COMUNICAZIONE SERIALE - USART

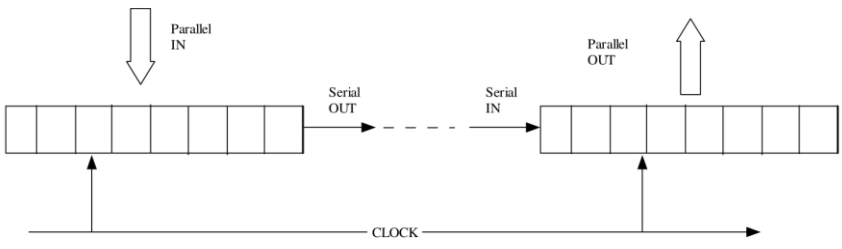
Queste periferiche, sia la USART che il PIC, sono da un lato programmabili e dall'altro utilizzabili con il processore 68K con il modello ad *interruzioni. La periferica seriale deve essere interfacciata all'interno di un'architettura di calcolatore più complessa per mettere in comunicazione nodi di sistemi differenti* che si basano su un protocollo diverso da quello parallelo, è infatti un protocollo di comunicazione seriale. **La comunicazione seriale, a differenza della parallela, permette di raggiungere distanze più elevate con hardware necessario per la comunicazione di gran lunga più semplice**, *abbiamo bisogno di pochi fili, due fili per la precisione per trasmettere e ricevere al fine di mettere in comunicazione nodi anche a grande distanza*. I vecchi modem, non quelli attuali in fibra, utilizzavano convertitori seriale-parallelo, cioè interfaccia seriale. **L'interfaccia seriale si interpone tra il calcolatore che è parallelo ed un mezzo di comunicazione seriale (un cavo)**, permette di trasformare il flusso di dati da parallelo, così come lo vede il calcolatore, a seriale come lo vede un cavo o un doppino telefonico. *È uno degli strumenti potentissimi perché permette di realizzare in modo semplice un protocollo di comunicazione*. Lo standard RS232 sembra abbandonato a favore dell'USB che è una sua evoluzione, in realtà in molti contesti industriali rimane il protocollo di comunicazione maggiormente adottato. Qualunque dispositivo hardware ha una porta seriale per collegare un terminale per accedere in maniera lite, con poca complessità, a tutte le comunicazioni. I protocolli all'interno di un'automobile che hanno il can bus o quelli delle catene di montaggio industriali modbus sono basati su **protocollo seriale,** cioè **un'unica linea che codifica sui singoli blocchi di bit, multipli di byte, il comando, controllo o messaggio che deve essere inviato**. Avendo un'unica linea, codificheremo una sequenza di bit in funzione dei messaggi che vogliamo scambiare, cioè dati o comandi, *dobbiamo capire allora come controllare sincronizzazione, errori, corretta ricezione, tutto su un'unica linea*. Dobbiamo capire le caratteristiche della comunicazione seriale, l'interfaccia ed il modello di programmazione associato.

Facciamo riferimento ad un componente dell'Intel, che è *8251A, la USART*, **è un ricevitore e trasmettitore universale che può essere programmato per poter lavorare in due modalità differenti, la modalità sincrona e asincrona**. Abbiamo usato più volte i termini sincrono ed asincrono relativamente ai bus oppure ai protocolli verso la memoria. In questo caso **facciamo riferimento a come i bit vengono mandati tra le due interfacce seriali per poter sincronizzare l'informazione che stiamo mandando, in altre parole per non perdere caratteri**. Progettata per la comunicazione con i processori della famiglia Intel, ne vedremo adattamento al 68k, completamente programmabile.

Questo trasmettitore converte dati che riceve dal processore in parallelo, dato il parallelismo del modello (nel nostro caso 8 bit), **e li converte con uno shift register in uno stream seriale di dati**. Riceviamo 8 bit alla volta, il cosiddetto carattere, li accogliamo in un SR ed in maniera opportunamente tempificata mandiamo su un'unica linea in uscita i dati serializzati. Viceversa, **in ricezione riceviamo un flusso seriale di bit attraverso un SR, una volta che l'SR è pieno degli 8 bit viene copiato in un buffer interno, registro con stesso parallelismo che permette di inviare questi dati direttamente al processore**. La USART può segnalare al processore se è disponibile ad accettare un nuovo carattere da trasmettere oppure se ha ricevuto un nuovo carattere che può comunicare alla CPU**. La periferica può comunicare con il processore attraverso due linee di interruzione, trasmission ready ed accept, mentre la CPU può leggere lo stato della USART in ogni istante perché si possono verificare errori, quindi dobbiamo capire se la trasmissione è avvenuta con successo oppure no**. Poi ci sono segnali di controllo che permettono di gestire le interruzioni, che possiamo realizzare interrogando il registro di stato della periferica.

Le modalità di trasmissione tra due interfacce seriali possono essere simplex, half duplex o full duplex. Nella ***modalità simplex***, esiste **un'unica linea monodirezionale**, avremo una periferica che trasmette sempre e l'altra che può solo ricevere. Un oggetto è programmato solo per trasmettere, l'altro solo per ricevere. Le ***modalità half duplex e full duplex*** permettono di realizzare una **comunicazione bidirezionale** con l'unica differenza è che nella prima esiste una linea bidirezionale, abbiamo sempre un'entità che trasmette e l'altro riceve, quindi una fa da trasmettitore e l'altra da ricevitore (le due periferiche possono essere programmate affinché si possano scambiare i ruoli); nella seconda modalità entrambe le periferiche possono comunicare, trasmettendo e ricevendo, contemporaneamente.

