Appunti IoT Systems

Capitolo 5

Le Interfacce

Simone Coco

Le interfacce

Ogni microcontrollore generalmente contiene determinate interfacce per la comunicazione, a volte anche delle istanze multiple di una particolare interfaccia.

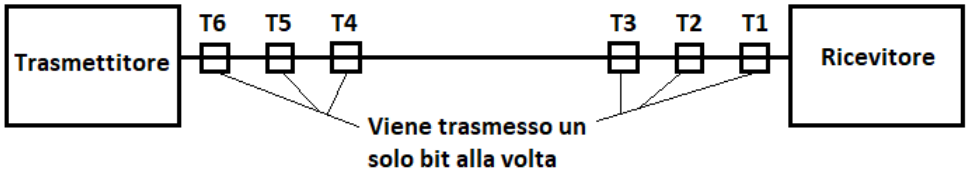
Le interfacce permettono al microcontrollore di comunicare con le altre unità (ovvero con le periferiche, con i PC o anche con altri microcontrollori)

Classificazione

Ogni interfaccia è caratterizzata da una categoria di appartenenza; questa ne contraddistingue il funzionamento e la tipologia.
NB: utilizzeremo il termine "trasmettitore" e "ricevitore" riferendoci, genericamente, a qualsiasi componente (microcontrollore, periferica, etc…)

1. Seriale o parallela

Un'interfaccia seriale trasmette un bit alla volta, quindi osservando il traffico di dati, osserveremo la trasmissione di un bit alla volta; per questo motivo vi è **una sola data line**: una sola linea per il trasporto dei bit.



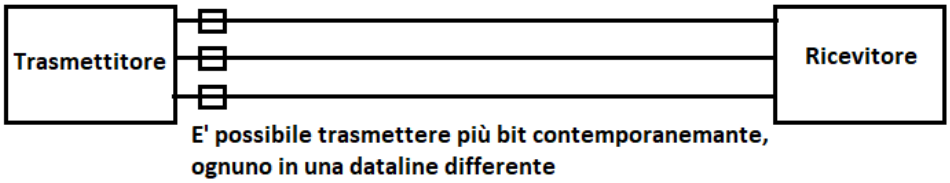
Vantaggi:

- √ La risorsa è efficiente: con un solo pin si possono trasmettere i dati (trascurando i dati di controllo!)

Svantaggi

- × Ogni cosa deve essere serializzata: ogni byte di dati da trasmettere deve essere "scomposto" in un flusso di bit: il **throughput** (quantità di informazioni che passano nell'unità di tempo) è molto bassa.

Un'interfaccia parallela è caratterizzata da una certa ampiezza, una certa **quantità di connessioni** (generalmente multipla di due) su cui ognuna può transitare un bit.



Vantaggi:

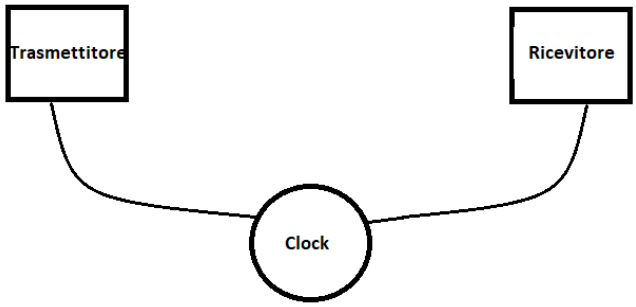
- √ Ovvia alla limitazione del **throughput** dell'interfaccia parallela

Svantaggi

- × La risorsa è meno efficiente poiché sfrutta più connessioni per trasmettere i dati

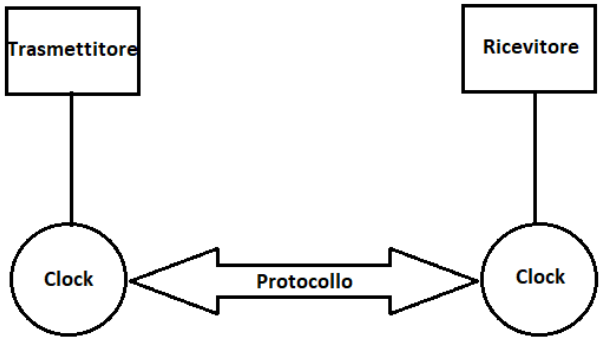
2. Sincrona o asincrona

In un'interfaccia sincrona, ricevitore e trasmettitore sono sincronizzati allo stesso tempo di clock. Per affermare che due interfacce siano sincrone non basta affermare che essi abbiano la stessa frequenza, bensì esse devono essere **fisicamente collegate**, inoltre, è anche possibile avere due interfacce sincrone a frequenze diverse: basti avere, ad esempio, il ricevitore che va a 1MHz e il ricevitore che va a 2MHz perché, magari, ha dimezzato la velocità del clock (es: prescaling)



Chiaramente, nelle interfacce sincrone non vi sono problemi di sincronizzazione.
La "condivisione" di un unico clock può essere implementata facendo si che uno dei due componenti invii un segnale di sincronizzazione all'altro così che entrambi seguano lo stesso clock.
L'implementazione infatti prescinde da questo particolare, l'unica cosa che conta è che sia trasmettitore che ricevitore siano sincronizzati dallo stesso clock.

In un'interfaccia asincrona i clock di trasmettitore e ricevitore sono indipendenti. Possono avere anche la stessa frequenza ma avranno clock separati.
In questo caso, poiché essi non sono sincronizzati "naturalmente", hanno bisogno di un **protocollo per gestire la comunicazione**: vi è uno scambio di messaggi per "concordare" alcuni parametri condivisi, quale, ad esempio la velocità comune ad entrambi (poiché potrebbe capitare che il trasmettitore non supporti la velocità del ricevitore o viceversa); un altro scambio di messaggi è quello dei messaggi che delimitano il dato, segnalando quando la trasmissione di quest'ultimo è effettivamente finita.
Tra questi protocolli vi è l'oversampling per la sincronizzazione, che specificheremo più avanti.



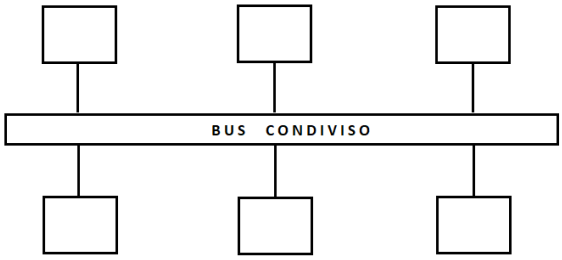
Svantaggi

- × Generalmente, una interfaccia asincrona potrebbe essere più lenta di quella sincrona e ciò è dovuto alla comunicazione che deve avvenire per la sincronizzazione dello scambio di informazioni.

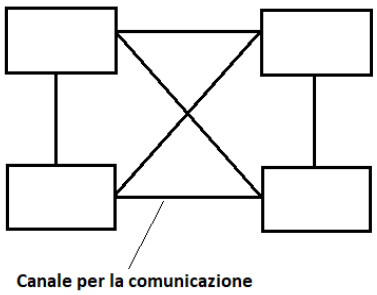
3. Interfacce connesse tramite un Bus o tramite connessione point-to-point

Un trasmettitore ed un ricevitore possono essere connessi mediante un bus o una connessione di tipo point-to-point

Nella comunicazione tramite Bus tutti i dispositivi sono connessi ad un unico bus dati centrale avente una massima dimensione di N bit. Chiaramente, in un dato momento, solo uno dei dispositivi può trasmettere attraverso un bus e tutti gli altri non dovranno essere in grado di trasmettere; ciò lo si può garantire con delle determinate politiche per la gestione della risorsa condivisa (il bus) e di indirizzamento per chiarire a quale dispositivo è diretto il messaggio trasmesso.



Nella connessione point-to-point ogni dispositivo ha un canale per parlare con gli altri dispositivi. Questo significa che non sorgono alcuni requisiti fondamentali nella comunicazione mediante bus come ad esempio l'indirizzamento: non vi è bisogno di alcun indirizzamento poiché ogni dispositivo è già direttamente collegato al destinatario del messaggio con un canale dedicato. Questo collegamento è difficilmente scalabile se vi è un numero elevato di nodi.



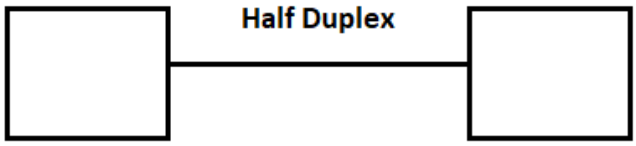
Comunicazione Bidirezionale

Prima di elencare le altre due tipologie di interfacce, è bene parlare della comunicazione bidirezionale e distinguerla da quella unidirezionale.
Fino ad ora abbiamo supposto che due dispositivi comunicassero con comunicazioni monodirezionali, ovvero delle comunicazioni in cui si ha un trasmettitore ed un ricevitore e la comunicazione può avvenire in un unico senso: dal trasmettitore al ricevitore.
Una comunicazione bidirezionale è una comunicazione che possa avvenire in entrambe le direzioni: il trasmettitore invia dati al ricevitore e può avvenire anche il viceversa.

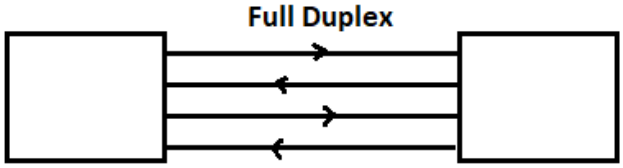
La comunicazione bidirezionale può essere:

- **Half Duplex:** i dispositivi possono trasmettere "a turno" i dati: alternando la trasmissione con la ricezione, in modo tale da non sovrapporsi.
  Per una comunicazione simile il numero minimo di connessioni è una poiché i dispositivi si alternano durante la comunicazione e non si rischia la sovrapposizione dei dati trasmessi
  [Se dovessimo fare un'analogia, si pensi alla comunicazione attraverso dei Walkie-Talkie: chi riceve il messaggio deve aspettare il segnale di "Passo" prima di rispondere a chi ha trasmesso il segnale.**]**
  Vantaggi:
  - √ Si risparmia nel numero di connessioni

  Svantaggi:
  - × Throughput meno efficiente poiché le due parti devono alternarsi durante la comunicazione



- **Full Duplex:** i dispositivi non necessitano di aspettare un turno e possono trasmettere contemporaneamente.
  Questa comunicazione necessita almeno di due connessioni così che se contemporaneamente vengono inviati dei dati, non vi siano interferenze.
  [Se dovessimo fare un'analogia, si pendi ad una comunicazione attraverso un telefono: gli interlocutori possono parlare quando lo ritengono necessario, rischiando anche di sovrapporsi durante la conversazione.**]**
  Vantaggi:
  - √ Throughput più efficiente poiché si possono trasmettere dati contemporaneamente



  Svantaggi:
  - × Costi di realizzazione più elevati dovuti alle connessioni multiple da instaurare.

