno *R1* come prefisso; anche qui un bit del primo byte è forzato a 0.

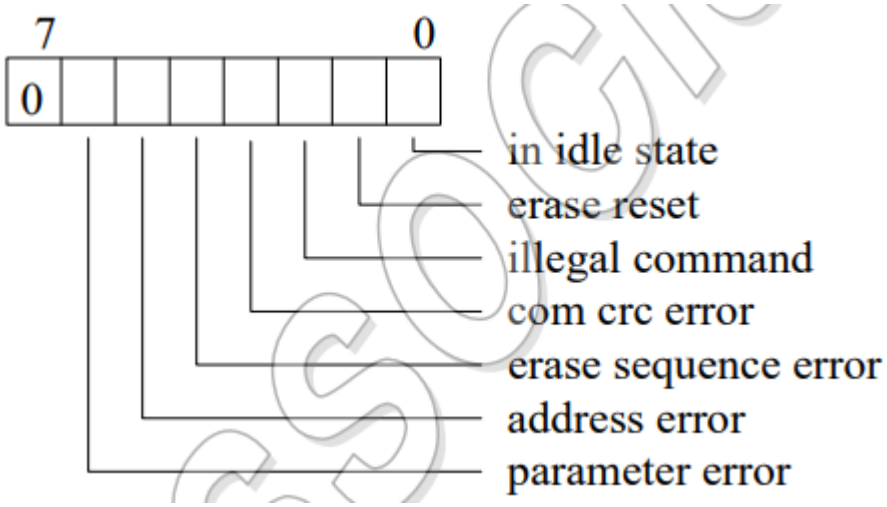


Figura 30 Struttura byte risposta principale, ***Physical Layer Simplified Specification***

Il controller della scheda SD potrebbe metterci del tempo a rispondere; di conseguenza, dobbiamo continuare a mandare dummy byte fino a che non riceviamo qualcosa di diverso da 0xFF sul bus. Per la lettura dei dati dalla memoria il ragionamento è lo stesso: viene inviato un comando (**CMD17**) contenente l'indirizzo dei dati da leggere (allineato per byte o per settore in base alla card capacity); la scheda SD prima risponderà al comando con la relativa risposta R1 e poi invierà un ***data/error token*** seguito dai dati e da 16 byte di CRC in caso di successo. Come abbiamo già analizzato, l'unico modo per essere sicuri che stiamo leggendo dei dati veritieri e non solo dummy bytes dopo il data token, è controllare il CRC.

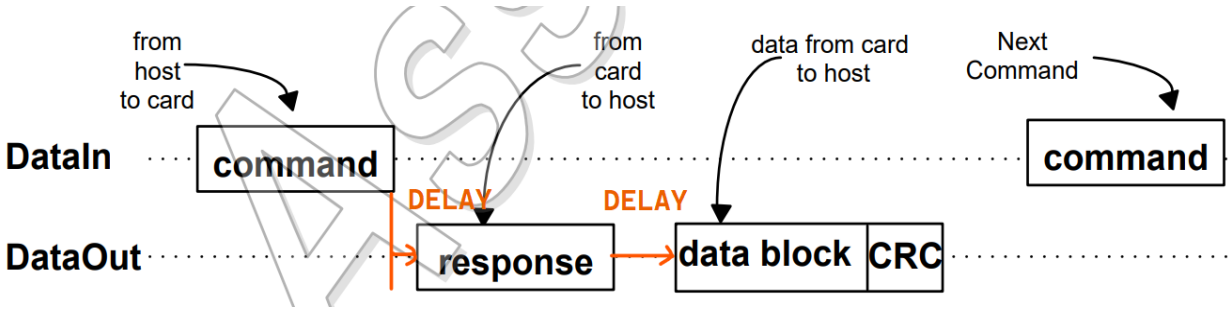


Figura 31 Flow dei dati per la lettura di un blocco di memoria dalla scheda, ***Physical Layer Simplified Specification***

4.6 Registri

Abbiamo già parlato di come comunichiamo con una scheda SD e di come si applica una corretta procedura di inizializzazione. Dobbiamo solo capire come leggere ed estrapolare i dati dai registri indicati nei vari step.
Per la lettura, il valore del registro può essere inviato direttamente nella risposta (OCR) oppure inviato come un normale data block (CID, CSD).
L' OCR (*Operation Condition Register*) è un registro su 32 bit che memorizza i profili del VDD supportati e, una volta terminata l'inizializzazione, le informazioni sulla capacità (*Card Capacity Status* bit).

