

***LaurTec***

**Libreria CAN per PIC**

**Autore :** *Mauro Laurenti*

**email:** [info.laurtec@gmail.com](mailto:info.laurtec@gmail.com)

**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.

Tutti i diritti di questa opera sono riservati. Ogni riproduzione ed ogni altra forma di diffusione al pubblico dell'opera, o parte di essa, senza un'autorizzazione scritta dell'autore, rappresenta una violazione della legge che tutela il diritto d'autore, in particolare non ne è consentito un utilizzo per trarne profitto.

La mancata osservanza della legge 22 Aprile del 1941 n. 633 è perseguibile con la reclusione o sanzione pecuniaria, come descritto al Titolo III, Capo III, Sezione II.

A norma dell'art. 70 è comunque consentito, per scopi di critica o discussione, il riassunto e la citazione, accompagnati dalla menzione del titolo dell'opera e dal nome dell'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.

Tutti i marchi citati in quest'opera sono dei rispettivi proprietari.

## 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 il progetto del lettore ma al tempo stesso di stimolarlo alla scrittura di una propria libreria. Il protocollo CAN può essere compreso a pieno solo se si percorrono i vari passi...dalla lettura delle specifiche alla scrittura di una libreria. La sua conoscenza permetterà lo sviluppo di applicazioni professionali e porterà le sue soddisfazioni.

## Il modulo CAN dei PIC

Scopo di questo paragrafo è quello di riassumere le caratteristiche principali del modulo CAN interno ai PIC della famiglia 18Fxx8. Durante la spiegazione si riterrà noto quanto scritto nel Tutorial "Il protocollo CAN". 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 bytes. 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 disabilita 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 trasmetterà 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<sup>1</sup>.

### **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 C18<sup>2</sup> ed utilizzando per semplicità la modalità di funzionamento mode 0 (Legacy mode). Il codice sorgente è possibile trovarlo al sito [www.laurtec.com](http://www.laurtec.com). 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<sup>3</sup>. 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 riesposti:

- 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

Al fine di utilizzare la libreria è necessario includere il file CANlib.h nel progetto o all'interno di ogni file in cui si fa uso delle funzioni CAN. La dichiarazione e definizione delle funzione è fatta all'interno dello stesso file dunque non è necessario includere nessun file \*.c.

Il modulo CAN fa utilizzo delle porte RB2 e RB3. 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.

---

<sup>1</sup> 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 Microchip e può creare rompicapi!

<sup>2</sup> Per maggiori dettagli sul linguaggio di programmazione C18 si rimanda al Tutorial "C18 step by step".

<sup>3</sup> La Microchip mette a disposizione una libreria sia per il modulo CAN che ECAN.

Le funzioni disponibili nella libreria sono:

<b>Nome</b>	<b>Operazione</b>
CANOperationMode	Imposta la modalità operativa del modulo CAN.
CANInitialize	Inizializza il modulo CAN.
CANsendMessage	Invia un messaggio
CANreceiveMessage	Legge un messaggio dal buffer di ricezione
CANsetMask	Imposta le maschere di ricezione del modulo CAN
CANsetFilter	Imposta i filtri di ricezione del modulo CAN
CANisTxReady	Controlla se il modulo CAN può trasmettere un nuovo messaggio
CANisRxReady	Controlla se è stato ricevuto un nuovo messaggio
CANisTXwarningON	Controlla se il modulo CAN ha TX warning attivo
CANisRXwarningON	Controlla se il modulo CAN ha RX warning attivo
CANisTXpassive	Controlla se il modulo CAN è TX passive
CANisRXpassive	Controlla se il modulo CAN è RX passive
CANgetTXerrorCount	Legge l'indice d'errore in trasmissione
CANgetRXerrorCount	Legge l'indice d'errore in ricezione
CANisBusOFF	Controlla se il modulo CAN è bus off
CANAbortMessages	Cancella i messaggi nel buffer di trasmissione non ancora inviati

## CANOperationMode

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 onorato 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 CANOperationMode (enum CAN_OP_MODE mode);
```

### Parametri

- `mode`

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

Valore	Significato
<code>CAN_OP_MODE_NORMAL</code>	Questa è la modalità che viene normalmente utilizzata in normali trasmissioni e ricezioni dati.
<code>CAN_OP_MODE_SLEEP</code>	Questa modalità mette in stato di sleep il modulo CAN, il quale non partecipa più alla comunicazione sul bus.
<code>CAN_OP_MODE_LOOP</code>	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.
<code>CAN_OP_MODE_LISTEN</code>	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.
<code>CAN_OP_MODE_CONFIG</code>	Per mezzo di questa modalità è possibile cambiare i filtri e le maschere del modulo CAN.

### Esempio

