PREMESSA

A coronamento del “corso” sulla programmazione con linguaggio Assembler dei microprocessori serie ST6, non poteva mancare la raccolta in un unico cd-rom di tutti gli articoli pubblicati nel corso degli anni sull’argomento.

E quando diciamo tutti, intendiamo proprio tutti: dai due programmatori in kit, ai circuiti di prova, dalla spiegazione teorica delle istruzioni del linguaggio Assembler, alla loro applicazione pratica in elettronica, dagli accorgimenti per utilizzare al meglio le istruzioni e la memoria dei micro, all’uso del software emulatore per testare i programmi.

L’intento didattico accompagna tutti gli articoli, anche quelli che, a prima vista, sono di carattere più propriamente pratico: gli stessi programmi-sorgente, che trovate in questo stesso cd-rom in una directory dedicata, servono soprattutto per capire come si deve scrivere un’istruzione per ottenere una determinata funzione.
Leggendo i commenti accanto ad ogni riga di programma, non solo vi impadronirete della materia, ma potrete addirittura utilizzare blocchi di istruzioni trasferendoli nei vostri programmi.

Inoltre, con i software emulatori che vi proponiamo diventa facilissimo controllare le istruzioni via via che vengono eseguite. E’ così possibile capire dove e perché si genera l’errore e come fare per correggerlo.
Per questo motivo, ci siamo premurati di mettere a vostra disposizione, sempre in questo cd-rom, l’ultima versione del software emulatore SimST62, che avete imparato a conoscere, ad usare e ad apprezzare nei nostri articoli.

In appendice trovate il kit di una lampada ad ultravioletti per cancellare i microprocessori con memoria Eprom e un inedito sulla funzione Timer dei microprocessori ST6, che tiene conto del fatto che in alcuni tipi di micro è possibile attivare alcune modalità di funzionamento particolari e molto interessanti.

Non poteva mancare l’indice analitico dei kit e degli argomenti teorici, che vi rimanda immediatamente agli articoli in cui l’argomento scelto è trattato.

la Direzione Editoriale

Bologna, Gennaio 2003

Nota: poiché negli articoli si fa spesso riferimento agli argomenti trattati specificando la rivista in cui sono apparsi, nel sommario abbiamo riportato, oltre al titolo dell’articolo, anche il numero di rivista in cui è stato pubblicato, per facilitarne il ritrovamento all’interno del cd-rom.

SOMMARIO — RIVISTA

PROGRAMMATORE per microprocessori serie ST6 172
Programmatore LX.1170 per gli ST62/10-15-20-25
CIRCUITO TEST per microprocessore ST6E10 172
Scheda test LX.1171 per provare gli ST6
IMPARARE a programmare i MICROPROCESSORI ST6 174
Istruzioni – Variabili – Registri
IMPARARE a programmare i microprocessori ST6 175
Watchdog – Porte – Interrupt – A/D converter – Timer
BUS per TESTARE i micro ST6 179
Bus LX.1202-1203 per testare i micro ST62/10-15-20-25
SCHEDA TEST per ST6 179
Schede test LX.1204-1205 per provare gli ST6
NOTA per il programmatore LX.1170 per micro ST6 179
Consigli per migliorare il programmatore LX.1170
SCHEDA con 4 TRIAC per microprocessori ST6 180
Scheda LX.1206: pilotare 4 diodi triac con un ST6
SCHEDA con DISPLAY LCD pilotata con un ST6 181
Scheda LX.1207: pilotare un display numerico LCD con un ST6
UNA SCHEDA per pilotare un DISPLAY alfanumerico 182
Scheda LX.1208/N: pilotare un display alfanumerico LCD con un ST6
SOFTWARE emulatore per TESTARE i micro ST6 184
Software simulatore DSE622
SOFTWARE simulatore per TESTARE i micro ST6 185
Formato e Opcode delle istruzioni – Carry flag e Z flag – Correzione degli errori con il DSE622
Windows 95 e ST6 185
Se i programmi in DOS per ST6 non girano sotto Windows 95
PER PROGRAMMARE correttamente i micro ST6 189
Cicli macchina – Reset – Watchdog – Gestione ottimale delle porte – Espressioni
NUOVO software SIMULATORE per micro ST6 190
Nuovo simulatore software per micro ST62/10-15-20-25
SOFTWARE emulatore per TESTARE i micro ST6 190
Le direttive .w_on, .ifc, .block
LE DIRETTIVE dell'assembler ST6 191
Le direttive .ascii, .asciz, .def
PROGRAMMATORE per MICRO ST62/60-65 192
Programmatore LX.1325 per micro ST62/60-65
BUS per TESTARE le funzioni PWM e EEPROM 192
Bus LX.1329 per testare i micro ST62/60-65 – Programmi di esempio per PWM e EEPROM
LE DIRETTIVE dell'assembler ST6 193
Le direttive .byte, .equ, .set
OPZIONI del compilatore Assembler 194
Opzioni del compilatore Assembler
Le memorie RAM–EEPROM 195
Tipi di registri – Memoria RAM e RAM aggiuntiva – Lettura e scrittura della memoria EEPROM
Software SIMULATORE per micro ST6 197
Nuovo simulatore software per micro ST62/60-65
LA funzione SPI per lo scambio DATI 198
La funzione SPI per lo scambio seriale dei dati
CIRCUITI test per la SPI 198
Schede test LX.1380-1381-1382 per la funzione SPI
COME PROGRAMMARE i nuovi MICRO serie ST6/C 202
Interfaccia LX.1430 per gli ST6 serie C – Option Byte della serie C
COME UTILIZZARE la DIRETTIVA .MACRO 203
La direttiva .macro
Per PROGRAMMARE i nuovi MICRO serie ST6/C 204
Le funzioni attivabili tramite l'Option Byte della serie C
LA DIRETTIVA .IFC dell'ASSEMBLER per ST6 205
La direttiva .ifc
IL programma LINKER per i microprocessori ST6 206
Il programma Linker – I formati .hex e .obj – Le direttive .pp_on, .extern, .window, .windowend
APPENDICE A: QUALCOSA in più sul TIMER ....
APPENDICE B: Lampada per cancellare le Eprom 174
INDICE ANALITICO ....

Molti **Istituti Tecnici** e non pochi **softwaristi** e **progettisti** ci chiedono con sempre maggiore insistenza di spiegare in modo molto semplice come si programmano i microprocessori **ST62**, ritenendo che se ci prendiamo questo impegno lo adempiremo come è nostra consuetudine nel migliore dei modi.
Per accontentarvi iniziamo subito dicendo che i microprocessori della famiglia **ST62** sono reperibili in due diverse versioni:
quelli siglati **ST62/E** e quelli siglati **ST62/T**.
La lettera **E** posta dopo la sigla **ST62** indica che il microprocessore si può **cancellare** e **riprogrammare** per almeno un **centinaio** di volte.
I microprocessori **ST62/E** si riconoscono facilmente perché al centro del loro corpo è presente una piccola **finestra** (vedi fig.1) che permette di **cancellare** la **EPROM** interna tramite una lampada a raggi ultravioletti.
La lettera **T**, posta dopo la sigla **ST62**, indica che i dati **memorizzati** all'interno del microprocessore

PROGRAMMATORE per

non si **possono** più cancellare e quindi nemmeno riscrivere.
Gli **ST62/T** si riconoscono facilmente perché sono **sprovvisti** della **finestra** per la cancellazione (vedi fig.1).
Solitamente i microprocessori **ST62/E** vengono usati per le prime prove, perché in presenza di un eventuale **errore** nei programmi è sempre possibile **cancellare** e riscrive il software.

Solo quando si ha la conferma che il programma funziona regolarmente, si preferisce utilizzare i microprocessori della serie **ST62/T**, perché oltre ad essere meno costosi, non è più possibile manometterli.

Nelle **Tabelle N.1** e **N.2** riportiamo le principali caratteristiche di queste due serie di microprocessori.
Tenete presente che nei microprocessori da **2 K** di **memoria** è possibile inserire circa **900 - 990 righe** di programma ed in quelli da **4 K** circa **1.800 - 2.000 righe** di programma.

Per completare i dati riportati nelle due tabelle, precisiamo che il numero a due cifre riportato dopo la sigla, ad esempio **ST62E.10 - 15 - 20 - 25**, ha un preciso significato.

La prima cifra indica la memoria disponibile:
- se la prima cifra è un **1** (vedi **1**0-**1**5) sono disponibili **2 K** di **memoria**,
- se la prima cifra è un **2** (vedi **2**0-**2**5) sono disponibili **4 K** di **memoria**.

La seconda cifra indica i piedini disponibili per i segnali d'ingresso e d'uscita:
- se la seconda cifra è uno **0** (1**0**-2**0**) sono disponibili **12 piedini**,
- se la seconda cifra è un **5** (1**5**-2**5**) sono disponibili **20 piedini**.

TABELLA N.1 micro NON CANCELLABILI

Sigla Micro	memoria utile	Ram utile	zoccolo piedini	piedini utili per i segnali
ST62T.10	2 K	64 byte	20 pin	12
ST62T.15	2 K	64 byte	28 pin	20
ST62T.20	4 K	64 byte	20 pin	12
ST62T.25	4 K	64 byte	28 pin	20

TABELLA N.2 micro CANCELLABILI

Sigla Micro	memoria utile	Ram utile	zoccolo piedini	piedini utili per i segnali
ST62E.10	2 K	64 byte	20 pin	12
ST62E.15	2 K	64 byte	28 pin	20
ST62E.20	4 K	64 byte	20 pin	12
ST62E.25	4 K	64 byte	28 pin	20

microprocessori serie ST6

Si parla spesso dei vantaggi che offrono i microprocessori ST62 senza però spiegare quello che interessa maggiormente, cioè come fare per programmarli e quale programmatore utilizzare. Al contrario noi vi spiegheremo come costruirvi un valido programmatore ed anche come si deve procedere per programmare questi microprocessori.

Se osservate la zoccolatura di questi microprocessori (vedi fig.2-3), potete leggere a fianco di ogni piedino una sigla, e poiché non sempre viene precisato il loro esatto significato, sarà utile spiegarlo.

Vcc - Piedino di alimentazione **positiva**. Su questo piedino va applicata una tensione continua **stabilizzata** di **5 volt**.

TIMER - Applicando su questo piedino un **livello logico 1**, la frequenza del **quarzo** (vedi piedini **3-4**) divisa **x12** potrà giungere sullo **stadio contatore**. Da questo piedino è possibile prelevare un segnale ad onda quadra, la cui frequenza può essere stabilita con le istruzioni del programma.

OSC./In-Out - Sui piedini **3-4** viene applicato un **quarzo** necessario per avere la frequenza di **clock** che serve per far funzionare il microprocessore.

NMI - Questo piedino va sempre tenuto a **livello logico 1**. Applicando a questo piedino un impulso negativo, si informa la **CPU** di interrompere il programma che sta eseguendo e di passare automaticamente ad eseguire una seconda e diversa **subroutine** (sottoprogramma).

Vpp - Questo piedino serve per la programmazione. Durante la fase di programmazione questo piedino, che normalmente si trova a **5 volt**, riceve dal computer una tensione di **12,5 volt**. Quando il microprocessore già programmato viene inserito nella sua scheda di utilizzazione, si deve sempre tenere questo piedino a **livello logico 0**, per evitare di danneggiare i dati in memoria.

RESET - Questo piedino, che si trova sempre a **livello logico 1**, resetta il microprocessore ogni volta che viene cortocircuitato a **massa**. Quando si u-

tilizza un microprocessore già programmato, su tale piedino occorre sempre collegare una resistenza al **positivo** ed un condensatore verso **massa**, in modo da avere un **reset** automatico ogni volta che si alimenta il microprocessore.

PA - PB - PC - Sono le **porte** che la **CPU** può utilizzare singolarmente come **ingressi** oppure come **uscite** tramite programma. Se le utilizzate come **uscite**, per non danneggiarle è consigliabile non collegare dei circuiti che assorbano più di **5 milliAmpere.** Per pilotare dei circuiti che assorbono più di **5 mA**, è necessario interporre tra il microprocessore ed il carico esterno dei **transistor** oppure un **integrato** tipo **SN.74244** o **74HC244** o **74LS244**.

GND - Piedino di alimentazione da collegare a **massa.**

SCHEMA ELETTRICO del PROGRAMMATORE

L'intero circuito programmatore visibile in fig.5 è molto semplice perché richiede solo 3 transistor, due **NPN** ed un **PNP**, due integrati stabilizzatori di tensione **uA.78L05** (vedi IC2-IC3), un integrato digitale **C/Mos** tipo **SN.74HC14** contenente sei inverter a trigger di Schmitt (vedi IC1) ed infine uno zoccolo **textool** a 28 piedini.
Su questo zoccolo andrà infilato il microprocessore **ST62** che si vuole programmare.
Tutte le tensioni necessarie al microprocessore **ST62** vengono prelevate dal secondario del trasformatore **T1**.
I **15 volt** alternati, raddrizzati dal ponte **RS1**, forniscono una tensione **continua** di circa **20 - 21 volt** che raggiunge l'Emettitore del transistor **PNP** siglato **TR2**.
Come si vede nel disegno dello schema elettrico, la Base di questo transistor risulta collegata, tramite la resistenza **R3**, al Collettore del transistor **NPN** siglato **TR1**.
Quando questo transistor riceve dai piedini **2-1** del **Connettore** collegato al computer la necessaria tensione di polarizzazione, porta in conduzione il transistor **TR2** ed in questo modo la tensione positiva di **20 - 21 volt** può raggiungere gli ingressi dei due integrati stabilizzatori siglati **IC2 - IC3.**
L'integrato **IC2** provvede a fornire sulla sua uscita una tensione stabilizzata di **5 volt** per alimentare l'integrato **IC1** ed i piedini **1-5** dell'**ST62** a 28 piedini o il solo piedino **1** dell'**ST62** a 20 piedini.
L'integrato **IC3** provvede a fornire una tensione stabilizzata, sempre di **5 volt**, sul piedino **10** dell'**ST62** a 28 piedini o sul piedino **6** dell'**ST62** a 20 piedini.
Quando tramite computer si desidera **memorizzare** un programma all'interno dell'**ST62**, il piedino **3** del **Connettore**, che normalmente si trova a **livello logico 1**, si commuta sul **livello logico 0** e così la Base del transistor **NPN** siglato **TR3** toglie il **cortocircuito** sul diodo zener **DZ1** da **7,5 volt**.
In questo modo la tensione sull'uscita dell'integrato stabilizzatore **IC3** sale dai **5 volt** iniziali a **12,5 volt** (5 + 7,5 = 12,5).
Da questo istante i dati in **scrittura** giungono dal computer sui terminali **4-6-5-7** del **Connettore** e, prima di raggiungere il **microprocessore ST62**, vengono **squadrati** dai quattro **inverter** siglati **IC1/E - IC1/A - IC1/B - IC1/F**.
Le resistenze **R7 - R5 - R6 - R8**, che abbiamo posto in serie agli ingressi di questi **inverter**, servono per proteggerli nell'eventualità che il **CONN.1** venga per **errore** collegato sulla presa **Seriale** del computer anziché su quella **Parallela**.
Poiché non l'abbiamo ancora precisato, vi segnaliamo fin da ora che il **CONN.1** va inserito nella **PRESA PARALLELA** del computer (presa **LPT1**), alla quale è normalmente collegata la **stampante**.
A **memorizzazione** completata, il computer riporta a **livello logico 1** il piedino **3** del **CONN.1** polarizzando così la Base del transistor **TR3**, che portandosi in conduzione, **cortocircuita** a massa il diodo zener **DZ1**.
Quando il diodo **zener** risulta cortocircuitato, sull'uscita dell'integrato stabilizzatore **IC3** la tensione scende da **12,5** a soli **5 volt** ed in queste condi-

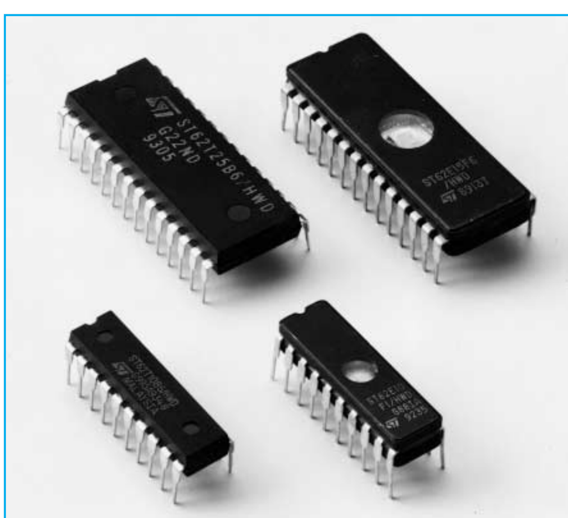


Fig.1 I microprocessori della serie ST62/T sprovvisti di finestra NON sono cancellabili, mentre i microprocessori della serie ST62/E disponendo di una piccola finestra SONO cancellabili. Il numero posto dopo la sigla T o E indica i Kilobyte di memoria e i piedini utili per i segnali di entrata e di uscita (vedi Tabelle 1-2).

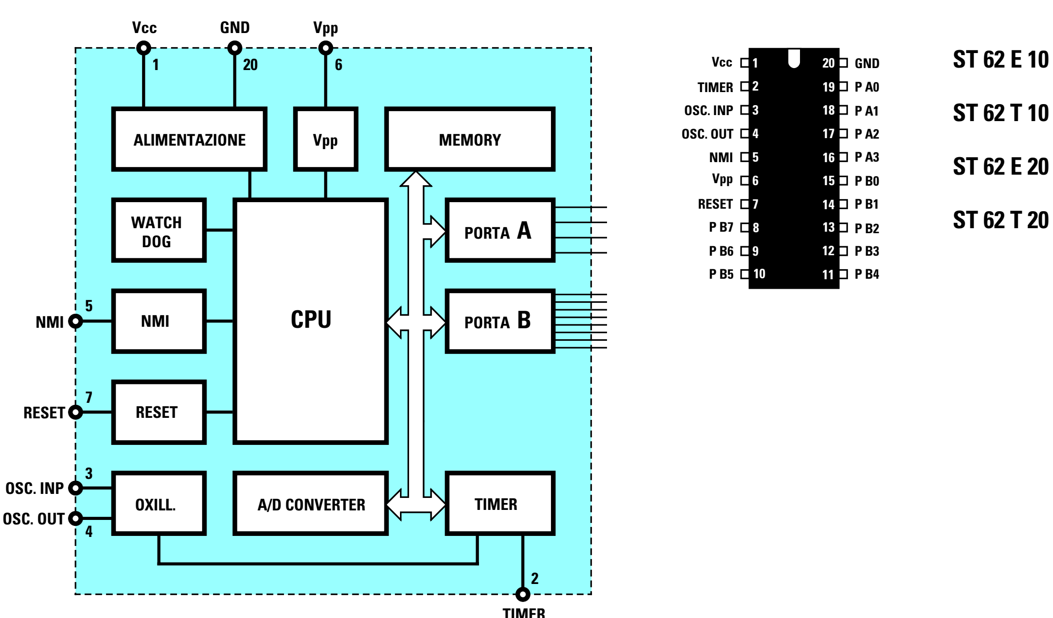


Fig.2 Tutti i microprocessori siglati ST62/E10 e ST62/T10 hanno 2K di memoria utile, mentre quelli siglati ST62/E20 e ST62/T20 hanno 4K di memoria utile. Questi microprocessori a 20 piedini dispongono di 12 porte di entrata o di uscita. La porta A dispone di 4 entrate/uscite (PA1-PA2 ecc.), mentre la porta B di 8 entrate/uscite (PB1-PB2 ecc.)

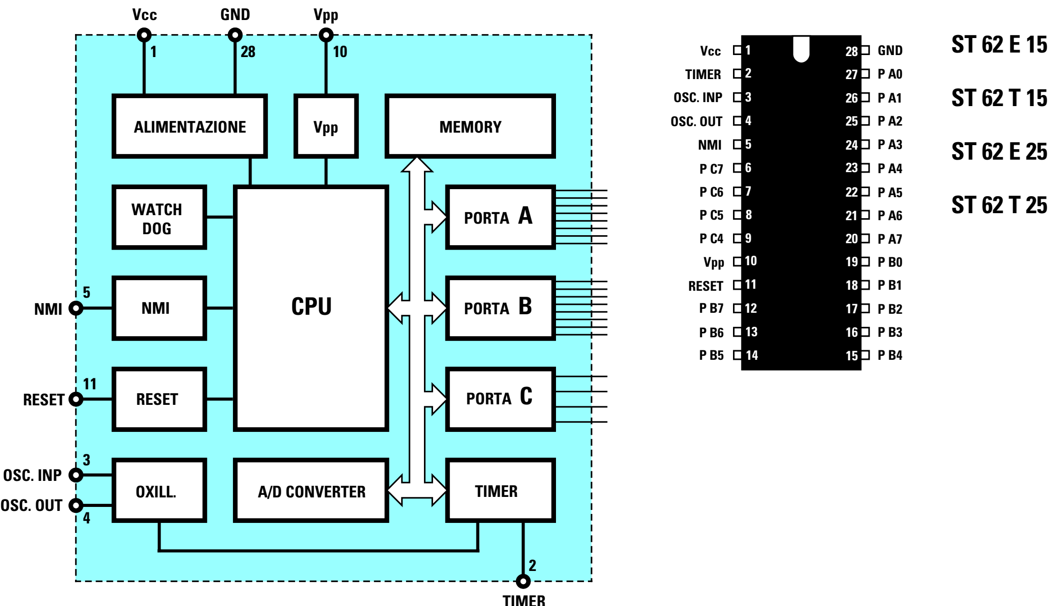


Fig.3 Tutti i microprocessori siglati ST62/E15 e ST62/T15 hanno 2K di memoria utile, mentre quelli siglati ST62/E25 e ST62/T25 hanno 4K di memoria utile. Questi microprocessori a 28 piedini dispongono di 28 porte di entrata o di uscita. Le porte A-B dispongono di 8 entrate/uscite (vedi PA1, PB1), mentre la porta C di 4 entrate/uscite (vedi PC1).

zioni nessun dato può più essere trascritto nella memoria del microprocessore.
I due inverter **IC1/C - IC1/D**, collegati in **parallelo** ed inseriti in senso inverso rispetto agli altri quattro inverter, vengono utilizzati dal computer per **leggere** i dati dall'**ST62**.
Grazie a questa **uscita** il computer può **rileggere** il programma caricato sul microprocessore e verificare che non vi siano **errori** nella trascrizione dei dati.
In presenza di un **errore** è possibile cancellare il microprocessore e ricopiare nella sua memoria i dati corretti, a patto che l'integrato inserito nel **textool** sia del tipo **ST62/E**.
Nello schema pratico visibile in fig.7 abbiamo raffigurato lo zoccolo **textool** per i microprocessori con **28** piedini e non per i microprocessori con **20 piedini**, ma come vi spiegheremo più avanti, lo stesso zoccolo viene utilizzato per entrambi i microprocessori.
A questo punto possiamo passare alla descrizione della realizzazione pratica e subito dopo vi spiegheremo come procedere per la **memorizzazione** dei **programmi-test** che troverete nel dischetto floppy fornito assieme al kit.
Sono inoltre in preparazione degli articoli teorico-pratici per insegnarvi a scrivere alcuni dei **programmi** che possono svolgere i microprocessori della serie **ST62**.
Vi chiediamo però di concederci un po' di tempo, perché oltre a testare i programmi, vogliamo ricercare tutte le possibili soluzioni per renderli comprensibili a tutti.