Dopo aver introdotto la comunicazione bidirezionale possiamo trattare altri due tipi di interfacce:

4. Interfacce basate su master-slave o equal partners

A prescindere dalla tipologia di comunicazione (bus, point-to-point, etc..), in un sistema di tipo master-slave, possiamo distinguere alcuni nodi (dispositivi) che ricoprono il ruolo di master e altri di slave. I nodi Master sono gli unici che possono far iniziare le trasmissioni, gli slave possono comunicare solo se interpellati dai master. Qualora vi fossero più Master servirà una politica di arbitraggio per decidere quale farà partire la trasmissione per primo.

Caratterizzazione a livello fisico

Un'altra differenza tra interfacce di comunicazioni è quella tra interfacce **single-ended** e interfacce **differenziali.**

5. Interfacce single-ended

Quando affermiamo che un pin ha un determinato valore, come ad esempio 1, quello che in realtà diciamo è che il valore della Vcc misurata tra un punto e la massa avrà un determinato valore che, codificato in binario corrisponde al valore alto, ovvero 1.

Quando un pin di un dispositivo in trasmissione deve comunicare un bit al pin del dispositivo in ricezione, ci si deve assicurare che esso assuma lo stesso valore in entrambi i dispositivi.

Per le interfacce single-ended questo problema si affronta assumendo che il bit 0 corrisponda al collegamento con la massa, i due microcontrollori fanno riferimento allo zero allo stesso valore di Vcc (la messa a terra), quindi il valore misurato rispetto ad esso sarà uguale in entrambi

Vantaggi:

- √ Semplicità della risoluzione del problema

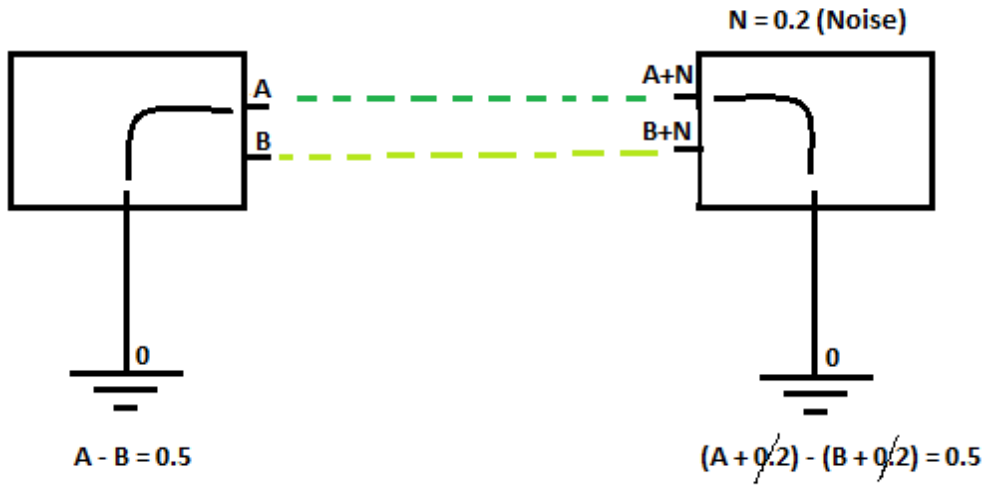
Svantaggi:

- × Se i microcontrollori sono lontani vi possono essere dei disturbi che perturbano il valore e i due valori misurati sono differenti.

6. Interfacce differenziali

Nelle interfacce differenziali per trasmettere un segnale non viene trasmesso solamente il bit ma vengono trasmessi almeno due segnali, in questo modo, se volessi inviare un valore, ad esempio 0.5V, faccio si che questo numero sia pari alla differenza tra i due segnali trasmessi.

Qualora vi sia un possibile disturbo, siccome il disturbo perturberebbe entrambi i segnali in ugual misura, otterrei comunque che la loro differenza coincide con quella inviata.



Chiaramente, devono essere utilizzate almeno due collegamenti: uno per A e uno per B.

Nelle interfacce differenziali non si comunica direttamente il segnale da trasmettere, esso si comunica sottoforma di differenza tra due segnali, così da rimediare ad eventuali problemi dovuti al rumore.

Vantaggi:

- √ Ogni rumore agisce sui due segnali in modo equo, mantenendo inalterata la loro differenza, in questo modo sono possibili trasmissioni a lunga distanza (rimedia ai limiti delle interfacce single-ended).

Svantaggi:

- × Richiede più connessioni rispetto all'interfaccia single ended (almeno due).

NB: Le interfacce servono per la comunicazione tra microcontrollori o tra il microcontrollore ed un PC (Qualora, ad esempio, sei stesse eseguendo un debug). Questa comunicazione tra microcontrollori non si deve però confondere con la comunicazione con il mondo esterno che avviene con i pin di I/O. Il microcontrollore può, infatti, comunicare con il mondo esterno e per farlo utilizza le tecniche di conversione da analogico a digitale e/o viceversa che abbiamo affrontato (PWM, circuiti di resistori Binary Waighted, etc..). L'interfaccia invece si può vedere come uno "standard" per la comunicazione tra due microprocessori: essi dovranno condividere le stesse interfacce per la corretta comunicazione, quindi non avviene come la comunicazione con il mondo esterno.

In ambito progettuale utilizzeremo le interfacce già pronte per essere utilizzate.

Distingueremo tre tipologie di interfacce che rappresentano tre tipi di comunicazioni diverse tra due entità (microcontrollori e/o PC):

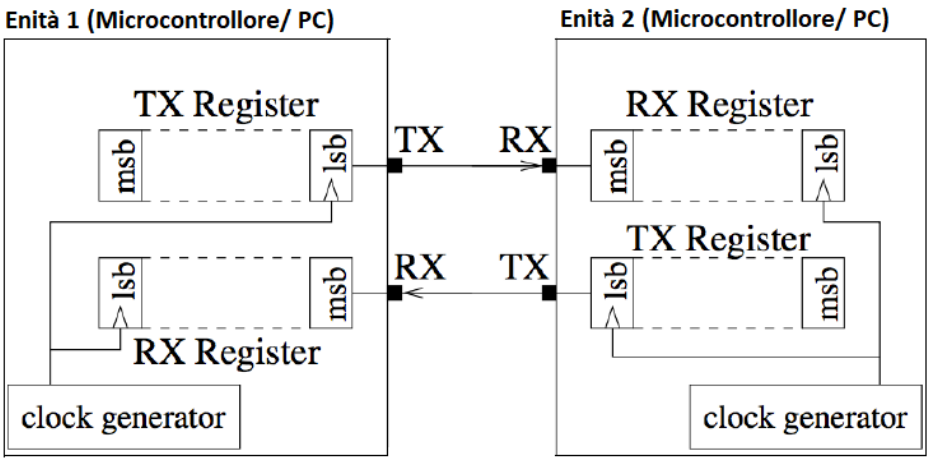
1. UART
2. SPI
3. IIC

SCI (UART)

L'interfaccia di comunicazione seriale: **Serial Communication Interface (SCI)** fornisce un sistema di comunicazione **asincrono** mediante **modulo UART**: **Universal Asynchronus Receiver Trasmitter.**
[Seriale : un bit alla volta]
[Asincrono: i clock in trasmissione ed in ricezione non sono sincronizzati, necessitano di una sincronizzazione durante il ciclo della conversazione]

Il modulo UART prevede l'utilizzo di **due cavi**: uno in trasmissione (TXD) ed uno in ricezione (RXD), supportando così la comunicazione full-duplex o half-duplex

Studiamo la struttura di funzionamento del modulo UART:



In figura possiamo notare due capi della trasmissione: non parleremo di lato trasmittente e lato ricevente poiché entrambi i lati sono in grado di ricevere e/o trasmettere.

Ognuno dei due capi ha un clock e questo ci permette di confermare una delle caratteristiche di questo modulo: entrambi i capi sono asincroni, quindi **il modulo UART è asincrono**.

Vi sono poi due registri, uno di trasmissione: **TX Register** e uno in ricezione: **RX Register,** aventi celle di memoria ordinate da sinistra (lsb: least significant bit) a a destra (msb: most significant bit).

Il registro di trasmissione è collegato ad una **porta** che sarà il **fronte di trasmissione verso l'altra entità**; il registro di ricezione è collegato ad un'altra **porta** che sarà il **fronte di ricezione** che riceve le informazioni dall'entità trasmittente dall'altro lato

Il modulo UART può essere utilizzato per le comunicazioni **asincrone e seriali,** ma si noti bene che esso **non è un protocollo di comunicazione** perché vi sono molti aspetti che sono configurabili; quindi il modulo UART è la struttura con il quale avviene la comunicazione ma non sancisce alcuna regola di comunicazione come fa un vero e proprio protocollo; molti parametri dell'aspetto comportamentale delle entità durante la comunicazione possono essere configurati.

