Versione N° 2.0

LaurTec

Libreria CAN per PIC

Autore : *Mauro Laurenti*

ID: PJ11002-IT

INFORMATIVA

Come prescritto dall'art. 1, comma 1, della legge 21 maggio 2004 n.128, l'autore avvisa di aver assolto, per la seguente opera dell'ingegno, a tutti gli obblighi della legge 22 Aprile del 1941 n. 633, sulla tutela del diritto d'autore.

AVVERTENZE

I progetti presentati non hanno la certificazione CE, quindi non possono essere utilizzati per scopi commerciali nella Comunità Economica Europea.

Chiunque decida di far uso delle nozioni riportate nella seguente opera o decida di realizzare i circuiti proposti, è tenuto pertanto a prestare la massima attenzione in osservanza alle normative in vigore sulla sicurezza.

L'autore declina ogni responsabilità per eventuali danni causati a persone, animali o cose derivante dall'utilizzo diretto o indiretto del materiale, dei dispositivi o del software presentati nella seguente opera.

Si fa inoltre presente che quanto riportato viene fornito così com'è, a solo scopo didattico e formativo, senza garanzia alcuna della sua correttezza.

L'autore ringrazia anticipatamente per la segnalazione di ogni errore.

Indice

Introduzione

Quando la comunicazione tra due sistemi elettronici richiede l'utilizzo del bus CAN, una libreria già funzionante può agevolare lo sviluppo del progetto. La libreria CAN qui proposta ha lo scopo di agevolare e stimolare il progettista alla scrittura di una propria libreria. Il protocollo CAN può essere compreso a pieno solo se si percorrono i vari passi, ovvero dalla lettura delle specifiche alla scrittura o utilizzo di una libreria.

Il modulo CAN dei PIC

Scopo di questo paragrafo è quello di riassumere le caratteristiche principali del modulo CAN interno ai PIC della famiglia PIC18Fxx8. Durante la spiegazione si riterrà noto quanto scritto nel Tutorial "Il protocollo CAN" scaricabile dal sito LaurTec. Per ulteriori dettagli sul modulo CAN si rimanda al datasheet del rispettivo PIC utilizzato.

La famiglia PIC18Fxx8 possiede un modulo CAN nominato ECAN (la E sta per Enhanced). Tale modulo è totalmente compatibile con il modulo CAN della famiglia PIC18Cxx8. Il miglioramento consiste nel fatto che possono essere gestite più maschere e filtri nonché un buffer di ricezione e trasmissione migliorato. Il modulo ECAN è compatibile con le specifiche CAN 2.0 A e B e supporta sia la modalità standard che estesa. Il massimo bit rate supportato è 1Mbits ed una lunghezza dati compresa tra 0 e 8 byte. Il modulo ECAN può essere impostato su tre modalità di funzionamento:

- mode 0: Legacy mode (compatibile con la serie PIC18Cxx8).
- mode 1: Modalità enhanced .
- mode 2: Modalità FIFO con supporto DeviceNet .

ogni modalità di funzionamento possiede sei modalità operative

- Configuration mode.
- Disable mode.
- Normal operation mode.
- Listen Only mode.
- Loopback mode.
- Error Recognition mode.

Configuration mode

Ogni volta che bisogna inizializzare il modulo ECAN, scrivere una maschera od un filtro è necessario prima impostare questa modalità.

Disable mode

Tale modalità disabilità sia il modulo di ricezione che di trasmissione del modulo ECAN.

Normal operation mode

Questa modalità è quella che viene normalmente utilizzata nelle normali fasi di trasmissione e ricezione.

Listen Only mode

Questa modalità rappresenta una modalità "silenziosa", ovvero il modulo ECAN è impostato solo in ricezione e non trasmette nulla sul bus.

Loopback mode

Questa modalità permette di collegare la linea di trasmissione del PIC con la linea di ricezione. Tale modalità può risultare utile per ragioni didattiche o diagnosi[1].

Error Recognition mode

Per mezzo di questa modalità il modulo ECAN viene impostato in maniera tale da ricevere tutti i messaggi ignorando ogni errore. Questa modalità può risultare particolarmente utile in fase di diagnosi.

Per ulteriori informazioni si rimanda al datasheet del microcontrollore utilizzato.