OCR bit position	OCR Fields Definition
0-3	reserved
4	reserved
5	reserved
6	reserved
7	Reserved for Low Voltage Range
8	reserved
9	reserved
10	reserved
11	reserved
12	reserved
13	reserved
14	reserved
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24[3]	Switching to 1.8V Accepted (S18A)
25-26	reserved
27[4]	Over 2TB support Status (CO2T)
28	reserved
29	UHS-II Card Status
30	Card Capacity Status (CCS)[1]
31	Card power up status bit (busy)[2]

VDD Voltage Window

1) This bit is valid only when the card power up status bit is set.
2) This bit is set to LOW if the card has not finished the power up routine.
3) Only UHS-I card supports this bit.
4) Only SDUC card supports this bit.

Figura 32 Struttura registro OCR, ***Physical Layer Simplified Specification***

Un altro registro che dobbiamo leggere per stabilire la massima velocità del clock che possiamo usare è il CSD (*Card-Specific Data*): è differenziato in base al tipo di Card Capacity della scheda ma la struttura base è la stessa.

Name	Field	Width	Value	Cell Type	CSD-slice
CSD structure	CSD_STRUCTURE	2	00b	R	[127:126]
reserved	-	6	00 0000b	R	[125:120]
data read access-time-1	TAAC	8	xxh	R	[119:112]
data read access-time-2 in CLK cycles (NSAC*100)	NSAC	8	xxh	R	[111:104]
max. data transfer rate	TRAN_SPEED	8	32h or 5Ah	R	[103:96]
card command classes	CCC	12	01x110110101b	R	[95:84]
max. read data block length	READ_BL_LEN	4	xh	R	[83:80]
partial blocks for read allowed	READ_BL_PARTIAL	1	1b	R	[79:79]
write block misalignment	WRITE_BLK_MISALIGN	1	xb	R	[78:78]
read block misalignment	READ_BLK_MISALIGN	1	xb	R	[77:77]
DSR implemented	DSR_IMP	1	xb	R	[76:76]
reserved	-	2	00b	R	[75:74]

Figura 33 Struttura registro CSD per la Standard Card Capacity, ***Physical Layer Simplified Specification***

Notiamo come il transfer rate possibile è fissato a rispettivamente 25MHz o 50MHz per High-Speed mode (il calcolo del valore della frequenza a partire dal byte segue una piccola tabella che omettiamo ma è disponibile nelle

specifiche); inoltre vediamo che, siccome l'indirizzamento è riferito al byte nelle SD Standard Capacity, è possibile (se abilitato) leggere blocchi parziali e disallineati. Nelle versioni successive, l'indirizzamento è sul settore, quindi, non è più possibile fare letture disallineate.
Un ultimo registro che è giusto menzionare è il CID, *Card Identification Register*, che contiene tutte le informazioni sul produttore e ID della scheda. Non contiene dati utili per il funzionamento/comunicazione.

5 Struttura e implementazione del progetto

5.1 SD Layer

Il layer per la comunicazione con la scheda SD è implementato nel file *sd.c* e mette a disposizione delle funzioni per la connessione, disconnessione, power cycle e per la lettura di uno o più blocchi sequenziali.
Il modulo deve essere opportunamente inizializzato per salvare l'interfaccia SPI e GPIO sulle quali inviare i comandi. L'assunzione è quella che i parametri della SPI siano già impostati correttamente (nel codice sono presenti delle asserzioni che in fase di debug verificano che siano rispettate le specifiche) perché si dovrà poi solo cambiare il prescaler del clock una volta terminata la procedura di inizializzazione della scheda. La lettura di settori multipli è differenziata da quella singola perché abbiamo a disposizione un comando aggiuntivo che simula uno "stream" dei dati fino a che non viene inviato un comando di stop.
Tutte le funzioni di interfaccia pubblica restituiscono un valore dell'enumerazione `SdStatus` per descrivere l'esito di quello che è stato eseguito.
Il calcolo e controllo di CRC viene abilitato (se possibile) sui comandi in entrata nella SD e viene sempre applicato quando vengono letti dei dati (ove disponibile).
Il layer SD verrà chiamato dallo stub di "Storage Device Control" generato dal CubeIDE, utilizzato da FatFs.