```
CANOperationMode (CAN_OP_MODE_CONFIG); // abilito la modalità configurazione
```

## CANInitialize

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 `CANOperationMode (CAN_OP_MODE_CONFIG);` viene automaticamente chiamata, dunque non è necessario impostare tale configurazione manualmente<sup>4</sup>.

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 bisognerà richiamare la funzione `CANOperationMode ()` con l'opportuno parametro. Durante l'inizializzazione la funzione `CANInitialize ()` resetta le maschere ed i filtri in modo da ricevere ogni messaggio. Se la funzionalità delle maschere e filtri dovesse essere richiesta sarà 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` (si veda specifiche CAN) sfruttato per compensare eventuali ritardi di propagazione sulla linea di trasmissione e dei bus driver.

- **phaseSeg1**

Il parametro **phaseSeg1** può assumere valori compresi tra 1 e 8. Viene utilizzato per impostare il valore `PHASE_SEG1` (si veda 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` (si veda 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` (si veda specifiche CAN) che è sfruttato dai vari nodi per effettuare la sincronizzazione della fase del clock interno. Questo avviene sommando o sottraendo quanti temporalidai parametri **phaseSeg1** e **phaseSeg2** <sup>5</sup>.

- **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

<sup>4</sup> Finora si era detto che ma modalità di configurazione era 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.

<sup>5</sup> Per maggiori informazioni riguardo il parametro `SJW`, si veda il paragrafo del datasheet del microcontrollore utilizzato, riguardante la risincronizzazione.

valore può essere compreso tra 0 e 63.

la divisione della frequenza principale avviene come segue.

$$111111 = TQ = (2 \times 64)/FOSC$$

$$111110 = TQ = (2 \times 63)/FOSC$$

:

$$000001 = TQ = (2 \times 2)/FOSC$$

$$000000 = TQ = (2 \times 1)/FOSC$$

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 avvenire, come riportato nell'esempio sotto riportato, per mezzo dell'operatore binario and &.

Valore	Significato
CAN_CONFIG_PHASE2_PRG_ON	Specifica di utilizzare il valore per il phase segment 2 impostato dall'utente.
CAN_CONFIG_PHASE2_PRG_OFF	Disabilita la funzione precedente ed ignora il valore impostato dal parametro <b>phaseSeg2</b> .
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 dei nodi è necessario che ognuno di essi sia capace di rigenerare il segnale 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. Tale frequenza viene determinata come:

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

ovvero

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

quindi il  $T_{BIT}$  minimo sarà 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:

$$T_{BIT} = T_Q \cdot (\text{Sync}_{SEG} + \text{Prop}_{SEG} + \text{Phase}_{SEG1} + \text{Phase}_{SEG2})$$

il valore  $T_Q$  è un valore fisso che viene a dipendere dalla frequenza di clock principale<sup>6</sup>, espressa in MHz, e il valore di prescaler **BRP** utilizzato.

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

Secondo le specifiche del PIC  $T_{BIT}$  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:

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

quindi:

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

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

<sup>6</sup> Qualora si faccia uso del PLL interno per moltiplicare la frequenza di clock, come frequenza bisogna considerare quella in uscita al PLL.

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:

```
CANInitialize (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.

## CANsendMessage

Per mezzo di questa funzione è possibile inviare un messaggio composto da un massimo di 8 bytes. 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 qual volta il modulo CAN deve inviare un messaggio presente in uno dei tre buffer di trasmissione, verrà inviato prima il messaggio a più alta priorità. Prima di inviare un messaggio è bene accertarsi, per mezzo della funzione `CANisTXready ( )`, se è possibile trasmetterlo.