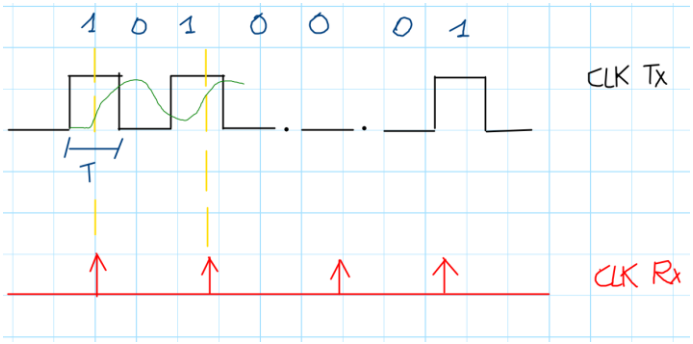
Per vedere qual è il principio di funzionamento, è come se avessimo due SR da una parte e dall'altra collegati con una linea seriale, in parallelo possiamo scrivere nel registro a sinistra tempificato da un clock che permette di mandare in uscita un bit alla volta in maniera seriale; sulla destra abbiamo un altro SR che in ingresso riceve i bit e con un clock della stessa frequenza permette di ricevere in ingresso i bit ricevuti e leggerli in parallelo per inviarli al processore. In questa figura, **il clock è in linea di principio condiviso tra i due SR perché devono essere fasati**. *Immaginiamo un clock di 100 Hz in trasmissione ed uno in ricezione a 50 Hz, il ricevitore prende un bit ogni due mandati, cioè perderebbe la metà delle informazioni*. **Non possiamo mandare sulla linea di comunicazione tra i due oggetti il clock, nasce il problema di sincronizzare i due oggetti**. L'informazione elementare



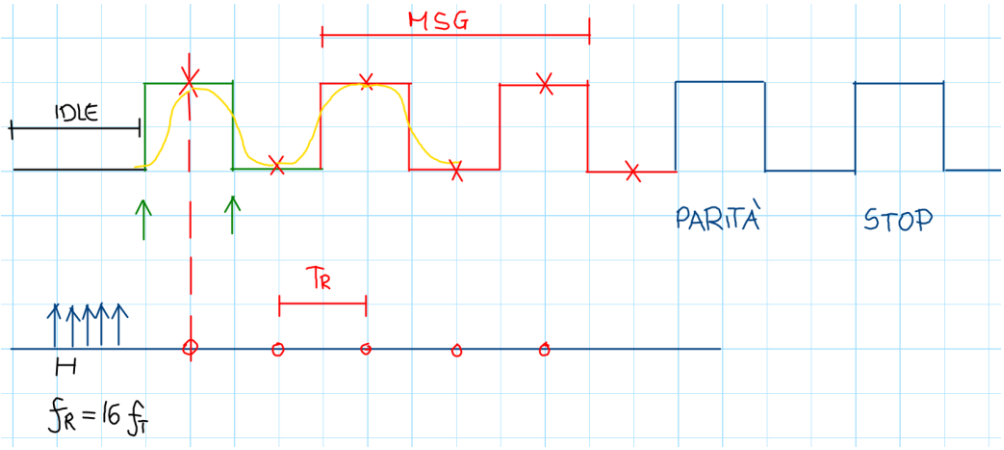
trasmessa è un carattere la cui lunghezza dipende dalla lunghezza dello SR, nel nostro caso il numero massimo di bit che possiamo mandare sarà di 8 bit**. In funzione della modalità di trasferimento dobbiamo fare in modo che in questi 8 bit ci sia non solo tutta l'informazione ma anche la parte di controllo**, lo **start bit** che indica l'inizio del messaggio, lo **stop bit o coda** per la fine, il messaggio centrale vero e proprio. Possiamo avere all'interno del singolo messaggio inviato anche bit di controllo, ***bit di parità***, perché **la linea seriale è soggetta a molti errori in quanto il segnale può essere degradato, affetto da disturbi data la lunghezza della linea stessa**. Dobbiamo gestire in maniera opportuna eventuali errori.

Comunicazione asincrona

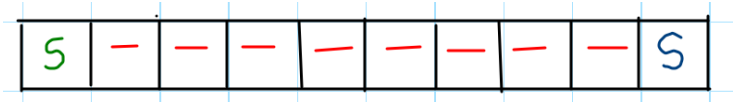
Abbiamo bisogno di mettere all'interno del messaggio una serie di informazioni che permettono di controllare il messaggio stesso. **Non dobbiamo perdere lo start bit perché altrimenti non riusciamo a sincronizzare i due oggetti**. Supponiamo di voler mandare il segnale blu, 10100010. *Che frequenza ha questo messaggio?* Ha una certa frequenza, sappiamo che il periodo di trasmissione è dato dall'intervallo T. Supponiamo che in ricezione invece abbiamo un clock fatto così (rosso). Cosa cogliamo del messaggio inviato? Alla prima freccia prendiamo 1, alla seconda 1, poi 0 e infine 1. **Se questi sono il clock in ricezione e il clock in trasmissione, abbiamo ricevuto un messaggio completamente diverso da quello mandato**. Abbiamo messo il clock giusto al centro, ma *il segnale reale non sarà mai squadrato, ha tipicamente l'andamento verde*, per cui se il clock del ricevitore fosse stato questo rosso avremmo letto 0 e non 1.