Analisi della libreria

La libreria è stata scritta facendo uso del linguaggio XC8[2] ed utilizzando per semplicità la modalità di funzionamento mode 0 (Legacy mode). Il codice sorgente è possibile trovarlo al sito www.laurtec.it. Poiché i registri WIN non sono stati utilizzati, il codice non risulta ottimizzato per quanto riguarda le dimensioni, ciononostante non perde nulla in termini di velocità di esecuzione[3]. Prima di utilizzare il modulo ECAN è necessario eseguire i seguenti passi:

- Selezionare la modalità di configurazione
- Selezionare la modalità operativa
- Selezionare il baud rate (velocità di trasmissione)
- Impostare i filtri e le maschere
- Selezionare la modalità operativa richiesta dall'applicazione

Dopo questi passi è possibile iniziare la ricezione e la trasmissione di messaggi. Quanto esposto risulta semplificato dall'utilizzo della libreria qui proposta, la quale permette di ottenere un buon livello di astrazione per l'utilizzo del modulo ECAN.
Per mezzo della libreria qui presentata i passi precedenti, grazie all'astrazione ottenuta, possono essere così riformulati:

- Inizializzazione del modulo CAN
- Selezionare la modalità di configurazione (se si richiedono i filtri e/o maschere)
- Impostare i filtri e le maschere (se richiesti)
- Selezionare la modalità operativa richiesta dall'applicazione

[1] Nonostante la linea TX ed RX vengono internamente collegate tra loro è necessario il collegamento di tale linee ad un Transceiver esterno, tipo l'MCP2551. Se così non si facesse il PIC non è in grado di cambiare modalità operativa. Questa necessità non è ben spiegata sui datasheet della Michrocip e può creare rompicapi!

[2] Per maggiori dettagli sul linguaggio di programmazione XC8 si rimanda al libro "XC8 Step by Step" scaricabile dal sito LaurTec.

[3] La Microchip mette a disposizione una libreria sia per il modulo CAN che ECAN.

Al fine di utilizzare la libreria è necessario includere il file module_CAN.h e module_CAN.c nel progetto.

Il modulo CAN fa utilizzo dei pin RB2 e RB3 della PORTB. Il programmatore deve accertarsi che RB3 sia posto ad 1 all'interno del registro TRISB. Il valore di RB2 è indifferente dal momento che viene automaticamente sovrascritto dal modulo CAN.

Le funzioni disponibili nella libreria sono:

Nome	Operazione
CAN_operation_mode	Imposta la modalità operativa del modulo CAN.
CAN_open	Inizializza il modulo CAN.
CAN_close	Disabilita il modulo CAN.
CAN_write_message	Invia un messaggio.
CAN_read_message	Legge un messaggio dal buffer di ricezione.
CAN_set_mask	Imposta le maschere di ricezione del modulo CAN.
CAN_set_filter	Imposta i filtri di ricezione del modulo CAN.
CAN_is_TX_ready	Controlla se il modulo CAN può trasmettere un nuovo messaggio.
CAN_is_RX_ready	Controlla se è stato ricevuto un nuovo messaggio.
CAN_is_TX_warning_ON	Controlla se il modulo CAN ha TX warning attivo.
CAN_is_RX_warning_ON	Controlla se il modulo CAN ha RX warning attivo.
CAN_is_TX_passive	Controlla se il modulo CAN è TX passive.
CAN_is_RX_passive	Controlla se il modulo CAN è RX passive.
CAN_get_TX_error_count	Legge l'indice degli errori in trasmissione.
CAN_get_RX_error_count	Legge l'indice degli errori in ricezione.
CAN_is_BUS_OFF	Controlla se il modulo CAN è bus off.
CAN_abort_messages	Cancella i messaggi nel buffer di trasmissione non ancora inviati.

CAN_operation_mode

La seguente funzione permette di cambiare la modalità di funzionamento del modulo CAN. Ogni qualvolta bisogna cambiare una maschera o filtro è necessario impostare la modalità di configurazione per mezzo del parametro CAN_OP_MODE_CONFIG. Tale funzione risulta bloccante, ovvero fino a che non viene cambiata la modalità la funzione rimane attiva. Il cambio di modalità potrebbe non essere effettuato subito qualora ci siano dei messaggi pendenti. Se si dovesse far uso degli interrupt la funzione può essere trasformata in non bloccante rimuovendo il ciclo `while` interno alla funzione stessa.

Prototype

```
void CAN_operation_mode (enum CAN_OP_MODE mode);
```

Parametri

- **mode**
  Il parametro `mode` può assumere uno dei valori riportati in Tabella.

Valore	Significato
CAN_OP_MODE_NORMAL	Questa è la modalità che viene normalmente utilizzata per la ricezione e trasmissione di messaggi.
CAN_OP_MODE_SLEEP	Questa modalità mette in stato di sleep il modulo CAN, il quale non partecipa più alla comunicazione sul bus.
CAN_OP_MODE_LOOP	Per mezzo di questa modalità è possibile creare un link tra il modulo TX ed RX del PIC. Si ricorda che il Transceiver deve essere comunque presente.
CAN_OP_MODE_LISTEN	Per mezzo di questa modalità il modulo CAN si mette in ricezione senza realmente partecipare alla comunicazione sul bus. Questa modalità può risultare utile in caso si voglia intercettare in maniera automatica la frequenza con cui avviene la comunicazione sul bus.
CAN_OP_MODE_CONFIG	Per mezzo di questa modalità è possibile cambiare i filtri e le maschere del modulo CAN.