### Prototype

```
void CANsendMessage (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 verranno effettivamente inviati.

- **flags**

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

Valore	Significato
CAN_CONFIG_STD_MSG	Specifica che il messaggio è in formato Standard
CAN_CONFIG_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**

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

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

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

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

## CANreceiveMessage

Per mezzo di questa funzione è possibile prelevare un messaggio dal buffer di ricezione. Il valore di ritorno 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 `CANisRxReady ()` se sono stati ricevuti messaggi<sup>7</sup>.

## Prototype

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

## Parametri

- **\*msg**

Tale parametro rappresenta un puntatore ad una struttura `CANmessage`. Tale struttura è definita all'interno del file `CANlib.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 riempirà i vari campi in funzione del messaggio ricevuto. Si noti che per passare l'indirizzo della struttura è necessario far uso dell'operatore `&`.

## Esempio

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

while (1) // ciclo infinito
{
    if (CANisRxReady ()) // controllo che sia presente un messaggio
    {
        CANreceiveMessage (&msg); //leggo il messaggio
        if (msg.data[0] == 0x03)
        {
            // se data[1]=0x03 fai questo...
        }
    }
}
```

<sup>7</sup> Con questa affermazione si considera il fatto che il programmatore non sta facendo uso delle interruzioni.

## CANSetMask

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 permetteranno 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<sup>8</sup>.

### Prototype

```
void CANSetMask(enum CAN_MASK code, unsigned long val,
                enum CAN_CONFIG_FLAGS type);
```

### Parametri

- **code**

I valori che può assumere il parametro **code** 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

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

CANSetMask (CAN_MASK_B1, 0xFFFFFFFF, CAN_CONFIG_XTD_MSG);
CANSetMask (CAN_MASK_B2, 0xFFFFFFFF, CAN_CONFIG_XTD_MSG);

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

<sup>8</sup> Questo significa che l'identificatore verrà riconosciuto come valido indipendentemente dal fatto che il valore del bit sia 1 o 0.

## CANSetFilter

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<sup>9</sup>, i messaggi verranno accettati solo se l'identificatore del messaggio ricevuto è uguale ad uno dei filtri.

### Prototype

```
void CANSetFilter(enum CAN_FILTER code, unsigned long val,
                 enum CAN_CONFIG_FLAGS type);
```

### Parametri

- **code**

I valori che può assumere il parametro **code** 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

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

CANSetFilter (CAN_FILTER_B1_F1, 0xFFFFFFFF0, CAN_CONFIG_XTD_MSG);
CANSetFilter (CAN_FILTER_B1_F2, 0xFFFFFFFFA0, CAN_CONFIG_XTD_MSG);
CANSetFilter (CAN_FILTER_B2_F1, 0x000A0000, CAN_CONFIG_XTD_MSG);
CANSetFilter (CAN_FILTER_B2_F2, 0x000A0000, CAN_CONFIG_XTD_MSG);
```

<sup>9</sup> I filtri risultano disattivi se le maschere sono poste a 0x00000000, ovvero ignorando il valore dell'identificatore.

```
CANSetFilter (CAN_FILTER_B2_F3,0x000AC911,CAN_CONFIG_STD_MSG);  
CANSetFilter (CAN_FILTER_B2_F4,0x001AB000,CAN_CONFIG_XTD_MSG);  
  
CANOperationMode (CAN_OP_MODE_NORMAL); //imposta modalità normale
```

### CANisTxReady

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 ritorna 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 CANisTxReady (void);
```

#### Esempio

```
if ( CANisTxReady ())
{
    // programma da eseguire in caso sia disponibile un buffer di trasmissione
}
```

### CANisRxReady

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

#### Prototype

```
BYTE CANisRxReady (void);
```

#### Esempio

```
if (CANisRxReady ())
{
    // programma da eseguire in caso sia stato ricevuto un dato
}
else
{
    // programma da eseguire in caso non sia stato ricevuto un dato
}
```

---

<sup>10</sup> 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.

### **CANisTXwarningON**

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

#### **Prototype**

```
BYTE CANisTXwarningON (void);
```

#### **Esempio**

```
if (CANisTXwarningON ())  
{  
    // programma da eseguire in caso di warning  
}
```

### **CANisRXwarningON**

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

#### **Prototype**

```
BYTE CANisRXwarningON (void);
```

#### **Esempio**

```
if (CANisRXwarningON ())  
{  
    // programma da eseguire in caso di warning  
}
```

### CANisTXpassive

Per mezzo di questa funzione è possibile controllare se il modulo CAN è affetto da un tasso di errore in trasmissione superiore a 128. Questo valore determina l'attivazione dello stato Error Passive. La funzione ritorna 1 se il modulo CAN è in stato TX passive altrimenti 0.