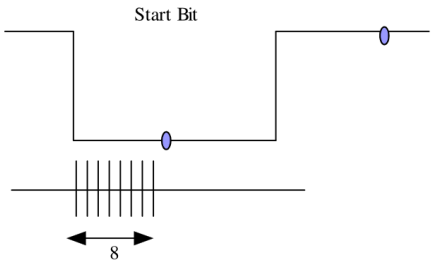
Il primo problema è capire **quando comincia il segnale che è di mio interesse**, per fare questo introduciamo in concetto di **start bit**. Questo è caratterizzato dal fatto che quando non stiamo usando la linea seriale, **dobbiamo mandare un segnale di idle, alto o basso, in modo tale che quando si presenta un nuovo bit significativo bruciamo il primo bit per dire che da quel momento inizia la trasmissione**. Dallo start bit ricevuto, inizia il messaggio vero e proprio. Se il clock in trasmissione è questo verde, in ricezione dobbiamo fare in modo che **in idle la frequenza di campionamento della linea in ricezione sia molto più elevata della frequenza di trasmissione**, tipicamente ad 8/16/32 a seconda della rumorosità del segnale la frequenza di trasmissione. In pratica dobbiamo avere una frequenza fatta come in figura (frecce blu). Con questa frequenza elevata **in qualunque punto si presenta lo start bit ce ne accorgiamo perché aumentando la frequenza non è possibile che perdiamo la transizione basso alto**. Il segnale reale potrebbe avere l'andamento giallo: **quando il ricevitore scopre che sta cambiando il segnale con la stessa frequenza si pone esattamente al centro del messaggio inviato**, dello start bit, e **da quel punto in poi abbassa la frequenza alla stessa del mittente affinché tutti gli altri messaggi siano campionati esattamente al centro del messaggio successivo**. In questo modo, *il periodo di ricezione viene aumentato ed è uguale alla frequenza di trasmissione*. Prima fr è circa sedici volte la frequenza del trasmettitore, dopo il primo bit di start viene abbassata la frequenza, quindi ***il periodo di ricezione viene aumentato e settato uguale al periodo di trasmissione, ma shiftato di un certo numero di colpi di clock per stare a centro bit*** *in questo modo tutti i messaggi in rosso, che sono il messaggio reale che vogliamo trasmettere, non vengono perduti perché campioniamo sulle x rosse*. Anche se avessimo la situazione in giallo o in generale una situazione molto rumorosa, non perdiamo il messaggio. Possiamo avere dei bit di controllo, per esempio un bit di parità e poi abbiamo lo stop bit che indica la terminazione della trasmissione. **Possiamo programmare il numero di caratteri del messaggio, da 5 ad 8, il numero di bit di controllo, il numero degli stop bit, il fattore moltiplicativo del periodo di ricezione**.



Questa è la modalità asincrona, dove la sincronizzazione è contenuta nel messaggio stesso, perché nel messaggio abbiamo una parte iniziale di start, una parte centrale di messaggio fatta da un certo numero di bit, una parte finale di controllo e di stop. *La sincronizzazione avviene all'interno del messaggio, inizio e fine, gli errori si possono gestire nel messaggio e il messaggio vero e proprio è il corpo centrale degli 8 bit*.



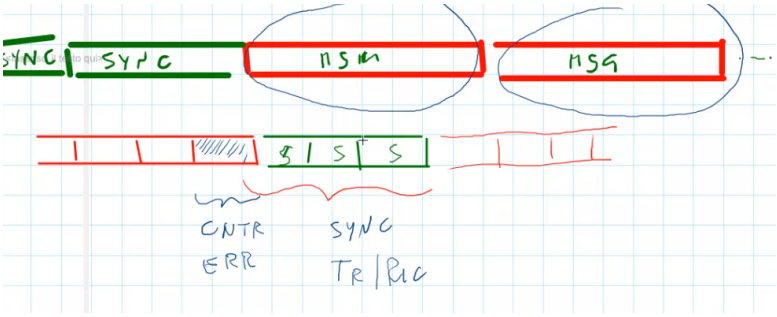
Il clock di ricezione è settato ad una velocità superiore in modo da non perdere la transizione dello start bit, abbiamo ipotizzato che il segnale di idle sia alto. Il clock della ricezione si rifasa, **il campionamento viene rifasato con uno shift di un certo numero di impulsi per portarsi a metà rispetto alla frequenza del clock di trasmissione**, supponendo di avere 16 volte come fattore moltiplicativo viene shiftato di 8 impulsi di clock per posizionarsi al centro dello start bit e campionare i bit successivi al centro. **Nella comunicazione asincrona sono necessarie informazioni aggiuntive all'interno del messaggio per controllare che non si sia persa questa sincronizzazione**, abbiamo un **overhead** per controllare messaggio per messaggio che non venga persa la sincronizzazione costituita da un bit di parità per ogni carattere. *Il controllo di parità può essere per l'interno messaggio e si potrebbe pensare alla parità longitudinale aggiungendo la check sum*, per ricostruire in qualche modo quale messaggio è stato corrotto.