5.2 FatFs Layer (from ChaN)[9]

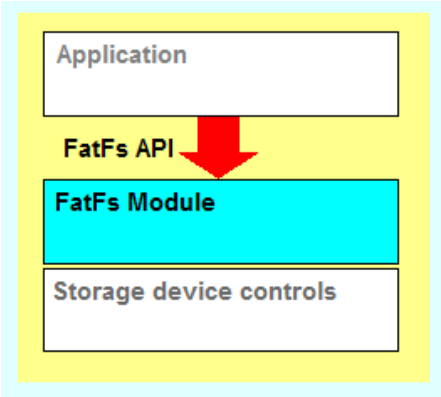


*Figura 34 Struttura FatFs API, **ChaN***

FatFs mette a disposizione per l'applicazione diverse funzioni per l'accesso ad un filesystem di tipo FAT (nelle varie versioni rilasciate nel tempo). Il modulo è completamente configurabile e nel nostro caso sono state disabilitate tutte le parti di codice inerenti a scrittura/cancellazione su file system; i nostri casi d' uso sono solamente l'enumerazione dei file di una cartella e la lettura di questi.
Il codice si interfaccia con un altro modulo chiamato *Storage Device Control* che deve implementare le funzioni di accesso al media da cui vogliamo leggere i dati; nel nostro caso sono implementate solamente le chiamate **`disk_initialize`** e **`disk_read`**; che CubeIDE definisce già per offrire un ulteriore livello di astrazione. La nostra implementazione è visibile nel file *user_diskio.c*.
Il codice delle due funzioni risulta lineare una volta incluso il layer SD:

- **`disk_initialize`**: applica un ciclo di accensione alla SD (per resettarla in caso di errore) e tenta la riconnessione
- **`disk_read`**: in base al numero di settori da leggere, chiama la relativa funzione di lettura da SD.

L' eventuale codice di errore restituito dalle funzioni del driver viene convertito in generico errore per il layer di controllo.

[9] http://elm-chan.org/fsw/ff/00index_e.html - **FatFs homepage**

5.3 VGA management e screen buffer

Il file *vgascreenbuffer.c* contiene il codice per la gestione, creazione e visualizzazione del nostro buffer a schermo. Tutto il contesto necessario è salvato nella struttura `VgaScreenBuffer` che "estende" la struttura `ScreenBuffer`.
La funzione `VgaCreateScreenBuffer()` implementa la logica di preparazione necessaria: deve scalare tutti i tempi dei pixel per poi allocare il frame buffer e settare i contatori dei timer. Il modulo mette poi a disposizione le funzioni per far partire, stoppare e sospendere l'output; a differenza dello stop, la sospensione blocca semplicemente il DMA lasciando però i timer attivi in modo da non disconnettere lo schermo.
Nel settaggio dei timer dobbiamo fare tre operazioni:

- ***Impostare il timer principale (pixel clock)***: viene calcolato il prescaler necessario per ottenere il pixel clock dalla frequenza dell'*APB1* (ottenibile tramite `HAL_RCC_GetPCLK1Freq() * 2`) e viene impostato come *ARR*. Questo timer farà da trigger per il timer del sync orizzontale abilitando il trigger output sull' update event.
- ***Impostare il timer del sync orizzontale***: l'*ARR* viene impostato sul numero di pixel di un intera linea; il primo canale genera il segnale effettivo dell'hsync mentre il secondo triggera il timer del sync verticale sul back porch del segnale. Gli ultimi due canali vengono impostati per generare un interrupt all'inizio e alla fine dell'area visibile di una linea. Tutti i counter sono calcolati sommando i relativi tempi di pixel già scalati in precedenza. Anche qui il timer ha abilitato il trigger output sull' evento di *output compare* del canale che fa da clock.
- **Impostare il timer di sync verticale**: viene fatto lo stesso processo del sync orizzontale ma un canale non verrà utilizzato (non abbiamo più un clock in uscita che ne triggera un'altro)