Modulo UART: parametri configurabili

Come abbiamo introdotto, vi sono alcuni parametri del modulo UART che sono **configurabili:**

1. **Numero dei bit dati:** alcuni bit vengono utilizzati per il controllo e per altre funzioni quindi non tutti i bit sono destinati ai dati, tuttavia il numero di bit utilizzabili per i dati si può configurare
2. **Bit di parità**
3. **Bit Stop**
4. **Baud rate**

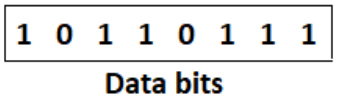
1. UART: Numero di data bit

Le informazioni inviate all'interno della parola di codice non consistono nei soli dati: abbiamo, infatti, oltre ai dati anche alcuni bit "speciali" che servono a scopi precisi, come ad esempio il bit di parità o quello di stop.
Il numero di data bit ci dice quanti bit, all'interno della parola di codice, sono destinati alla sola trasmissione dei dati; questo numero varia solitamente da 5 a 9 bit per parola di codice, che verranno contornati, come già detto, da altri bit che ne aggiungono dettagli e informazioni aggiuntive per diversi scopi.
Per trasmettere più bit dati si trasmetteranno, chiaramente, più parole codice (o simboli) [Fino a formare, ad esempio, un byte].
Il numero di data bits è configurabile a monte: nella programmazione dei programmi che gestiscono il microcontrollore è possibile configurare il numero di bit per i dati; è importante precisare che **trasmettitore e ricevitore devono avere lo stesso numero di data bits** altrimenti la comunicazione risulterebbe falsata.

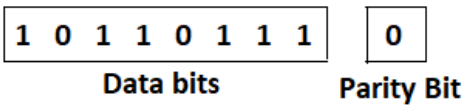
2. UART: Parity bit

Il bit di parità (in inglese parity bit) è un bit "speciale" addetto al semplice controllo di eventuali errori nella trasmissione;
NB: **esso serve per controllare se vi sia stato un errore e non a correggere gli eventuali errori.**
Il nome "parità" non deve indurre a pensare che questo bit può essere programmato solo come bit pari; infatti, il bit di parità può essere configurato come bit pari o bit dispari: ai dati normali verrà inviato un bit extra che non trasporta alcuna informazione (quindi non è un data bit) e che terrà il conto del numero di 1 all'interno dei data bit; in base alla politica scelta (se odd: pari o even: dispari) verrà deciso se settare il bit di parità o meno. Per capire meglio questo concetto ci serviamo del seguente esempio:

Supponendo di avere 8 data bit per la trasmissione delle informazioni, avremo una situazione del genere:

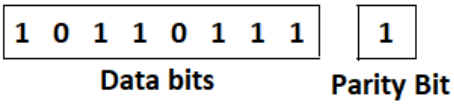


Utilizzando una politica **odd** avremo:



Poiché il numero di uno presente nei data bits è pari (6 bits) e quindi non vi è bisogno di aggiungere un bit per avere un numero di bit che sia pari (il parity bit non si setta: rimane 0).

Utilizzando una politica **even** avremo:



Siccome il numero di uno è pari (6 bit), aggiungeremo il bit di parità per far si che essi risultino dispari (settiamo il parity bit a 1).

Si noti che il parity bit non ha alcun significato informativo ma serve solo per controllare se vi è stato un errore: nella trasmissione, se si corrompe un bit il parity bit non quadra più.

Chiaramente non è un'ottima soluzione:
1. Il bit che corrotto potrebbe proprio essere il parity bit
2. Se vengono corrotti due data bit non ci si accorgerebbe dell'errore perché si ritornerebbe ad avere un numero di uno pari/dispari.

Conviene usarlo quando è poco probabile che vi siano errori e, qualora vi fossero, chiaramente è più probabile che questo errore ricada nei dati (visto che sono una quantità di bit maggiore) e non sul singolo parity bit; se è molto raro che avvenga un errore allora si suppone che l'errore avvenga al massimo per un bit alla volta. Sotto queste opportune condizioni è utilizzabile il parity bit.

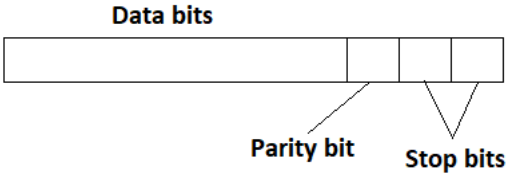
Quando viene ricevuta una parola codice errata (con l'errore che viene fuori grazie al parity bit) viene richiesta la ritrasmissione di quest'ultima.

Per l modulo UART un errore sulla comunicazione dovuta al bit di parità errato non è un errore critico
NB: **il bit di parità non è obbligatorio,** la comunicazione può avvenire con assenza di bit di pairtà

3. UART: Stop Bits

Alla fine del pacchetto di bit (la parola di codice) trasmesso, vi è lo stop bit; vi possono essere uno e due stop bit che indicano la fine della parola trasmessa.

Lo stop bit è necessario poiché con il modulo UART la comunicazione è asincrona e quindi i clock in trasmissione ed in ricezione, non essendo sincronizzati, devono comunicare tra loro quando finisce una determinata parola di codice, questo è possibile proprio grazie allo stop bit, che indica la fine di ogni simbolo.



4.UART: Baud Rate

Abbiamo visto che possiamo configurare: i data bits (decidendo quanti bit destinare al trasporto dell'informazione vera e propria), i bit di parità (scegliendo se controllare se il numero di uno siano pari o dispari), gli stop bit (settandone uno o due per segnalare la fine del simbolo trasmesso).

Altro importante parametro riguarda la **durata di ogni singolo bit**: questo parametro può essere configurato e deve essere concordato in trasmissione ed in ricezione così che le due entità comunicanti possano capire quando finisce un determinato bit e ne inizia un altro.

Nel nostro caso la "frequenza di trasmissione" di un bit si misura in Baud;
NB: in realtà "Baud" è l'unità di misura della symbol rate, ovvero della "frequenza di trasmissione" dei simboli; nel nostro caso, però, parleremo del trasferimento dei singoli bit e quindi il simbolo corrisponde al singolo bit.
Per un simbolo che viaggia ad una symbol rate pari a 9600 baud avrà un tempo di trasmissione (tempo di simbolo) pari a $t_s = \frac{1}{9600}$ s.

Per la trasmissione dei singoli bit, si ha un baud range (range di "frequenze di trasmissione") pari a: [9600 - 115.200 baud]; quando si setta la baud rate si modifica tale valore all'interno di questo range finito. Conoscendo la baud rate in entrambi i lati (trasmissione e ricezione), si può esser a conoscenza della durata di ogni singolo bit.

In questo modo, ad esempio, conoscendo quanti bit sono destinati all'informazione (data bits), si può conoscere dopo quanto tempo arriverà il bit di parità così da poterlo riconoscere e trattare diversamente dai bit informativi; successivamente, lo stesso discorso è applicabile agli stop bits.

Chiaramente, sia la sorgente che la destinazione devono avere un clock tale da permettere la baud rate; NB: deve "reggere" la velocità esplicitata e anche un eventuale scaling di quest'ultima.
Più aumenta la velocità e più è possibile incappare in errori.

UART: Nomenclature

L'insieme di configurazioni dei parametri descritti viene riassunto in un'unica stringa del tipo:

D{E|O|N}S

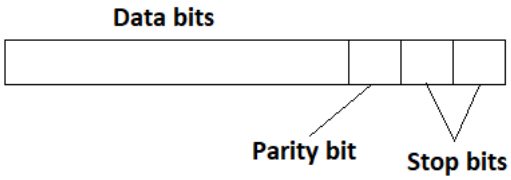
Dove:

- **D:** indica il **numero di data bits** (bit destinati alla rappresentazione dell'informazione)
- **S:** indica il **numero degli stop bits** (bit utilizzati per la segnalazione di fine stringa di codice)
- **E|O|N:** indica il **tipo di bit di parità**:
  - **E: Even,** il numero di uno trasmessi (compreso il bit di parità) deve essere **dispari**
  - **O: Odd,** il numero di uno trasmessi (compreso il bit di parità) deve essere **pari**
  - **N: No Parity,** indicherà l'assenza del bit di parità

Facciamo un esempio di stringa di configurazione:

8E1: 8 bit sono utilizzati per la trasmissione dei dati, è previsto il bit di parità che tiene conto di un numero di uno dispari (E: Even) ed 1 bit di stop per indicare la fine della stringa trasmessa [Possibile domanda da esame].

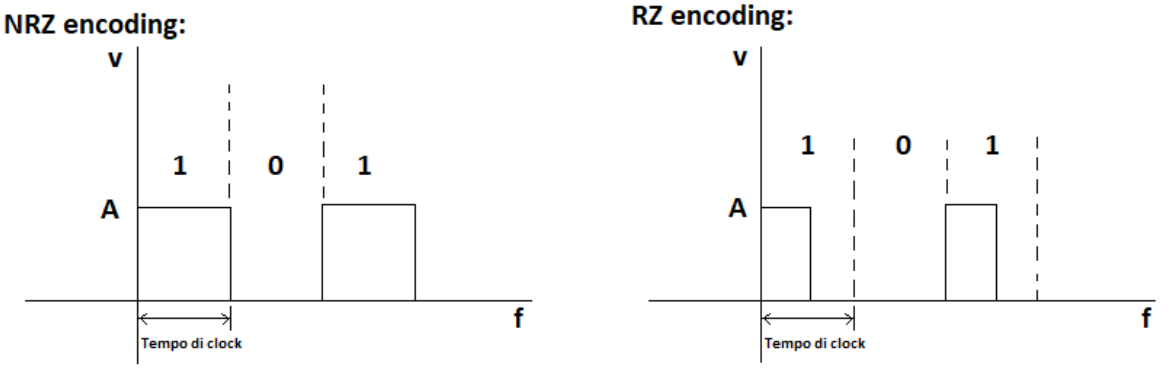
NB: questa stringa indica la forma del frame trasmesso; in questa stringa **non vi è il settaggio della baud rate.** Quest'ultimo è un parametro configurabile ma non viene configurato con la stringa che abbiamo appena trattato.

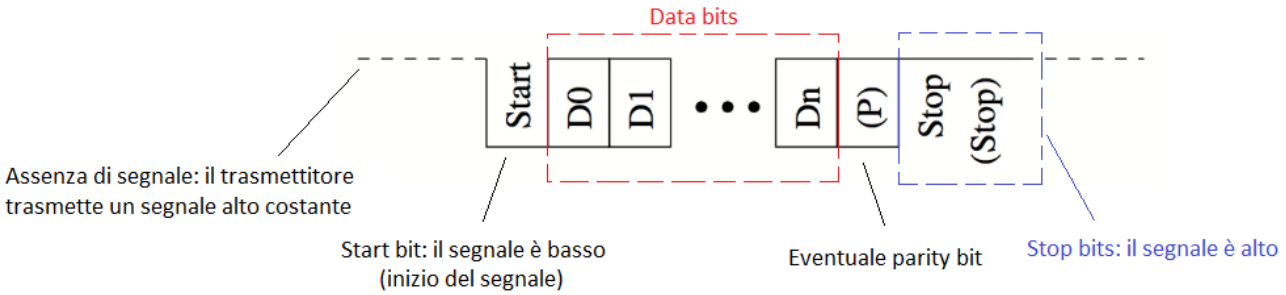


Trasmissione fisica dei dati

Dopo aver trattato la forma del pacchetto informativo da trasmettere, trattiamo cosa effettivamente viene inviato dal punto di vista fisico.