Comunicazione sincrona

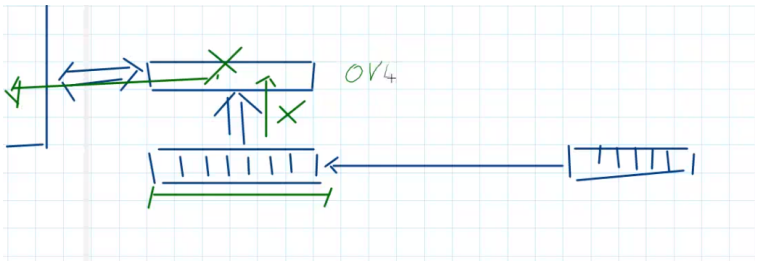
La modalità sincrona cerca di evitare questi problemi con approccio differente poiché i bit di parità rappresentano un overhead notevole, possiamo eliminare lo start e stop bit, i bit di parità e fare rilevamento degli errori più sofisticato. Nella modalità sincrona, **mandiamo tutti gli 8 bit che costituiscono il carattere rappresentanti una parte di dati significativi e la sincronizzazione viene fatta in maniera separata usando dei byte speciali**, *che in numero predeterminato rappresentano sequenze speciali di sincronizzazione*, **per la correzione degli errori si usano codici ridondanti sull'intero frame indipendente dalla lunghezza del messaggio**.

Il **singolo messaggio** che vogliamo mandare sarà **codificato su tutti gli 8 bit**, avremo **per la sincronizzazione** dei caratteri speciali della stessa lunghezza che sono dei **sync**. *Ogni tanto possiamo decidere di mandare un certo numero di sync che è programmabile*, per esempio 3 e poi di nuovi il messaggio. Grazie a questi caratteri di sync possiamo periodicamente risincronizzare trasmettitore e ricevitore. In aggiunta a questi possiamo mettere alla fine del messaggio uno o più byte di controllo (in blu in figura). Questo permette di avere dei dati da mandare estremamente più interessanti, perché i **dati sono a tutta lunghezza, non abbiamo un forte overhead sul singolo messaggio come prima**, **sulla linea seriale mandiamo tutti i dati utile, solo alla fine facciamo un controllo ed ogni tanto con una certa periodicità risincronizziamo ricevitore e trasmettitore per essere sicuri che i clock non si siano sfasati**. Sulle lunghe linee ciò è estremamente utile, tuttavia se ci sono degli errori siamo costretti a rimandare tutto, mentre nella modalità asincrona se avessimo degli errori potremmo rimandare solo il singolo byte perché ce ne accorgiamo subito. Quindi, **nella modalità sincrona nel caso di errore dobbiamo mandare tutto il frame, mentre nella modalità asincrona anche se paghiamo overhead notevole nel singolo messaggio, nella modalità asincrona abbiamo controllo maggiore**. Nelle due modalità **abbiamo trade off tra possibilità di errore e "velocità"** perché la velocità dipenderà anche dai clock di ricezione e trasmissione e dall'affidabilità del canale, se il canale fosse molto rumoroso una trasmissione del genere potrebbe costringere a rispedire stesso dato che il controllo degli errori darà spesso esito negativo (genera errori) e dovremo rimandare i messaggi. Nella comunicazione sincrona la sincronizzazione viene fatta con un numero predeterminato di caratteri che hanno codice prestabilito e sono detti caratteri di sync, mentre per la correzione degli errori in genere si applica un codice ridondante sull'intero frame (check sum).



Tipologie di errori

Ci sono *alcuni errori legati alla trasmissione ed altri legati al malfunzionamento della periferica*. **L'overrun si verifica nel dispositivo ed è causato dal fatto che abbiamo una coppia di registri**, *quello che legge/scrive verso il processore e lo SR*. **Cosa succede se un dato viene copiato da SR a buffer prima che questi venga spedito alla CPU?** Il registro (blu) comunica in maniera bidirezionale con la CPU. Supponendo di voler leggere, questo registro comunica direttamente con l'SR che è invece collegato in ingresso alla linea seriale. *Appena arrivano 8 bit nell'SR vengono copiati nel registro buffer da dove saranno prelevati e copiati nella CPU*. Cosa succede se questa scrittura x viene prima che il buffer sia stato letto? Se si verifica questa condizione, abbiamo l'errore di overrun, **il dato viene copiato dallo shift register nel registro prima che quest'ultimo fosse stato svuotato nella CPU, cioè il dato è perso**. **L'errore di parità si verifica quando fallisce il controllo sulla parità**, quando facciamo nella modalità asincrona un controllo sul bit di parità scopriamo che c'è stato qualche errore sul messaggio e quindi **dobbiamo richiedere la**



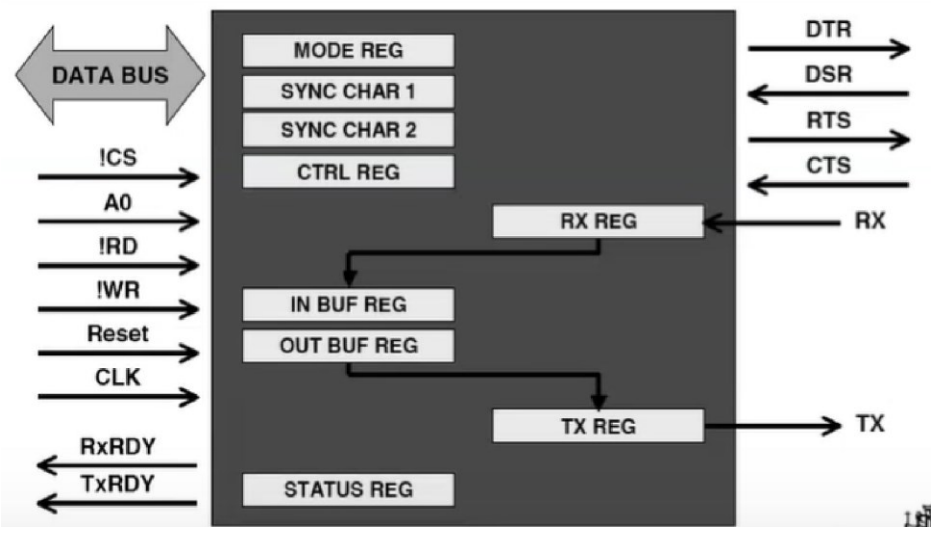
ritrasmissione del messaggio perché il messaggio è arrivato corrotto. **L'errore di framing avviene quando si perde la sincronizzazione** (tipicamente nella modalità sincrona), quindi ci aspettavamo un carattere di sync, ne troviamo un altro, abbiamo perso la sincronizzazione tra ricevente e trasmettitore. La comunicazione seriale avviene su canali molto rumorosi, quindi bisogna tener conto di questi errori.