REALIZZAZIONE PRATICA

La realizzazione pratica è così semplice che in brevissimo tempo avrete già disponibile il vostro programmatore montato e funzionante.
Sul circuito stampato a fori metallizzati siglato **LX.1170**, dovete montare tutti i componenti richiesti disponendoli come visibile in fig.7.
Potete iniziare inserendo e stagnando i piedini degli zoccoli per l'integrato **IC1** e per il **textool**.
Quest'ultimo deve essere inserito nello stampato rivolgendo la leva di bloccaggio verso il basso, come appare chiaramente visibile nello schema pratico di fig.7.
Dopo questi due componenti potete inserire i due diodi: la fascia **bianca** presente sul corpo plastico del diodo siglato **DS1** va rivolta verso la resistenza R3, mentre la fascia **nera** presente sul corpo in vetro del diodo **zener** siglato **DZ1** va rivolta verso l'alto.
Proseguendo nel montaggio inserite tutte le **resistenze**, i **condensatori** poliestere e l'**elettrolitico**

ELENCO COMPONENTI LX.1170

R1 = 10.000 ohm 1/4 watt
R2 = 47.000 ohm 1/4 watt
R3 = 4.700 ohm 1/4 watt
R4 = 10.000 ohm 1/4 watt
R5 = 220 ohm 1/4 watt
R6 = 220 ohm 1/4 watt
R7 = 220 ohm 1/4 watt
R8 = 220 ohm 1/4 watt
*R9 = 1.500 ohm 1/4 watt
C1 = 22 mF elettr. 25 volt
C2 = 100.000 pF poliestere
C3 = 100.000 pF poliestere
C4 = 100.000 pF poliestere
C5 = 100.000 pF poliestere
C6 = 100.000 pF poliestere
*C7 = 1.000 mF elettr. 35 volt
DS1 = diodo EM.513 o 1N.4007
*RS1 = ponte raddriz. 100 V. 1 A.
DZ1 = zener 7,5 volt
*DL1 = diodo led
TR1 = NPN tipo BC.547
TR2 = PNP tipo BC.327
TR3 = NPN tipo BC.547
IC1 = C/Mos tipo 74HC14
IC2 = uA.78L05
IC3 = uA.78L05
*F1 = fusibile autoripr. 145 mA
*T1 = trasformatore 3 watt (TN00.01)
sec. 15 volt 0,2 Ampere
*S1 = interruttore
CONN.1 = connettore 25 poli

Nota = I componenti contraddistinti dall'asterisco andranno montati sul circuito stampato siglato LX.1170/B.

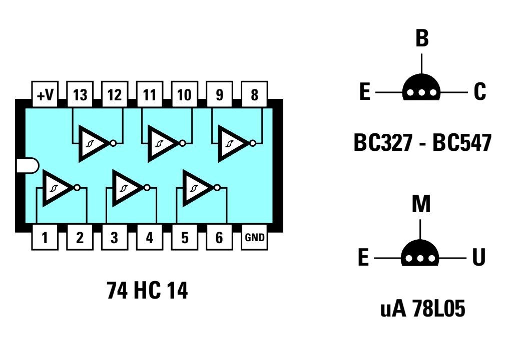


Fig.4 Connessioni dell'SN.74HC14 viste da sopra e dei transistor NPN e PNP e dell'integrato stabilizzatore uA.78L05 viste da sotto.

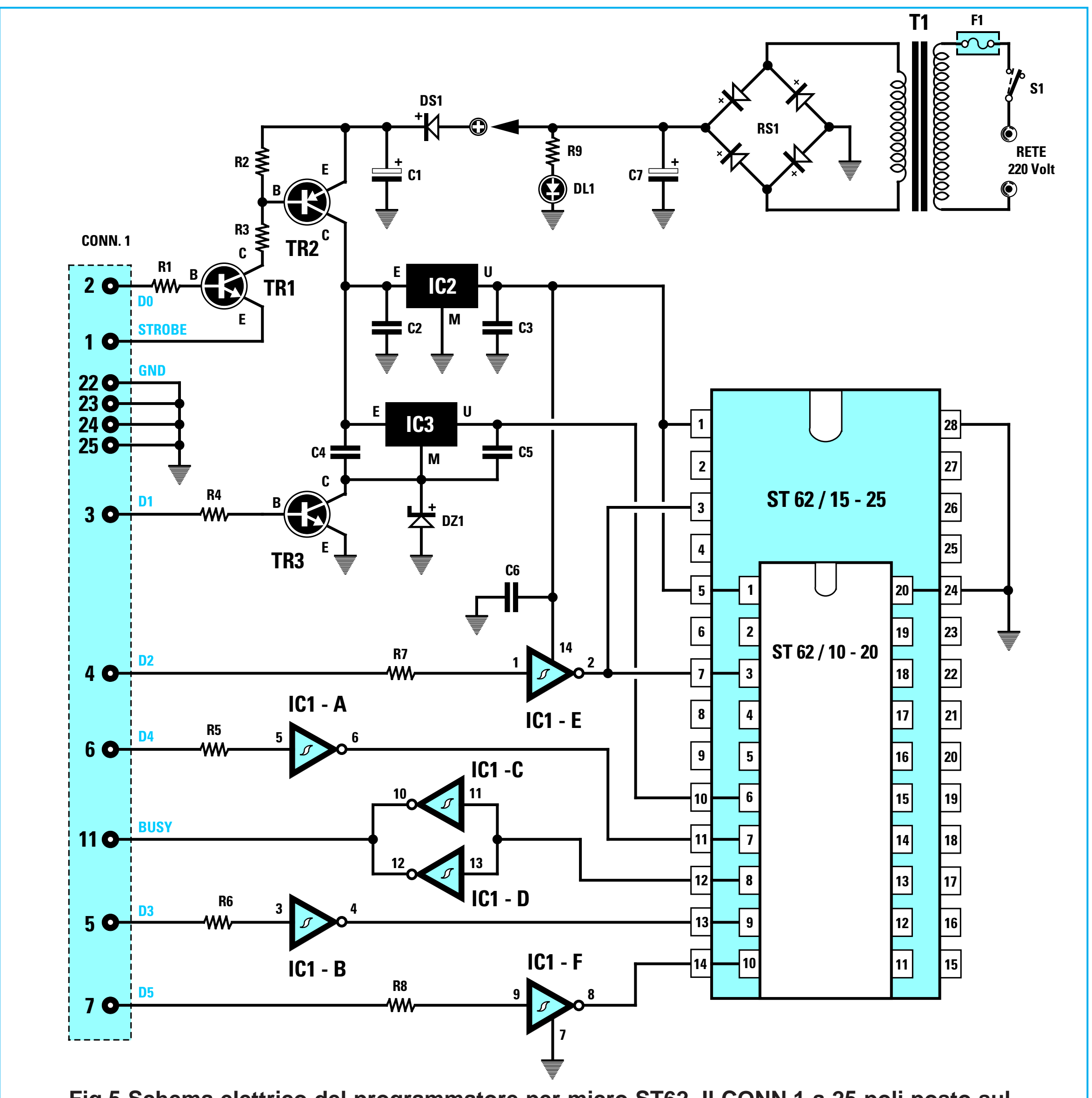


Fig.5 Schema elettrico del programmatore per micro ST62. Il CONN.1 a 25 poli posto sulla destra andrà collegato con un cavetto seriale alla porta PARALLELA del computer, cioè dove ora risulta collegata la STAMPANTE. Dopo aver sfilato il connettore della stampante, dovrete innestare il connettore proveniente da questo PROGRAMMATORE.

C1, che come visibile nello schema pratico di fig.7, deve essere collocato in posizione **orizzontale**.
A questo punto potete inserire i tre **transistor** ed i due integrati **stabilizzatori** e poiché questi ultimi hanno le stesse dimensioni dei transistor, dovete controllare attentamente la loro sigla prima di saldarli sullo stampato.
Come potete vedere nello schema pratico di fig.7, la parte **piatta** dei due **78L05** (IC2 - IC3) va rivolta verso destra e così dicasi per il transistor **BC.547** siglato **TR1**.Gli altri due transistor, siglati **TR2** (un **BC.327**) e **TR3** (un **BC.547**), vanno inseriti rivolgendo la parte **piatta** del loro corpo verso il basso e controllando con molta attenzione le loro sigle, in quanto uno è un **PNP** e l'altro un **NPN**.
Per completare il montaggio non vi resta che inserire sulla parte alta dello stampato il connettore **maschio** d'uscita ed infilare nel suo zoccolo l'integrato **74HC14**, rivolgendo la sua tacca di riferimento verso destra.

Lo stadio di alimentazione verrà montato sul circuito stampato siglato **LX.1170/B**, e poiché questo è un monofaccia, in fig.8 potete osservare le sue dimensioni a grandezza naturale.
Su questo stampato potete inserire come primo

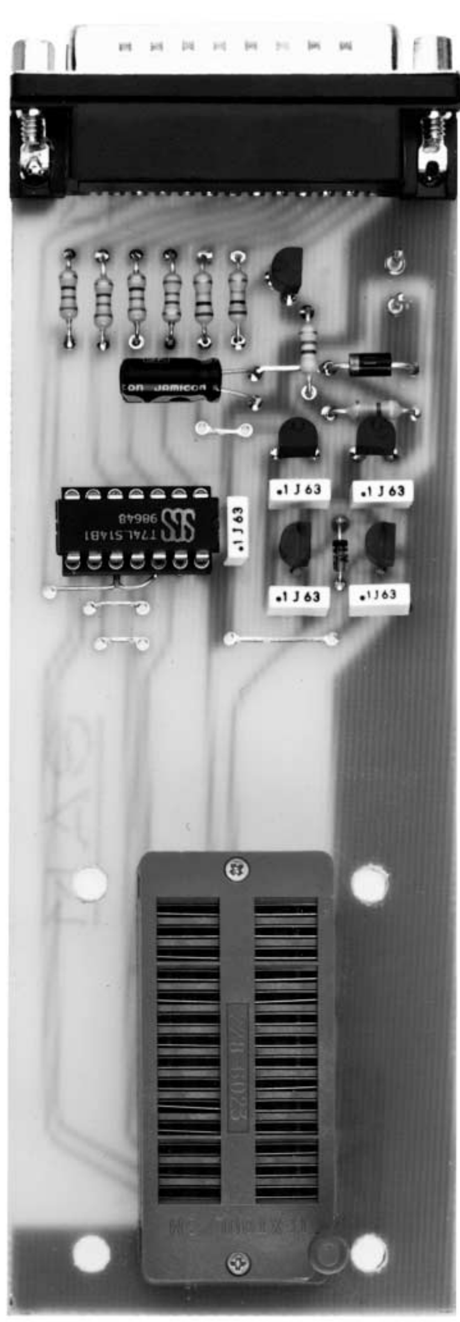


Fig.6 In questa foto potete vedere come si presenta questo programmatore dopo aver montato tutti i suoi componenti. Si noti sulla parte inferiore del circuito stampato lo zoccolo “textool”, che vi permetterà di inserire tutti i microprocessori da programmare senza sforzare i loro piedini.

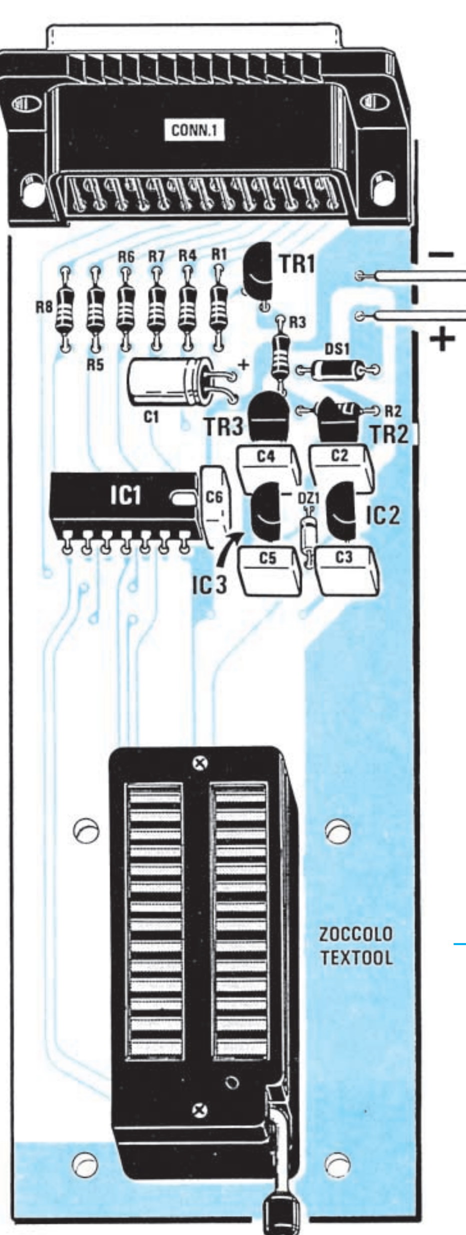


Fig.7 Schema pratico di montaggio dello stadio siglato LX.1170 e, a destra, del suo alimentatore siglato LX.1170/B. Facciamo presente che il CONN.1 può avere una forma diversa da come l’abbiamo disegnata. Se sul connettore fossero presenti due “torrette” (vedi foto), occorrerà toglierle.

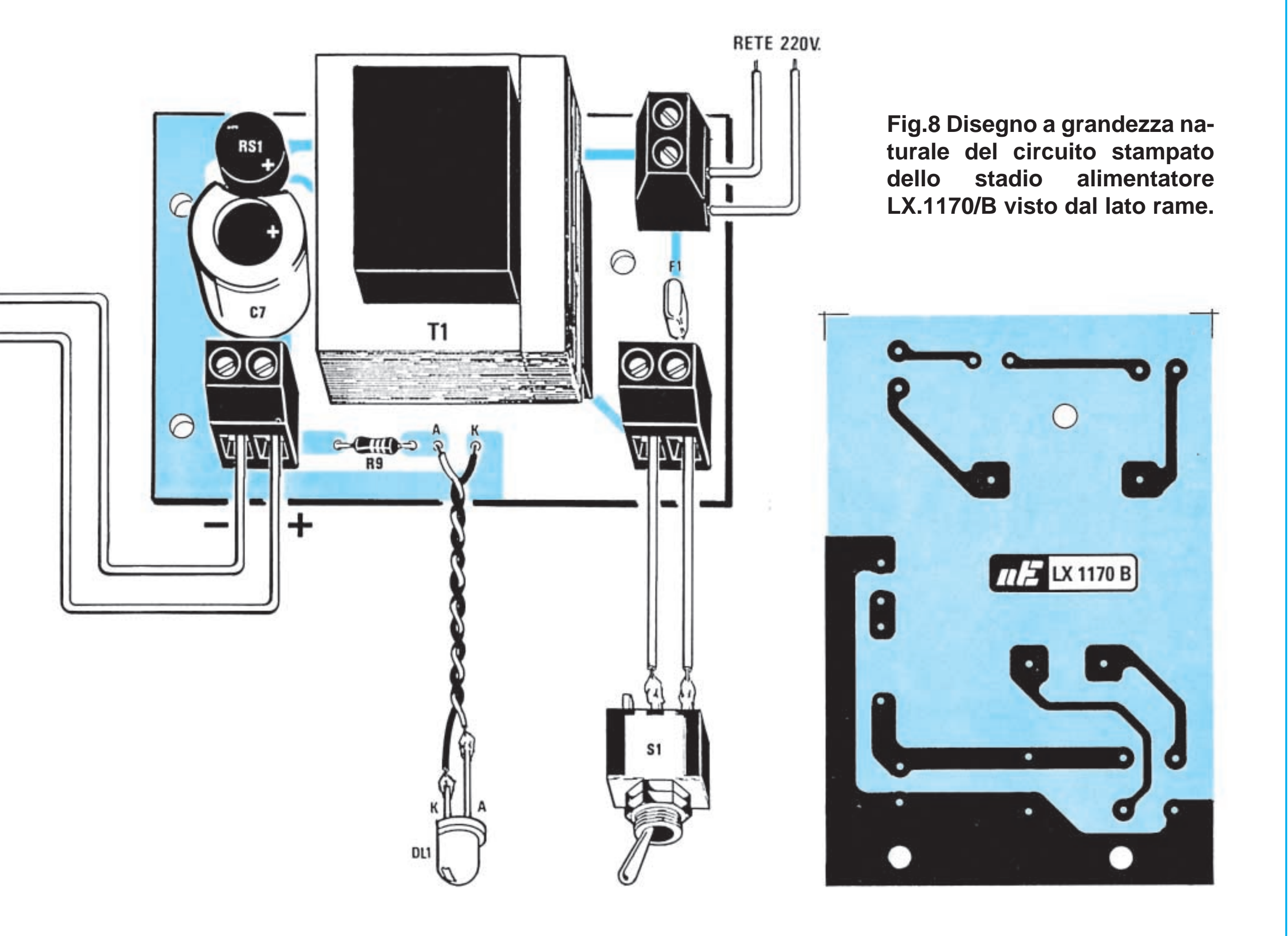


Fig.8 Disegno a grandezza naturale del circuito stampato dello stadio alimentatore LX.1170/B visto dal lato rame.

componente il trasformatore di alimentazione, i cui piedini risultano già predisposti per entrare solo nel loro giusto verso.
Quindi proseguite e completate il montaggio anche di questo stampato inserendo il **ponte raddrizzatore**, il condensatore elettrolitico **C7** rispettando la polarità dei due terminali, la resistenza **R9**, che serve ad alimentare il diodo **led**, ed il **fusibile** autoripristinante siglato **F1**.

MONTAGGIO NEL MOBILE

L'interfaccia verrà fissata dentro un piccolo mobile plastico tipo consolle (vedi fig.11).
Come prima operazione fissate sul mobile il suo **pannello frontale** utilizzando delle viti del diametro di **2 mm** o delle piccole viti autofilettanti.
Su tale pannello fissate con quattro viti lo stampato **LX.1170**, ma prima di eseguire questa operazione dovete stagnare sui due terminali di alimentazione uno spezzone di filo **rosso** per il positivo ed uno di filo **nero** per il **negativo**.
Sul piano del mobile fissate lo stampato dell'alimentatore utilizzando i distanziatori plastici con base **autoadesiva** che trovate nel kit.
Sul piccolo pannello della consolle va invece fissato il **portaled** e l'interruttore di rete **S1**.

A questo punto dovete effettuare i pochi collegamenti richiesti per portare la tensione di alimentazione all'interfaccia **LX.1170**, al diodo **led** ed all'interruttore di rete (vedi figg.7-8).

COME COLLEGARLO al COMPUTER

Dopo aver montato il programmatore siglato **LX.1170** dovete collegarlo alla **presa** della porta **parallela** del computer, cioè a quella che ora utilizzate per la **stampante**. Questa porta si distingue da quella **seriale** perché è **femmina**.
Per questo collegamento non potete usare il connettore che sfilerete dalla stampante, perché questo **non può** innestarsi nel connettore **maschio** presente sull'uscita del programmatore.
Per collegare il programmatore al computer potete usare un qualsiasi **cavo seriale** provvisto ad una

estremità di un connettore **maschio** che va innestato nel **computer**, e dall'altra di un connettore **femmina** che va innestato nel **programmatore**.

IL COMPUTER da USARE

Per programmare gli **ST62** bisogna disporre di un qualsiasi personal computer **IBM compatibile**, non importa se europeo o se costruito ad Hong-Kong o a Taiwan.
A tutti coloro che ci chiedono perché presentiamo programmi per soli IBM compatibili rispondiamo che la maggior parte dei programmi reperibili funzionano sotto **DOS** e poiché questo è il sistema operativo usato su tutti i computer **IBM compatibili** non è possibile adattare i programmi scritti per **DOS** per i computer tipo **APPLE - AMIGA - AMSTRAD** ecc.
Questa scelta non è nostra, ma delle Case di software che avendo constatato che i computer **IBM compatibili** sono i più diffusi in Europa - America - Asia, si sono orientate a realizzare solo programmi per **DOS**.
In questo modo le Case di software vendono un numero maggiore di programmi, quindi riducono i costi di **copyright** ed in più hanno la certezza che questi programmi funzioneranno su qualsiasi modello e marca di computer, perché usati sul **sistema operativo** più diffuso.
Ritornando al computer **IBM compatibile**, non importa di quale marca o tipo e neanche se il modello è vecchio o nuovo, deve soltanto essere dotato di

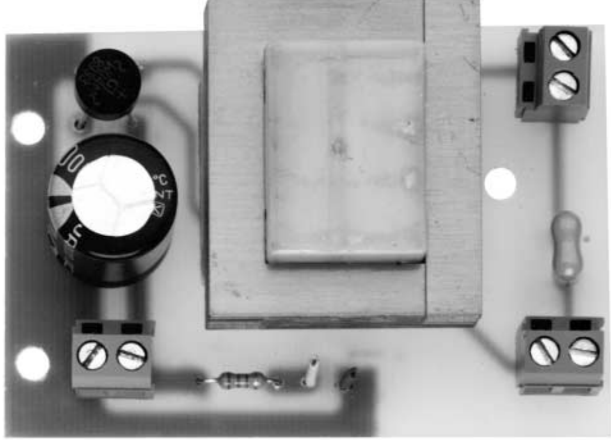


Fig.9 Foto dello stadio di alimentazione che una volta montato dovrete fissare sul coperchio del mobile con tre distanziatori plastici autoadesivi (vedi fig.10).