La codifica di linea utilizzata dal modulo UART per trasmettere uno e zero è una **codifica NRZ: Non Return to Zero**: per indicare l'1 il segnale sta alto per tutta la durata del ciclo di clock, per indicare 0 il segnale sta basso per la durata del ciclo di clock.
[Nella codifica RZ: Return to Zero, invece, il segnale 1 viene rappresentato da un segnale che sta alto per la prima metà del tempo di clock e scenderà a zero per la restante metà di ciclo di clock]





Inizialmente, in assenza di segnali da trasmettere, il segnale vale 1 e quindi la tensione è alta.
Non appena il trasmettitore vuole inviare un determinato segnale, invia un bit 0 rappresentato da una tensione bassa, questo particolare bit viene chiamato **start bit** e server per **comunicare al ricevitore l'inizio della trasmissione** di un determinato segnale. **Lo start bit viene praticamente utilizzato per sincronizzare ricevitore e trasmettitore** (che sono nativamente asincroni).
In ricezione il segnale si accorge che sta iniziando la trasmissione dei dati quando constata che il segnale è basso per un certo lasso di tempo pari a $\frac{1}{baud\ rate}$: in questo modo capisce che questo è lo start bit e che sta per iniziare la trasmissione.

Successivamente, **il ricevitore, una volta che scorre il tempo destinato allo start bit, si aspetta che inizi la trasmissione dei dati:** se capta un cambiamento allora assimilerà questo cambiamento come il data bit 1, altrimenti, se nota un "prolungamento" del segnale 0, capterà il data bit 0.

Dopo di che, dato che ricevitore e trasmettitore condividono lo stesso numero di data bit e gli stessi bit di controllo, dopo un tot di data bit il ricevitore capterà l'eventuale presenza di un bit di parità, che servirà per il controllo di eventuali errori sui bit (NB: il bit di parità non corregge l'errore ma lo "segnala").

Una volta ricevuto il bit di parità attenderà uno o due stop bits; questi devono essere pari a 1. Qualora il ricevitore ricevesse un bit 0 quando si aspetta uno stop bit segnala un errore, vuol dire che la trasmissione del segnale si è, in qualche modo, "sfasata".

Il controllo dei singoli bit avviene ogni $\frac{1}{baud\ rate}$; ribadiamo dunque che è fondamentale conoscere il parametro di baud rate per "scandire" i bit, distinguendoli nel loro significato.

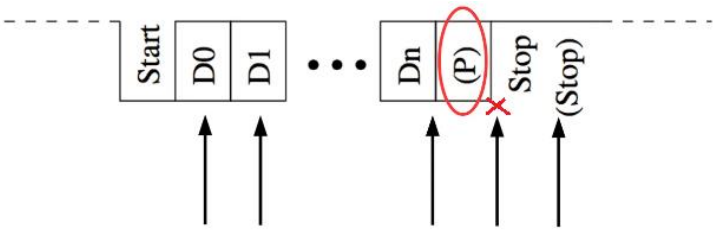
Errori di sincronizzazione e di ricognizione dei dati

Problematica:

Avendo dei clock asincroni (ognuno con una frequenza propria) abbiamo che le due differenti frequenze dei singoli clock portano ad uno sfasamento tra i due che spesso può non essere costante (le frequenze si discostano tra loro per quantità non costanti, che possono variare di volte in volta); quindi i clock non solo hanno velocità differenti ma hanno anche dei clock sfasati in maniera non costante.

Ciò che ne viene fuori è che il singolo bit non verrà captato al proprio centro di simbolo ma verrà "shiftato" verso destra e questo è causa di errore;

ad esempio, nell'immagine possiamo notare che il ricevitore crede di leggere il parity bit ma in realtà sta leggendo il valore dello stop bit (1), e ciò potrebbe quindi essere causa d'errore.

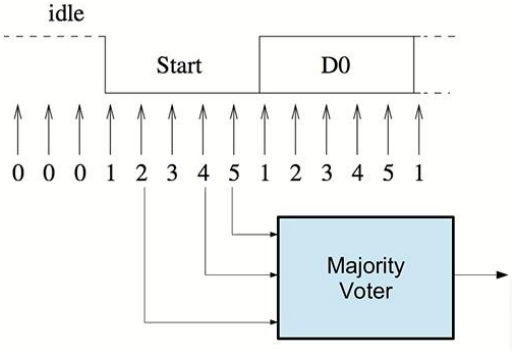


Per risolvere questo problema vi è l'**oversampling (sovra-campionamento):** viene stabilito un fattore di oversampling e l'idea è quella di far eseguire un campionamento per s volte al ricevitore (con s pari al fattore di oversampling).

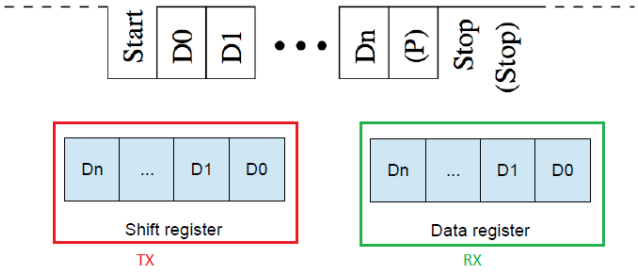
Nel momento in cui inizia lo start bit (appena viene riscontrato la discesa a zero), si imposta il primo valore di oversampling e, siccome si conosce a priori il fattore di oversampling, si sa quante volte viene campionato il segnale di start.

Avendo campionato s volte il segnale (nell'esempio in figura s = 5), si ottengono s valori, che potrebbero essere diversi qualora vi fosse stato un determinato sfasamento del segnale. Per questo motivo viene presa la maggioranza: se per la maggior parte delle volte è stato campionato 1 verrà preso il valore 1, altrimenti 0.

Chiaramente, questo discorso vale solo se le frequenze dei due clock sono in grado di "gestire" questa soluzione.

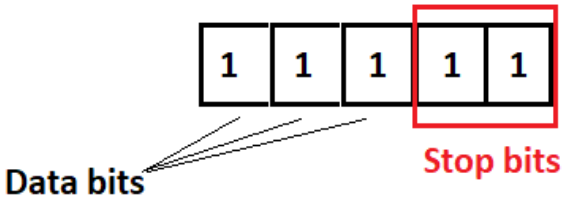


[NB: nei registri di ricevitore e trasmettitore ci stanno solo i data bit e sono quindi esclusi gli altri bit di controllo]



Se si perde troppo la sincronizzazione (quindi se non si riesce a sincronizzare nonostante l'oversampling), si può arrivare ad avere ulteriori problemi difficilmente risolvibili.

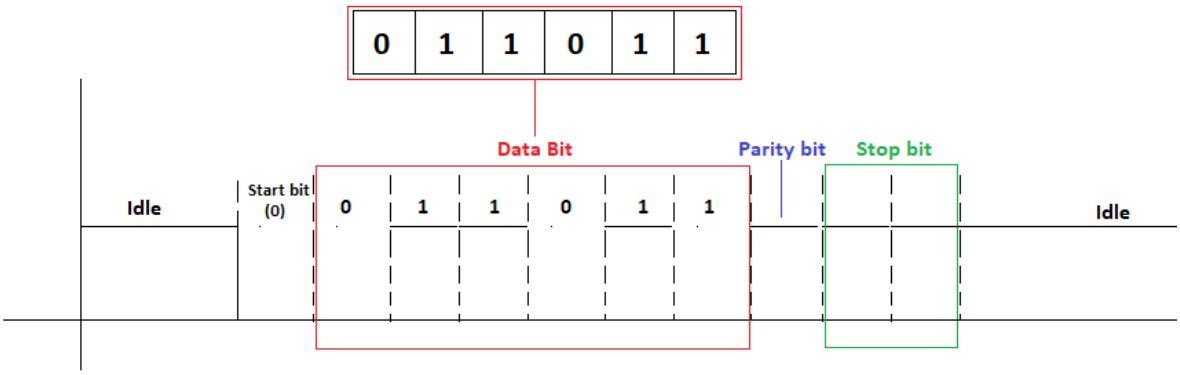
Se, ad esempio, il segnale viene shiftato a tal punto che i bit dati assumono il posto dei bit di stop, potrebbe accadere che, qualora i data bit assumessero i valori 1, la trasmissione viene data per buona poiché vengono correttamente captati i due bit di stop.



Questo problema è anche legato alla natura della trasmissione: **con una trasmissione NRZ, un tot di bit consecutivi settati ad 1 formano, praticamente, un segnale costante.**
In una situazione seguente non si avrebbe modo di distinguere se i bit letti come stop bits sono effettivamente degli stop bits o se sono sei data bits traslati.
Quindi se si ha una forte traslazione di bit 1, si potrebbero avere degli errori "impercettibili".

Quando vi sono degli errori negli stop bits (o nello stop bit se ne viene utilizzato solamente uno), si annuncia l'errore all'applicazione e viene richiesta la ritrasmissione del frame.

Un altro problema avviene quando **il ricevitore è troppo lento rispetto al trasmettitore**: potrebbe accadere che il trasmettitore finisce di trasmettere e, dopo di che, si mette in **idle** (ovvero in assenza di trasmissione dati), settando il **segnale alto**, in questo modo potrebbe accadere che il ricevitore (più lento del trasmettitore), arrivando in ritardo, interpreti il segnale alto dello stato di idle del trasmettitore come bit dati o come stop bits, captando quindi delle informazioni errate



Generalizzando, il problema principale è quello di **interpetrare dei bit aventi un determinato ruolo con altri bit aventi ruolo differente.**

Un altro problema è quello del **data overrun:** i dati ricevuti vengono "immagazzinati" nel data register, se questo non riesce a "smaltirli", ovvero a processarli, nei giusti tempi potrebbe accadere che i dati salvati nel registro vengano sovrascritti dai nuovi dati entranti in quest'ultimo.