Possiamo identificare **quattro sezioni differenti**, **una sezione che permette la programmazione del dispositivo, quindi lo scambio di comandi e dati con il processore**. Abbiamo **due sezioni per la trasmissione e ricezione** che *contengono rispettivamente lo shift register e il buffer register per la trasmissione e la coppia SR e buffer temporaneo per la ricezione*, infine abbiamo **una sezione di tempificazione che si occupa del rate generator**, cioè *la frequenza di clock per poter trasmettere i dati*. **Il modello di programmazione consiste nell'inizializzare il dispositivo per poter funzionare in un modo**. Per fare questo abbiamo a disposizione due registri differenti (a differenza della PIA che ne aveva uno solo) che sono il **registro MODE** e il **registro COMMAND**. *Abbiamo un modo particolare per scrivere nei vari registri e* ***non abbiamo tutto lo spazio di indirizzamento lineare così come accadeva la PIA***. **Il registro MODE ed il registro COMMAND sono nello stesso indirizzo, ma la prima volta che accediamo alla periferica, dopo un'operazione di reset, il device considera come mode instruction il primo comando ricevuto**. La prima operazione di scrittura che facciamo sul dispositivo accederà al registro di modo, **le istruzioni successive sono tutte trattate come parole di comando, indirizzate al registro COMMAND**. Se invece lo abbiamo programmato per funzionare in modalità sincrona, **accederemo a dei registri particolari che memorizzano la modalità di sincronizzazione che vogliamo utilizzare**.

Abbiamo un modello memory mapped, dobbiamo associare ai registri interni della periferica. La tecnica utilizzata per accedervi è differente da quella vista nella PIA. *Nella PIA la prima volta che accediamo entriamo nel registro di controllo CR e il prossimo accesso all'indirizzo viene reindirizzato internamente al registro successivo*. Nella USART, invece, **usiamo sempre una tecnica di indirizzamento interna ma che funziona in base all'ordine con cui accediamo**. Inizializziamo la periferica, subito dopo il reset accediamo per la prima volta al registro di modo, quindi programmiamo la periferica per funzionare in un certo modo. **Da quel momento in poi ogni volta che accederemo a quello stesso indirizzo, invece, invieremo specifiche linee di comando. Se però nella modalità modo abbiamo indicato una comunicazione sincrona, dobbiamo indicare anche quali sono i caratteri di sincronizzazione**, il sync-sync è un carattere speciale predeterminato ma dobbiamo programmare la periferica a definire la codifica del carattere (tutti 0 o tutti 1, conviene un carattere con valore esadecimale riconoscibile e distinguibile).

Segnali di comunicazione

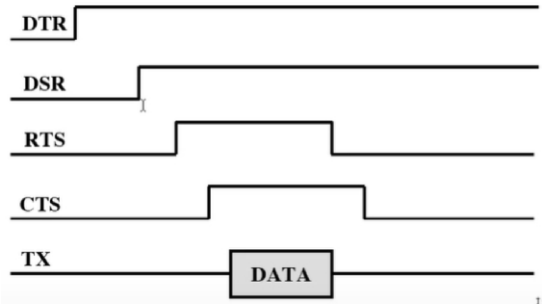
La periferica seriale può comunicare con un'altra interfaccia seriale ma anche con un'altra periferica seriale, quindi possiamo avere periferica seriale-interfaccia seriale – processore, per esempio il mouse è una periferica che si collega via interfaccia seriale al processore, ma possiamo voler utilizzare due interfacce seriali che mettano in comunicazione due PC completamente differenti. Per esempio, un modem è un'interfaccia seriale che metteva in collegamento due PC da remoto. I vari segnali che abbiamo a disposizione sono **i segnali che conterranno i dati veri e propri**, che secondo lo standard RS232 si chiamano **TD** ed **RD**, ***trasmission data e receiving data***. Ci sono **quattro segnali per stabilire protocollo di comunicazione tra periferica e la sua interfaccia**, *in altre parole per stabilire l'handshake.* Prima i modem su linee a 56k facevano dei rumori assurdi all'inizio della comunicazione, questi rumori erano associati al protocollo di handshake che metteva in comunicazione il processore con il terminale di trasmissione e quando finivano i rumori la trasmissione era stabilita e da quel momento in poi potevano essere mandati i dati, stava avvenendo la sincronizzazione tra la periferica e l'interfaccia. Tra i 4, **RTS, request to send, e CTS, clear to send, indicano lo stato della periferica a trasmettere**: abbiamo un'interfaccia che manda la periferica una richiesta di trasmissione, la periferica risponde con un clear to send, cioè è pronta a ricevere i dati. **Il protocollo di handshake vero e proprio è dato dalla coppia di segnali DSR, data set ready, e DTR, data terminal ready, per indicare che trasmittente e ricevente sono collegati in trasmissione ed il terminale è pronto a comunicare.**