Fig.10 Lo stampato del programmatore siglato LX.1170 andrà fissato sul pannello del mobile con quattro viti più dado. Sul pannello inclinato dello stesso mobile fisserete il portaled e l'interruttore di accensione.

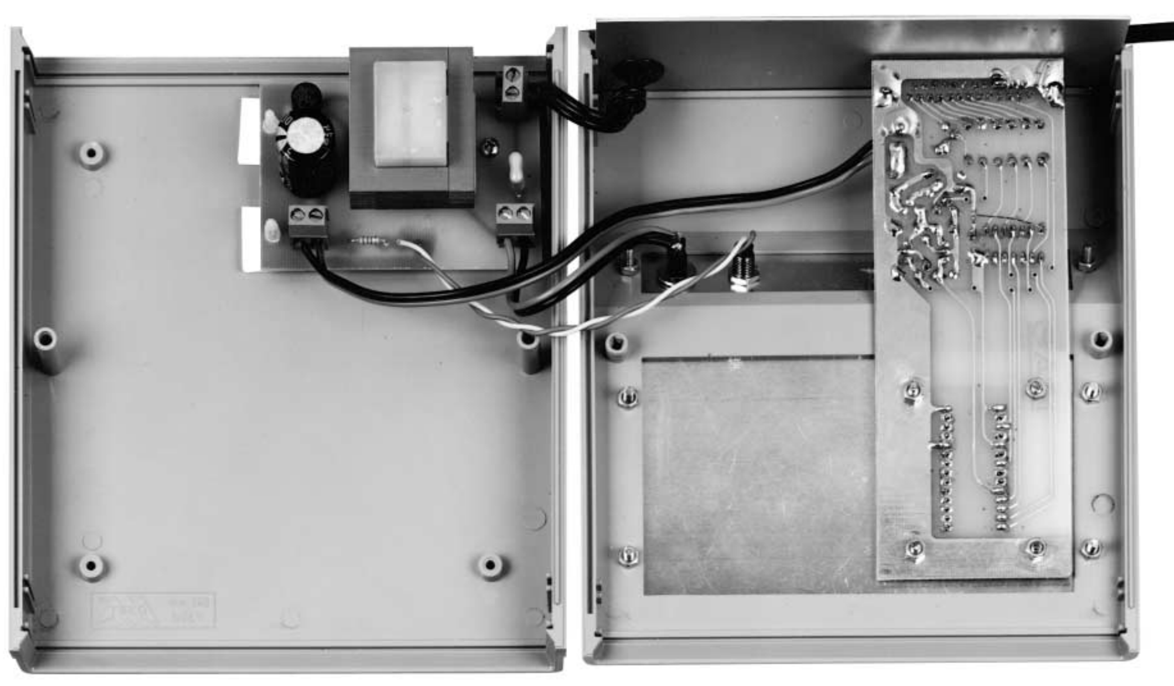


Fig.11 Il mobile scelto per questo programmatore completo delle sue mascherine già forate e serigrafate fornisce al progetto un aspetto molto professionale. Quello che più apprezzerete di questo progetto è la facilità con cui riuscirete, con il dischetto da noi fornito, a programmare qualsiasi tipo di microprocessore ST62.

una **scheda grafica** che rientri nel tipo **CGA - EGA - VGA - SuperVGA**.

INSTALLAZIONE del PROGRAMMA

Con il kit riceverete il **dischetto floppy** fornito dalla **SGS Thomson**, indispensabile per poter programmare tutti i microprocessori della serie **ST62**.
In questo dischetto abbiamo inserito dei **programmi** che vi permetteranno di semplificare tutte le operazioni necessarie per **scrivere** un programma, per **modificarlo** e poi **assemblarlo** ed ovviamente per trasferirlo all'interno della **memoria** di un microprocessore **ST62**.
Il programma vi indicherà inoltre se avete commesso degli **errori**, se avete inserito un **ST62** bruciato, se la memoria del microprocessore è **vergine** o già occupata da un altro programma.

Per iniziare a prendere confidenza con i microprocessori ed imparare a trasferire su questi un programma presente nell'**Hard-Disk**, abbiamo **aggiunto** nello stesso dischetto tre **semplici programmi**, che una volta trasferiti all'interno di un **ST62** vi permetteranno di verificare se avete eseguito correttamente tutte le operazioni di trasferimento dati.
Per copiare nell'Hard-Disk quanto è contenuto in questo **dischetto** dovete eseguire soltanto poche semplici istruzioni.
Quando, dopo aver acceso il computer, sul monitor appare la scritta **C:\>**, inserite il dischetto nell'unità floppy poi digitate:

C:\>A: poi Enter
A:\>installa poi Enter

Nota: Usate solo queste istruzioni e non altre, come ad esempio il **COPY** del **DOS** o le istruzioni dei

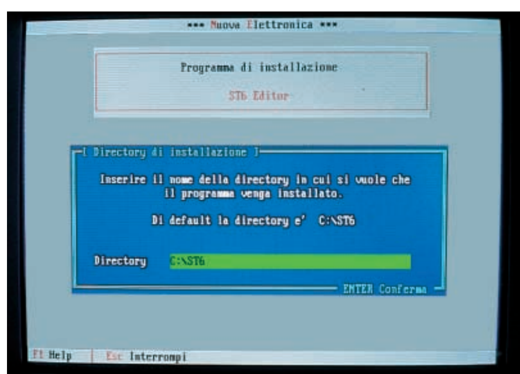


Fig.12 Per trasferire nell'Hard-Disk i programmi contenuti nel dischetto dovete digitare A:\>INSTALLA poi premere Enter. Tutti i programmi verranno memorizzati nella directory C:\ST6.

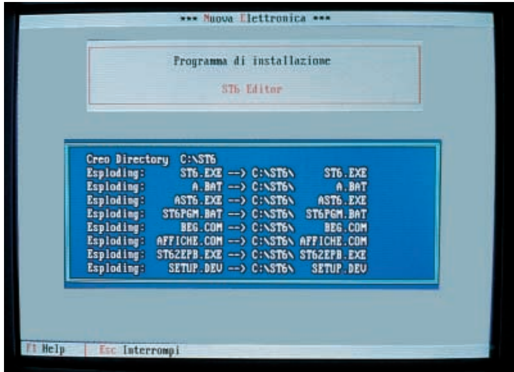


Fig.13 Poiché i programmi nel dischetto risultano compattati, durante l'operazione di scompattazione apparirà sul monitor l'intero elenco dei files. Il programma occupa 1 Mega circa di memoria.

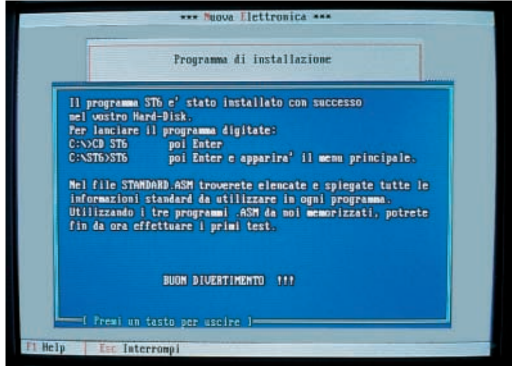


Fig.14 Scompattati tutti i programmi con successo, il computer ve lo segnalerà facendo apparire sul monitor questa scritta. Per uscire da questa finestra pigiate un tasto qualsiasi.

programmi tipo **PCSHELL - PCTOOLS - NORTON Commander**, perché il programma non funzionerebbe.

Con le due semplici istruzioni trascritte sopra, create **automaticamente** una **directory** chiamata **ST6**, nella quale vengono memorizzati tutti i files contenuti nel dischetto.
Durante l'operazione di scompattazione appare sul monitor l'elenco dei **files** (vedi fig.13).
Quando il programma è interamente memorizzato, appare un messaggio a conferma che l'installazione è stata **completata**.
Il programma **scompattato** occupa circa 1 Megabyte di memoria.
Se non premete nessun tasto, dopo qualche minuto compare la scritta:

C:\ST6>

Se volete uscire dalla **directory ST6** sarà sufficiente digitare:

**C:\ST6>CD ** poi Enter

e comparirà così sul monitor **C:\>**.

Una volta installato il programma nell'Hard-Disk potete mettere da parte il dischetto **floppy**, perché non vi servirà più.

COME si RICHIAMA il PROGRAMMA

Tutte le volte che volete richiamare il programma **ST6**, quando sul monitor appare **C:\>** dovete digitare queste semplici istruzioni:

C:\>CD ST6 poi Enter
C:\ST6>ST6 poi Enter

Se dovesse comparire una **directory** diversa da **C:**, ad esempio:

C:\JVFAX>

dovete digitare:

**C:\JVFAX>CD ** poi Enter
C:\>CD ST6 poi Enter
C:\ST6>ST6 poi Enter

Sul monitor comparirà così il **menu principale** (vedi fig.15).

Nota: Le scritte colorate in **azzurro** appaiono direttamente sul monitor, quelle senza colore dovrete digitarle dalla tastiera.

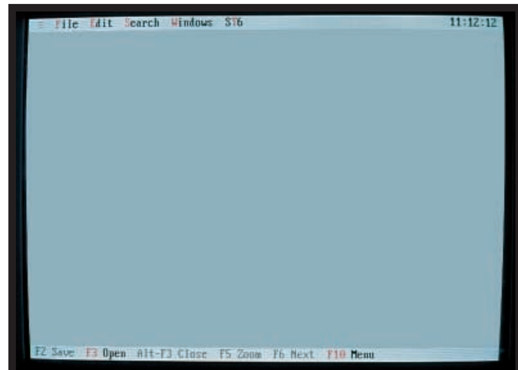


Fig.15 Richiamando il programma con C:\>ST6 Enter, C:\ST6>ST6 Enter, vedrete apparire sul monitor questo "menu". Se premete il tasto funzione F3 apparirà la finestra di fig.16.

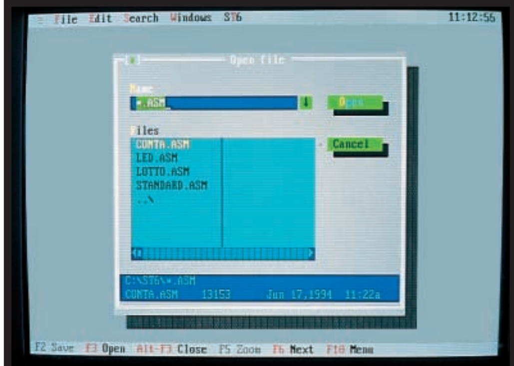


Fig.16 Premendo F3, appariranno in questa finestra i programmi "test" da noi inseriti, cioè Conta - Led - Lotto che potrete trasferire, come spiegato nell'articolo, su un microprocessore ST6 vergine.

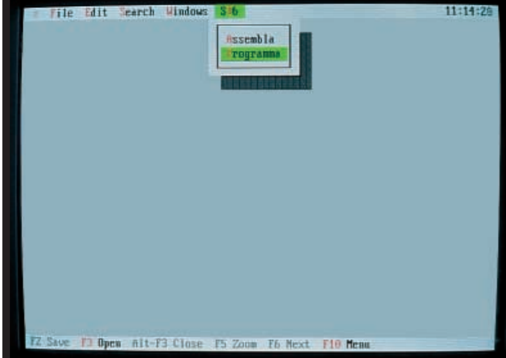


Fig.17 Se portate il cursore sulla scritta ST6 e premete Enter o pigiate i tasti Alt+T, apparirà questa finestra che vi permetterà di programmare l'ST6 inserito nello zoccolo textool del programmatore.

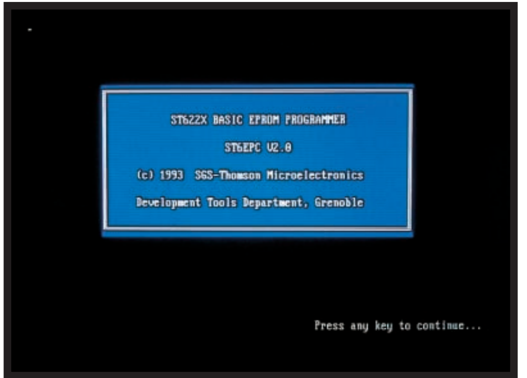


Fig.18 Premendo il tasto P = Programma dopo pochi secondi apparirà sul monitor del computer il software della SGS scritto in lingua inglese. Per continuare pigiate un qualsiasi tasto.

Fig.19 Sullo schermo apparirà una lista con tutti i tipi di ST6 che potete programmare e che sono circa 20. Per selezionare la sigla del vostro microprocessore usate i tasti freccia su e giù.

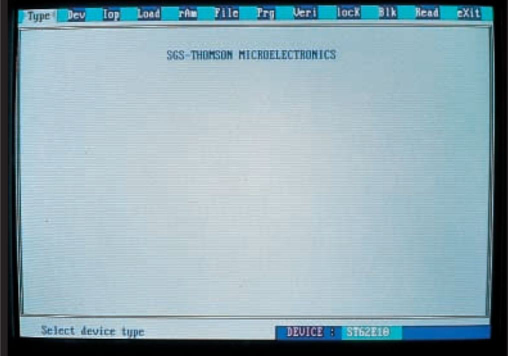


Fig.20 Poiché dovete programmare un ST62E10 portate il cursore su questa sigla poi pigiate Enter. Sullo schermo apparirà questa finestra con in basso l'indicazione dell'ST62E10.

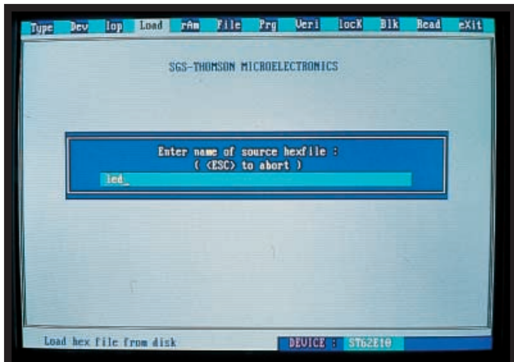


Fig.21 Dalla finestra di fig.20 premete il tasto L = Load e apparirà questa finestra. Qui dovete scrivere il nome del programma che volete trasferire dall'Hard-Disk al microprocessore ST62E10.

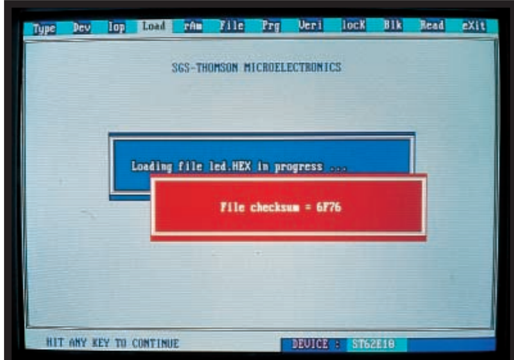


Fig.22 Dopo aver pigiato Enter apparirà la scritta "File checksum" per avvisarvi che il computer ha selezionato il programma, ma non l'ha ancora trasferito sul micro vergine. Per continuare premete un tasto.

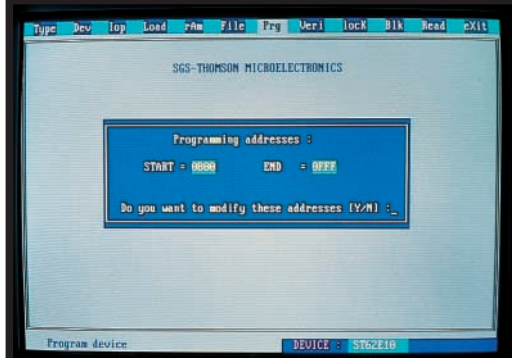


Fig.23 Pigiando un qualsiasi tasto apparirà la finestra di fig.20. Per programmare l'ST62E10 che avete inserito nello zoccolo textool del programmatore pigiate il tasto P = Prg e di seguito il tasto N.

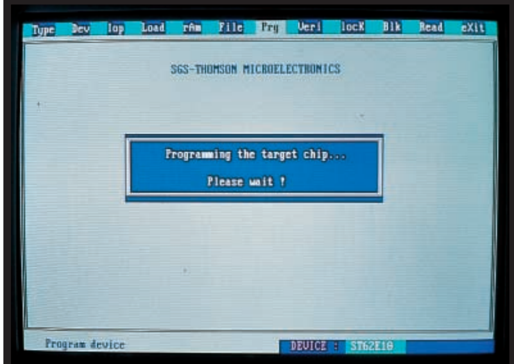


Fig.24 Quando compare questa scritta, non toccate più nessun tasto, perché il computer dopo aver verificato che l'ST62E10 è vergine, provvede a programmarlo impiegando circa 9-15 secondi.

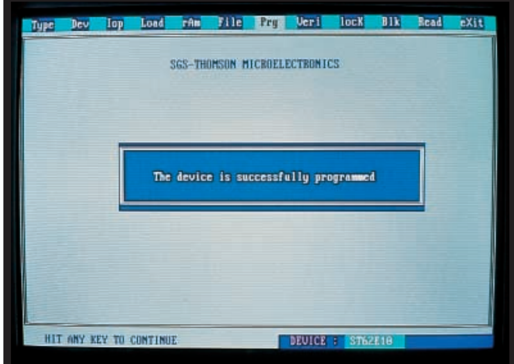


Fig.25 Completata la programmazione, sullo schermo apparirà questa scritta. A questo punto pigiate un qualsiasi tasto e così ritornerete al menu di fig.20. Per uscire basterà premere X.

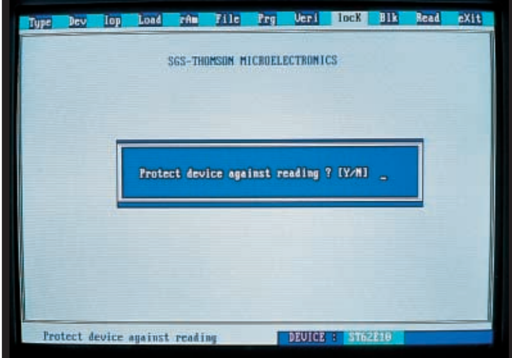


Fig.26 Quando sul monitor appare il menu di fig.20, se volete proteggere il micro dalla lettura dovete premere il tasto K = locK poi Y. L'ST62E10 anche se "protetto" si può cancellare.

A questo punto molti penseranno di aver già risolto tutti i loro problemi, ma poiché non è nostra abitudine illudere nessuno, vogliamo subito precisare che se non conoscete l'architettura di un **microprocessore** e non avete ancora una seppure minima conoscenza generale di come scrivere un programma, saranno necessari dai **3** ai **6 mesi** di pratica per poter diventare **autosufficienti**.
Per questo motivo abbiamo inserito nel dischetto **tre** semplici programmi che oltre a servirvi per effettuare le prime prove pratiche di trasferimento di **dati** verso le **memorie** del **microprocessore**, potranno esservi utili per capire come si imposta un **programma** per **ST62**. Vi spiegheremo infatti anche come richiamare e visualizzare tutte le istruzioni dei vari programmi.

CARICARE un PROGRAMMA

Per trasferire all'interno della **memoria vergine** di un microprocessore **ST62** uno dei tre programmi che noi abbiamo scritto, bisogna innanzitutto inserire il microprocessore nello zoccolo **textool** e **bloccarlo** spostando verso il basso la levetta.
Nel kit del programmatore troverete un **ST62E10** che ha una memoria **EPROM** utile di **2 Kbyte**.
Ovviamente potete caricare uno dei nostri programmi anche su un **ST62E25** da **4 Kbyte** di memoria **EPROM**, che però oltre ad essere più costoso, non può essere utilizzato sulla scheda sperimentale **LX.1171**, pubblicata su questa rivista, perché ha 28 piedini.
Poiché L'**ST62E10** ha soltanto **20 piedini**, dovete collocarlo nello zoccolo come visibile in fig.27, cioè in **basso** e rivolgendo la **tacca** di riferimento verso l'**alto**.
Eseguita questa operazione potete richiamare il programma (vedi paragrafo **Come si richiama il Programma**).
Quando sul monitor del vostro computer appare il menu di fig.15, per proseguire dovete conoscere il **nome** del file da trasferire e per questo dovete semplicemente premere:

F3

Sullo schermo apparirà una nuova finestra con l'elenco dei programmi presenti in memoria (vedi fig. 16). I programmi scritti da noi hanno questi nomi:

CONTA.ASM
LED.ASM
LOTTO.ASM

Nota: Oltre a questi tre files ne troverete un quarto chiamato **STANDARD.ASM**, che a differenza degli altri, **non contiene** un programma da caricare nel microprocessore. In questo file abbiamo voluto inserire tutte le istruzioni **standard** che occorre richiamare in **ogni** programma e che vi risulteranno utilissime nel prossimo articolo, dedicato alle istruzioni dei programmi per **ST62**.

Di questi files ne dovete scegliere **uno solo**, perché all'interno di un microprocessore potete inserire **un solo** programma alla volta.
Ammesso di aver scelto il primo, cioè **LED.ASM**, dovete ricordare il solo nome **LED** tralasciando l'estensione **.ASM**, che **non** vi serve durante la programmazione del microprocessore.
L'estensione **.ASM** è l'abbreviazione della parola **Assembler**.
A questo punto potete uscire da questa **finestra** premendo il tasto **Escape** e vedrete riapparire la pagina del **menu** principale (vedi fig.15).

Tenendo premuto il tasto **ALT** dovete premere il tasto **T = ST6** ed apparirà una finestra con in alto la scritta **Assembla - Programma** (vedi fig.17).

Premete ora il tasto **P = Programma**, e dopo alcuni secondi comparirà l'intestazione del software di programmazione della **SGS** in lingua inglese (vedi fig.18).

Per continuare dovete pigiare un **qualsiasi** tasto e così comparirà la finestra di fig.19.

Premendo i tasti **frecce** giù o su, potete **visualizzare** e **selezionare** tutti i tipi di **microprocessori ST62** che questa interfaccia è in grado di programmare.
Poiché dovete programmare un **ST62E10**, andate con il **cursore** sulla riga in cui appare questa scritta e pigiate **Enter**.

Sul monitor comparirà la pagina di fig.20 ed in basso a destra vedrete la sigla del tipo di microprocessore selezionato, che nel nostro caso è:
DEVICE: ST62E10.

Pigiate il tasto **L = Load** e nella maschera che appare scrivete il nome del file che volete **memorizzare** all'interno dell'**ST62E10**.