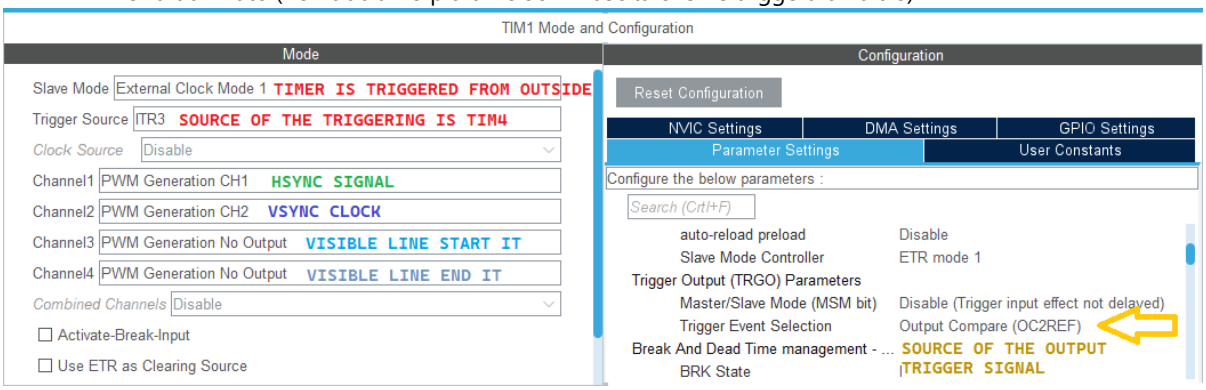


Figura 35 Configurazione del timer designato alla generazione del sync orizzontale

Una volta impostati i timer, viene infine configurato il DMA per scrivere i dati sulla periferica corretta.
Le altre funzioni sono invece più semplici: `VgaSuspendOutput()` e `VgaResumeOutput()` vanno semplicemente a cambiare un valore nel `VgaScreenBuffer` che impedisce l'abilitazione del DMA alla fine di una line; `VgaStopOutput()` ferma semplicemente il DMA e i timer mentre `VgaStartOutput()` avvia i timer di hsync e vsync, aspetta che il DMA riempia la FIFO e poi avvia il timer principale, mettendo effettivamente in moto la catena di timing e di triggering del DMA.
Il file implementa inoltre le callback richieste dal `ScreenBuffer` per il disegno di un singolo pixel, sia in modalità normale che in modalità packed. Gli interrupt dei timer vengono gestiti direttamente senza passare dall' HAL per diminuire il più possibile l'overhead.
Tutte le funzioni di interfaccia pubblica restituiscono un valore dell'enumerazione `VgaError` per indicare un eventuale problema riscontrato durante la chiamata. Per l'allocazione del frame buffer è stata creata una fittizia funzione di $ralloc()$ e $rfree()$ che tramite la gestione di un contatore, spostano semplicemente un puntatore di un array statico che copre l'intera RAM a disposizione.