Questo è lo schema del dispositivo in ASIM. Partiamo dal basso, ci sono 4 registri: ***RX REG***, che è l'SR in ricezione che comunica direttamente con **IN BUF REG** che è il reg temporaneo che accoglie gli 8 bit ricevuti; la coppia di registri in trasmissione ***TX REG***, SR in trasmissione, e ***OUT BUF REG***, buffer a parallelismo 8 nel processore. Nella parte in alto abbiamo il ***MODE REG*** con alcuni registri per memorizzare uno o più caratteri di sync, poi abbiamo il ***CTRL REG*** e alla fine lo ***STATUS REG***.

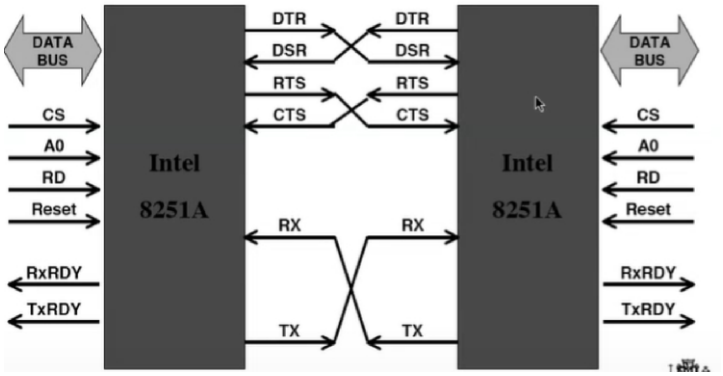
Nella comunicazione verso il dispositivo seriale o verso un'altra interfaccia seriale (lato destro dello schema) ci sono le due linee di comunicazione RX e TX su cui viaggiano i dati e poi i 4 segnali che implementano il protocollo. Sulla sinistra invece ci sono tutti i segnali di comunicazione verso il processore, quindi **A0 è la sola linea di indirizzamento per indirizzare tutti i registri interni** (nella PIA 2 per indirizzare 6 registri, adesso 1 per indirizzare 7 registri perché **gli SR non sono accessibili direttamente dal processore**). *Avremo bisogno di una tecnica di indirizzamento interna molto più complessa*. **Ci sono due linee di interruzione, RxRDY e TxRDY, che comunicano la disponibilità dell'interfaccia a trasmettere o ricevere, a seconda di come abbiamo programmato la periferica**. Infine, c'è in ingresso il bus ad 8 bit che permette di leggere e scrivere i buffer temporanei e di avviare lo scambio dei messaggi con il processore.

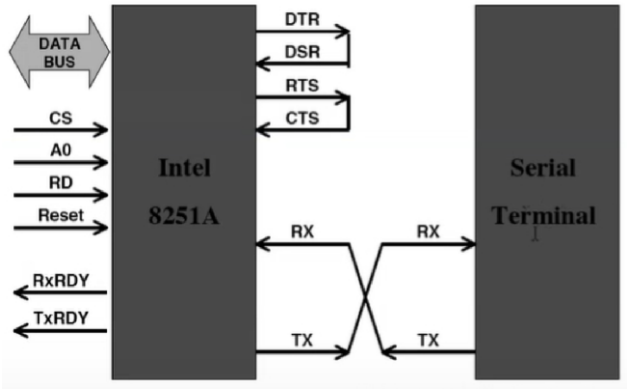
Funzionamento handshake

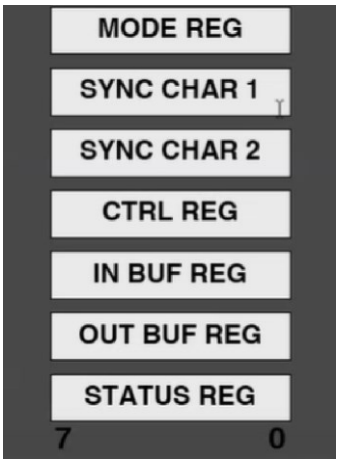


Inizialmente settiamo la periferica a comunicare con l'interfaccia per dire che siamo pronti a trasmettere e ricevere. Il terminale dice che vuole trasmettere, la periferica comunica di essere pronta a trasmettere. Da questo momento in poi richiediamo di trasmettere, diamo l'ok e trasmettiamo veramente i dati. Appena i dati sono finiti chiudiamo la trasmissione e finiamo. Possiamo sia spegnere la periferica abbassando DTR e DSR, sia continuare così per accettare i nuovi dati dopo.

Lo ***schema di connessione completo*** è il seguendo, abbiamo **due interfacce seriali che comunicano tra loro**, i dati di ricezione e trasmissione sono incrociati tra loro. Anche i segnali del protocollo sono intrecciati tra loro, il DTR va in ingresso al DSR dell'altro e viceversa, così come RTS e CTS. La modalità che noi usiamo spesso è la **modalità diretta**, anche detta **parziale**, in cui **tutti i segnali di protocollo sono ritenuti abilitati di default**, soprattutto DTR e DSR, quindi bypassiamo questi segnali per non implementare questa parte del protocollo, ma consentire direttamente la comunicazione tra una periferica seriale e la sua rispettiva interfaccia. **Il modello di programmazione che avremo a disposizione è dato da questi registri** in figura.