Poiché per questo esempio abbiamo scelto il file **LED**, scrivete questo nome nella riga (vedi fig.21) poi premete **Enter**.

Dopo pochi secondi comparirà una seconda finestra rossa (vedi fig.22) con scritto **File checksum =** un **numero esadecimale** di controllo.
Poiché questo numero non vi serve, pigiate un **qualsiasi** tasto.

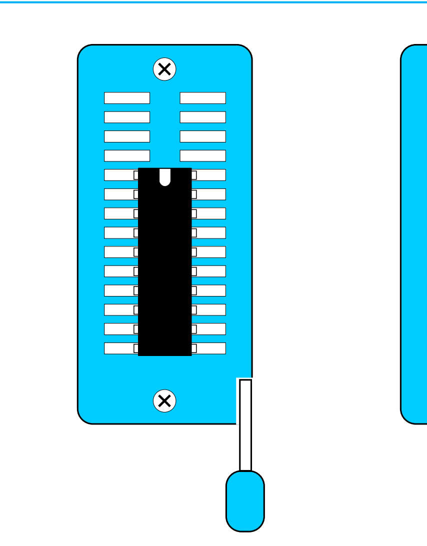
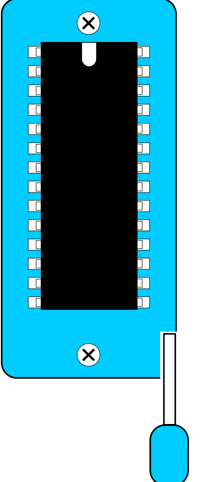



Fig.27 Tutti gli ST62 a 20 piedini andranno inseriti nello zoccolo textool in basso rivolgendo verso l'alto la tacca di riferimento. Anche gli ST62 a 28 piedini andranno inseriti nello zoccolo textool rivolgendo verso l'alto la tacca di riferimento.

Apparirà così la finestra **bianca** visibile in fig.20 e a questo punto dovete solo pigiare il tasto **P = Prg** e sul monitor vedrete la pagina visibile in fig.23.

Ora pigiate il tasto **N** in modo che il computer inizi a **controllare** il microprocessore inserito nello zoccolo **textool**.

Nota: **Non pigiate** mai il tasto **Y** e se per **sbaglio** lo premete, annullate questo comando pigiando il tasto **Escape**, quindi premete ancora il tasto **P** e di seguito **N.**

Dopo aver premuto **N** sul monitor apparirà questa scritta:

Verifying the target chip ... Please Wait
Verifica chip da programmare ... attendi

Se tutto risulta regolare, dopo pochi secondi apparirà sul monitor la finestra di fig.24 con la scritta:

Programming the target chip ... Please wait!
Programmazione in corso ... attendi!

L'operazione di scrittura dei dati dal computer verso le **memorie** del **microprocessore ST62** richiede circa **9 - 15 secondi**.

A programmazione completata sul monitor appare questa scritta (vedi fig.25):

The device is successfully programmed
Microprocessore programmato con successo

Poiché l'operazione di **caricamento dati** nell'**ST6** è completata, potete già estrarre l'**ST62** dallo zoccolo **textool** per inserirlo nel circuito siglato **LX.1171** (vedi articolo su questa rivista a pag.56).

Per uscire dal programma premete un tasto **qualsiasi** e di seguito il tasto **X.** Ritornerete così al menu principale di fig.15.

GLI ERRORI che possono COMPARIRE

Può succedere che per disattenzione premiate il tasto sbagliato o che il microprocessore che inserite nello zoccolo **textool** sia difettoso.
In questi casi sarà il programma a segnalarvi con alcuni messaggi in **inglese** l'anomalia o l'errore commesso cosicché possiate correggerlo.

Target Chip not presente or defective
L'integrato non c'è o è difettoso

Questo messaggio appare ogni volta che vi dimenticate di **inserire** il microprocessore nello zoccolo **textool** oppure quando il microprocessore che avete inserito è **bruciato**.
Non sempre però il microprocessore è fuori uso, perché questo identico messaggio appare anche quando:

- avete inserito il microprocessore nello zoccolo **textool** rivolgendo la **tacca** di riferimento verso il **basso** anziché verso l'**alto**, come visibile in fig.27.

- non avete innestato bene i **connettori** nel computer o nell'interfaccia **LX.1170**.

- vi siete dimenticati di accendere l'**interfaccia** del programmatore.

Device already programmed
Continue Programming? Y/N
L'integrato è già programmato
vuoi continuare? Si/No

Questo messaggio compare quando nello zoccolo **textool** avete inserito un microprocessore **ST62** che risulta **già programmato**.
In questo caso dovete premere il tasto **N** per ritornare così alla finestra di fig.20.
A questo punto potete togliere dallo zoccolo **textool** il microprocessore per **cancellarlo** (vedi paragrafo **Per cancellare un ST62/E**) e quindi riprogrammarlo oppure inserire nello zoccolo un **ST62** vergine e ripetere tutte le operazioni per la programmazione.Vi chiederete allora a cosa serve il comando **Y**, che conferma al programma di proseguire nella programmazione.
Se premete il tasto **Y** lasciando nello zoccolo **textool** l'**ST62** già programmato, non accadrà nulla, cioè il programma presente al suo interno **non si cancellerà** ed il nuovo **non** sarà mai **memorizzato** nella sua memoria.
Poiché nessuno ha mai chiarito quando è possibile usare il comando **Y**, cercheremo di spiegarvelo noi utilizzando degli esempi.
Se durante la fase di programmazione, quando all'interno della memoria del microprocessore è già stato trasferito un **50%** di dati, venisse improvvisamente a mancare la corrente di rete, voi vi trovereste con un microprocessore **programmato** per **metà** che risulterebbe inutilizzabile.
Una volta ritornata la corrente, il **computer** leggendo all'interno dell'**ST62** anche solo una parte di programma, lo considererà **già programmato**, ma se in questo caso premerete il tasto **Y**, il computer trasferirà nella memoria dell'**ST62** il restante **50%** di programma mancante.
Sempre **durante** la fase di programmazione, se si alzasse inavvertitamente la **levetta** dello zoccolo **textool**, i piedini dell'integrato non sarebbero più a contatto e quindi non entrerebbe più alcun dato nel microprocessore.
Poiché qualche dato può già essere entrato nell'**ST62**, ripetendo tutte le operazioni di trasferimento il computer si accorgerà che nelle memorie è già presente un programma e subito lo segnalerà.
Anche in questo caso premendo il tasto **Y**, il computer **completerà** l'inserimento dei dati che in precedenza non erano stati **memorizzati**.

Program result: Device fail at address xxx
Trovato un errore all'indirizzo xxx

Dove **xxx** è un numero esadecimale.
Questo messaggio appare ogniqualvolta il computer non riesce a trasferire correttamente i dati nella **memoria** del microprocessore.
Normalmente questo si verifica quando il microprocessore **ST62** è già stato riprogrammato più di un **centinaio** di volte.
Se questo messaggio compare spesso, è consigliabile sostituire il microprocessore.

Per CARICARE un altro PROGRAMMA

Se volete riutilizzare un microprocessore già programmato per trasferire nella sua memoria un **diverso** programma, dovete prima di tutto **cancellare** i dati al suo interno, dopodiché potete ripetere tutte le operazione già descritte.
Proseguendo nel nostro esempio, se dopo aver **memorizzato** il programma **LED** volete provare le funzioni del programma **CONTA** ed in seguito quelle del programma **LOTTO**, solo dopo aver **cancellato** il microprocessore potrete trasferire dal computer i dati contenuti in uno di questi **files**.

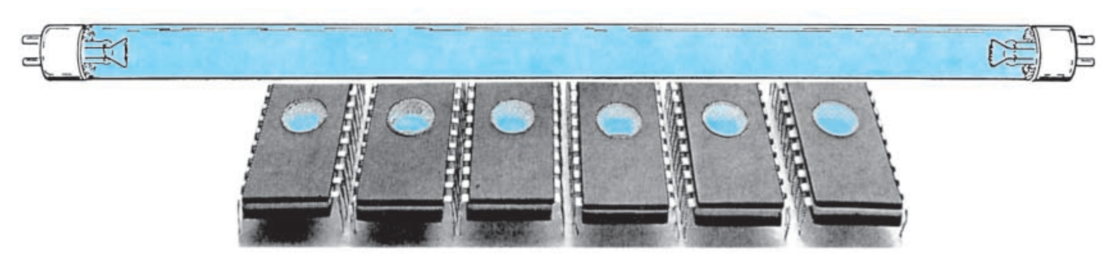


Fig.28 Per cancellare i microprocessori della serie ST62/E e tutti i tipi con EPROM, occorre esporre la loro finestra alla luce emessa da una lampada ultravioletta da 2.300-2.700 Angstrom. Poiché queste lampade non sono facilmente reperibili, abbiamo provveduto ad ordinarne un certo numero ed appena ci perverranno (è prevista una consegna entro settembre) vi presenteremo un completo progetto provvisto di temporizzatore.

Per PROTEGGERE un ST62

Dopo aver constatato che il microprocessore **programmato** funziona correttamente e siete certi che non volete più apportare modifiche al **programma**, ed inoltre non avete più alcuna necessità di rileggere i dati memorizzati al suo interno, vi conviene **proteggerlo**.
Un microprocessore **protetto** tipo **ST62/E** si può **cancellare** per renderlo idoneo a ricevere altri programmi.
Per proteggere un microprocessore, sia del tipo **ST62/T** che del tipo **ST62/E**, lo si deve lasciare inserito nello zoccolo **textool** e procedere come ora vi spiegheremo.

Quando sul monitor appare il menu principale (vedi fig.15), tornate nel menu di programmazione premendo **Alt+T** e di seguito **P** e apparirà la fig.19.
Selezionata la sigla del microprocessore che avete inserito nello zoccolo textool, quando appare il menu di fig.20 premete il tasto **K = Lock** e così apparirà sul monitor la finestra di fig.26.
Per **proteggerlo** sarà sufficiente premere il tasto **Y**, se **non** lo volete proteggere premete il tasto **N**.

Per CANCELLARE un ST62/E

Tutti i microprocessori della serie **ST62/E**, cioè quelli provvisti di una piccola finestra (vedi fig.1), una volta **programmati** si possono **cancellare** e poi nuovamente riprogrammare per utilizzarli con un diverso programma.
Per cancellare questi microprocessori occorre una **lampada ultravioletta** che lavori su una lunghezza d'onda compresa fra i **2.300** e i **2.700 Angstrom**.
Sotto questa lampada va collocato il microprocessore tenendo la sua finestra ad una distanza di circa **2 centimetri**.
A questa distanza per cancellare un microprocessore occorrono dai **15** ai **20 minuti**, sempre che la **finestra** risulti pulita.
Se sopra tale finestra c'è della sporcizia, ad esempio rimangono dei residui di collante dopo aver rimosso un'etichetta autoadesiva, dovrete prima pulirla con un batuffolo di cotone imbevuto di alcool o di acetone.
Poiché la lunghezza del bulbo di una lampada ultravioletta è di circa **30 cm**, potete cancellare contemporaneamente più **ST62/E** disponendoli uno di fianco all'altro (vedi fig.28).

NOTE per la LAMPADA UV

Se vi dimenticate il microprocessore sotto la lampada a **raggi ultravioletti** per una tempo superiore ai **50 minuti** non sono garantite più di **70 - 80 cancellazioni**.
Se volete usare un solo microprocessore per effettuare tantissime prove di **memorizzazione** e **cancellazione**, potete collegare la lampada ad uno dei tanti temporizzatori o timer per lampade da 220 volt pubblicati sulla nostra rivista (ad esempio il Kit **LX.1068** pubblicato sulla rivista **N.153**), che potrete regolare per una accensione massima di **10 minuti** circa.

A lampada **accesa** non fissate **ASSOLUTAMENTE** la luce **viola** che emette, perché **nuoce** gravemente agli occhi.
Per evitare questo inconveniente si potrà mettere sopra la lampada un panno o una scatola di cartone.

CONCLUSIONE

Su questo stesso numero troverete un semplice progetto che oltre a permettervi di controllare se il microprocessore programmato con uno dei tre programmi da noi inseriti nel dischetto, cioè **LED - CONTA - LOTTO**, funziona correttamente, vi consentirà di fare un po' di pratica sulla **cancellazione** di un **ST62/E** e sulla **riprogrammazione**.
In questo articolo vi insegneremo anche ad apportare delle **semplici** varianti sul programma, mentre nei prossimi articoli vi spiegheremo tutto il **set** di **istruzioni** per i microprocessori **ST62**, perché solo conoscendo il significato di queste istruzioni potrete un domani realizzare programmi personalizzati per far svolgere agli **ST62** tutte le funzioni a voi necessarie.

COSTO DI REALIZZAZIONE

Costo di realizzazione dello stadio LX.1170 (vedi figg.6-7) completo di circuito stampato, zoccolo Textool, connettore d'uscita, transistor, integrati con INSERITO un microprocessore ST62/E10, un dischetto floppy contenenti i programmi richiesti, ed il CAVO seriale completo di connettori, ESCLUSI il mobile e lo stadio di alimentazione .... € 49,10

Costo di realizzazione dello stadio di alimentazione LX.1170/B (vedi fig.8) completo di cordone di alimentazione .... € 11,60

Il mobile MO.1170 completo delle due mascherine forate e serigrafate .... € 16,01

Costo del solo stampato LX.1170 .... € 5,42
Costi del solo stampato LX.1170/B .... € 1,55

Vogliamo subito precisare che questo circuito serve per testare i programmi che avete imparato a trasferire nel microprocessore **ST62E10** fornito nel kit del programmatore.
Lo stesso circuito può essere utilizzato anche per i programmi che vorrete scrivere, a patto che configuriate le porte come le abbiamo configurate noi, diversamente non potrete sfruttarlo.

In questo circuito di prova, che potete vedere in fig.1, vi sono due integrati, ma quello che abbiamo siglato **IC1** è in pratica il **microprocessore ST62E10** che dovete prima programmare, mentre l'integrato **IC2**, che trovate inserito nel kit, è un **74LS244** utilizzato come **buffer** di corrente.
Infatti dovete tenere presente che sulle uscite dell'**ST62E10** non è possibile applicare dei carichi che assorbano più di **5 mA**, e poiché questo circuito viene utilizzato per accendere dei **diodi led** e dei **display** che assorbono una corrente maggiore, abbiamo dovuto adoperare l'integrato **74LS244**, che è in grado di sopportare carichi fino ad un massimo di **20 mA**.
Sempre guardando lo schema elettrico, sui piedini **3-4** dell'**ST62E10** trovate collegato un quarzo da **8 MHz**, che serve al microprocessore per generare la frequenza di **clock** necessaria per il suo funzionamento.
La frequenza di questo **quarzo** non è critica, quindi potrete utilizzare anche quarzi di frequenza inferiore, ad esempio **7 - 6 - 4 MHz**, tenendo comunque presente che più si scende di frequenza, **più lenta** risulta la velocità di esecuzione del programma.
Non utilizzate quarzi con una frequenza maggiore di **8 MHz**, perché il microprocessore non riuscirà a generare la necessaria frequenza di clock.
Dei tre pulsanti presenti nel circuito, quelli siglati **P1 - P2** svolgono le funzioni rese disponibili dal programma, mentre **P3** serve sempre e solo come comando di **reset**.

CIRCUITO TEST per

Dopo aver imparato come memorizzare un programma all'interno di un microprocessore ST62E10, e aver constatato di persona che non è poi così difficile come viene invece descritto in altre parti, sarete assaliti dalla curiosità di "testarlo" e per questo vi occorre soltanto il semplice circuito che ora vi presentiamo.

Nel **CONN.1** dovete inserire la **scheda** con i **diodi Led**, se avete memorizzato nell'**ST62E10** il programma **LED** o la **scheda** con i due **Display** se avete memorizzato nell'**ST62E10** il programma **CONTA** o **LOTTO**.
Per alimentare questa scheda occorre una tensione **stabilizzata** di **5 volt 200 milliAmpere** circa.

REALIZZAZIONE PRATICA

Sul circuito stampato siglato **LX.1171** dovete montare tutti i componenti disponendoli come visibile in fig.12.
Lo schema è così semplice che non ha certo bisogno di particolari consigli, comunque una volta stagnati tutti i terminali degli zoccoli e del connettore è consigliabile controllare con una lente d'ingrandimento che non vi sia una goccia di stagno tra due piedini che provochi un **corto**.
Come visibile nel disegno dello schema pratico conviene collocare il quarzo in posizione **orizzontale**, saldando il suo corpo sul circuito stampato con una goccia di stagno.
Nello zoccolo **IC2** (quello posto in alto verso il **CONN.1**) inserite l'integrato **74LS244** rivolgendo la tacca di riferimento verso il condensatore **C2**.
Nello zoccolo **IC1** inserite dopo averlo **programmato** il microprocessore **ST62E10**, rivolgendo la tacca di riferimento verso il condensatore **C1**.
Completato il montaggio di questo stampato potete prendere lo stampato siglato **LX.1171/B** e su questo saldare il **connettore maschio**, tutte le resistenze dalla **R3** alla **R10** ed i diodi **led**, come visibile in fig.14.
Quando inserite i diodi led nel circuito stampato dovete rivolgere il terminale **più corto** (terminale **K**) verso le resistenze.
L'ultimo stampato, quello siglato **LX.1171/D**, provvisto di due **display** vi servirà per testare i programmi **CONTA** e **LOTTO**.

microprocessore ST62E10

Come visibile in fig.15 su questo stampato fissate le resistenze da **R1** ad **R8**, che sono da **220 ohm**, e le due resist,enze **R9 - R10** che sono invece da **4.700 ohm**, poi il connettore maschio, i due transistor **TR1 - TR2** rivolgendo la parte piatta del loro corpo verso destra e per ultimi montate i due display, rivolgendo il lato con i **punti decimali** verso il basso.

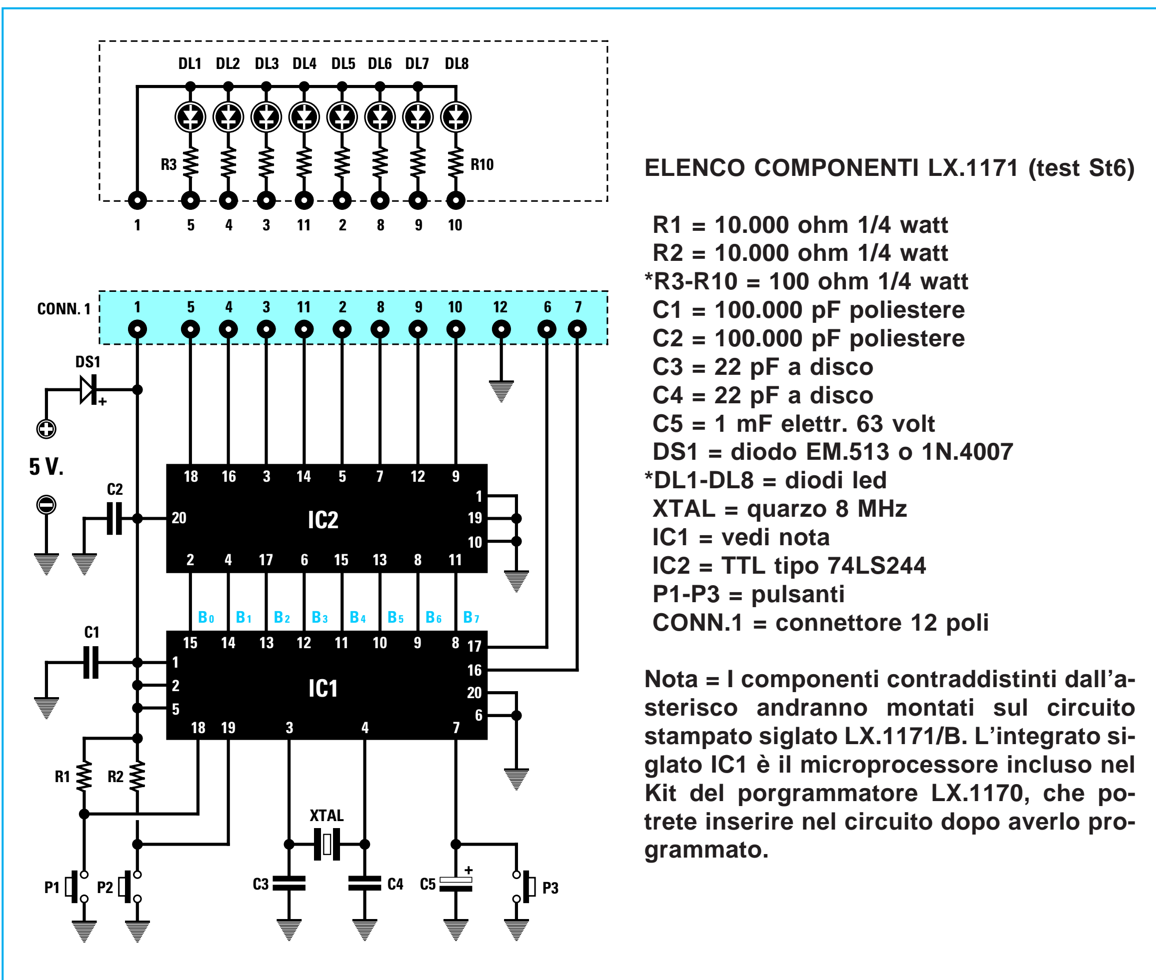
IMPORTANTE

Se nel microprocessore **ST62E10** avete memorizzato i dati del **programma LED**, dovrete inserire nel **CONN.1** della scheda **LX.1171** la scheda con gli **8 diodi led**, se avete memorizzato i dati del **programma CONTA** o del **programma LOTTO**, dovrete inserire la scheda con i **2 display**.
Se per errore scambiate le schede, **non causerete** nessun danno né all'integrato **IC2** né al microprocessore **IC1**, quindi basterà inserire la scheda giusta per vedere il circuito **funzionare**.
Se il circuito **non funziona**, potreste esservi sbagliati nel **memorizzare** il programma nelle memorie del microprocessore.
In questo caso dovrete esporlo sotto una luce ultravioletta per **cancellare** i dati dalla sua memoria, quindi dovrete **riprogrammarlo**.

Questa stessa operazione va effettuata se dopo aver **trasferito** il programma **LED** volete sostituirlo con il programma **CONTA** o con il programma **LOTTO**.