Esempio

```
CAN_operation_mode(CAN_OP_MODE_CONFIG);
```

CAN_open

Per mezzo della seguente funzione è possibile configurare il modulo CAN in modo che possa divenire operativo. Questa deve essere la prima funzione ad essere eseguita prima di adoperare il modulo CAN.
La funzione

```
CAN_operation_mode (CAN_OP_MODE_CONFIG);
```

viene automaticamente chiamata, dunque non è necessario impostare tale configurazione manualmente[4]. Al termine dell'inizializzazione la funzione cambia automaticamente la modalità da CAN_OP_MODE_CONFIG a CAN_OP_MODE_NORMAL. Se è richiesta una differente modalità bisogna richiamare la funzione CAN_operation_mode () con l'opportuno parametro. Durante l'inizializzazione la funzione CAN_open () resetta le maschere ed i filtri in modo da ricevere ogni messaggio. Se la funzionalità delle maschere e filtri dovesse essere richiesta è necessario impostarli individualmente (fra non molto si vedrà come).

Prototype

```
void CANInitialize (BYTE propSeg, BYTE phaseSeg1, BYTE phaseSeg2,
                    BYTE SJW, BYTE BRP, enum CAN_CONFIG_FLAGS flags);
```

Parametri

- **propSeg**
  Il parametro propSeg può assumere un valore compreso tra 1 e 8. Viene utilizzato per impostare il valore PROP_SEG (per maggiori dettagli si vedano le specifiche CAN) sfruttato per compensare eventuali ritardi di propagazione sulla linea di trasmissione e transceiver.

- **phaseSeg1**
  Il parametro phaseSeg1 può assumere valori compresi tra 1 e 8. Viene utilizzato per impostare il valore PHASE_SEG1 (per maggiori dettagli si vedano le specifiche CAN). Al termine del PHASE_SEG1 avviene il campionamento del bit ricevuto.

- **phaseSeg2**
  Il parametro phaseSeg2 può assumere valori compresi tra 1 e 8. Viene utilizzato per impostare il valore PHASE_SEG2 (per maggiori dettagli si vedano le specifiche CAN).

- **SJW**
  Il parametro SJW (Synchronized Jump Width) può assumere il valore compreso tra 1 e 4. Viene utilizzato per impostare il SYNC_SEG (per maggiori dettagli si vedano le specifiche CAN) che è sfruttato dai vari nodi per effettuare la sincronizzazione della fase del clock interno. Questo avviene sommando o

[4] Finora si è detto che la modalità di configurazione è necessaria solo per cambiare la maschera ed i filtri. Questo non è in realtà vero, infatti la modalità di configurazione dovrebbe essere impostata anche per cambiare le impostazioni del modulo CAN. La libreria qui introdotta nasconde questo particolare.

sottraendo quanti temporali dai parametri phaseSeg1 e phaseSeg2.

- **BRP**
  Il parametro BRP rappresenta il valore del prescaler per mezzo del quale si effettua la divisione della frequenza generata dall'oscillatore principale (interno o esterno che sia). Il suo valore può essere compreso tra 0 e 63.

la divisione della frequenza principale avviene come segue.

111111 = TQ = (2 x 64)/FOSC
111110 = TQ = (2 x 63)/FOSC
:
000001 = TQ = (2 x 2)/FOSC
000000 = TQ = (2 x 1)/FOSC

cioè

$$T_Q = \frac{2 \cdot (BRP + 1)}{F_{OSC}}$$

il valore T_Q rappresenta il quanto temporale.

- **flags**
  Il parametro flags può assumere una combinazione dei valori riportati in Tabella. Una combinazione di più valori deve essere fatta come nell'esempio sotto riportato, ovvero per mezzo dell'operatore binario and &.