5.4 Application level – Palette

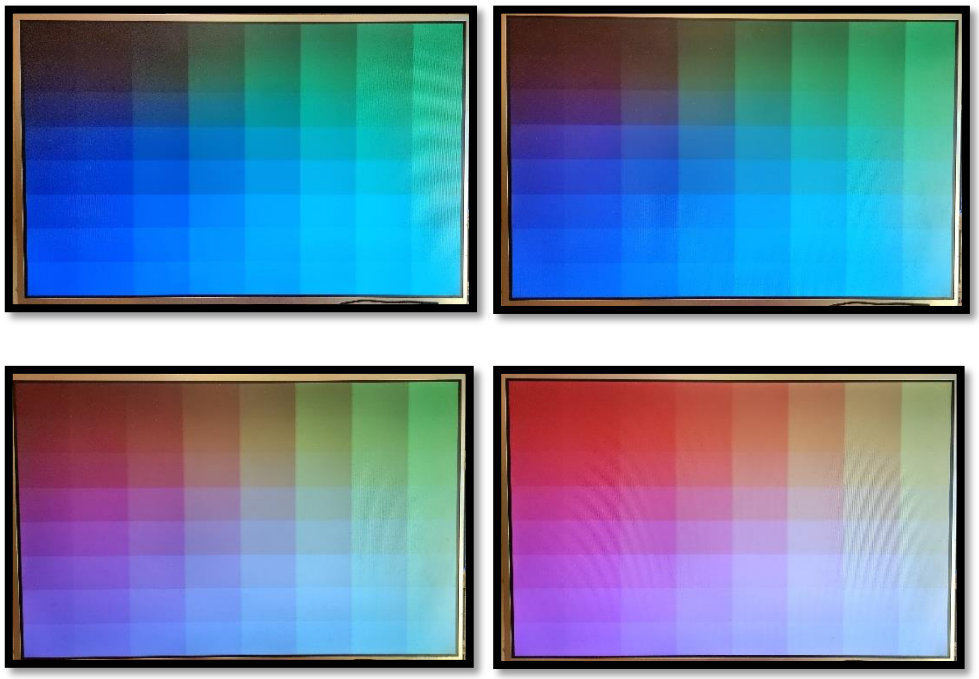


Figura 36 Palette di colori rappresentabile sullo schermo

L' applicazione **Palette** mostra semplicemente a schermo tutte le 256 combinazioni di colori possibili secondo la configurazione del nostro screen buffer. Le righe dello schermo rappresentano tonalità diverse di blu mentre le colonne diverse tonalità di verde. La tonalità di rosso può essere cambiata inviando i comandi + e – da seriale

5.5 Application level – Explorer

L' applicazione **Explorer**, una volta avviata, monta il nostro file system e mostra a schermo la lista di file presenti nella root della nostra scheda SD (è stata implementata l'enumerazione solo della top directory). Navigando nella lista, è possibile aprire i file BMP e visualizzarli a schermo. Se un file BMP è della stessa dimensione del nostro screen buffer, l'immagine viene caricata e mostrata direttamente, altrimenti viene scalata. Nel caso di errore, viene visualizzato un messaggio a schermo.