COLLAUDO MICROPROCESSORE nel CIRCUITO TEST

Dopo aver realizzato il circuito test siglato **LX.1171** potete collaudare il microprocessore che avete imparato a programmare con uno dei tre semplici programmi **LED - CONTA - LOTTO**, come vi abbiamo spiegato nell'articolo precedente.
Per prima cosa dovete inserire nello zoccolo a **20 piedini** del circuito test **LX.1171** il microprocessore appena programmato, rivolgendo la sua tacca di riferimento verso il condensatore **C1** (vedi fig.12).
Dopo questa operazione, se avete programmato il microprocessore con il programma **LED** dovete innestare nel connettore **femmina** del circuito test **LX.1171** il circuito applicativo a diodi led siglato **LX.1171/B**.
Se invece avete programmato il microprocessore con uno qualsiasi dei due programmi **CONTA** o **LOTTO**, dovete innestare nel connettore **femmina**



ELENCO COMPONENTI LX.1171 (test St6)

R1 = 10.000 ohm 1/4 watt
R2 = 10.000 ohm 1/4 watt
*R3-R10 = 100 ohm 1/4 watt
C1 = 100.000 pF poliestere
C2 = 100.000 pF poliestere
C3 = 22 pF a disco
C4 = 22 pF a disco
C5 = 1 mF elettr. 63 volt
DS1 = diodo EM.513 o 1N.4007
*DL1-DL8 = diodi led
XTAL = quarzo 8 MHz
IC1 = vedi nota
IC2 = TTL tipo 74LS244
P1-P3 = pulsanti
CONN.1 = connettore 12 poli

Nota = I componenti contraddistinti dall'asterisco andranno montati sul circuito stampato siglato LX.1171/B. L'integrato siglato IC1 è il microprocessore incluso nel Kit del porgrammatore LX.1170, che potrete inserire nel circuito dopo averlo programmato.

Fig.1 Schema elettrico del circuito che dovrete realizzare per poter verificare se il programma LED (vedi pagg.37-38) è stato correttamente memorizzato all'interno del micro ST62E10. Per alimentare questo circuito occorre una tensione esterna stabilizzata di 5 volt.

del circuito test il circuito applicativo con i due **display** siglato **LX.1171/D**.
A questo punto potete passare al collaudo vero e proprio del programma caricato nelle memorie del microprocessore.

COLLAUDO PROGRAMMA LED

Dopo aver programmato il microprocessore con il programma **Led** ed averlo inserito nel circuito stampato siglato **LX.1171**, dovete innestare il connettore **maschio** del circuito a **diodi led** nel connettore **femmina** presente sul circuito test, poi dovete alimentare quest'ultimo con una tensione di **5 volt** stabilizzati.

Il programma **LED** vi dà la possibilità di far lampeggiare gli **8 diodi** led presenti nel circuito con **5** diverse **modalità**, che potete selezionare pigiando ripetutamente il pulsante **P1**.

1° Lampeggio
I led si accendono in sequenza uno alla volta da **sinistra** verso **destra**. Il ciclo continua all'infinito.

2° Lampeggio
I led si accendono due alla volta dall'**esterno** verso l'interno (prima **DL1** e **DL8**, poi **DL2** e **DL7** ecc.), fino ai due led centrali (**DL4** e **DL5**), poi i led si accendono sempre due alla volta, ma in senso inverso, cioè dall'interno verso l'esterno. Il ciclo continua all'infinito.

3° Lampeggio
Si accende tutta la fila di led, iniziando dal primo a **sinistra** e proseguendo verso **destra**. Quando sono tutti accesi si spengono tutti insieme. Il ciclo riprende all'infinito.

4° Lampeggio
Lampeggiano uno alla volta prima i led **pari** poi i **dispari**, poi si spengono e si accendono tutti insieme. Il ciclo continua all'infinito.

5° Lampeggio
I led si accendono prima tutti insieme, poi si spengono tutti insieme. Il ciclo si ripete all'infinito.

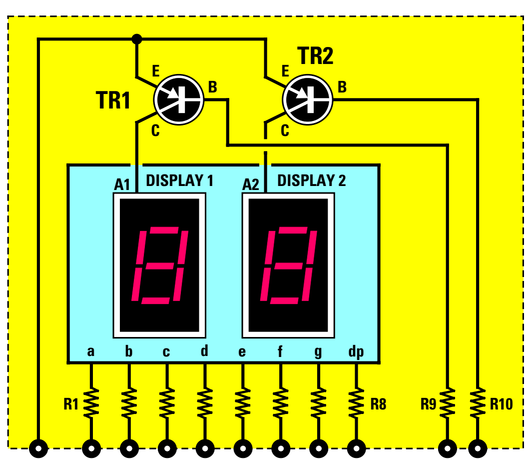
Non appena il circuito viene alimentato, il microprocessore esegue il **1° motivo**. Premendo ripetutamente il pulsante **P1** vengono eseguiti uno di seguito all'altro il **2° - 3° - 4° - 5° motivo**. Se mentre è in corso il **5°** premete nuovamente **P1**, il microprocessore eseguirà di nuovo il **1° lampeggio**.
L'intervallo fra un'accensione dei diodi led e l'altra è di circa **1/2 secondo**, ma è possibile **diminuire** questo tempo premendo ripetutamente **P2**.
La massima velocità di lampeggio consentita dal programma viene raggiunta dopo aver premuto questo pulsante per **8 volte**.
Premendolo ancora una volta, il lampeggio riprenderà con la stessa velocità iniziale.
Premendo in qualunque momento il pulsante **P3** (**RESET**), il microprocessore tornerà ad eseguire il programma da capo, cioè ripartirà dal **primo** lampeggio come se aveste alimentato solo in quel momento il circuito.

Se volete passare al collaudo di uno degli altri due programmi **CONTA - LOTTO**, dovete togliere l'alimentazione al circuito ed estrarre il circuito a led **LX.1171/B**. Ovviamente dovete pure estrarre il microprocessore, e dopo averlo **cancellato**, dovete **riprogrammarlo** con uno degli altri due programmi.

COLLAUDO PROGRAMMA CONTA

Dopo aver programmato il microprocessore con il programma **Conta** ed averlo inserito nel circuito stampato siglato **LX.1171**, dovete innestare il connettore **maschio** del circuito a **display** nel connettore **femmina** presente sul circuito test, poi dovete alimentare quest'ultimo con una tensione di **5 volt** stabilizzati.

Dopo aver alimentato il circuito vedrete comparire sui display il numero **00**, che aumenterà di una unità ogni **1/2 secondo**.
Pertanto ogni 5 decimi di secondo leggerete **01 - 02 - 03 - ecc.** fino a **99**, dopodiché il conteggio ripartirà sempre in avanti da **00**.
Per ottenere un conteggio all'**indietro**, potete premere in qualunque istante il pulsante **P2**.
Supponendo di premere **P2** quando sui display compare ad esempio il numero **74**, vedrete apparire, sempre ad intervalli di **1/2 secondo**, i numeri



ELENCO COMPONENTI LX.1171/D

R1-R8 = 220 ohm 1/4 watt
R9 = 4.700 ohm 1/4 watt
R10 = 4.700 ohm 1/4 watt
TR1 = PNP tipo BC327
TR2 = PNP tipo BC327
DISPLAY1-2 = display Anodo comune tipo HP.5082 o 7731

Fig.2 Se all'interno del micro ST62E10 avete memorizzato il programma CONTA o LOTTO, per poterlo controllare dovrete realizzare questo circuito elettrico. La scheda dei diodi led o dei display andrà inserita nel connettore dell'LX.1171 (vedi figg.19-20).

73 - 72 - 71 - ecc., fino a **00**, dopodiché il conteggio riprenderà da **99** per tornare a **00** e così all'infinito.
Premendo in qualunque momento il tasto **P1** il conteggio proseguirà di nuovo in **avanti** e così pure premendo in qualsiasi momento il pulsante **P2** il conteggio riprenderà all'**indietro**.
Infine potrete riprendere l'esecuzione del programma da capo premendo il pulsante **P3** (**RESET**), perché in tal modo sarà come se aveste appena alimentato il circuito.
In questo caso il conteggio ripartirà da **00** e verrà effettuato in **avanti**.

COLLAUDO PROGRAMMA LOTTO

Dopo aver programmato il microprocessore con il programma **Lotto** ed averlo inserito nel circuito stampato siglato **LX.1171**, dovete innestare il connettore **maschio** del circuito a **display** nel connettore **femmina** presente sul circuito test, poi dovete alimentare quest'ultimo con una tensione di **5 volt** stabilizzati.

Dopo aver alimentato il circuito vedrete comparire sui display due lineette (**--**) ed ogni volta che premerete il pulsante **P1** comparirà un numero sempre diverso compreso fra **01** e **90**, cioè i numeri della tombola o del **lotto**.
Ogni volta che premete **P1** il numero non sarà **mai uguale** ai **precedenti**, quindi potrete simulare una reale estrazione di numeri.
Una volta estratti tutti i **90 numeri**, vedrete comparire sui display le due lineette (**--**), quindi saprete che sono stati estratti tutti i **90 numeri** disponibili.
Quando compaiono le due lineette (**--**), per iniziare una nuova estrazione basterà premere **P1**, e così sempre in maniera casuale ricompariranno i numeri compresi fra **00** e **90**.
Se invece volete iniziare una nuova estrazione interrompendo quella in corso, sarà sufficiente **resettare** il microprocessore premendo il pulsante **P3** (**RESET**). In tal modo il programma verrà eseguito da capo, esattamente come se aveste appena alimentato il microprocessore, anche se i numeri non sono stati tutti estratti.
In questo programma il pulsante **P2** non viene utilizzato.

PER VEDERE il LISTATO di un PROGRAMMA

Le informazioni seguenti vi saranno utili quando vorrete entrare nel listato di un programma per modificarlo.
Per visualizzare un qualunque listato di uno dei pro-

Fig.3 Per poter vedere il listato dei programmi, quando sullo schermo appare il menu principale, pigiate F3 quindi scegliete quello che vi interessa, cioè Conta, Led o Lotto e premete Enter.

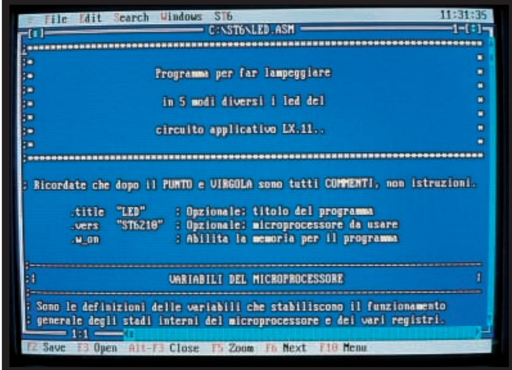


Fig.4 Se sceglierete il programma Led, sullo schermo del computer appariranno tutte le righe di tale programma. Per uscire da questo listato dovete premere Alt ed F3 e apparirà il menu principale.

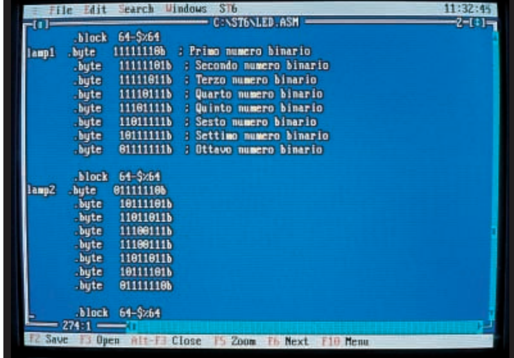


Fig.5 I più esperti potranno apportare personali modifiche a questi programmi pigiando il tasto F2 per memorizzarle. Pigiando Alt+F3 potrete non confermare le modifiche apportate.

gramma per **ST62**, anche senza bisogno di modificarlo, dovete innanzitutto caricare il programma. Quando sul monitor del computer compare:

C:\>

digitate:

C:\>CD ST6 poi Enter
C:\ST6>ST6 poi Enter

Così entrerete nel menu principale di fig.3.

Premete il tasto **F3** per visualizzare l'elenco dei files contenenti i programmi per **ST62**.
Premete **Enter** e dopo aver portato il cursore sul nome del file desiderato, premete ancora **Enter**. In questo modo comparirà il listato del programma contenuto in quel file.

Per muovervi all'interno del listato e vedere così tutte le istruzioni usate i tasti freccia **su/giù** oppure i due tasti **pagina su/giù**.

Per **uscire** dal listato di un file, dovete tenere premuto **Alt** e premere **F3**. Ritornerete così al menu principale (vedi fig.3).

Nota: Se mentre visualizzate il listato premete per errore i tasti scrivendo nel file dei caratteri indesiderati, senza curarvi di andarli a cancellare, potete uscire dal file **senza** salvare le modifiche.
Per compiere questa operazione è sufficiente tenere premuto il tasto **Alt** e premere **F3**, e quando appare la finestra di conferma di fig.7 dovete premere il tasto **N**.
Ricordate che se dopo aver modificato il file senza volerlo, premete inavvertitamente **F2**, le modifiche verranno **salvate**, quindi premendo **Alt+F3** la finestra di conferma modifiche (vedi fig.7) **non apparirà**.
In questo caso l'unico modo per correggere le modifiche è **entrare** di nuovo nel file, cercare la riga del listato dove avete apportato le modifiche e correggerla.
Nel caso non riusciate a correggere l'errore neanche in questo modo, non vi rimane altro che **installare** di nuovo il programma, perché in tal caso caricherete nell'Hard-Disk i **programmi originali** contenuti nei files **LED - CONTA - LOTTO**.

Per MODIFICARE un PROGRAMMA

Chi sa già programmare potrebbe trovare questo paragrafo poco interessante, ma poiché dobbiamo pensare anche a tutti i principianti, riteniamo necessario spiegare anche quello che per molti potrebbe essere ovvio.

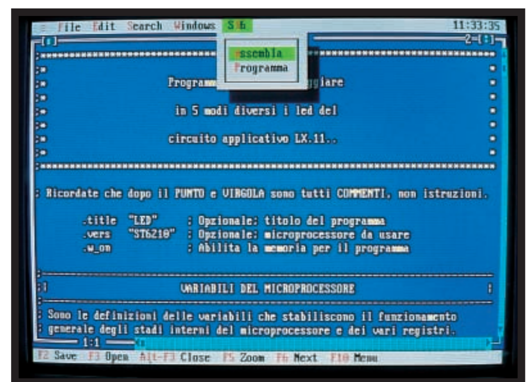


Fig.6 Una volta salvate le modifiche con F2 (vedi fig.5) dovrete riassemblare tutto il programma. Premete Alt+T e quando apparirà questa finestra pigiate il tasto A. L'assemblaggio dura solo pochi secondi.

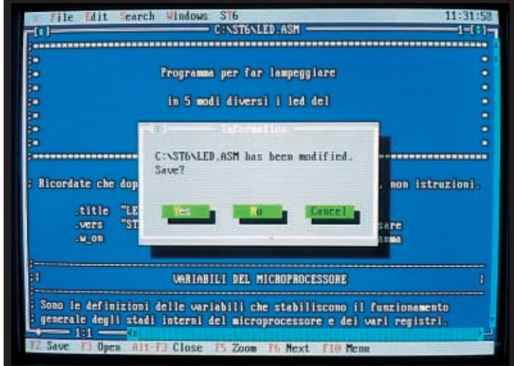


Fig.7 Se non volete memorizzare le modifiche effettuate NON dovrete pigiare il tasto F2, ma solo Alt+F3 e così apparirà questa finestra. A questo punto dovrete semplicemente pigiare N.

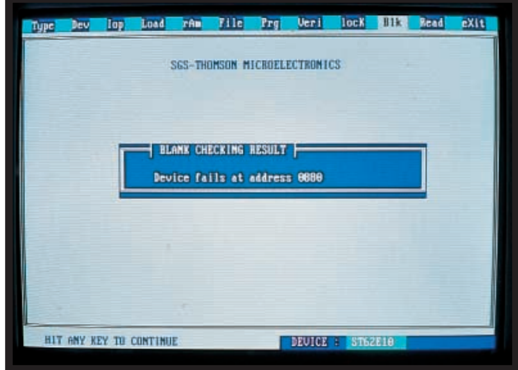


Fig.8 Se prima di programmare un ST62E10, seguendo quanto descritto da pag.26, pigiate il tasto B, il computer vi dirà se il microprocessore inserito nello zoccolo textool è vergine.

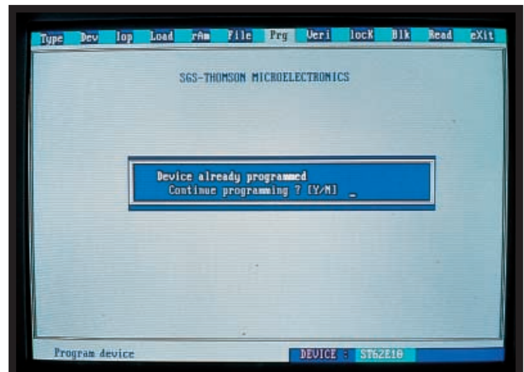


Fig.9 Se tentate di programmare un ST62E10 già programmato il computer vi mostrerà questa scritta. Se volete inserire un diverso programma, dovrete premere N e cancellare il micro.

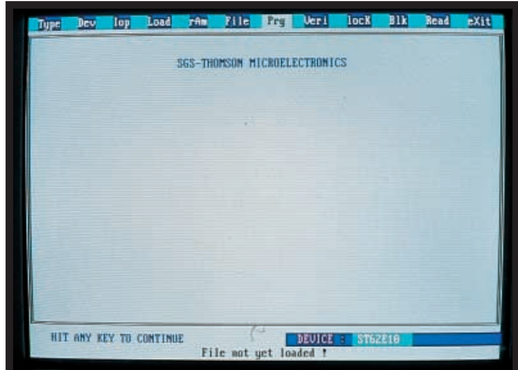


Fig.10 Se nel programmare un microprocessore vi dimenticate di scrivere il nome del file del programma, Conta - Led - Lotto, il computer lo segnalerà con questo messaggio.

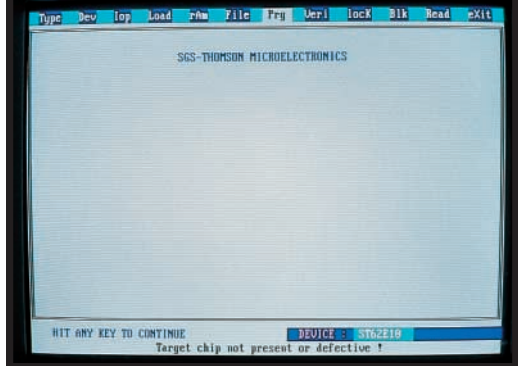


Fig.11 Se il microprocessore è stato inserito nello zoccolo textool in modo errato o se il programmatore non è alimentato, sullo schermo del computer apparirà questo messaggio.

Sapere come **entrare** in un file e come procedere per **modificare** qualcuna delle istruzioni dei programmi che vi abbiamo fornito, costituisce un primo importante passo per tutti coloro che non hanno mai visto come sono scritte le diverse righe di un programma e che in un secondo tempo vorranno provare a realizzare dei semplici e personali programmi.

Per spiegarvi queste prime cose prenderemo spunto dal programma più semplice che vi abbiamo proposto, quello cioè chiamato **LED**, e su questo vi insegneremo come si deve procedere per **cambiare** le modalità di lampeggio degli 8 diodi led, cioè per fare in modo che i diodi led possano lampeggiare in modo **diverso** da quello da noi proposto.

Quando, dopo aver caricato il programma, compare il menu principale di fig.3, premete il tasto **F3** (tasto per l'**apertura dei files**).

Apparirà la finestra con l'elenco dei file dei programmi, cioè:

CONTA.ASM
LED.ASM
LOTTO.ASM
STANDARD.ASM

Nota: Il file **STANDARD.ASM** non contiene un programma vero e proprio, ma delle utili indicazioni per capire il significato, l'uso e l'importanza delle varie istruzioni di ogni programma per **ST62**. Come vi spiegheremo nel prossimo paragrafo, potrete entrare in questo file e leggere tutti gli utili commenti che abbiamo inserito.

A questo punto premete **Enter**, portate il **cursore** sulla riga **LED.ASM** e premete ancora **Enter**.

Sul monitor comparirà tutto il **listato** del programma contenuto nel file **LED.ASM** (vedi fig.4).

In basso a sinistra sono presenti due numeri separati dai **due punti** (:). Il **primo** numero vi permette di identificare la riga del programma, il **secondo** la colonna del file.
Con i tasti **freccia giù** o **pagina giù** portate il cursore in prossimità della riga **255**, cioè scendete con il cursore fino a quando in basso a sinistra non leggete **255:1**.
Dalla riga **255** in poi (vedi fig. 5) compaiono delle istruzioni del tipo:

```
lamp1    .byte 11111110b ; Prima istruzione
         .byte 11111101b ; Seconda istruzione
```

```
.byte 11111011b ; Terza istruzione
.byte 11110111b ; Quarta istruzione
.byte 11101111b ; Quinta istruzione
.byte 11011111b ; Sesta istruzione
.byte 10111111b ; Settima istruzione
.byte 01111111b ; Ottava istruzione
```

dove **lamp1** sta ad indicare che le istruzioni successive sono relative alla prima modalità di lampeggio dei diodi led.

Nota: Le scritte dopo il **punto e virgola** (;) **non** sono **istruzioni**, ma **commenti**, che abbiamo inserito appositamente nel listato per rendere più comprensibili le spiegazioni che ora vi daremo. Per questo motivo vi consigliamo di **non** cambiarle.

Per capire in che modo è possibile cambiare queste istruzioni per variare la modalità di accensione dei vari diodi led, cercheremo di spiegarvi in modo molto semplice come funzionano queste istruzioni.

Le **cifre** siglate **1** e **0** che trovate dopo l'istruzione **.byte** compongono un numero **binario**, riconoscibile per la presenza della lettera **b = b**inario.
Come noterete, queste **cifre binarie** sono **8** e ad ognuna di esse corrisponde un diverso **diodo led** del circuito test **LX.1171/B** (vedi fig.1).
Ad esempio, alla prima cifra da **destra** corrisponde **DL1**, alla **seconda** corrisponde **DL2**, e così via fino all'**ottava** cifra da destra, alla quale corrisponde **DL8**.
Ogni volta che il microprocessore esegue un'istruzione come:

```
.byte 11111011b ; Terza istruzione
```

i diodi led corrispondenti alle **cifre binarie** uguali a **0** vengono **accesi**, mentre i diodi led corrispondenti alle **cifre binarie** uguali ad **1** rimangono **spenti**.
Quindi quando il microprocessore esegue questa istruzione, viene **acceso** il solo diodo **DL3**, mentre tutti gli **altri** rimangono **spenti**.