Valore	Significato
CAN_CONFIG_PHSEG2_PRG_ON	Specifica di utilizzare il valore per il phase segment 2 impostato dall'utente.
CAN_CONFIG_PHSEG2_PRG_OFF	Disabilita la funzione precedente ed ignora il valore impostato dal parametro phaseSeg2.
CAN_CONFIG_LINE_FILTER_ON	Disabilita il filtro di linea.
CAN_CONFIG_LINE_FILTER_OFF	Abilita il filtro di linea.
CAN_CONFIG_SAMPLE_ONCE	Imposta un solo impulso di campionamento.
CAN_CONFIG_SAMPLE_THRICE	Imposta tre impulsi di campionamento.
CAN_CONFIG_DBL_BUFFER_ON	Abilita il doppio buffer in ricezione.
CAN_CONFIG_DBL_BUFFER_OFF	Disabilita il doppio buffer in ricezione
CAN_CONFIG_ALL_MSG	Permette la ricezione di tutti i messaggi, sia in modalità standard che estesa, con o senza errori. Questa funzione è utile in casi di debug.
CAN_CONFIG_VALID_XTD_MSG	Permette di accettare solo messaggi in modalità estesa.
CAN_CONFIG_VALID_STD_MSG	Permette di accettare solo messaggi in modalità standard.
CAN_CONFIG_ALL_VALID_MSG	Permette di accettare tutti i messaggi sia in modalità standard che estesa, ignorando però i messaggi con errori.

Esempio

Dal momento che il segnale di clock non è trasmesso assieme al messaggio, al fine di permettere la comunicazione tra i nodi, è necessario che ognuno di essi sia capace di rigenerare il clock internamente. Questo mette in evidenza il fatto che ogni nodo deve trasmettere alla stessa frequenza nominale al fine di semplificare la ricostruzione del clock. La frequenza massima che il bus CAN attualmente ammette è 1Mbits/s[5]. Tale frequenza viene determinata come:

$$f_{NOM}=1/T_{BIT}$$

ovvero

$$T_{BIT}=1/f_{NOM}$$

quindi il T_{BIT} minimo è pari a $1\,\mu s$.
Il periodo T_{BIT} può essere pensato come composto dall'unione dei seguenti campi:

- Synchronization Segment, SYNC_SEG.
- Propagation Segment, PROP_SEG.
- Phase Segment 1, PHASE_SEG1.
- Phase Segment 2, PHASE_SEG2.

ognuno di questi campi è composto da un numero intero di quanti T_Q , variabile a seconda delle esigenze. Si capisce dunque che:

$$T_{BIT}=T_Q\cdot(Sync_{SEG}+Prop_{SEG}+Phase_{SEG1}+Phase_{SEG2})$$

il valore T_Q è un valore fisso che viene a dipendere dalla frequenza di clock principale[6], espressa in MHz, e il valore di prescaler BRP utilizzato, ovvero:

$$T_Q=\frac{2\cdot(BRP+1)}{F_{OSC}}$$

T_{BIT} deve essere composto da un minimo di $8T_Q$ e un massimo di $25T_Q$.

Se per esempio si ha un $F_{OSC}=16\text{MHz}$ e si vuole trasmettere alla frequenza di 125KHz avendo $T_{BIT}=16T_Q$ si ha:

$$T_{BIT}=\frac{1}{f_{NOM}}=\frac{1}{125000}=0{,}000008\,s=8\,\mu s$$

quindi:

$$T_Q=\frac{T_{BIT}}{16}=0{,}5\,\mu s$$

[5] Le specifiche del CAN FD estendono il massimo bit rate a valori anche superiori a 10Mb/s.
[6] Qualora si faccia uso del PLL interno per moltiplicare la frequenza di clock, come frequenza bisogna considerare quella in uscita dal PLL.

$$BRP = \left(\frac{F_{OSC} \cdot T_Q}{2} \right) - 1 = \left(\frac{16 \cdot 0{,}5}{2} \right) - 1 = 3$$

gli altri valori possono essere scelti in maniera tale che la somma dei quanti sia 16, ovvero:

- Synchronization Segment, SYNC_SEG = 1
- Propagation Segment, PROP_SEG = 4
- Phase Segment 1, PHASE_SEG1 = 6
- Phase Segment 2, PHASE_SEG2 = 5

in generale Synchronization Segment uguale ad 1 è sufficiente. Per gli altri valori si deve rispettare quanto segue:

$$Prop_{SEG} + Phase_{SEG1} \geq Phase_{SEG2}$$

e

$$Phase_{SEG2} \geq SJW$$

L'inizializzazione potrebbe avvenire come segue:

```
CAN_opern (4,6,5,1,3, CAN_CONFIG_LINE_FILTER_OFF &
                      CAN_CONFIG_SAMPLE_ONCE &
                      CAN_CONFIG_ALL_VALID_MSG &
                      CAN_CONFIG_DBL_BUFFER_ON);
```

Per ulteriori informazioni si rimanda al datasheet del microcontrollore utilizzato.

Una volta aperto il modulo CAN è possibile disattivarlo per mezzo della funzione `CAN_close ()` o metterlo in stato di sleep per mezzo della funzione `CAN_operation_mode(CAN_OP_MODE_SLEEP);`

CAN_close

Per mezzo di questa funzione è possibile chiudere il modulo CAN. L'esecuzione della funzione close è equivalente a mettere in stato di sleep il modulo CAN.