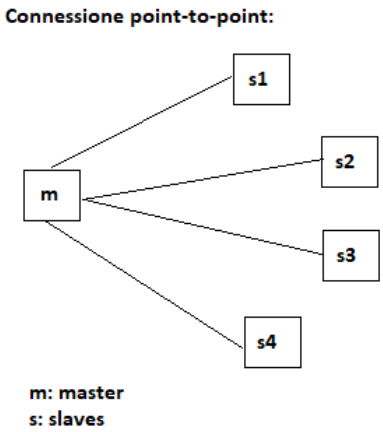
USART

Per ovviare ai problemi introdotti nel modulo UART è nato il modulo USART: esso è praticamente identico al modulo UART con l'unica differenza che implementa una comunicazione **sincrona: il clock del ricevitore e quello del trasmettitore sono sincronizzati**.
L'USART utilizza una third line (segnale) che trasporta il clock signal (la frequenza del clock); dato che i clock sono sincroni e non vi è bisogno di creare un metodo di sincronizzazione, alcune cose studiate per l'UART sono superflue, come ad esempio l'oversampling: esso è un meccanismo non necessario poiché i clock sono già sincroni e non vi sarà lo shift del segnale verso destra. Questo comporta che **il modulo USART sarà s volte più veloce dell'UART poiché non dovrà compiere gli s campionamenti.**

Nei casi discussi fino ad ora abbiamo dato per scontato che la comunicazione avvenga tra due controllori che avranno dei pin di I/O che saranno opportunamente configurati come ricevitori o trasmettitori e che permetteranno la comunicazione. Non avremo bisogno di alcun connettore.

SPI

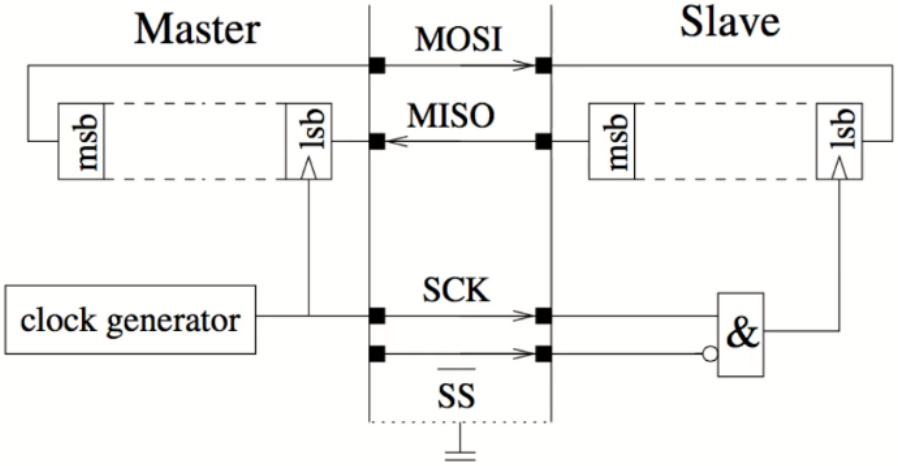
Parleremo adesso di un'altra tipologia di interfaccia, differente dalla UART: parleremo della **SPI: Serial Peripheral Interface.**
La SPI è una interfaccia **seriale**: permette la trasmissione di un solo bit alla volta, è **sincrona**: i clock di ricevitore e trasmettitore sono sincroni: nello specifico, abbiamo un unico clock per entrambe le entità; è un'interfaccia **poin-to.point:** ogni entità è connessa singolarmente con un unico microcontrollore, non vi sarà la condivisione di un unico bus.
Ed è basata su **connessione master-slave**: un microcontrollore "centrale" avvia la trasmissione (master) e gli altri microcontrollori potranno ricevere (slaves).



Nelle SPI vi sono **4 linee single-ended** (linea che va da sorgente a destinazione, valuteremo le due tensioni al capo di essa, in altre parole: **viaggerà un solo segnale nella linea**):

1. **MOSI: Master Out Slave In,** linea in cui il master trasmette e gli slave ricevono
2. **MISO: Master In Slave Out,** linea in cui il master riceve e gli slave trasmettono
3. **SCK: System Clock,** linea per il clock, che ricordiamo essere in comune tra master e slave
4. **SS: Slave Select,** linea che permette di selezionare fisicamente un singolo slave per la connessione point-to-point

Facciamo un esempio con un master ed un unico slave:



Il master avrà il clock generator che genera il segnale che verrà utilizzato dal singolo clock in comune; questo viene passato al ricevitore attraverso la linea single-ended SCK (System Clock).

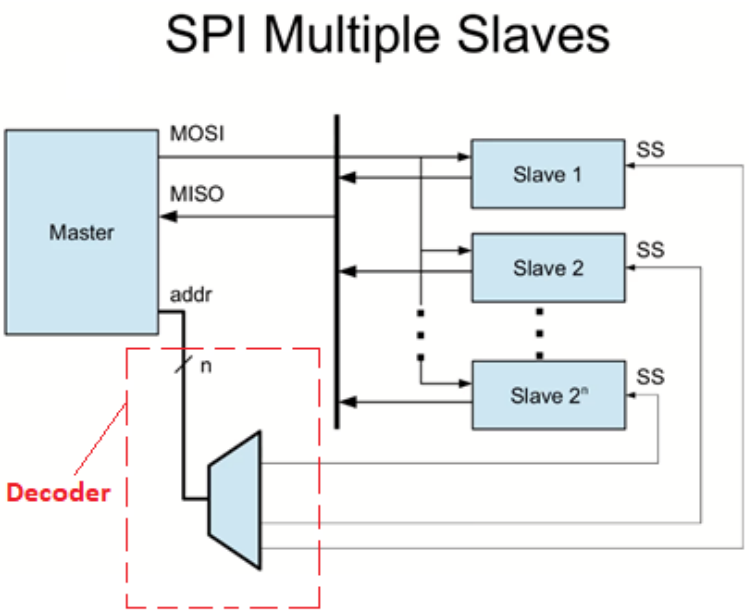
Si noti che il segnale di clock, in ricezione, è posto in ingresso ad una porta and, insieme ad segnale slave select (SS), questo perché in questo modo, fintanto che la linea slave select è “disattivata”, ovvero fintanto che lo slave non è stato selezionato, essa vale zero e quindi l’and tornerà 0: avremo che il clock dello slave non sarà sincronizzato fintanto che non verrà selezionato per la comunicazione.

Le linee MISO e MOSI collegano i registri dei data bits, e in base alla loro funzione connettono dal master allo slave (MOSI) o dallo slave al master (MISO)

Nella realtà **il master può essere connesso a più slaves**:

In questo caso, il master avrà un **decoder** che gli permetterà di scegliere uno degli slaves a disposizione e la comunicazione avviene con lo slave scelto.

Il decoder funziona con delle combinazioni binarie [quindi con n = 3bit potranno essere indirizzati 8 slaves], ognuna identifica uno slave alla volta; il decoder può anche essere chiamato slave select



IIC (I^2C)

L'IIC (Inter-Integrated Circuit) è un modulo (protocollo) molto utilizzato. È **sincrono**: ha un unico clock che sincronizza trasmettitore e ricevitore, è una **bus-based interface:** vi è un bus centrale che connette più interfacce di più microcontrollori;
con l'introduzione del bus centrale si introducono alcune funzionalità in più, necessarie per il corretto funzionamento, tra le quali **accesso alla risorsa condivisa** (il bus), l'**addressing:** il processo di indirizzamento utilizzato per raggiungere una determinata interfaccia mediante il bus.

I microcontrollori collegati tramite bus continuano però ad essere basati sul principio **master-slave** (come per la SPI): quindi anche qui vi è un master che inizia la comunicazione.

L'I^2C è molto semplice dal punto di vista realizzativo: vengono utilizzate **due linee in totale (compresa la linea del clock)**:

1. **SCL: Serial Clock Line,** linea per il segnale di clock (per sincronizzare ricevitore e trasmettitore)
2. **SDA: Serial Data Line,** linea per il segnale informativo (per i dati)

È possibile l'utilizzo di sole due linee poiché utilizza l'informazione del clock, oltre per la sincronizzazione delle due parti, anche per far si che delle operazioni sui dati trasmessi sulla linea Serial Data (SDA) abbiano un significato diverso a seconda di quale sia il valore attuale della Serial Clock Line (SLC); in poche parole **il vantaggio sta nel fatto che la linea di clock non serva solo alla sincronizzazione dei clock in ricezione ed in trasmissione ma essa serve anche ad indicare che tipo di operazione effettuare sui bit dati (che circolano nella linea dei dati).** [Tra poco lo vedremo meglio]

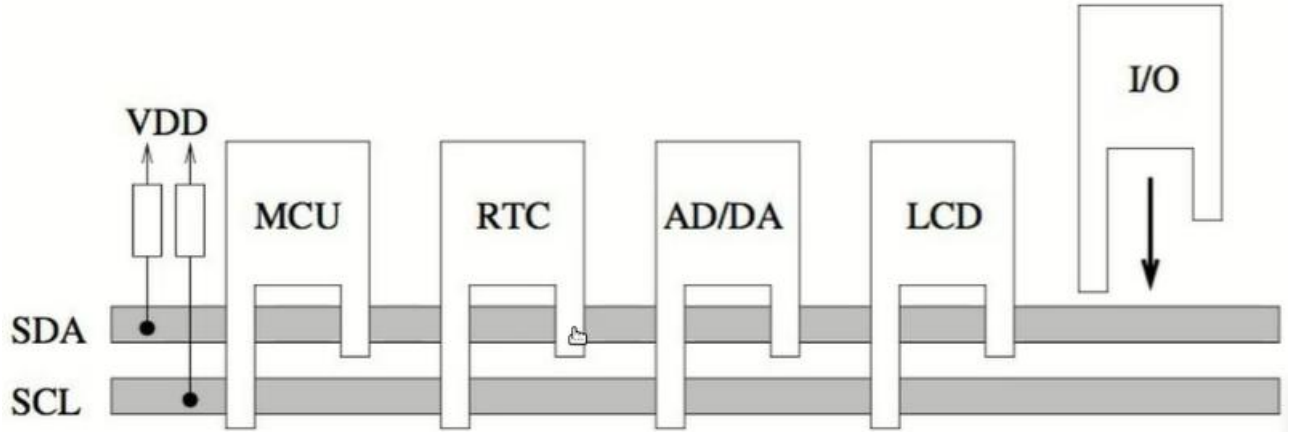
La comunicazione con I^2C è di tipo **half-duplex,** questo perché essendoci un'unica linea dei dati, non è possibile far circolare l'informazione in due sensi.