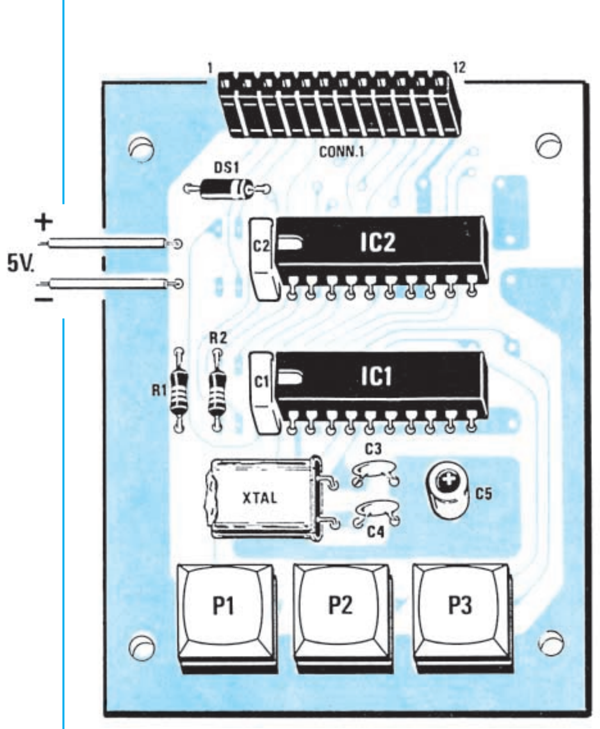


Fig.12 Schema pratico di montaggio della scheda sperimentale LX.1171. L'integrato IC1 è il microprocessore ST62E10 che vi abbiamo fatto programmare con il progetto pubblicato a pag.26. I pulsanti P1 - P2 vi serviranno per modificare le funzioni o la velocità (leggere articolo), mentre il pulsante P3 serve per resettare il circuito.

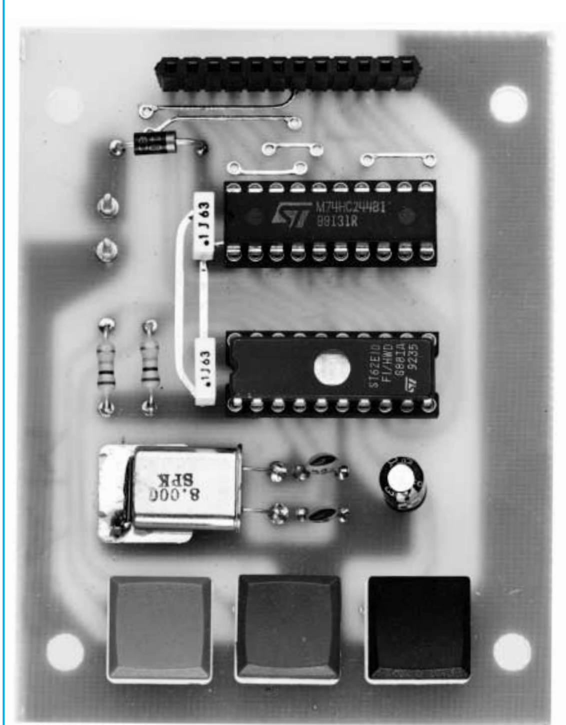


Fig.13 Foto del circuito LX.1171 come si presenta a montaggio ultimato. Si noti nello zoccolo IC1 il microprocessore ST62E10 provvisto della finestra di cancellazione ed in alto il connettore per poter inserire la scheda con i diodi led (vedi fig.14) o con i display (vedi fig.15) Il circuito va alimentato con una tensione di 5 volt.

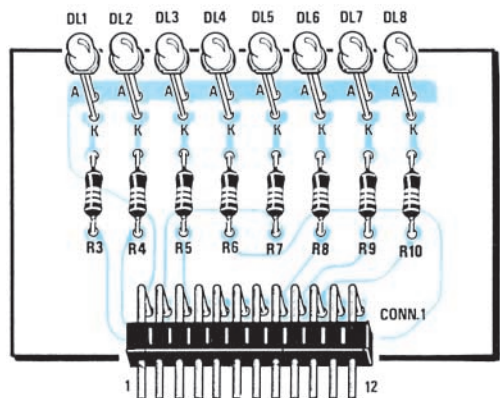


Fig.14 Schema pratico di montaggio della scheda a diodi led da usare se avete memorizzato nell'ST62E10 il programma LED.

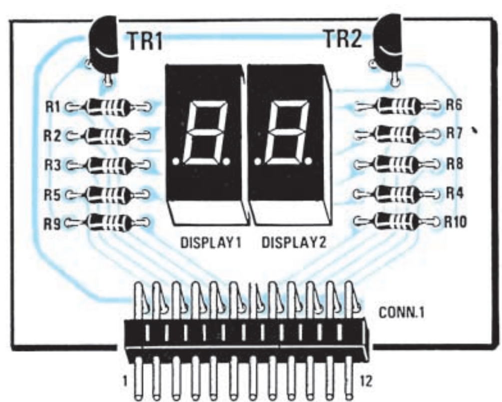


Fig.15 Schema pratico della scheda display da usare se nell'ST62E10 avete memorizzato il programma CONTA oppure LOTTO.

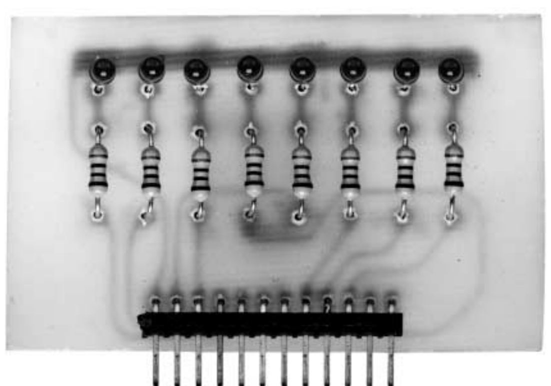


Fig.16 Foto della scheda LX.1171/B dei diodi Led a montaggio ultimato.

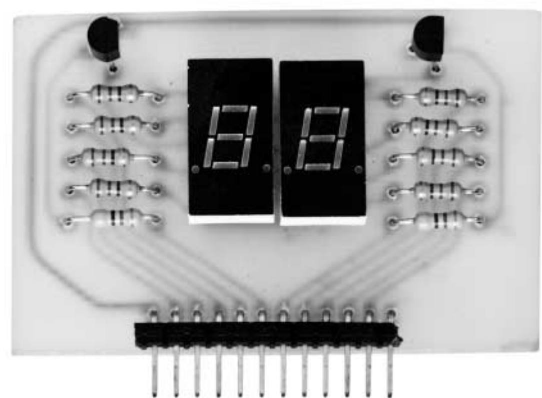


Fig.17 Foto della scheda LX.1171/D dei display a montaggio ultimato.

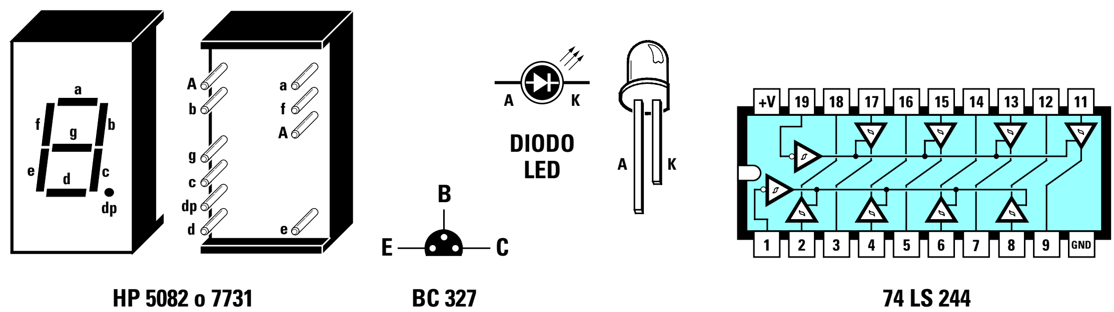


Fig.18 Connessioni dei display viste da dietro, del transistor BC.327 viste da sotto e dell'integrato 74LS244 viste da sopra. Il terminale più lungo presente nei diodi led è l'Anodo.

Se nelle diverse righe di istruzione si scrivono differenti numeri binari, il microprocessore accenderà ogni volta dei diodi led diversi, ed in questo modo potrete creare differenti giochi di luce.

Se ad esempio considerate le istruzioni del lampeggio chiamato **lamp1**, vedete che quando viene eseguita la **prima istruzione** si accende solo il diodo **DL1**, perché solo la cifra più a destra è uno **0**, poi quando viene eseguita la **seconda** istruzione si accende solo **DL2** e così via. In questo modo si è realizzata una semplice accensione in sequenza di un solo diodo alla volta.

Cambiando le cifre **1** e **0** che compongono i vari numeri binari potete realizzare con un po' di fantasia tutti i giochi di lampeggio che vorrete.

Per cambiare le cifre che compongono i numeri binari è sufficiente che vi portiate col cursore sulla cifra che volete modificare, dopodiché potete scrivere **1** o **0**.Per cancellare la cifra binaria che volete sostituire portate il cursore su quella cifra e premete il tasto **Canc**.

Ricordate che ogni numero binario deve essere composto da **non più** di **8 cifre**, altrimenti il programma non funzionerà.
Se scrivete un numero di cifre **inferiori** ad **8** il programma funzionerà ugualmente, ma i diodi corrispondenti alle cifre **non utilizzate** rimarranno sempre **accesi**. Ad esempio, scrivendo **11010b**, cioè tralasciando le tre cifre corrispondenti ai diodi **DL6 - DL7 - DL8**, questi diodi rimarranno sempre **accesi**.

Se volete potete cambiare le cifre binarie di tutti i **5 giochi** proposti, quindi potete modificare anche le istruzioni scritte dopo le etichette **lamp2 - lamp3 - lamp4 - lamp5**. Troverete queste scritte scorrendo con il cursore il listato del programma.
I giochi di luce supportati da questo programma sono solo **5**, quindi non aggiungete altre etichette del

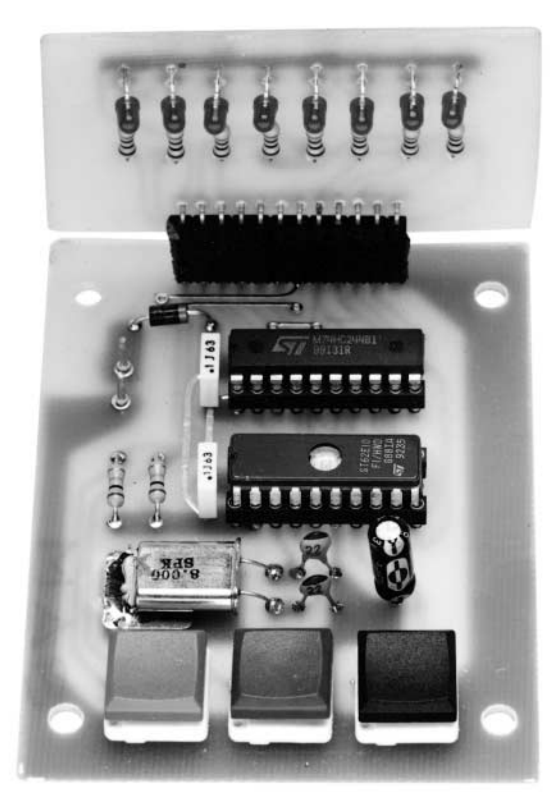


Fig.19 Se avete programmato il microprocessore con il programma LED dovrete inserire nel connettore dell'LX.1171 il connettore maschio della basetta di fig.14.

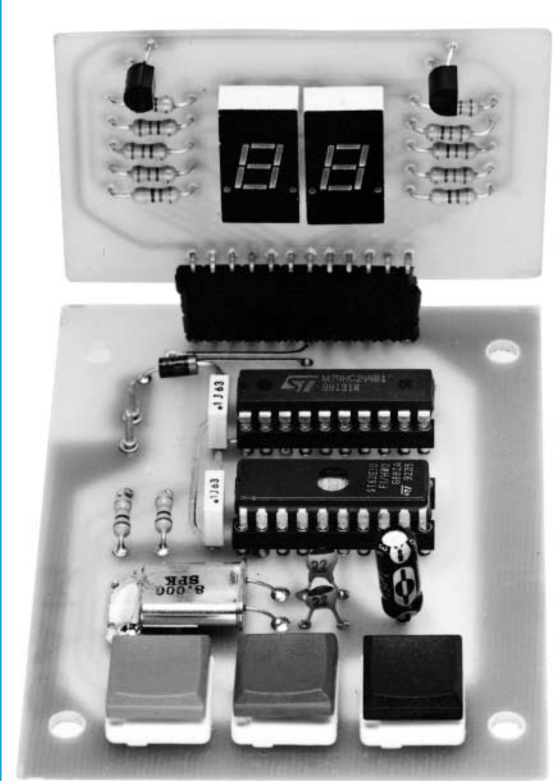
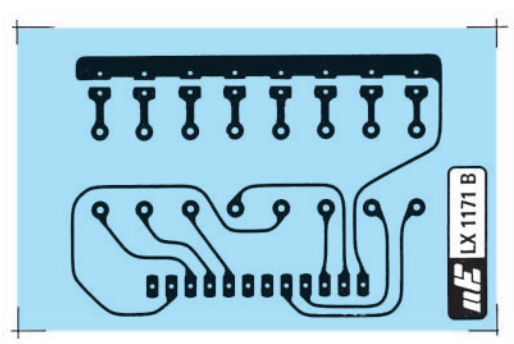


Fig.20 Se avete programmato il microprocessore con il programma CONTA o LOTTO dovrete inserire nel connettore femmina dell'LX.1171 la basetta visibile in fig.15.



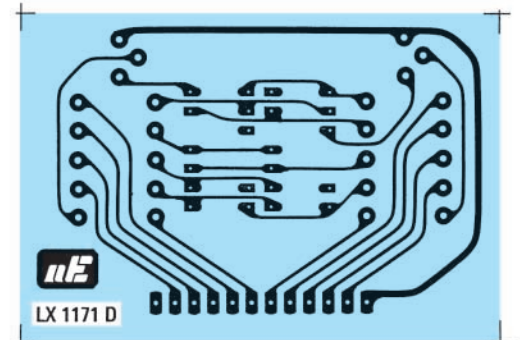


Fig.21 Disegno a grandezza naturale dei due stampati da inserire nella scheda LX.1171 (vedi figg.19-20) visti dal lato rame. Il circuito di sinistra siglato LX.1171/B serve per il programma Led, mentre quello siglato LX.1171/D serve per i programmi Conta e Lotto.

tipo **lamp6 - lamp7** ecc. seguite dalle istruzioni **.byte**, perché il programma non funzionerebbe.

NOTA IMPORTANTE: Come avrete notato, le istruzioni di tipo **.byte** per accendere i led, che seguono le cinque etichette **lamp1 - lamp2 - ecc.**, sono **8**, e così devono sempre rimanere. Se cancellate alcune di queste istruzioni oppure ne aggiungete altre analoghe, il programma **non funzionerà**.

Eseguite le vostre modifiche dovete **salvarle**, altrimenti anche se leggerete sul monitor le istruzioni che avete appena scritto, il programma non risulterà modificato.

Per **salvare** le variazioni basterà **premere** il tasto **F2**: dopo pochi istanti vedrete accendersi la luce dell'Hard-Disk, quindi sarete sicuri che il file modificato è stato aggiornato.

Se **non** desiderate **salvare** le modifiche dovete tenere premuto **Alt** e poi premere **F3**.
Comparirà la finestra di fig.7 dove vi verrà chiesto se volete salvare (tasto **Y**) oppure no (tasto **N**) le modifiche apportate.
Premendo uno qualsiasi di questi tasti tornerete nel menu di fig.3.

Una volta apportate e salvate le modifiche, **prima** di trasferire i dati del programma nelle memorie del microprocessore dovete eseguire un'operazione supplementare, cioè lanciare il programma **assembla**, che serve per convertire le istruzioni del programma in dati che il microprocessore utilizza per eseguire il programma.

Senza uscire dal listato del programma, dopo aver apportato le modifiche ed averle salvate con il tasto **F2**, premete i tasti **Alt+T** e di seguito **A** (vedi fig.6).
In questo modo lo schermo del vostro computer diventerà tutto **nero** e dopo alcuni secondi vedrete apparire questa scritta:

*** **SUCCESS** ***

che conferma che l'**assemblaggio** è stato completato **senza errori**.
Finita l'operazione di assemblaggio, premendo un tasto qualsiasi tornerete al listato del programma.

Se anziché apparire la scritta *** **SUCCESS** *** compare ad esempio:

ERROR C:\ST62\LED.ASM 256:

significa che nella riga **256** del file **LED.ASM** avete involontariamente inserito un **errore**.

Nota: Un errore molto comune nel quale si può incappare è quello di scrivere un numero binario con un numero di **cifre** superiore ad **8**. Se per esempio scrivete un numero binario a **9 cifre** del tipo:

.byte 110101001b ; Seconda istruzione

dopo aver lanciato il programma **assembla** comparirà il messaggio di errore:

ERROR C:\ST62\LED.ASM 256:
(81) 8-bit value overflow

Il numero tra parentesi **(81)** identifica il tipo di errore e non vi interessa, mentre la scritta **8-bit va-**

lue overflow significa che avete utilizzato un numero binario con più di **8 bit**

Per correggere questo errore dovete tornare al listato del programma e per questo basterà premere un tasto qualsiasi.

Una volta nel listato, poiché l'errore era stato segnalato nella riga **256**, dovete portarvi con il cursore su questa riga e controllare che risulti effettivamente scritto un numero di **8 bit** (**8 cifre**).

Questo **errore** si verificherà raramente perché scrivere un'istruzione così semplice non sarà per voi un problema, comunque lo abbiamo voluto segnalare, perché se un domani dovesse apparire per un vostro programma un **qualunque** messaggio di errore, sappiate che questo è presente nella riga indicata prima dei **due punti** (**:**).

Per **uscire** dal file **LED.ASM** dovete tenere premuto **Alt** e poi premere **F3**. Se non avete ancora salvato la correzione, comparirà la finestra di fig.7, in cui dovrete indicare se volete salvare (tasto **Y**) oppure no (tasto **N**) le modifiche.
Premendo **Y** registrerete queste modifiche, premendo **N** invece **non** le **salverete**.
Facciamo presente che premendo uno qualsiasi di questi tasti, **Y** o **N**, il listato del file **LED** scomparirà e ritornerete nel menu principale.

Il FILE STANDARD.ASM

Come abbiamo già avuto modo di sottolineare, premendo **F3** dal menu principale, oltre ai tre files di tipo **.ASM** contenenti i programmi (–) test per **ST62**, compare un file chiamato **STANDARD.ASM**, che **non** contiene un **programma** vero e proprio.
In questo file trovate l'elenco delle istruzioni che devono comparire sempre in ogni programma, e la cui conoscenza è **basilare** se si desidera realizzare programmi personali.
Sono inoltre presenti tantissimi **commenti**, che vi aiuteranno a capire il significato delle varie istruzioni e tante note utilissime per realizzare programmi per **ST62** senza incorrere negli errori più banali.
Per visualizzare questo file dovete eseguire di nuovo le operazioni spiegate nel paragrafo **Per vedere il listato di un programma**, e quando compare l'elenco dei files dovete premere **Enter**, portare il cursore sul nome **STANDARD.ASM** e premere di nuovo **Enter**.
Questo file risulterà molto utile sia ai più esperti, che vogliono cimentarsi nella realizzazione di programmi senza attendere l'uscita della prossima rivista, sia a chi è alle prime armi, perché potrà iniziare a riconoscere le istruzioni principali che ricorrono in qualsiasi programma per **ST62**.
Probabilmente a molti di voi queste scritte appariranno ancora oscure e prive di significato, quindi non perdete il prossimo numero in cui vi spiegheremo il significato di **tutte** le istruzioni.

Per TORNARE al DOS

Quando avete terminato le operazioni di programmazione potete uscire dal programma e ritornare al **DOS**.

Se vi trovate nel menu di fig.3, basterà tenere premuto **Alt** e poi pigiare **X** e così sul monitor comparirà:

C:\ST6>

a questo punto per uscire dalla directory **ST6** e lanciare altri programmi dovrete digitare:

**C:\ST6>CD ** poi Enter

e così comparirà:

C:\\>

Arrivederci al prossimo numero.

COSTO DI REALIZZAZIONE

Costo del kit LX.1171 completo di circuito stampato, dell'integrato 74LS244 (non c'è l'ST62E10 perché inserito nel kit LX.1170) più il quarzo, i pulsanti ed il circuito stampato LX.1171/B (vedi fig.14) con gli 8 diodi led .... € 12,90

Il kit LX.1171/D con i due display ed i due transistor BC.327 (vedi fig.15)............................ € 4,90

Costo del solo stampato LX.1171 .... € 3,72
Costo del solo stampato LX.1171/B........... € 0,93
Costo del solo stampato LX.1171/D........... € 1,08

I prezzi sopra riportati sono già compresi di IVA, ma non sono incluse le spese postali di spedizione a domicilio.

IMPARARE a programmare i

Come vi abbiamo anticipato nella precedente rivista, per programmare un **ST6**, come del resto un qualunque altro microprocessore, è assolutamente necessario conoscere le **basi** del linguaggio di programmazione, perché senza queste è impossibile scrivere un programma.
Tanto per fare un esempio, se vi proponessimo di progettare un amplificatore utilizzando un **integrato operazionale**, senza precisare come si collega il piedino **invertente** o quello **non invertente** o quali modifiche vanno apportate per alimentare il circuito con una tensione **singola** anziché **duale**, incontrereste parecchie difficoltà nella sua realizzazione.