Prototype

```
void CAN_close (void);
```

Esempio

```
// Chiude il modulo CAN
CAN_close ();
```

CAN_write_message

Per mezzo di questa funzione è possibile inviare un messaggio composto da un massimo di 8 byte. Ogni messaggio è identificato da un identifier e alcuni parametri aggiuntivi (si veda il parametro flags). Il livello di priorità che è possibile impostare non ha nulla a che vedere con le specifiche CAN ma è una caratteristica dei microcontrollori PIC. Ogni volta che il modulo CAN deve inviare un messaggio presente in uno dei tre buffer di trasmissione, invia prima il messaggio a più alta priorità. Prima di inviare un messaggio è bene accertarsi, per mezzo della funzione `CAN_is_TX_ready ()`, se è possibile trasmetterlo.

Prototype

```
void CAN_write_message (unsigned long identifier, BYTE *data,
                        BYTE dataLength, enum CAN_TX_MSG_FLAGS flags);
```

Parametri

- **identifier**
  Per mezzo di questo valore è possibile impostare l'identificatore del messaggio.

- ***data**
  Questo parametro rappresenta un puntatore all'array di caratteri (`unsigned char`) che deve contenere i dati da trasmettere. La dimensione massima dell'array è di 8 caratteri.

- **dataLength**
  Per mezzo di questo parametro è possibile impostare la lunghezza del pacchetto dati, ovvero il numero di byte dell'array di caratteri precedentemente descritto, che devono essere effettivamente inviati.

- **flags**
  Il parametro `flags` può assumere una combinazione dei valori riportati in Tabella. Una combinazione di più valori deve avvenire come nell'esempio sotto riportato, ovvero per mezzo dell'operatore binario and &.

Valore	Significato
CAN_TX_STD_MSG	Specifica che il messaggio è in formato Standard.
CAN_TX_XTD_MSG	Specifica che il messaggio è in formato Esteso.
CAN_REMOTE_TX_FRAME	Specifica che il messaggio è un Remote Frame (senza dati).
CAN_NORMAL_TX_FRAME	Specifica che il messaggio contiene dati (messaggio normale).
CAN_TX_PRIORITY_0	Imposta priorità di trasmissione 0 (bassa priorità).
CAN_TX_PRIORITY_1	Imposta priorità di trasmissione 1.
CAN_TX_PRIORITY_2	Imposta priorità di trasmissione 2.
CAN_TX_PRIORITY_3	Imposta priorità di trasmissione 3 (alta priorità).

Esempio

```
// array per i dati
BYTE info[8];

info[0] = 0x56;
info[1] = 0x01;

// controlla che sia possibile inviare il messaggio
while (!CAN_is_Tx_ready());

CAN_write_message (0x000FFFF0, info,2, CAN_TX_XTD_FRAME &
                                       CAN_NORMAL_TX_FRAME &
                                       CAN_TX_PRIORITY_2);
```

CAN_read_message

Per mezzo di questa funzione è possibile prelevare un messaggio dal buffer di ricezione. Il valore restituito, se maggiore di 0, identifica un errore di overflow del buffer. Se il valore ritornato è 1 significa che si è avuto un errore di overflow sul buffer 1. Se il valore ritornato è 2 significa che si è avuto un errore di overflow sul buffer 2. Prima di usare questa funzione è bene controllare per mezzo della funzione `CAN_is_Rx_ready ()` se sono stati ricevuti messaggi[7].

Prototype

```
BYTE CAN_read_message (CANmessage *msg);
```

Parametri

- ***msg**
  Tale parametro rappresenta un puntatore ad una struttura CANmessage. Tale struttura è definita all'interno del file module_CAN.h e può essere liberamente usata dal programmatore. La definizione della struttura è:

```
typedef struct {

 unsigned long identifier;
 BYTE data[8];
 BYTE type;            //1 = IDE   0=standard
 BYTE length;          // data length
 BYTE RTR;             //Remote flag. 1 it means remote frame

} CANmessage;
```

La funzione, una volta ricevuto il puntatore alla struttura, riempe i vari campi in funzione del messaggio ricevuto. Si noti che per passare l'indirizzo della struttura è necessario far uso dell'operatore &.

Esempio

```
// creo la struttura dati per la lettura
CANmessage msg;

while (1) {

     // controllo che sia presente un messaggio
     if (CAN_is_RX_ready ()) {

           //leggo il messaggio
           CAN_read_message (&msg);

           if (msg.data[0] == 0x03) {

                      // se data[1]=0x03 fai questo...
           }
     }
}
```

[7] Con questa affermazione si considera il fatto che il programmatore non sta facendo uso delle interruzioni.