Il protocollo IIC è stato sviluppato da Philips ed è molto utilizzato per **comunicazioni ad alta velocità, a breve distanza** tra uno o più microcontrollori e periferiche.

L'IIC può essere configurato per lavorare a diversi **profili di velocità**:

- **Standard mode:** fino a **100Kbit/s**
- **Fast mode:** fino a **400Kbit/s**
- **High-speed mode:** fino a **3.4 Mbit/s**

È molto flessibile nell'aspetto della velocità e può "mixare" devices a diverse velocità.



Come si vede in figura, tutti i dispositivi connessi con IIC vengono connessi ai due bus. Questi ultimi sono connessi con delle resistenze di pull-up: quando una delle due linee impone 0 vanno entrambe a 0, quando non viene imposto, di default vengono settati a 1.
Quindi la linea SDA sarà 1 quando nessuno dei dispositivi collegati danno 0, basta un solo 0 per circuitare la linea intera.

Il protocollo IIC gode di **estensibilità:** si possono connettere dispositivi fintanto che il valore del bus non eccede ai 400pF

Addressing

Avendo un unico bus, nella trasmissione dell'informazione deve essere specificato il destinatario (lo slave) e quindi ci servirà conoscere il suo **indirizzo** così da avviare la comunicazione con esso.

[Differenza con la SPI: nella SPI vi è lo slave select che permette di "tracciare" una connessione punto-punto con lo slave destinatario mentre nella I^2C viene inviato un indirizzo nel bus, il destinatario avente tale indirizzo sarà l'unico a instaurare la comunicazione con il master]

Gli indirizzi possono essere a **7 bit** o a **10 bit** (estensione dell'indirizzo a 7 bit);
Nell'addressing a 7 bit si ha che i 4 bit più alti sono **hard-coded** dal fabbricatore: sono 4 bit che non possono essere manipolati (a meno che non venga riprogrammato l'intero codice);
["Hard coding": "codifica fissa", indica la prassi di introduzione di un codice sorgente che non può essere programmato, a meno che non venga riprogrammato per intero].
I 4bit più alti sono dunque utilizzati per identificare il creatore del device.
La Philips ha, inoltre, riservato una serie di indirizzi: $(0000XXX)_2$ e $(1111XXX)_2$, questo significa che, invece di $2^7 = 128$ bit di indirizzamento avremo un totale di **112 indirizzi utilizzabili.**
[Gli indirizzi 0000XXX e 1111XXX sono in totale 16:i bit che variano (contrassegnati dalla "X") sono 3 per indirizzo, quindi si hanno $2^3 = 8$ indirizzi: 8 per 0000XXX e 8 per 1111XXX per una totale di 16; quindi avremo 128 - 16 = 112 indirizzi utilizzabili (abbiamo sottratto i 16 indirizzi riservati)].

Più avanti si è utilizzato un **addressing a 10 bit** per **estendere** il già esistente addressing a 7 bit. Chiaramente, l'addressing utilizzante 10bit è **compatibile con quello a 7 bit**

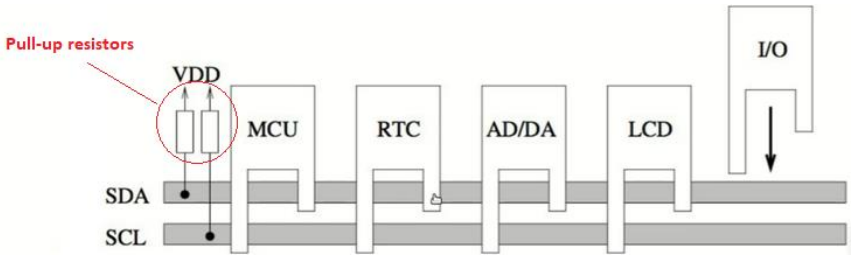
Trasmissione dei dati

Le linee sono collegate a dei **resistori pull-up esterni**: questo significa che non si può mai controllare il livello alto: **è possibile solo controllare il livello basso (0) e per avere il livello alto (1) si deve disconnettere, lasciando che sia il pull-up resistor a gestire il livello alto.** Quindi se si impone il valore 0 allora si avrà un'uscita bassa, altrimenti, non imponendolo, si avrà l'uscita alta.

Programmando quindi a basso livello non potremo guidare l'alto livello (1) e dovremo quindi proseguire come segue:

- Per programmare un output alto si setta il controller pin come input: è come se lo si disconnettesse (collegandolo al pull-up resistor). Scollegandolo è come se si passasse il livello di dafault (Quello del pull-up resistor) che è pari a 1, quindi si è ottenuto un output alto
- Per programmare un output basso basta settare l'output del pin a 0

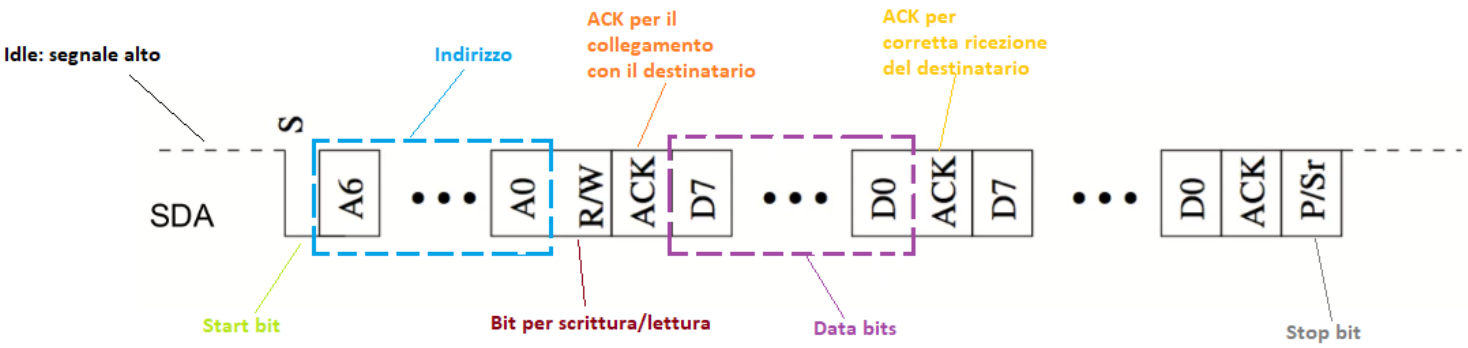
Quindi, in poche parole, viene programmato soltanto il segnale basso, poiché il segnale è alto di default.



Questo funzionamento ci porta ad affermare il concetto di **stato dominante e stato recessivo:** se vi è uno zero trasmesso da un devices, questo vincerà su tutti i possibili 1 degli altri devices: definiamo lo stato 0 come **stato dominante**, ogni volta che esso viene trasmesso vince su tutti gli altri stati. Definiremo allora lo stato 1 come **stato recessivo**.
Allora, ricapitolando: **lo stato di low (0) è uno stato dominante, lo stato di high (1) è recessivo.** Questa definizione è, praticamente, analoga a quanto detto: viene scritto solo 0, in quanto esso è il valore dominante.

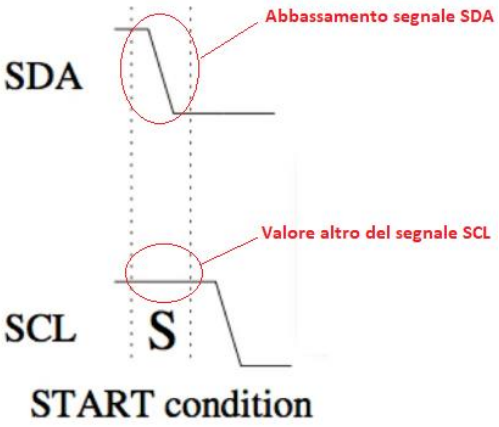
Il Frame IIC

Lungo la linea SDA vi è la trasmissione dei dati; vediamo la forma dei frame IIC che circolano lungo questa linea.

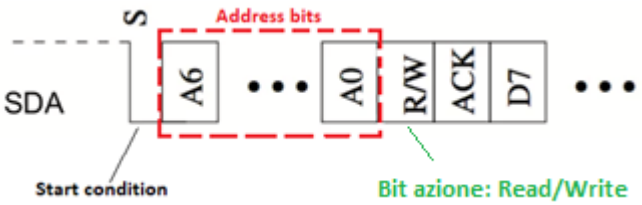


Come per la SPI, la situazione di quiete (stato di **idle**) (senza alcun segnale che passi lungo il canale) viene identificata dal **segnale alto**.
Quando deve iniziare la conversazione vi sarà, anche per questo frame, uno **start bit** che avrà valore **basso** per abbassare il segnale di idle e iniziare a trasmettere i dati**.**
Vi saranno poi **7 bit per l'indirizzamento**
Dopo la sezione di indirizzamento, vi è **un bit** che indica se il master (poiché, ricordiamo, la comunicazione parte sempre dal master) deve **leggere o scrivere.**
Dopo di che vi è un **ACK per il collegamento** (Acknowledgement): un particolare bit che sta ad indicare che il corretto collegamento con il destinatario (solo ed esclusivamente l'unico slave destinatario della conversazione).
A seguire vi sono **8 data bits** per la trasmissione dei dati
Successivamente, troviamo un altro **ACK per segnalare la corretta ricezione** del destinatario.
I bit dati si continuano a ripetere, insieme al bit di ACK per la corretta ricezione, fintanto che duri la comunicazione delle informazioni.
Infine, vi è uno **stop/repeat bit**, questo bit, diverso da quello dell'SPI, consente di terminare la conversazione o di iniziare una nuova trasmissione, facendo ripartire il ciclo spiegato (intraprendendo una conversazione anche con un altro slave).