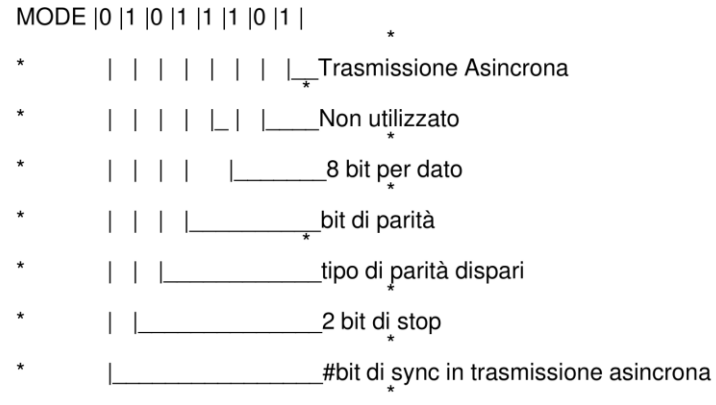






Per attivare la trasmissione dobbiamo definire la modalità di scambio, la prima cosa da fare è decidere se comunicare in maniera sincrona o asincrona. Se vogliamo comunicare in maniera **sincrona** dobbiamo indicare **quanti saranno i caratteri di sincronizzazione e quali saranno** (il codice di sincronizzazione). Se vogliamo utilizzare la **modalità asincrona**, dobbiamo indicare **il fattore moltiplicativo del clock in ricezione, la lunghezza effettiva del carattere da 5 ad 8 bit, il tipo di controllo di parità ed il numero di stop bit**. **Successivamente nella parola di command scriviamo in cui possiamo abilitare la trasmissione oppure chiedere il reset di una periferica**. Poi possiamo attivare il segnale verso la CPU, cioè ***possiamo abilitare l'interruzione di trasmission ready in attesa che il carattere da trasmettere sia nel buffer***. **Il carattere viene trasferito nel serializzatore se libero, si abilita il segnale TxC che scandisce la serializzazione del canale e poi si ricomincia di capo con il prossimo carattere**. *Se la CPU ritarda nel caricare il buffer vengono inviati dei caratteri di sync nel caso di modalità sincrona oppure caratteri di idle nel caso di modalità asincrona*.

Mode Register



Il MODE contiene 8 bit: **il bit 0 indica la comunicazione sarà asincrona o sincrona** (0 per sincrona, 1 per asincrona); **il bit 1 non viene utilizzato**; **i bit 2 e 3 indicano quanti bit effettivi costituiscono il messaggio** (00 indica 5 bit, 01 indica 6 bit, 10 indica 7 bit, 11 indica 8 bit); **il bit 4 indica se è previsto o meno il bit di parità** (1 se è previsto il bit di parità); **il bit 5 indica se la parità è pari o dispari** (1 per pari, 0 per dispari); **il bit 6 determina in una trasmissione asincrona il numero di bit stop** (0 per 1 e 1 per 2); **il bit 7 determina in una trasmissione sincrona il numero di caratteri di sincronizzazione** (0 per un sync, 1 per 2 sync). Se abbiamo indicato modalità sincrona dobbiamo settare questo bit e scrivere successivamente il codice del carattere di sincronizzazione.

Questo registro è accessibile solo in write e vi si accede la prima volta che accediamo in write ad un indirizzo dispari della periferica. **La periferica ha un'unica linea di indirizzo A0, se il bit meno significativo è 1 (è dispari) e l'operazione asserita è di write stiamo accedendo al registro di MODE**.

Registro INBUFREG

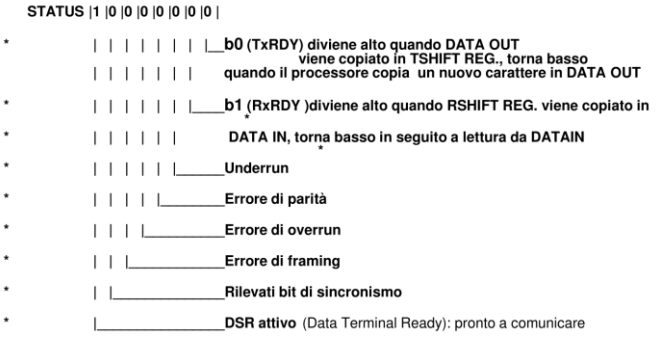
Questo viene **scritto dal processore per trasferire un byte alla volta, viene riempito con il dato ricevuto serialmente e convertito in parallelo. Non fa parte del modello di programmazione del device. Vi si accede con un read asserito all'indirizzo pari**. Se accediamo ad indirizzo pari con operazione di lettura, stiamo leggendo il dato ricevuto. **L'accesso ad esso azzera il bit 1 del registro di stato perché significa che abbiamo completato l'interruzione di servizio che andava a leggere il valore.**