CAN_set_mask

Per mezzo di questa funzione è possibile impostare le due maschere associate ai buffer di ricezione. Si ricorda che sono presenti due maschere una per ogni buffer di ricezione. Per poter impostare le maschere è necessario impostare la modalità di configurazione. Se le maschere vengono attivate permettono di mascherare alcuni bit dell'identificatore del messaggio ricevuto, prima di essere confrontato con il valore dei filtri. Se un bit della maschera vale 1 vuol dire che il corrispondente bit dell'identificatore verrà confrontato con il valore dei filtri (spiegati a breve). Se uno dei bit della maschera vale 0 il bit verrà ignorato e non confrontato con il filtro[8].

Prototype

```
void CAN_set_mask(enum CAN_MASK mask_number, unsigned long val,
                                  enum CAN_CONFIG_FLAGS type);
```

Parametri

- **mask_number**
  I valori che può assumere il parametro `mask_number` sono:

Valore	Significato
CAN_MASK_B1	Permette di selezionare la maschera del primo buffer.
CAN_MASK_B2	Permette di selezionare la maschera del secondo buffer.

- **val**
  La variabile val permette di impostare il valore effettivo del filtro.

- **type**
  I valori che può assumere il parametro `type` sono:

Valore	Significato
CAN_CONFIG_STD_MSG	Specifica che la maschera è per un messaggio in formato Standard.
CAN_CONFIG_XTD_MSG	Specifica che la maschera è per un messaggio in formato Esteso.

Esempio

```
//imposta modalità configurazione
CAN_operation_mode(CAN_OP_MODE_CONFIG);

CAN_set_mask (CAN_MASK_B1, 0xFFFFFFF0, CAN_CONFIG_XTD_MSG);
CAN_set_mask (CAN_MASK_B2, 0xFFFFFF00, CAN_CONFIG_XTD_MSG);

//imposta modalità normale
CAN_operation_mode (CAN_OP_MODE_NORMAL);
```

[8] Questo significa che l'identificatore viene riconosciuto come valido indipendentemente dal fatto che il valore del bit sia 1 o 0.

CAN_set_filter

Per mezzo di questa funzione è possibile impostare i filtri associati ai buffer di ricezione. Si ricorda che sono presenti due buffer di ricezione. Il primo buffer possiede due filtri mentre il secondo buffer ne possiede 4. Per poter impostare i filtri è necessario impostare la modalità di configurazione. Se i filtri vengono attivati[9], i messaggi vengono accettati solo se l'identificatore del messaggio ricevuto è uguale ad uno dei filtri.

Prototype

```
void CAN_set_filter(enum CAN_FILTER filter_number, unsigned long val,
                                enum CAN_CONFIG_FLAGS type);
```

Parametri

- **filter_number**
  I valori che può assumere il parametro `filter_number` sono:

Valore	Significato
CAN_FILTER_B1_F1	Permette di selezionare il primo filtro del primo buffer.
CAN_FILTER_B1_F2	Permette di selezionare il secondo filtro del primo buffer.
CAN_FILTER_B2_F1	Permette di selezionare il primo filtro del secondo buffer.
CAN_FILTER_B2_F2	Permette di selezionare il secondo filtro del secondo buffer.
CAN_FILTER_B2_F3	Permette di selezionare il terzo filtro del secondo buffer.
CAN_FILTER_B2_F4	Permette di selezionare il quarto filtro del secondo buffer.

- **val**
  Il parametro `val` permette di impostare il valore effettivo del filtro.

- **type**
  I valori che può assumere il parametro `type` sono:

Valore	Significato
CAN_CONFIG_STD_MSG	Specifica che il filtro è per un messaggio in formato Standard.
CAN_CONFIG_XTD_MSG	Specifica che il filtro è per un messaggio in formato Esteso.

Esempio

```
CAN_operation_mode(CAN_OP_MODE_CONFIG); //imposta modalità configurazione

CAN_set_filter (CAN_FILTER_B1_F1,0xFFFFFFF0,CAN_CONFIG_XTD_MSG);
CAN_set_filter (CAN_FILTER_B1_F2,0xFFFFFFA0,CAN_CONFIG_XTD_MSG);
CAN_set_filter (CAN_FILTER_B2_F1,0x000A0000,CAN_CONFIG_XTD_MSG);
CAN_set_filter (CAN_FILTER_B2_F2,0x000A0000,CAN_CONFIG_XTD_MSG);
CAN_set_filter (CAN_FILTER_B2_F3,0x000AC911,CAN_CONFIG_STD_MSG);
CAN_set_filter (CAN_FILTER_B2_F4,0x001AB000,CAN_CONFIG_XTD_MSG);

CAN_operation_mode (CAN_OP_MODE_NORMAL); //imposta modalità normale
```

[9] I filtri risultano disattivi se le maschere sono poste a 0x00000000, ovvero ignorando il valore dell'identificatore.