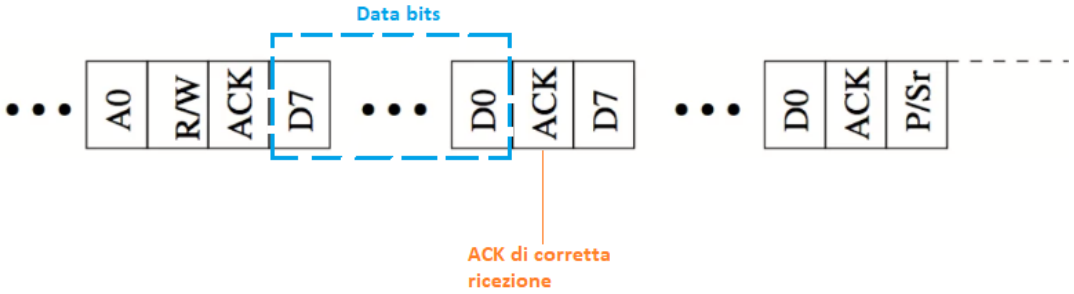
1. **Idle state:**
   Nell'idle state entrambi i segnali: SCL (System Clock) ed SDA (Segnale dati) sono **alti**
2. **Inizio della trasmissione:**
   La trasmissione viene avviata dal master; essa inizia quando si abbassa il segnale SDA (più o meno rapidamente), nel frattempo il senale SCL (System Clock) rimane alto. Quindi la comunicazione inizia con l'abbassamento del segnale SDA + il valore alto del segnale SCL [vale questa "doppia condizione"]. Una volta che il si abbassa il segnale SDA e viene mantenuto alto il segnale SCL, tutti gli slave si mettono in "allerta" per ricevere la richiesta di comunicazione attraverso l'addressing.



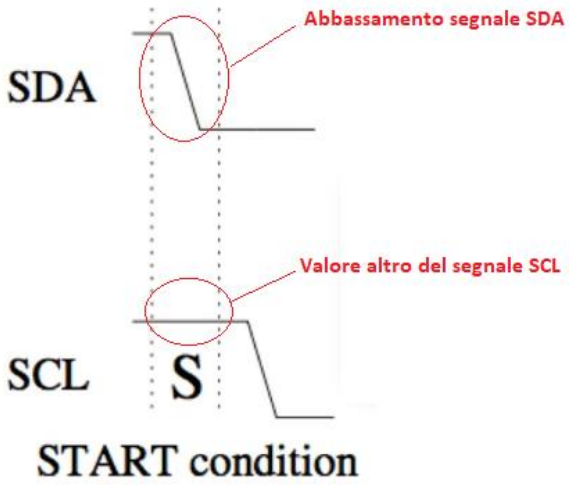
3. **Invio bit di addressing:**
   Dopo aver abbassato il segnale SDA, il master si occupa di inviare i bit di addressing che formano l'indirizzo dello slave destinatario
4. **Invio bit lettura/scrittura:**
   Dopo di che specifica l'operazione che vuole che vuole eseguire attraverso il bit R/W: lettura (Read: R) o scrittura (Write: W).



5. **Identificazione del destinatario (ACK):**
   Il destinatario deve "identificarsi": una volta che tutti ricevono l'indirizzo trasmesso dal master durante la "ricerca" dello slave destinatario, uno (ed uno solo) di essi dovrà in qualche modo identificarsi: lo fa attraverso il **bit di acknowledgement ACK.** A questo punto inizia la trasmissione: può avvenire che il master scrive verso lo slave o che il master legga dallo slave (dipende dal bit settato precedentemente: Read/Write);
6. **Invio dei data bits:**
   Una volta che è stato identificato il destinatario, inizia la trasmissione delle informazioni, ovvero dei data bits trasportanti le informazioni; ricordiamo che la trasmissione può essere in lettura o in scrittura a seconda del bit appositamente settato.
7. **Segnalazione di corretta ricezione:** viene infine mandato un altro **ACK** dal destinatario per comunicare la corretta comunicazione (che tutto è andato a buon fine); l'ACK lo invierà comunque l'entità che **riceve** l'informazione.
8. **Fine della trasmissione:**
   La comunicazione avviene fintanto che il master non generi una **stop condition (P)** che indica che la trasmissione è finita ed il bus sarà nuovamente libero: **si ritorna nello stato di idle (segnale SCL ed SDA alti);**
   Per una questione di ottimizzazione è stata creata la condizione di **Repeated Start** per evitare che si debba ricominciare tutta la conversazione da capo (con tanto si start bit): l'idea è quella di non stoppare la comunicazione liberando il bus ma di continuare ad utilizzarlo per trasmettere altre informazioni verso, eventualmente, altri indirizzi (sarà dunque necessario mettere in allerta tutti gli slaves).



Come abbiamo specificato, il bit di START è caratterizzato dal fatto che durante la sua trasmissione il segnale SDA si abbassa e quello di SCL è alto (high). Questa caratteristica, in realtà rispecchia solo i particolari bit di START E STOP: solo per essi, infatti, si ha un segnale SDA variabile ed un segnale SCL alto costante; nello specifico, **per lo START il segnale SDA passerà da alto a basso, per lo STOP il viceversa: passa dal basso all'alto mentre in entrambi i casi il segnale SCL resta costante.**

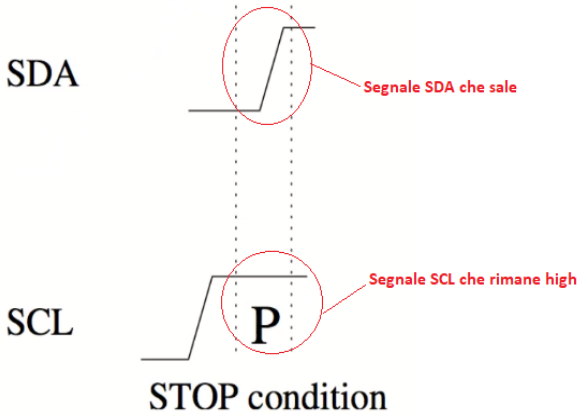


START condition

Differente è invece la situazione per gli altri segnali: **per tutti i segnali che non siano start e stop (ACK, DATA BIT, ADDRESSING) la trasmissione avviene con il segnale SCL basso (low).**

Quindi il master, stabilendo il clock alto o basso, può già distinguere l'invio di bit di start e stop dall'invio di bit informativi o di quei bit che servono ai fini della comunicazione: ACK, Data bit, Addressing. Ecco perché abbiamo detto che il segnale del clock SCL non serve solo per sincronizzare i clock di master e slaves ma ha anche delle funzionalità "in più" (a differenza della SPI).

[**NB:** si parla di livello low e/o high quando il master deve trasmettere, ovvero deve inviare informazioni sul bus]



STOP condition

Lo stesso discorso vale per il segnale START REPEAT (SR) che è identico alla normale condizione di start ma è posta al posto dello stop [In poche parole: si fa partire uno START al posto di uno STOP]. Se vi fossero più master vi sarebbero delle fasi di arbitraggio per scegliere il master che dovrà trasmettere prima della trasmissione. Questo funzionamento risulta utile nel caso di sistema multi-master perché evita di perdere cicli di clock con l'arbitraggio: continua a trasmettere lo steso master.

Gli Indirizzi (Address)

Gli indirizzi vengono trasmessi come dei normali bit informativi (data bit); si ricordi che in questo caso il segnale SCL è basso poiché SOLO I SEGNALI DI START/STOP SONO TRASMESSI CON SEGNALE SCL ALTO.

Dopo i 7 bit dell'indirizzo (qualora venisse utilizzato un addressing a 7 bit), **l'ottavo bit è utilizzato per il controllo della direzione: è il bit di R/W che stabilisce se il master deve leggere o scrivere**:

- **R/W high**: il master **legge** i dati dallo slave indirizzato
- **R/W low**: il master **invia** i dati allo slave indirizzato

Chiaramente in base a questo bit R/W si capirà chi dovrà inviare l'ACK: esso verrà inviato dall'entità che avrà ricevuto i dati.

Acknowledgement

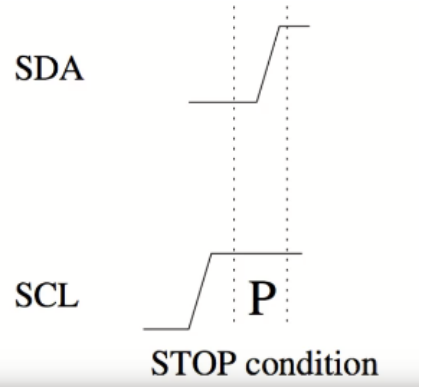
Ogni 8 Data bit il ricevitore dovrà inviare un segnale di ACK di conferma. Anche durante la trasmissione dei segnali di ACK il livello del clock sarà low. L'ACK è sempre 0 nella linea SDA: se dopo 8 bit viene ricevuto il tutto, il destinatario invierà 0.
Si ricordi che lo zero è il valore dominante: se nel bus vi è 0, questo "vincerà" su tutti gli 1 presenti.

Dati

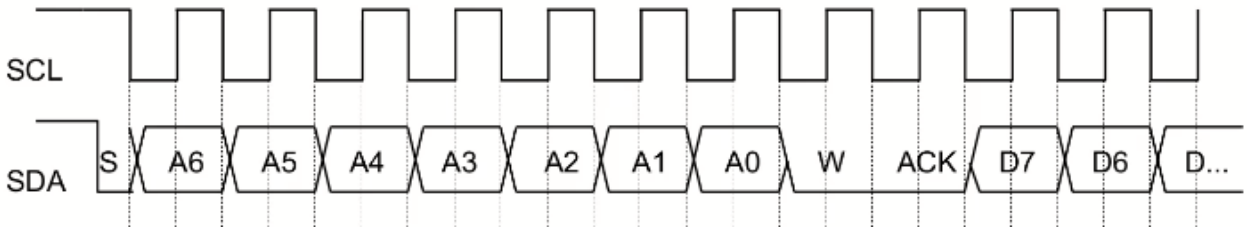
Nella trasmissione dei dati non vi è alcun limite: vengono inviati i dati ogni 8 bit ed essi sono seguiti da un ACK; questo procedimento va avanti fino allo STOP bit.