#### Prototype

```
BYTE CANisTXpassive (void);
```

#### Esempio

```
if ( CANisTXpassive ())
{
    // programma da eseguire in caso di error passive
}
```

### CANisRXpassive

Per mezzo di questa funzione è possibile controllare se il modulo CAN è affetto da un tasso di errore in ricezione superiore a 128. Questo valore determina l'attivazione dello stato Error Passive. La funzione ritorna 1 se il modulo CAN è in stato RX passive altrimenti 0.

#### Prototype

```
BYTE CANisRXpassive (void);
```

#### Esempio

```
if ( CANisRXpassive ())
{
    // programma da eseguire in caso di error passive
}
```

### **CANgetTXerrorCount**

Per mezzo di questa funzione è possibile sapere l'indice d'errore in trasmissione. Il contatore viene azzerato ogni volta che avviene l'inizializzazione del modulo CAN.

Il valore di ritorno è BYTE ovvero unsigned char. Il valore massimo che viene ritornato è 255, superato il quale il modulo CAN viene disattivato (bus off)

#### **Prototype**

```
BYTE CANgetTXerrorCount (void);
```

#### **Esempio**

```
unsigned char val = 0;  
  
val = CANgetTXerrorCount ();
```

### **CANgetRXerrorCount**

Per mezzo di questa funzione è possibile sapere l'indice d'errore in ricezione. Il contatore viene azzerato ogni volta che avviene l'inizializzazione del modulo CAN.

Il valore di ritorno è BYTE ovvero unsigned char. Il valore massimo che viene ritornato è 255, superato il quale il modulo CAN viene disattivato (bus off)

#### **Prototype**

```
BYTE CANgetRXerrorCount (void);
```

#### **Esempio**

```
unsigned char val = 0;  
  
val = CANgetRXerrorCount ();
```

### **CANisBusOFF**

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

#### **Prototype**

```
BYTE CANisBusOFF (void);
```

#### **Esempio**

```
if ( CANisBusOFF ())  
{  
    // programma da eseguire in caso di bus off  
}
```

### **CANAbortMessages**

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

#### **Prototype**

```
void CANAbortMessages (void);
```

#### **Esempio**

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

**Esempi d'utilizzo**

```
// impostazioni dell'applicazione...

#include "CANlib.h"

void main (void)

{
    BYTE info [8];
    CANmessage msg;

    TRISB = 0b00001000; // inizializzazione RB3 è posto ad 1
    PORTB = 0x00 ;

    //le altre impostazioni delle porte dipendono dall'applicazione

    //*****
    // CAN Library Test
    //*****

    //Inizializzazione a 125Kbits/s    16xTq
    // 16MHz 125Kb/s --> 2,7,6,1,3
    // 20MHz 125Kb/s --> 2,7,6,1,4

    CANInitialize (2,7,6,1,3, CAN_CONFIG_LINE_FILTER_OFF &
                  CAN_CONFIG_SAMPLE_ONCE &
                  CAN_CONFIG_ALL_VALID_MSG &
                  CAN_CONFIG_DBL_BUFFER_ON);

    info[0] = 0x30;      //messaggio da trasmettere
    info[1] = 0x45;
    info[2] = 0x02;

    while (!CANisTxReady());
    CANsendMessage (0x0A005510, info,3, CAN_TX_XTD_FRAME &
                  CAN_NORMAL_TX_FRAME &
                  CAN_TX_PRIORITY_2);

    while (1) //polling per controllare i messaggi ricevuti
    {
        if (CANisRxReady ())
        {
            CANreceiveMessage (&msg);

            //analisi del messaggio...dipende dall'applicazione
        }
    }
}
```

**Bibliografia**

[www.LaurTec.com](http://www.LaurTec.com) : sito di elettronica dove poter scaricare gli altri articoli menzionati, aggiornamenti e progetti.

<http://www.can.bosch.com/index.html> : sito dove poter scaricare tutta la documentazione originale della Bosch

[www.microchip.com](http://www.microchip.com) : sito dove poter scaricare la seguente documentazione:

- [1] : AN713 “Controller Area Network (CAN) Basics
- [2] : AN738 “PIC18C CAN Routine in C”
- [3] : AN916 “Comparing CAN and ECAN modules”
- [4] : DS39500A “PICmicro<sup>®</sup> 18C MCU Family Reference Manual”
- [5] : Datasheet PIC18F4580