CAN_is_TX_ready

Per mezzo di questa funzione è possibile controllare se almeno uno dei tre buffer di trasmissione è disponibile per inviare un novo dato. La funzione restituisce 1 se almeno un buffer è disponibile. Prima di inviare un messaggio è bene controllare per mezzo di questa funzione se almeno un buffer è disponibile.

Prototype

```
BYTE CAN_is_TX_ready (void);
```

Esempio

```
if ( CAN_is_TX_ready  ()){

     // programma da eseguire nel caso sia disponibile
    // un buffer di trasmissione
}
```

CAN_is_RX_ready

Per mezzo di questa funzione è possibile controllare se uno dei due buffer in ricezione contiene un dato. La funzione restituisce 1 se uno o più messaggi sono stati ricevuti. La funzione ritorna utile per gestire in polling[10] la ricezione di dati.

Prototype

```
BYTE CAN_is_RX_ready (void);
```

Esempio

```
if (CAN_is_RX_ready ()) {

     // programma da eseguire nel caso sia stato ricevuto un dato

} else {

     // programma da eseguire nel caso non sia stato ricevuto un dato
}
```

[10] La ricezione di dati può essere controllata per mezzo delle interruzioni o interrogando continuamente il modulo CAN per vedere se questo ha ricevuto un dato; questa seconda modalità viene detta polling.

CAN_is_TX_warning_ON

Per mezzo di questa funzione è possibile controllare se il modulo CAN è affetto da un indice di errori in trasmissione superiore a 96. Questo valore determina uno stato di warning. La funzione restituisce 1 se il modulo CAN è in stato TX warning altrimenti 0.

Prototype

```
BYTE CAN_is_TX_warning_ON (void);
```

Esempio

```
if ( CAN_is_TX_warning_ON ()){

   // programma da eseguire nel caso di warning

}
```

CAN_is_RX_warning_ON

Per mezzo di questa funzione è possibile controllare se il modulo CAN è affetto da un indice di errori in ricezione superiore a 96. Questo valore determina uno stato di warning. La funzione restituisce 1 se il modulo CAN è in stato RX warning altrimenti 0.

Prototype

```
BYTE CAN_is_RX_warning_ON (void);
```

Esempio

```
if ( CAN_is_RX_warning_ON ()){

   // programma da eseguire nel caso di warning

}
```

CAN_is_TX_passive

Per mezzo di questa funzione è possibile controllare se il modulo CAN è affetto da un indice di errori in trasmissione superiore a 127. Questo valore determina l'attivazione dello stato Error Passive. La funzione restituisce 1 se il modulo CAN è in stato TX passive altrimenti 0.

Prototype

```
BYTE CAN_is_TX_passive (void);
```

Esempio

```
if ( CAN_is_TX_passive ()) {
  // programma da eseguire nel caso di error passive
}
```

CAN_is_RX_passive

Per mezzo di questa funzione è possibile controllare se il modulo CAN è affetto da un indice di errori in ricezione superiore a 127. Questo valore determina l'attivazione dello stato Error Passive. La funzione restituisce 1 se il modulo CAN è in stato RX passive altrimenti 0.

Prototype

```
BYTE CAN_is_RX_passive (void);
```

Esempio

```
if ( CAN_is_RX_passive ()) {
 // programma da eseguire nel caso di error passive
}
```

CAN_get_TX_error_count

Per mezzo di questa funzione è possibile sapere l'indice degli errori in trasmissione. Il contatore viene azzerato ogni volta che avviene l'inizializzazione del modulo CAN.
Il valore restituito è di tipo BYTE ovvero `unsigned char`. Il valore massimo che viene restituito è 255, superato il quale il modulo CAN viene disattivato (bus off)

Prototype

```
BYTE CAN_get_TX_error_count (void);
```

Esempio

```
unsigned char val = 0;

val = CAN_get_TX_error_count ();
```

CAN_get_RX_error_count

Per mezzo di questa funzione è possibile sapere l'indice degli errori in ricezione. Il contatore viene azzerato ogni volta che avviene l'inizializzazione del modulo CAN.
Il valore restituito è di tipo BYTE ovvero `unsigned char`. Il valore massimo che viene restituito è 255, superato il quale il modulo CAN viene disattivato (bus off)

Prototype

```
BYTE CAN_get_RX_error_count (void);
```

Esempio

```
unsigned char val = 0;

val = CAN_get_RX_error_count ();
```

CAN_is_BUS_OFF

Per mezzo di questa funzione è possibile controllare se il modulo CAN è in stato bus off dovuto ad un numero di errori in ricezione o trasmissione maggiore di 255. La funzione restituisce 1 se il modulo CAN è in stato bus off.