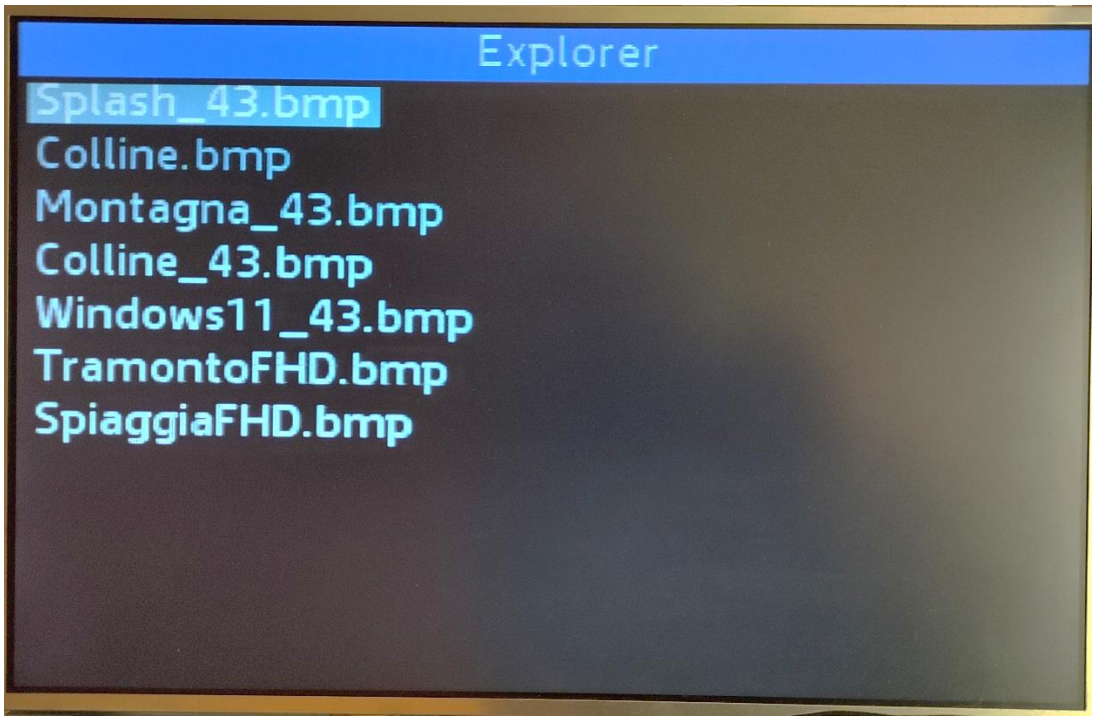


Figura 37 Interfaccia applicazione con la possibilità di scegliere il file da visualizzare

5.6 Application level - ASCII table

L' applicazione **ASCII table**, una volta avviata, mostra semplicemente tutti i caratteri ASCII a schermo per verificare come viene disegnato il font caricato.

5.7 Main

Il corpo principale dell'applicazione non fa altro che preparare i due task di FreeRTOS e lanciare il relativo scheduler. Il primo task è quello che si occupa di attendere una risposta alla richiesta dell'EDID inviata al monitor. Una volta ricevuto un messaggio e validato il checksum, questo task viene fermato per poi lanciare il task principale dove viene allocato il framebuffer, avviato l'output a monitor e viene gestito l'input utente ed il controllo di disconnessione del monitor.

Come già specificato, FreeRTOS non è necessario ed introduce solo latenze se non usato correttamente, ma è stato mantenuto solo per valutare appunto eventuali effetti indesiderati sulla visualizzazione durante lo sviluppo.

6 Prestazioni e consumi

6.1 Profiling via Serial Wire

Il processore Cortex M4 supporta una serie di estensioni hardware per il debugging avanzato e la profilazione del nostro applicativo indispensabili per trovare e ottimizzare i punti critici in un sistema. Nel nostro caso è stato utilizzato l'**ITM**, ovvero l'*Instrumentation Trace Macrocell*, una trace source che supporta un debugging stile printf(); la peculiarità sta nel fatto che ogni pacchetto emesso porta con se un timestamp; quindi possiamo, con una certa precisione, stimare quanto tempo occupa una certa routine del nostro programma.

Per utilizzare l'*ITM*, il CubeIDE dopo averlo configurato comunica con la porta *Serial Wire* della nostra board (una delle due interfacce di debugging disponibili).