Evidenziamo questo perché non vogliamo comportarci come tanti altri, che spiegano poco o niente ed illudono i loro lettori sostenendo che non c'è nulla di più facile che programmare un ST6.
Prima di insegnarvi a programmare sarà quindi utile spiegare, anche solo a grandi linee, cosa sono un **registro** e una **subroutine**, il significato di tutte le istruzioni, quali **jp - jrr - jrs - ld - cp** ecc., e come si utilizza una **memoria**.
A questo proposito vogliamo sottolineare che non è sufficiente imparare a memoria il significato di tutte le istruzioni, ma occorre anche sapere **come - dove - quando** utilizzarle, se si vuole che il programma funzioni correttamente.
Non illudetevi pensando di diventare esperti programmatori in pochi giorni, perché andreste incontro ad una delusione: sono necessari infatti alcuni mesi di pratica per acquisire una sufficiente padronanza della materia.
Per accelerare l'apprendimento vi suggeriamo di iniziare a leggere i programmi già funzionanti, perché studiare le soluzioni adottate per scrivere una subroutine, utilizzare un interrupt, o ancora vedere come si sfrutta la memoria, vi sarà di valido aiuto.
In altre parole, anche se sapete scrivere una lettera o una cartolina, non è detto che siate capaci di scrivere un libro giallo o un bel romanzo di avventura, quindi se decidete di dedicarvi all'attività di scrittore, dovete prima acquisire un po' di esperienza facendovi seguire da un letterato o leggendo testi d'autore per apprendere come impostare i vari capitoli.
Chi sa già programmare in **Basic - Fortran - Pascal - C** è avvantaggiato rispetto a chi inizia da "zero", anche se come vedrete, tutti i microprocessori **ST6** utilizzano un **linguaggio assembler** molto semplificato.

PER SCRIVERE un PROGRAMMA

Prima di scrivere qualsiasi programma è necessario sapere quali operazioni deve eseguire il microprocessore, perché in funzione della memoria occupata, dovrete scegliere il micro più idoneo.
Infatti se avete un programma che non supera i **2K**, potete utilizzare un **ST62E10**, se invece avete un programma che occupa più di **2K** e non supera i **4K**, dovete necessariamente adoperare un **ST62E20**.
Nelle **Tabelle N.1-2** riportiamo per ogni microprocessore la **memoria** disponibile ed il massimo numero di **ingressi/uscite** utilizzabili, cioè il numero di piedini che potete adoperare per i segnali.

TABELLA N.1 micro NON CANCELLABILI

Sigla Micro	memoria utile	Ram utile	zoccolo piedini	piedini utili per i segnali
ST62T.10	2 K	64 byte	20 pin	12
ST62T.15	2 K	64 byte	28 pin	20
ST62T.20	4 K	64 byte	20 pin	12
ST62T.25	4 K	64 byte	28 pin	20

TABELLA N.2 micro CANCELLABILI

Sigla Micro	memoria utile	Ram utile	zoccolo piedini	piedini utili per i segnali
ST62E.10	2 K	64 byte	20 pin	12
ST62E.15	2 K	64 byte	28 pin	20
ST62E.20	4 K	64 byte	20 pin	12
ST62E.25	4 K	64 byte	28 pin	20

Quando scrivete un programma, dovete inserire tutte le istruzioni nella sequenza in cui volete che siano eseguite dal microprocessore.
Ad esempio, se voleste programmare un **robot** per

MICROPROCESSORI ST6

Dopo avervi presentato sul N.172/173 un programmatore per microprocessori ST6 ed un circuito per i test, sarete curiosi di conoscere le procedure per scrivere i vostri programmi, ad esempio per realizzare un orologio, per pilotare dei display alfanumerici LCD, per realizzare generatori d'impulsi ecc. Se ci seguirete, cercheremo di spiegarvi con facili esempi tutte le istruzioni necessarie per scrivere i programmi per l'ST6.

cuocere degli spaghetti, dovreste fornirgli nell'ordine queste istruzioni:

- **Prendi una pentola**
- **Riempila per metà di acqua**
- **Metti il tutto sul fornello**
- **Accendi il gas sotto la pentola**
- **Attendi che l'acqua bolla**
- **Versaci un po' di sale**
- **Immergi gli spaghetti nell'acqua**
- **Attendi 5-6 minuti**
- **Spegni il fornello**
- **Togli la pentola dal fornello**
- **Scola la pasta**

Se vi dimenticate anche una sola di queste istruzioni, quale ad esempio quella di **accendere il fornello**, non riuscirete mai a cuocere gli spaghetti.
E così se vi dimenticate di inserire l'istruzione **riempi la pentola con acqua**, non potrete mai arrivare alla condizione di vedere l'acqua **bollire**.

Dunque prima di apprestarvi a scrivere un programma dovete sapere:

Come aprire un file per il programma
Come scrivere le istruzioni richieste
Come impostare il programma
Come utilizzare la memoria

COME CREARE un FILE SORGENTE

Chi ha già richiesto il kit **LX.1170** del **Programmatore per ST6** pubblicato sulla rivista **N.172/173** avrà ricevuto un **dischetto floppy**, che oltre a servire per trasferire un programma dall'Hard-Disk nella memoria di un microprocessore **ST6**, serve per creare i **files sorgenti** necessari per scrivere qualsiasi vostro programma.
Infatti in questo dischetto è stato memorizzato un ottimo **editor**, corredato di tantissime **opzioni** che vi saranno utili per scrivere le istruzioni, per duplicarle, per cancellarle ed anche per salvare i files modificati; nel floppy è stato inoltre incluso un **assemblatore**.
Nella rivista precedente (se non ne siete in possesso potete richiedercela, perché abbiamo ancora delle copie disponibili), vi abbiamo spiegato co-

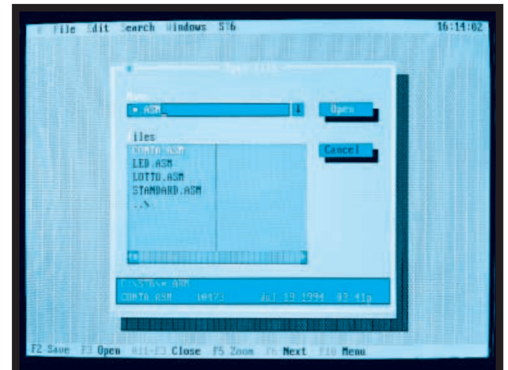


Fig.1 Per aprire un nuovo file dovete richiamare il menù principale e a questo punto se premete il tasto F3 appariranno tutti i nomi dei files del programma ST6 con estensione .ASM.

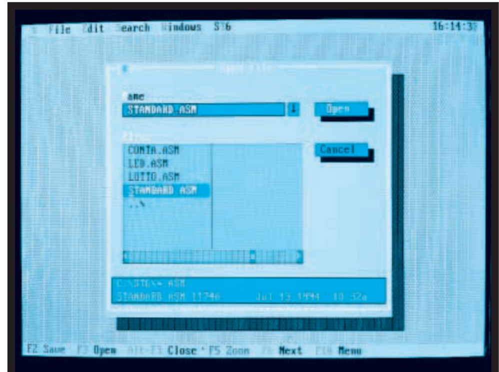


Fig.2 Dalla fig.1 premete Enter, poi portate il cursore sulla riga STANDARD.ASM e premete nuovamente Enter. Aprirete così la SORGENTE STANDARD per scrivere un nuovo programma.

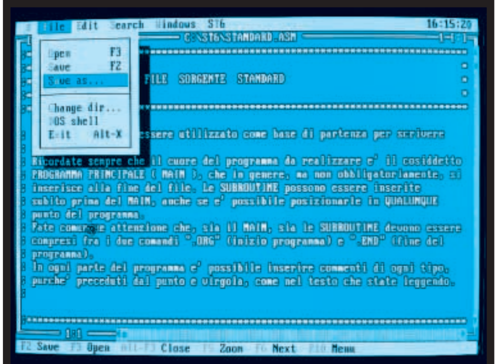


Fig.3 Per ricopiare la SORGENTE STANDARD dovete assegnarle un altro nome. Per fare questo dovete premere i tasti ALT+F poi premere il tasto A. Da questa figura si passerà alla fig.4.

me caricare il **programma ST6** nel vostro computer.

Per richiamare il programma dovete semplicemente digitare:

C:\ST6>ST6 poi Enter

e così compare sul monitor il menu principale.

A questo punto dovete **aprire** un **nuovo file**, nel quale scrivere il vostro programma.

Per aiutarvi fin da queste prime fasi, abbiamo inserito nel dischetto floppy il file **STANDARD.ASM**, che racchiude l'elenco delle istruzioni che devono necessariamente comparire in **ogni** programma.
Copiando il file **STANDARD** con un altro nome, avrete subito a disposizione la struttura base per scrivere il vostro programma.

Quindi per aprire un nuovo **file** dovete procedere come segue:

1° - Quando appare il menu principale premete il tasto funzione **F3** (vedi fig.1), poi premete Enter e portate il cursore sul nome **STANDARD.ASM** (vedi fig.2), quindi premete ancora Enter.

2° - Prima di qualsiasi altra cosa, dovete salvare questo file con un altro nome, quindi premete i tasti **Alt+F** e di seguito selezionate l'opzione **Save as** (che significa "salva con nome") premendo la lettera **A** (vedi fig.3).

3° - Sul monitor appare una finestra nella quale dovete digitare, oltre a **C:\ST6**, il nome del vostro **programma** (vedi fig.4).
Questo **nome**, che vi servirà quando vorrete trasferire il programma dall'Hard-Disk al microprocessore **ST6**, non deve mai superare gli **8 caratteri**.
Dopo il nome non dovete dimenticarvi di aggiungere l'estensione **.ASM**, che sta ad indicare che si tratta di un programma in **assembler**.
Cercate un nome che abbia una logica attinenza col programma che scriverete, per poterlo poi facilmente riconoscere tra gli altri. Per esempio potreste chiamare i programmi:

LED.ASM
LOTTO.ASM
OROLOGIO.ASM
TIMER.ASM ecc.

4° - Dopo aver scritto il nome per esteso (ad esempio, **C:\ST6\TIMER.ASM**) premete Enter, ed in

alto, nella pagina dell'editor visibile in fig.5, vedrete apparire la scritta:

C:\ST6\TIMER.ASM

che vi conferma che il file chiamato **TIMER.ASM** è stato creato.

COME ASSEMBLARE un PROGRAMMA

Importante: Quando avrete terminato di scrivere il programma, come più avanti vi spiegheremo, e l'avrete controllato apportando le modifiche necessarie, dovrete **assemblarlo**, altrimenti non potrete memorizzarlo nel microprocessore.
Per questo motivo, prima di chiudere l'editor, cioè il file del programma, premete i tasti **Alt+T** e di seguito il tasto **A** = assembla (vedi fig.6).
Se non avete commesso errori, dopo qualche secondo apparirà sul monitor la scritta **SUCCESS**.
In caso contrario, apparirà un messaggio che vi indicherà il tipo di errore commesso e la riga di istruzione in cui si trova.
Per correggere l'errore dovete tornare all'editor premendo un tasto qualsiasi.
Per trasferire il programma all'interno dell'**ST6**, seguite le istruzioni ampiamente descritte sulla rivista **N.172/173**.

COME si SCRIVE un'ISTRUZIONE

Quando scrivete un programma dovete rispettare alcune semplici regole che ora vi indicheremo, altrimenti quando l'**assemblerete** compariranno dei messaggi relativi agli **errori**, che dovrete **correggere** per poter proseguire.
Ogni istruzione deve essere scritta su una diversa riga di programma e deve essere composta da un'**etichetta**, da un'**istruzione** e da un **operando** dell'istruzione.
Ad esempio nella riga di programma:

pippo **ldi** **a,10h**

pippo è l'**etichetta**
ldi l'**istruzione**
a,10h è l'**operando** dell'istruzione

ETICHETTA

L'**etichetta** è un riferimento **non obbligatorio** che deve partire **sempre** dall'estremo sinistro della riga.
Un'etichetta serve come **punto di riferimento** per poter ritornare nuovamente, tramite l'istruzione di

Fig.4 Ammesso che vogliate chiamare il nuovo programma TIMER (ricordate che il nome non può mai superare gli 8 caratteri) scrivete per esteso C:\ST6\TIMER.ASM poi premete Enter.

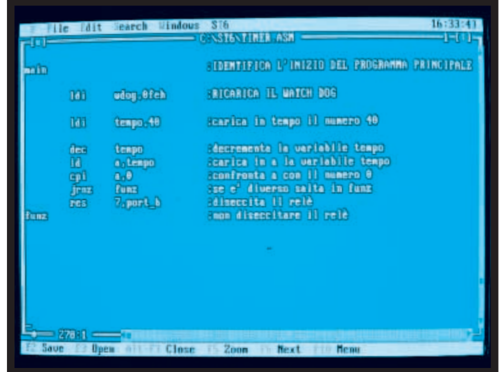


Fig.5 Una volta creato il file TIMER vedrete apparire nella prima riga in alto della finestra blu dell'editor C:\ST6\TIMER.ASM. A questo punto potete iniziare a scrivere il vostro nuovo programma.

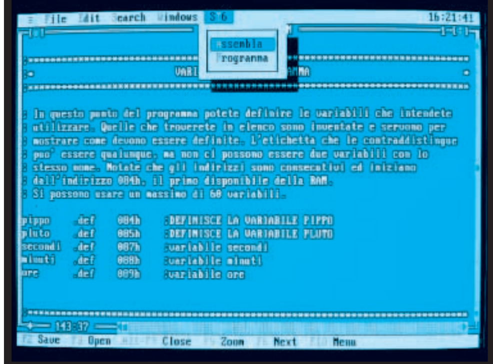


Fig.6 Una volta completato il programma prima di memorizzarlo nell'ST6 lo dovete Assemblare. Premete quindi i tasti ALT+T poi il tasto A. Se avete commesso un errore nel programma, vi verrà segnalato.

salto (indicata con la sigla **jp**, abbreviazione di **jump**), su quella riga di programma.
Quando scrivete un'**etichetta** dovete rispettare queste regole:

La parola non deve superare gli 8 caratteri

Potete scrivere **nota** - **PIPPO** - **RIF** ecc., ma non **parole** che superino gli **8** caratteri.

L'etichetta non deve iniziare con un numero

Non potete scrivere **1nota** - **2nota** - **1PIPPO** - **2PIPPO** ecc., ma potete posizionare un **numero** dopo la parola, senza interporre **spazi** tra parola e numero. E' quindi **corretto** scrivere **nota1** - **PIPPO1** ecc.

Non si può usare lo stesso nome due volte

E' scorretto definire più etichette con lo stesso nome, cioè scrivere **PIPPO - PIPPO**. Potete usare la stessa parola solo se inserite un numero progressivo, ad esempio **PIPPO1 - PIPPO2 - PIPPO3** ecc.

Non si può lasciare alcuno spazio a sinistra dell'etichetta

Se premete **spazio**, poi scrivete:

	PIPPO		

commettete un **errore**, quindi dovete partire da sinistra senza spazio:

PIPPO			

Carattere dell'etichetta

Anche se è possibile scrivere l'etichetta sia in **minuscolo** sia in **maiuscolo**, vi consigliamo di scrivere sempre in **minuscolo** così non vi sbaglierete mai.
Quindi scrivete **pippo1 - pippo2 - nota - rif** ecc.

ISTRUZIONE

Le istruzioni da inserire dopo l'etichetta sono proprie dell'**assembler** degli **ST6**.
Queste istruzioni devono sempre essere scritte dopo aver lasciato **uno spazio**.
Se è già presente il nome dell'**etichetta**, dovete comunque separare l'istruzione con **uno spazio**.
Quindi se avete l'etichetta **pippo1** e l'istruzione **ldi x,10**, dovete scrivere:

pippo1	ldi	x,10

Se nella riga di istruzione manca il nome dell'etichetta, dovete comunque lasciare **uno spazio** :

	ldi	x,10

Data l'importanza di scrivere correttamente l'**istruzione**, vi consigliamo di utilizzare la funzione **tabulazione** premendo il tasto **TAB** prima di scrivere qualsiasi **istruzione**.
In questo modo avrete tutte le istruzioni perfettamente incolonnate ed il programma risulterà più comprensibile quando dovrete rileggerlo.

OPERANDO

Nella riga riportata precedentemente, cioè:

	ldi	x,10	

ldi è l'**istruzione**, che significa **carica**
x,10 è l'**operando**

Questa riga indica: **carica** nella **cella di memoria X** il numero **10**.

L'**operando** deve sempre essere separato dall'**istruzione** tramite **uno spazio**, quindi se scrivete:

ldix,10

commette un grosso errore, mentre se scrivete:

ldi **x,10**

l'intera istruzione è corretta.

Dopo l'**istruzione** e l'**operando** potete inserire, se lo ritenete opportuno, un **commento**.

ETICHETTA	ISTRUZIONE	OPERANDO	;	COMMENTO RIGA

Fig.7 Dovrete sempre ricordare che ogni "riga" di programma è composta da quattro blocchi principali: Etichetta - Istruzione - Operando - Commento. I primi tre blocchi andranno tenuti separati tra loro da uno o più SPAZI (o ancora meglio usate il tasto TAB della tastiera), mentre l'ultimo blocco del COMMENTO dovrà essere separato da un punto e virgola. Se non userete l'ETICHETTA dovrete comunque lasciare uno o più spazi, mentre se non scriverete il COMMENTO non dovrete mettere il punto e virgola.

Questo deve essere sempre preceduto da un **punto e virgola** (;), diversamente il computer segnalerà **errore**.
Utilizzando l'istruzione precedente potete scrivere:

```
        ldi     x,10;   inserire 10 in x
```

I commenti possono essere scritti anche all'inizio di una riga, ma senza lasciare spazi e ricordando di mettere sempre prima un **punto e virgola** (;).

Ogni volta che completate un'istruzione dovete sempre e necessariamente andare a capo premendo il tasto **Enter**.
Tenete presente che le **istruzioni** possono essere scritte sia in **minuscolo** sia in **maiuscolo**:

ldi **x,10** oppure **LDI** **X,10**

COME scrivere i NUMERI

Nell'esempio sopra riportato noi abbiamo scritto **x,10**, in altre parole abbiamo utilizzato un numero **decimale**.
Tuttavia può risultare più **vantaggioso** in alcune istruzioni scrivere i numeri in base **esadecimale - ottale - binaria**.
Per far capire al **computer** che tipo di numero avete inserito, dovete scrivere una **lettera** come qui sotto specificato:

o oppure **O** se il numero è **ottale**
h oppure **H** se il numero è **esadecimale**
b oppure **B** se il numero è **binario**

Ad esempio:

10o = numero **ottale**
01Ah = numero **esadecimale**
00100101b = numero **binario**

Se dopo il numero **non mettete** nessuna lettera, il computer considererà questo numero **decimale**.
Quando scrivete un numero **esadecimale**, dovete sempre mettere **davanti** ad ogni numero uno **0** (**zero**), quindi **01A - 0ED - 0AC** ecc., ed alla fine deve seguire la lettera **H**, per indicare che il numero è **esadecimale**, quindi i numeri sopra riportati vanno scritti 01A**H** - 0ED**H** - 0AC**H**.

I **numeri decimali**
iniziano da **0** e terminano a **255**

I **numeri ottali**
iniziano da **0** e terminano a **377**

I **numeri esadecimali**
iniziano da **0** e terminano a **FF**

I **numero binari**
iniziano da **0** e terminano a **11111111**

STRUTTURA di un PROGRAMMA

Per scrivere un programma per **ST6** si devono seguire delle precise regole che sono:

Definire lo spazio in MEMORIA
Definire le VARIABILI
Definire i REGISTRI
Scrivere il PROGRAMMA PRINCIPALE
Scrivere le SUBROUTINE
Scrivere eventuali subroutine di INTERRUPT
Definire i VETTORI di INTERRUPT

Per facilitarvi, abbiamo inserito nel **dischetto floppy**, che avete ricevuto assieme al **programmatore LX.1170** (vedi rivista N.172/173), un **file** chiamato **STANDARD.ASM** che vi spiega come impostare il programma, dove scrivere le varie istruzioni, come definire lo spazio di memoria ed i registri, dove posizionare le subroutine, come inizializzare l'**ST6**, insomma tutti i consigli e le informazioni necessarie per non sbagliare.
Come abbiamo già descritto nel paragrafo "Come creare un file sorgente", tutte le volte che dovrete scrivere un **nuovo** programma **copiate** il file **STANDARD.ASM** con il **nome** del programma che volete scrivere e tutto risulterà più facile.

La MEMORIA dell'ST6

All'interno dei microprocessori tipo **ST62E10 - ST62T10 - ST62E15 - ST62T15** risultano disponibili per il programma **2K** di **memoria ROM**, mentre nei microprocessori **ST62E20 - ST62T20 - ST62E25 - ST62T25** sono disponibili **4K** di **memoria ROM**.
All'interno di ciascun microprocessore sono presenti anche **64 byte** di **memoria RAM** che servono per i **registri** e le **variabili**.
La memoria **ROM** mantiene tutte le informazioni, cioè il programma, inserite durante la programmazione del microprocessore anche in assenza di alimentazione.
La memoria **RAM** viene usata per le variabili, cioè per i dati che devono essere di volta in volta letti, scritti e modificati, e quindi può essere "aggiornata" dallo stesso microprocessore durante il funzionamento del programma.

La **memoria** (sia ROM sia RAM) può essere considerata come un insieme di piccole **celle** ed all'interno di ognuna può essere inserito un solo **dato**.
Per portarvi un esempio pratico, potete paragonare queste **celle** a quelle presenti in un **favo** per **api**. Quando le api hanno riempito una cella, passano a riempire la seconda, poi la terza ecc. fino a riempire tutto il **favo**.
In un microprocessore da **2K** ci sono esattamente **1.828 celle** in grado di contenere un programma composto da circa **900 - 990 righe** di **programma**.
In un microprocessore da **4K** ci sono esattamente **3.872 celle** in grado di contenere un programma composto da circa **1.800 - 2.000 righe** di **programma**.

DEFINIZIONE delle VARIABILI

Innanzitutto la **variabile** è un numero, **decimale - ottale - binario** oppure **esadecimale**, che il microprocessore può modificare tramite una particolare **istruzione**.
Le **variabili** non vengono inserite nella **memoria ROM**, ma sempre e solo nella **memoria RAM**; in questo modo è possibile variare questi numeri secondo le diverse esigenze.
Ad esempio, per realizzare un **orologio** servono **3 variabili**, una per le **ore**, una per i **minuti** ed una per i **secondi**, che tramite opportune e precise istruzioni, si possono incrementare per ottenere le seguenti funzioni.
La **prima variabile** dei **secondi** viene aumentata dal programma di **1** per ogni **secondo** trascorso.
Raggiunto il numero **60**, il programma aumenta di **1** la **seconda variabile** dei **minuti** e porta a **00** la **prima variabile** dei **secondi**.
Quando la **variabile** dei **minuti** raggiunge il numero **60**, il programma aumenta di **1** la **terza variabile** delle **ore** e porta a **00** le variabili dei **minuti** e dei **secondi**.
Quando la **variabile** delle **ore** raggiunge il numero **24**, il programma porta a **00** le tre variabili **ore - minuti - secondi**.