Stop

Il bit di stop è duale a quello di start: il segnale SCL resterà costante alto (high), mentre il segnale SDA, dualmente al segnale bit di start, passerò da basso ad alto.



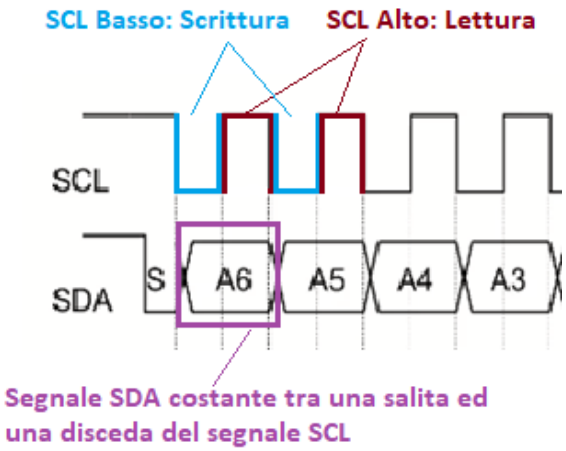
Esempio: funzionamento del protocollo I^2C - scrittura di 1 Byte



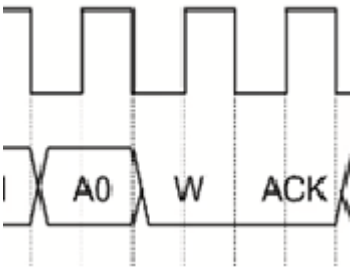
Inizialmente abbiamo la condizione di **idle**, fin quando non si ha il primo abbassamento del segnale SDA con segnale SCL ancora alto: questo sta ad indicare lo start; viene iniziata la trasmissione.

La linea SCL avrà allora il segnale low e nella linea SDA verrà allora trasmesso il primo bit di indirizzamento.
NB: i bit di indirizzamento, come quelli dei dati, hanno quella particolare formula per mantenere la genericità dei valori che possono assumere questi dati: essi infatti possono assumere arbitrariamente valore zero o uno, a seconda dell'indirizzo dello slave da "cercare" o dei dati da trasmettere. Quindi quella forma sta ad indicare che quel dato bit può essere o alto o basso

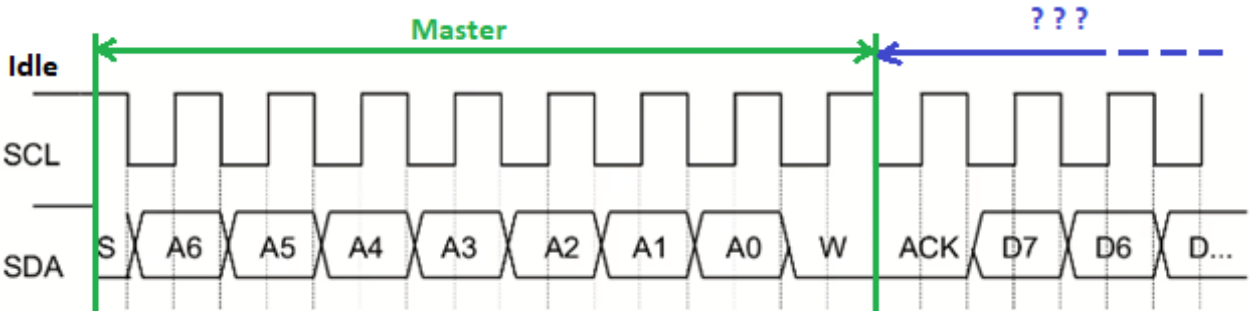
Durante il segnale clock basso (SCL low) verrà effettuata un'operazione di scrittura (come già ripetuto), mentre durante il segnale di clock alto (SCL Low) verrà effettuata una lettura (infatti il segnale si mantiene alto o basso sia in fase di scrittura che in fase di lettura: si osservi infatti che il segnale di clock è "altalenante" tra alto e basso - e questo alterna la fase di lettura e scrittura - mentre il segnale SDA resta costante per almeno una salita e una discesa del segnale SCL.



Dopo i bit di indirizzamento vi sarà il bit per il controllo della direzione della conversazione, che darà informazione se il master vorrà leggere o scrivere. Ricordiamo che: **0 → Write** e **1 → Read.** Anche in questo caso il bit di scrittura o lettura viene trasmesso quando il clock si trova sul fronte basso (quando il segnale SCL è basso), come si può notare in figura.

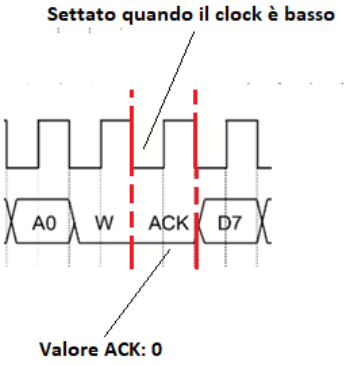


Fino a questo punto della trasmissione siamo sicuri che i bit sono stati trasmessi dal master poiché sarà quest'ultimo sarà ad avviare la trasmissione (che sia in scrittura o in lettura): i bit per ricercare l'address corrispondente allo slave con cui si vuole comunicare ed il bit per settare la lettura e la scrittura saranno quindi sicuramente trasmessi dallo slave che dovrà "cercare" lo slave e settare la conversazione come Read e Write.



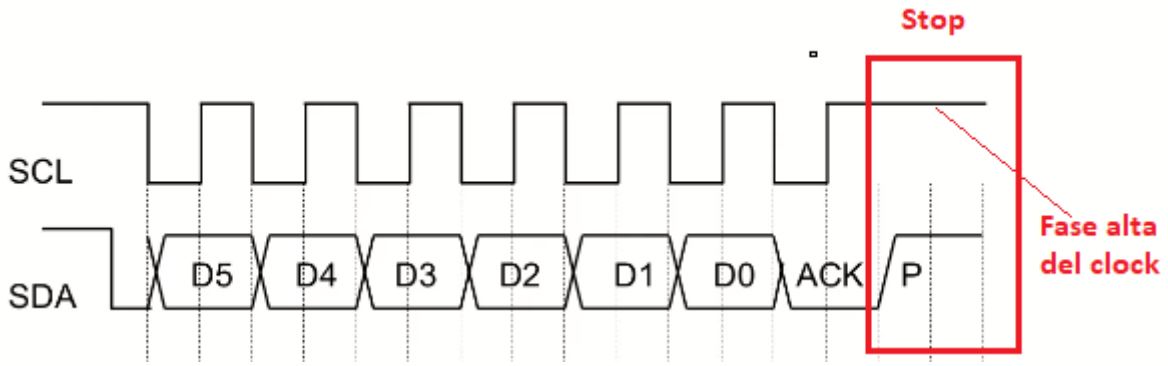
Dopo i primi 8 bit non possiamo sapere con certezza se la comunicazione sta spettando al master o ad uno slave.

Il vantaggio di questo protocollo sta proprio nell'utilizzo del segnale di clock non solo come segnale per la sincronizzazione dei clock in ricezione ed in trasmissione ma come parte integrante della comunicazione, che ci permette di distinguere la trasmissione di un segnale "speciale" per iniziare/finire la comunicazione (START e STOP o START REPEAT) dalla trasmissione dei semplici dati, inclusi tutti i bit di indirizzamento e di ACK



Arrivati alla fine dei data bit (o address bit) trasmessi abbiamo un ACK per Segnalare la corretta ricezione (o per identificare lo slave qualora il master avesse solamente trasmesso i bit dell'indirizzo dello slave che sta cercando. Si ricordi che l'ACK si comunica settando a 0 tale bit quando il segnale di clock è basso.

Alla fine della comunicazione avremo un segnale di stop (o eventualmente un segnale di start repeat per evitare che la comunicazione debba fermarsi per poi, eventualmente, riprendere subito dopo). Il segnale stop viene implementato quando si ha la fase alta del clock.

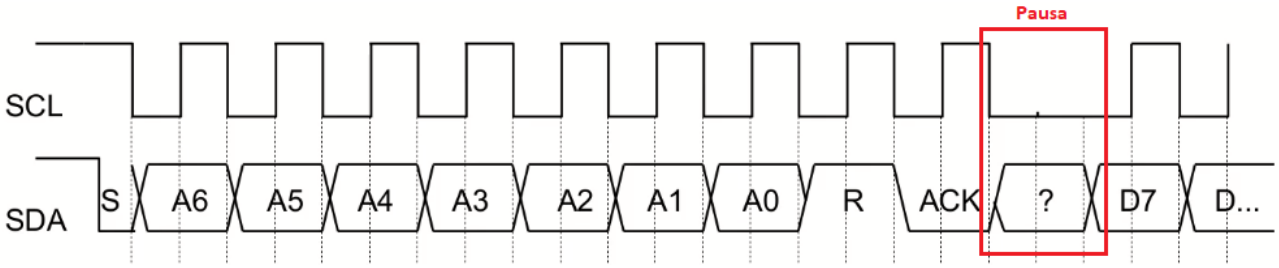


Speed Control

Il protocollo IIC è molto flessibile per quanto riguarda lo **speed control** da parte dello slave: se lo slave non ha la stessa velocità del master, ovvero se esso non riesce a stare "al passo" con la trasmissione, **il protocollo permette di mettere in pausa la trasmissione;**

La scrittura dei dati (insieme ad indirizzi ed ACK) avviene nelle fasi basse del clock, mentre le letture nelle fasi alte; quindi indipendentemente dalla velocità dei singoli, è il clock a stabilire entro quanto tempo verrà trasmesso il prossimo bit.

Quindi qualora lo slave ricevitore non riuscisse a "smaltire" i bit che riceve perché, magari, ha un buffer pieno, **può settare il segnale di clock a livello basso: setterà nella linea di clock il livello 0.** In questo modo, lo slave setterà il livello 0 e il master il livello 1 e dato che il livello 0 è quello dominante, lo slave riuscirà a "fermare il tempo" poiché così facendo il master capirà che non sarà momentaneamente possibile trasmettere bit. Il master riprenderà a trasmettere non appena lo slave setterà nuovamente ad 1 il segnale SCL.



Questo controllo è come se fosse un "**controllo di flusso".**