Prototype

```
BYTE CAN_is_BUS_OFF (void);
```

Esempio

```
if ( CAN_is_BUS_OFF ()){

   // programma da eseguire nel caso di bus off

}
```

CAN_abort_messages

Per mezzo di questa funzione vengono abortiti, ovvero annullati, tutti i messaggi non ancora trasmessi.

Prototype

```
void CAN_abort_messages (void);
```

Esempio

```
// tutti i messaggi non trasmessi vengono abortiti
CAN_abort_messages ();
```

Esempio d'utilizzo

Dopo mille parole è bene scrivere un esempio per il PIC18F4580, al fine di mettere un poco di chiarezza sul come utilizzare le varie funzioni. In particolare l'esempio riportato sotto permette di inviare dei dati sul Bus in base alla pressione di uno dei tasti posizionati su RB4, RB5, RB6, RB7. I dati inviati sono poi visualizzati sui LED posizionati sulla PORTD. Caricando lo stesso programma su due schede Freedom II è possibile creare una semplice rete CAN, senza dover discriminare tra master e slave. Ogni scheda visualizzerà i dati inviati dall'altra.

```
#include <xc.h>

#include "LTlib.h"

#include "module_IO.h"
#include "module_IO.c"

#include "module_CAN.h"
#include "module_CAN.c"

#define BUTTON_PRESSED 0x00

#define BUTTON1 PORTBbits.RB4
#define BUTTON2 PORTBbits.RB5
#define BUTTON3 PORTBbits.RB6
#define BUTTON4 PORTBbits.RB7

int main (void) {

    BYTE info [8];
    CANmessage msg;

    IO_set_all_ports_as_inputs();

    IO_set_port_direction(IO_PORTD, IO_ALL_PORT_OUTPUT);

    IO_enable_pull_up_resistors(IO_PORTB, IO_BIT4 + IO_BIT5 + IO_BIT6 +
IO_BIT7);

    //******************************************************
    // CAN Library Test
    //******************************************************
    //Initialize at 125Kbits/s 16xTq
    // 16MHz 125Kb/s --> 2,7,6,1,3
    // 20MHz 125Kb/s --> 2,7,6,1,4
    CAN_open (2,7,6,1,4, CAN_CONFIG_LINE_FILTER_OFF &
    CAN_CONFIG_SAMPLE_ONCE &
    CAN_CONFIG_ALL_VALID_MSG &
    CAN_CONFIG_DBL_BUFFER_ON);


    //Monitor all the buttons
    while (1){

        info[0] = 0x00;

        if (BUTTON1 == BUTTON_PRESSED){
           info[0] = 0x01;
        }
```

```
        if (BUTTON2 == BUTTON_PRESSED){
           info[0] = 0x02;
        }

        if (BUTTON3 == BUTTON_PRESSED){
           info[0] = 0x04;
        }

        if (BUTTON4 == BUTTON_PRESSED){
           info[0] = 0x08;
        }

        //Check if any button was pressed
        if (info[0] > 0) {

            while (!CAN_is_TX_ready());

            CAN_write_message (0x0A005510, info,1, CAN_TX_XTD_FRAME &
            CAN_NORMAL_TX_FRAME & CAN_TX_PRIORITY_2);
        }

        //Check if any data was received
        if (CAN_is_RX_ready ()) {
            CAN_read_message (&msg);

            IO_write_port (IO_PORTD, msg.data[0]);

        }
    }
}
```

Indice Alfabetico

Bibliografia

[1] www.LaurTec.it : sito dove poter scaricare la libreria CAN e gli altri articoli menzionati.
[2] www.can.bosch.com : sito dove poter scaricare tutta la documentazione originale della Bosch
[3] www.michrochip.com : sito dove poter scaricare la seguente documentazione:

[4] : AN713 "Controller Area Network (CAN) Basics
[5] : AN738 "PIC18C CAN Routine in C"
[6] : AN916 "Comparing CAN and ECAN modules"
[7] : DS39500A "PICmicro® 18C MCU Family Reference Manual"
[8] : Datasheet PIC18F4580

History

Data	Versione	Autore	Revisione	Descrizione Cambiamento
16/01/16	2.0	Mauro Laurenti	Mauro Laurenti	Aggiornato formato, libreria ed esempio.
01/01/06	1.0	Mauro Laurenti	Mauro Laurenti	Versione Originale.