Registro OUTBUFREG

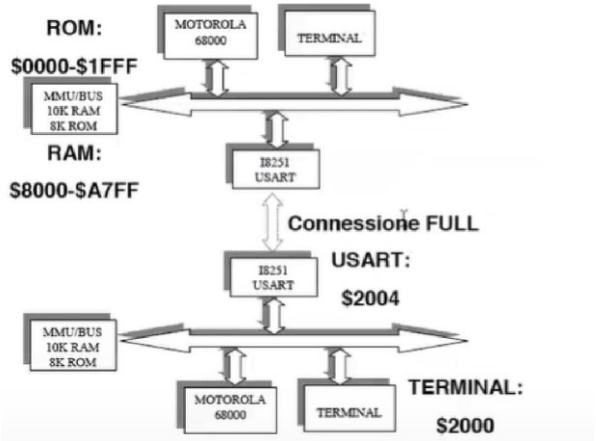
Viene riempito con il dato proveniente dal processore che deve essere inviato serialmente. Il dato viene copiato nel registro seriale che esegue la conversione. Anche questo non fa parte del modello di programmazione del device. **E' di tipo write only, si accede con write asserito ad indirizzo pari**. Quando accediamo ad indirizzo pari, se siamo in ready accediamo ad INBUF, se siamo in write accediamo ad OUTBUF. **L'accesso a questo registro azzera il bit 0 del registro stato**.

Status Register

Il registro stato conterrà gli ultimi due bit 0 ed1 per le interruzioni associate alle due line TxRDY e RxRDY. Il registro stato può essere usato dal programmatore, settando **bit 0 e bit 1 che abilitano le richieste di interrupt**. Gli altri bit sono usati o per la gestione degli errori oppure per il protocollo. **Il bit 2 in una trasmissione sincrona indica che il processore non ha alcun carattere da trasmettere**; **i bit 3-4-5 indicano se si è verificato uno dei 3 errori** (parità, overrun o framing). **Il bit 6 indica se sono stati rilevati i caratteri di sincronismo previsti**; **il bit 7 indica se è stato attivato handshaking da DSR**. Anche questo registro è di sola lettura e **vi si accede ad indirizzo pari in lettura**.



Un esempio di configurazione è la comunicazione tra due processori che comunicano tra loro tramite USART, così come visto la comunicazione tra due processori tramite PIA. Possiamo usare un componente aggiuntivo di ASIM che è il ***componente terminal***, interessante perché c'è sia terminale seriale che il terminale parallelo (possiamo vedere a video i messaggi che vogliamo mandare). Anche il terminale andrà programmato. Vedremo le configurazioni in ASIM di questi oggetti. Il dispositivo ASIM si chiama I8251 USART ed è caratterizzato dal fatto di avere **come insieme di indirizzi a cui andare a mappare in memoria questo dispositivo un indirizzo base che sarà un indirizzo pari ed un secondo indirizzo che è l'indirizzo successivo a quello base**. Con due indirizzi, pari e dispari, riusciamo ad accedere ai diversi registri in funzione del tipo di operazione che stiamo andando a fare. Ci sarà una logica



leggermente più complessa nella periferica che indica dove andare se stiamo accedendo in lettura.
Per abilitare la ricezione, dopo aver inizializzato la periferica con una MODE word, **dobbiamo abilitare nella parola command il segnale di RxEN**. *La periferica cerca lo start bit per sincronizzarsi ed abilita lo SR a ricevere gli n bit, dove n è stato definito nella parola di comando precedente*. Viene eseguito il test sia sul singolo carattere sia sul frame, in funzione del tipoo di controllo che abbiamo settato prima. Fatto questo, trasferisce il frame nel buffer e viene alzata l'interruzione verso la CPU, la RxRDY, per indicare che è pronta la ricezione**. Il processore legge sul buffer temporaneo e se RxRDY era già alto verrà generato un errore di ovverrun. RxRDY già alto significa che il processore non aveva completato la lettura precedente e quindi è stato sovrascritto il registro ed il dato perduto**. Inoltre, *abbiamo una modalità di esecuzione in cui si riceve il carattere di sync e lo si controlla per verificare che sia uguale a quello previsto*, si chiama **HUNT mode**. Una volta riconosciuto il bit di sincronizzazione viene inviato l'intero flusso di byte, il clock di campionamento è settato sulla variazione del fronte 1-0 presente nel messaggio stesso. Il flusso viene fatto ripartire e non più arrestato fino alla ricezione del carattere di sync successivo, posto in genere in una posizione prefissata.

Il **control register** è questo in figura.

```
CNTRL |1 |1 |1 |1 |0 |1 |1 |1 |
*          | | | | | | | |__Abilita trasmettitore (TxEN)
*          | | | | | | |____Attiva DTR (Data Terminal Ready):
                                    pronto a comunicare
*          | | | | | |______Attiva ricevitore  (RxEN)
*          | | | | |________Non utilizzato  (forza TxD a 0)
*          | | | |__________Azzera bits di errore in STATUS
*          | | |____________Attiva RTS (Request to Send): indica
                                    intenzione a trasmettere;
*          | |______________Resetta la periferica
*          |________________Ricerca sincronizzazione
```

Dal datasheet del dispositivo vediamo i 4 blocchi principali, la parte del blocco di ricezione, di trasmissione, di programmazione della logica di controllo e di gestione di trasferimento con il buffer. Poi c'è la parte di controllo che si chiama *modem control che fa riferimento ai 4 segnali di handshake*. Il buffer di trasmissione è collegato direttamente con il data bus buffer che permette di mandare sulla linea seriale TxD ed è collegato direttamente alla logica di controllo del blocco di trasmissione, che invia verso il processore due segnali specifici, l'interruzione TxRDY e TxE (in asim non lo vediamo) per avvisare il processore per l'abilitazione di tutte le fasi di trasmissione.