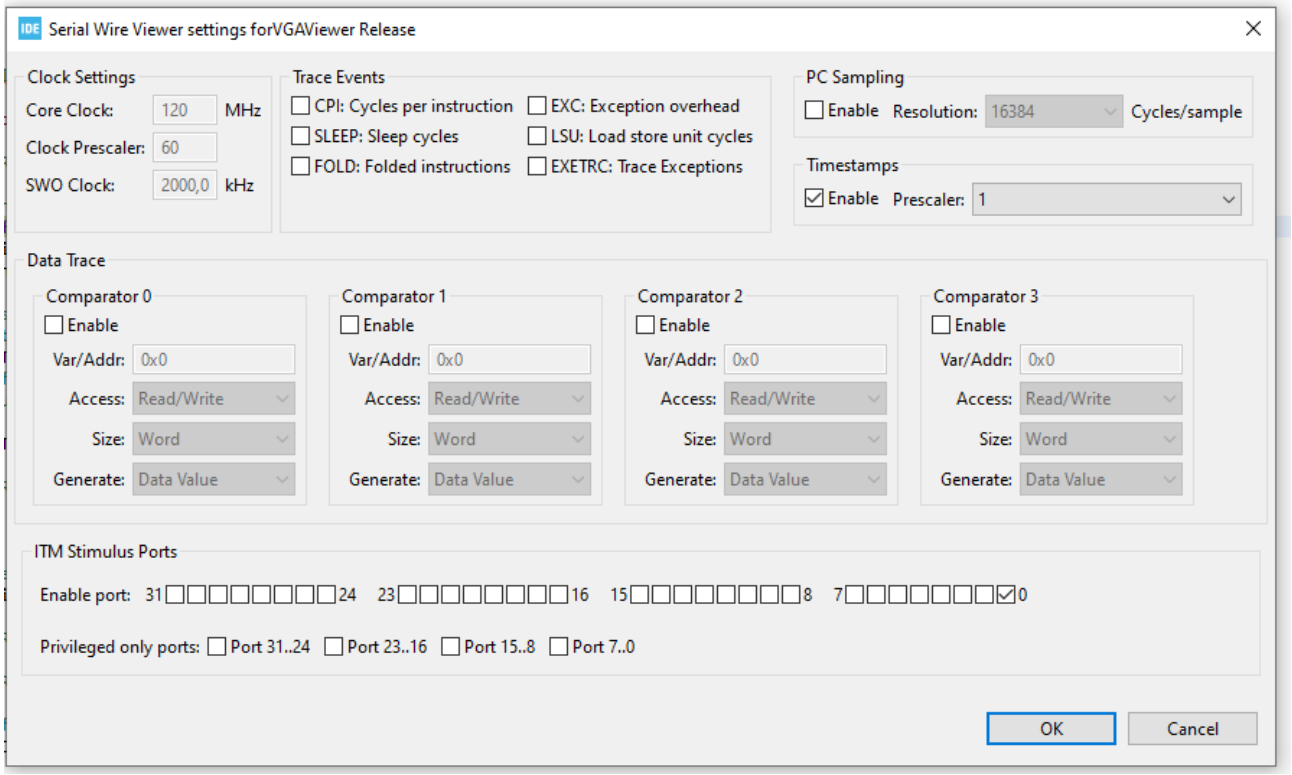


Figura 38 Interfaccia gestione Serial Wire una volta avviato il debug. Serial Wire deve essere stata precedentemente configurata attraverso il setup del progetto

Come vediamo dalla figura, l'interfaccia Serial Wire su CubeMx è completamente configurabile. Per i nostri scopi è necessario impostare correttamente il clock a cui viaggia l'MCU, abilitare i timestamp e selezionare la porta ITM corretta.

HSYNC VISIBLE AREA TIMING VIA ITM

Index	Type	Data	Cycles	Δ ~ 20us
0	ITM Port 0	83 DRAW START	753229	6.276908 ms
1	ITM Port 0	69 DRAW END	755773	6.298108 ms
2	ITM Port 0	83	756374	6.303117 ms
3	ITM Port 0	69	758930	6.324417 ms
4	ITM Port 0	83	759540	6.329500 ms
5	ITM Port 0	69	762096	6.350800 ms

Figura 39 Esempio di utilizzo pacchetti ITM per verificare il tempo che passa tra l'interrupt di inizio linea e fine linea (parte visibile). Il delta tra i due tempi rispetta quello delle specifiche della risoluzione che stiamo usando

6.2 Tempo di riempimento dell'intero schermo

Per riempire una certa area interamente di un colore abbiamo diverse possibilità. La più semplice è (rispetto alla nostra implementazione del layer grafico) chiamare la funzione nativa di disegno sul frame buffer per ogni pixel. Questa funzione deve compiere i seguenti passi:

- Calcolare l'indirizzo di memoria da scrivere basandosi sulle coordinate del pixel
- Se il colore da disegnare è opaco, sovrascrivere la memoria direttamente
- Altrimenti, applicare l'alpha blending e sovrascrivere il colore.

Con la giusta struttura del codice (evitando di chiamare un'altra funzione quando il colore è opaco ma solo quando dobbiamo applicare l'alpha), il riempimento dello schermo impiega circa $80ms$. Essendo il pixel solo di 1 byte, possiamo sfruttare delle scritture di word intere e aumentare le prestazioni (quando dobbiamo disegnare lo stesso colore senza alpha), andando così ad abbassare il tempo richiesto a circa $20ms$.

ITM PACKETS – 1 BYTE DRAW

Index	Type	Data	Cycles	Time(s)
0	ITM Port 0	100 DRAW START	3079945330	25,666211 s
1	ITM Port 0	68 DRAW END	3089563325	25,746361 s
2	ITM Port 0	100	3089563402	25,746362 s
3	ITM Port 0	68	3099181322	25,826511 s
4	ITM Port 0	100	3099181397	25,826512 s
5	ITM Port 0	68	3108799322	25,906661 s

ITM PACKETS – 4 BYTES DRAW

Index	Type	Data	Cycles	Time(s)
2	ITM Port 0	100 DRAW START	163760002	1,364667 s
3	ITM Port 0	68 DRAW END	166621665	1,388514 s
4	ITM Port 0	100	166621744	1,388515 s
5	ITM Port 0	68	169483272	1,412361 s
6	ITM Port 0	100	169483351	1,412361 s
7	ITM Port 0	68	172345007	1,436208 s

Figura 40 Lista di pacchetti ITM per le due tipologie di chiamate di disegno

6.3 Tempo di lettura settore dalla scheda SD

Come già specificato nel paragrafo relativo alle schede SD, la velocità di lettura dati è solamente dipendente dalla velocità del clock a cui lavora l'SPI (per quanto riguarda il lato comunicazione). Nel nostro caso, la periferica SPI è attaccata al bus che lavora a 30 MHz e, una volta terminata la procedura di inizializzazione, il prescaler viene portato a 2 (valore minimo), per un clock in uscita che lavora a 15MHz. Osservando il timing SPI nell' application note, in linea teorica ogni ciclo di clock corrisponde alla lettura di un bit; quindi, dovremmo leggere 512 byte in

$$\frac{1}{15 * 10^6} * 8 * 512 = 0.273\, ms$$

Una volta calcolato il nostro limite inferiore, possiamo avere una stima reale della velocità effettiva usando SWV. Misurando la distanza tra due pacchetti di tracing, otteniamo un tempo attorno ai $\sim 0.650\, ms$, con esclusi i calcoli sul CRC e le asserzioni sulla SPI. Notiamo che, pur essendo l'assembly ridotto al minimo in termini di operazioni eseguite, abbiamo comunque un piccolo overhead.

```
PerformByteTransaction:
        080032cc:   ldr     r3, [pc, #20]   // (0x80032e4 <PerformByteTransaction+24>)
        080032ce:   ldr     r3, [r3, #0]    // Loading _spiInstance address in r3
        080032d0:   str     r0, [r3, #12]   // Storing r0 (byte parameter) in r3 + 12 [SPI Data Register]
   +--> 080032d2:   ldr     r2, [r3, #8]    // Loading r3 + 8 (SPI Status Register) value into r2
   |    080032d4:   and.w   r2, r2, #3      // We need to check Tx event and Rx event
   |    080032d8:   cmp     r2, #3
   +--- 080032da:   bne.n   <PerformByteTransaction+6> // Loop waiting
        080032dc:   ldr     r0, [r3, #12]   // Tx and Rx completed; We need to read Data Register again
        080032de:   uxtb    r0, r0          // Zero extend byte
        080032e0:   bx      lr              // Return to caller. Read byte will be in r0
```

Figura 41 Disassembly della funzione che legge e scrive un byte sul bus SPI

6.4 Consumo di corrente

Lo scopo del progetto è visualizzare dei dati a schermo, di conseguenza non si è ritenuto utile implementare uno schema di power saving mettendo l'MCU in uno stato di sleep. Anche con lo schermo disconnesso, a meno di un circuito più complesso per il rilevamento della presenza di questo, dobbiamo comunque inviare periodicamente dei messaggi sul bus I^2C.

Per rilevare il consumo di corrente dobbiamo collegare un multimetro in serie al ***Jumper1*** come specificato nel pdf di presentazione della scheda discovery.

La baseline è calcolata a schermo connesso ma totalmente nero, dove il consumo risulta essere di $40mA$.

A schermo totalmente bianco, il consumo risulta essere invece di $\sim 75mA$, congruo con il calcolo della corrente sulle resistenze del DAC mostrato precedentemente ($\sim 13\ mA$ per ogni colore)