Quando inserite una variabile dovete ricordare che se dopo il numero **non mettete** nessuna lettera, il computer lo considera un numero **decimale**.
Per informare il computer che avete inserito un numero con base diversa da 10, seguite le istruzioni spiegate nel paragrafo "Come scrivere i numeri".

In ogni programma potete inserire un **massimo** di **60 variabili** e poiché queste sono situate nelle celle della **memoria RAM**, dovete dare ad ogni **cella** un numero di **riferimento**, così da poter ritrovare la variabile.

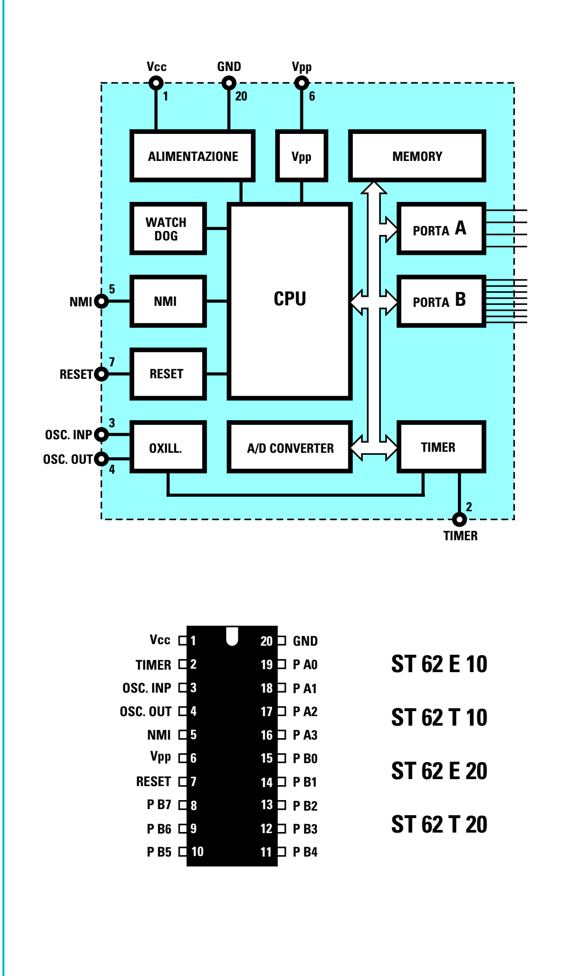
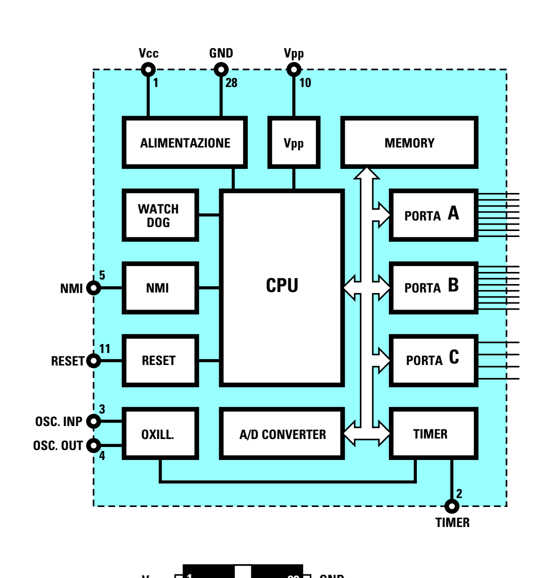


Fig.8 Nei microprocessori ST6 tipo E10-T10 ed E20-T20 sono disponibili due sole porte, la A e la B. La porta A è siglata A0-A1-A2-A3 mentre la porta B è siglata B0-B1-B2-B3-B4-B5-B6-B7. Tutti questi piedini possono essere utilizzati sia come Entrate sia come Uscite. Nei prossimi numeri vi spiegheremo come predisporli.

Questo numero di riferimento si chiama **indirizzo** e poiché ogni variabile occupa una sola **cella** di memoria, queste hanno un numero progressivo.
Se usate i numeri in base **dieci**, la prima cella porta il numero **132** e l'ultima il numero **191**.
Se usate i numeri in base **sedici**, la prima cella porta il numero **084H** e l'ultima il numero **0BFH**.

Ricordate che il **nome** che assegnate alla **variabile** deve sempre essere scritto partendo da **sinistra**, senza lasciare **nessuno spazio**, ed a questo nome deve seguire, spaziandola, la dicitura **.def**. Dopo questa abbreviazione, dovete lasciare un altro



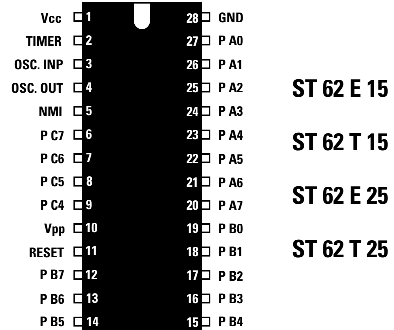


Fig.9 Nei microprocessori ST6 tipo E15-T15 ed E25-T25 sono disponibili tre porte, la A, la B e la C. La porta A è siglata A0-A1-A2-A3-A4-A5-A6-A7, la porta B è siglata B0-B1-B2-B3-B4-B5-B6-B7, mentre la porta C è siglata C4-C5-C6-C7. Anche tutti questi piedini possono essere utilizzati sia come Entrate sia come Uscite.

spazio e poi scrivere il **numero**, che rappresenta l'**indirizzo** della cella di memoria in cui volete allocare questo dato.

Ricordatevi inoltre che se assegnate la stessa cella di memoria a due **diverse variabili**, il microprocessore **non segnalerà** nessun **errore**, ma in questo caso il programma funzionerà in modo **anomalo** e voi non otterrete le funzioni che vi eravate prefissati.

Per il programma che serve a far funzionare un orologio, potete definire nel seguente modo le **3 variabili**:

```
secondi  .def 132
minuti   .def 133
ore      .def 134
```

Potete definire queste **variabili** anche con un numero **esadecimale**, senza che ciò modifichi il funzionamento dell'orologio:

```
secondi  .def 084H
minuti   .def 085H
ore      .def 086H
```

Come ultima indicazione, tenete presente che le **variabili** vanno definite fin dal principio, vanno cioè inserite all'**inizio** del programma e non a metà o alla fine.
Vi consigliamo di scrivere il nome delle **variabili** sempre in **minuscolo**.

I REGISTRI del MICROPROCESSORE

Nella **memoria RAM** del microprocessore **ST6**, oltre allo spazio riservato alle **variabili** in precedenza descritte, sono presenti delle altre **celle di memoria** chiamate **registri**, che permettono di eseguire precise funzioni già definite.
Ad esempio, c'è un **registro** che permette di definire quali piedini delle porte **A - B - C** vanno utilizzati come **ingressi** e quali come **uscite** (vedi figg. 8-9).
Questi **indirizzi** devono essere definiti sempre all'inizio di ogni programma. Questa è un'operazione che **non dovete** eseguire se utilizzate, come abbiamo spiegato, il file **STANDARD.ASM**, perché abbiamo già **definito** noi tutti i registri, così da evitarvi errori.

IL REGISTRO ACCUMULATORE

Nel microprocessore **ST6** c'è un particolare **registro** chiamato **accumulatore** ed indicato sempre con la lettera **a**, molto importante, perché esegue le seguenti **operazioni matematiche**:

fa la somma
fa la sottrazione
fa la tavola della verità di un AND
fa una comparazione tra due numeri
fa il complemento di un numero

Tutte queste operazioni **matematiche** si possono eseguire **solo** con il numero che avete provveduto ad inserire nell'**accumulatore;** il **risultato** ottenuto subentra poi automaticamente a sostituire il numero prima presente nel **registro accumulatore**.

Ritorniamo all'esempio dell'**orologio** e supponiamo che siano le **10:25:30**.
Poiché la funzione dell'orologio presuppone che si debba sempre **sommare 1**, tramite l'istruzione **ld** (load) spostiamo nell'accumulatore **a** il numero presente nella **variabile** dei **secondi**, cioè **30**. A questo punto possiamo sommare a questo il numero **1,** ottenendo **31**.
Ora sempre con l'istruzione **ld**, spostiamo nuovamente il risultato dal registro accumulatore alla **variabile** dei **secondi**, che di conseguenza risulta ora **31**.
Trascorso un secondo, si ripete il ciclo di istruzioni: spostiamo il numero dalla **variabile** dei **secondi** nel registro **accumulatore**, **sommiamo** a questo **1** ed il nuovo risultato, **32**, lo spostiamo nuovamente nella **variabile** dei **secondi.**

L'intero ciclo appena descritto si riduce a tre istruzioni:

	ld	**a,secondi**	**;** carica i secondi in **a**
	addi	**a,1**	**;** somma ad **a** il numero **1**
	ld	**secondi,a**	**;** carica somma nella **variabile**

REGISTRI SPECIALIZZATI

All'interno del microprocessore **ST6** ci sono dei registri **specializzati** che noi abbiamo definito nel nostro file **STANDARD.ASM** con le lettere **x - y - v - w** e che vi potrebbero servire per semplificare particolari operazioni.
Ad esempio se voleste ottenere un **impulso** della durata di **1 millisecondo** potreste eseguire queste istruzioni:

	set	**0,port_b**	**;** metti a **1** il piedino **PB0**
	ldi	**x,103**	**;** assegna **103** a **x**
ripeti	**dec**	**x**	**;** sottrai **1** a **x**
	jrnz	**ripeti**	**;** ripeti se **x** non è a **0**
	res	**0,port_b**	**;** metti a **0** il piedino **PB0**

Questa sequenza di istruzioni fa sì che, iniziando dal numero **103** e continuando a sottrargli **1** fino a quando non si è raggiunto lo **0**, passi esattamente **1 millisecondo**.

Il SET di ISTRUZIONI

Le istruzioni del linguaggio **assembler** usate dal microprocessore **ST6** sono molto semplici e possono essere così suddivise:

1° CARICAMENTO DATI

LD - spostamento di **dati** tra due registri
LDI - caricamento di un **numero** in un registro

2° ARITMETICHE

ADD - somma tra **variabile** e **accumulatore**
ADDI - somma tra **numero** e **accumulatore**
AND - funzione **And** tra **variabile** e **accumulatore**
ANDI - funzione **And** tra **numero** e **accumulatore**
CLR - azzera una **variabile**
COM - complementa i **bit** nell'**accumulatore**
CP - comparazione tra **variabile** e **accumulatore**
CPI - comparazione tra **numero** e **accumulatore**
DEC - sottrae **1** ad una **variabile**
INC - somma **1** ad una **variabile**
RLC - sposta una **variabile** a sinistra con **riporto**
SLA - sposta una **variabile** a sinistra senza **riporto**
SUB - sottrazione tra **variabile** e **accumulatore**
SUBI - sottrazione tra **accumulatore** e **numero**

3° SALTI CONDIZIONATI sull'ETICHETTA

JRC - salta se c'è un **riporto**
JRNC - salta se **non** c'è un **riporto**
JRZ - salta se l'operazione dà **0**
JRNZ - salta se l'operazione **non** dà **0**
JRR - salta se un **bit** è **0**
JRS - salta se un **bit** è **1**

4° SALTI INCONDIZIONATI sull'ETICHETTA

CALL - esegui una **subroutine**
JP - salta sempre sull'**etichetta**

5° SETTAGGIO dei BIT

SET - metti un **bit** a **1**
RES - metti un **bit** a **0**

6° CONTROLLO CPU

NOP - serve per ottenere dei **ritardi**
RET - ritorna da una **subroutine**
RETI - ritorna da un **interrupt**
STOP - blocca tutte le funzioni del **micro**
WAIT - arresta l'esecuzione del **programma**

TEMPI di ESECUZIONE

I tempi per eseguire un'istruzione si calcolano a **cicli macchina** e vanno da un **minimo** di **2 cicli** ad un **massimo** di **5 cicli macchina.**
Tutte le istruzioni di **Caricamenti Dati - Funzioni Aritmetiche - Salti Incondizionati - Settaggio Bit** impiegano **4 cicli macchina**.
Tutte le istruzioni **Controllo CPU** e le istruzioni **JRC - JRNC - JRZ - JRNZ** dei **Salti Condizionati** impiegano **2 cicli macchina.**
Tutte le istruzioni **JRR - JRS** dei **Salti Condizio-**

nati impiegano **5 cicli macchina**.
Il **tempo** di un **ciclo macchina** dipende dalla frequenza del **quarzo** utilizzato per il **clock**.
Per calcolare il **tempo** di un'istruzione potete usare questa formula:

microsecondi = (13 : MHz quarzo) x N cicli

Ad esempio, se usate un quarzo da **2 MHz** per eseguire un'istruzione **aritmetica** che necessita di **4 cicli**, il microprocessore per svolgerla impiegherà:

(13 : 2) x 4 = 26 microsecondi

Se usate un quarzo da **8 MHz**, la stessa istruzione sarà eseguita in un tempo di:

(13 : 8) x 4 = 6,5 microsecondi

Per il **clock** potete usare dei quarzi di qualsiasi frequenza, da **2,45 - 3,4 - 4,7 - 6,5 - 7 - 8 MHz**. Normalmente si utilizza la frequenza massima di **8 MHz** per rendere più veloce l'esecuzione di un programma.
Non usate mai quarzi **superiori** agli **8 MHz**, perché questa è la frequenza **massima** accettata dall'oscillatore interno del microprocessore **ST6**.

COME usare le varie ISTRUZIONI

Le istruzioni vanno scritte secondo precisi criteri, ed è quindi abbastanza facile che un principiante incontri qualche difficoltà nell'impostarle.
Ripetiamo nuovamente che se non mettete il nome di un'**etichetta** dovete sempre lasciare **uno spazio** prima di scrivere l'istruzione o, ancora meglio, premete il tasto **TAB**, così da avere tutte le istruzioni incolonnate.
Per ognuna delle istruzioni utilizzate dal linguaggio di programmazione Assembler, diamo di seguito una semplice spiegazione correlata da un esempio.

ADD

Per eseguire una **somma** tra il numero presente nella **variabile** ed il numero presente nell'**accumulatore**, dovete scrivere l'istruzione in questo modo:

```
        add     a,secondi
```

Per questa istruzione abbiamo usato come **variabile** il nome **secondi**, ma potevamo utilizzare un nome diverso, a patto che fosse stato sempre di **8 caratteri**, come ad esempio **gradi - metri - litri** ecc.
Se nell'**accumulatore** è presente il numero **22** e nella **variabile secondi** è presente il numero **15**, dopo questa istruzione nell'accumulatore **a** è presente il numero **37**, perché **22 + 15 = 37**.

ADDI

Questa istruzione è identica alla precedente con la sola differenza che il **numero** da **sommare** a quello presente nel registro **accumulatore** non è preso dalla variabile, ma immesso direttamente da voi.
Ad esempio, se al numero presente nell'**accumulatore**, che potrebbe essere **37**, volete sommare il **numero 30**, dovrete scrivere:

```
        addi    a,30
```

Dopo questa istruzione nell'accumulatore **a** è presente il numero **30 + 37 = 67**.

AND

Questa istruzione permette di eseguire un'operazione **AND** tra il numero contenuto nell'**accumulatore** e quello nella **variabile**.
Per farvi comprendere meglio come viene effettuata questa operazione, riportiamo la **tavola della verità** con i numeri **binari**.

Tavola della verità

accumulatore	**0 0 1 1**
variabile	**0 1 0 1**
risultato	**0 0 0 1**

Secondo questa tavola, quando è presente un **valore logico 1** sia nell'**accumulatore** sia nella **variabile**, si ha come risultato **1**; in ogni altra condizione si ha sempre come risultato **0**.
L'istruzione va scritta così:

```
        and     a,secondi
```

Se nella **variabile secondi** è presente il numero decimale **30**, che convertito in **binario** è uguale a **00011110**, e nell'**accumulatore** è presente il numero decimale **25**, che convertito in **binario** è uguale a **00011001**, il risultato dell'operazione **AND** tra questi numeri è:

00011110
00011001

00011000 risultato

che corrisponde al numero **decimale 24**.

ANDI

A differenza della precedente, questa istruzione esegue un'operazione **AND** tra il numero contenuto nell'**accumulatore** ed un **numero binario** scritto direttamente da voi sulla stessa riga dell'istruzione.
Quando nell'**accumulatore** e nel **numero** inserito è presente un valore **logico 1** si ha come risultato **1**; in ogni altra condizione si ha sempre come risultato **0**.
Ammesso di avere nell'**accumulatore** il numero **binario 00011110** e di voler eseguire l'operazione **AND** con il **numero binario 11111001**, l'istruzione va scritta:

```
        andi    a,11111001B
```

Come avrete notato, alla fine di questo numero abbiamo messo una **B** affinché il **computer** possa riconoscere che il numero è **binario**.
Il risultato di questa operazione è:

```
  00011110
  11111001
----------
  00011000  risultato
```

Le due funzioni **AND** e **ANDI** possono essere utilizzate per modificare il livello logico sugli **otto** piedini di uscita di una **porta** del **microprocessore**.
Sapendo su quali di questi piedini è presente un **livello logico 0** e su quali è presente un **livello logico 1**, è possibile accendere ad esempio dei diodi led.
Nel nostro esempio, poiché il risultato è **00011000**, saranno **spenti** i primi tre diodi led, **accesi** i successivi due e **spenti** gli ultimi tre.
Per portare il **risultato** dell'istruzione **ANDI** sulla **porta** di uscita **B**, bisogna scrivere questa istruzione:

```
        ld      port_b,a
```

che in pratica significa: **carica** (**ld**) sulla **porta B** il risultato contenuto nell'**accumulatore a**.

CALL

Questa istruzione viene adoperata quando si vuole far eseguire al microprocessore una **subroutine**, cioè un parte di programma identificata con un'**etichetta**.
Le **subroutine** sono utili per eseguire **più volte** lo stesso set di **istruzioni**.

Ad esempio potrebbe verificarsi di dover ripetere più volte la seguente funzione di **ritardo**.

```
ritardo   ldi    x,103    ; assegna 103 a x
ripeti    dec    x        ; sottrai 1 a x
          jrnz   ripeti   ; ripeti se x non è a 0
          ret             ; ritorna al programma
```

Tutte le volte che vorrete ripetere queste istruzioni in una parte del programma, basterà scrivere:

```
          call   ritardo  ; chiama subroutine ritardo
```

Come potete notare con due sole parole, **call ritardo**, farete ripetere esattamente le stesse istruzioni contrassegnate dall'etichetta **ritardo**, senza bisogno di riscriverle.
In questo modo non solo eviterete di occupare altra memoria nel microprocessore, ma soprattutto non correrete il rischio di compiere qualche **errore** nel riscrivere le istruzioni.
Eseguita la subroutine **ritardo**, il microprocessore proseguirà con le **istruzioni** successive alla riga **call ritardo**, perché alla fine della **subroutine** è presente l'istruzione **ret**, che significa: ritorna al programma nella riga successiva alla quale era scritto **call ritardo**.

CLR

Questa istruzione serve per portare a **0** una **variabile**.
Per spiegarci meglio riconsideriamo il programma per realizzare un orologio.
Per ottenere la funzione **orologio** è necessario che il numero delle **ore** riparta da **zero** quando si è raggiunto il numero **24**; allo stesso modo quando i **minuti** ed i **secondi** hanno raggiunto il numero **60**, devono ripartire a contare da **zero**.
Quando volete che le **variabili** chiamate **ore - minuti - secondi** diventino **0** dovete scrivere:

```
          clr    ore
          clr    minuti
          clr    secondi
```

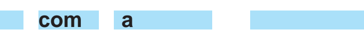
La funzione **CLR** può essere usata anche per azzerare il registro **accumulatore** scrivendo semplicemente:

```
        clr     a
```

In questo modo **cancellerete** eventuali numeri rimasti nell'**accumulatore** da un'operazione precedente.

COM

Questa funzione serve per **complementare** il **numero binario** presente nell'**accumulatore**.
In altre parole, questa istruzione **inverte** ogni singolo **bit**, quindi dove c'è **0** si ha **1** e dove c'è **1** si ha **0**.
Se nell'**accumulatore** è presente il numero **binario 00011000** scrivendo:



```
        com     a
```

il contenuto dell'**accumulatore** diventerà **11100111**.

CP

Questa istruzione **confronta** il numero contenuto nell'**accumulatore** con quello presente nella **variabile**.
Da questo confronto il microprocessore ricava queste tre sole **condizioni**:

- il numero dell'**accumulatore** è **minore** rispetto a quello della **variabile**.

- il numero dell'**accumulatore** è **uguale** a quello della **variabile**.

- il numero dell'**accumulatore** è **maggiore** rispetto a quello della **variabile**.

Questo confronto è utile quando occorre far compiere dei **salti condizionati** al programma, per eseguire le operazioni che desiderate.

Prendiamo ancora una volta l'esempio dell'orologio.
Quando il numero nella variabile **secondi** giunge a **60**, bisogna ripartire da **0** ed **aumentare** di **1** il numero nella variabile **minuti**.
Quando il numero nella variabile **minuti** giunge a **60** , bisogna ripartire da **0** ed **aumentare** di **1** il numero nella variabile **ore**.
Quando il numero nella variabile **ore** giunge a **24**, bisogna riportare a **0** le variabili **ore - minuti - secondi**.
Il programma per eseguire queste funzioni va scritto nel seguente modo:

```
        inc     secondi     ; incrementa di 1 i secondi
        ldi     a,60        ; carica in a il numero 60
        cp      a,secondi   ; confronta a con variabile sec.
        jrnz    fine        ; salto condizionato a fine
        clr     secondi     ; azzera i secondi
        inc     minuti      ; incrementa di 1 i minuti
        ldi     a,60        ; carica in a il numero 60
        cp      a,minuti    ; confronta a con variabile min.
        jrnz    fine        ; salto condizionato a fine
        clr     minuti      ; azzera i minuti
        inc     ore         ; incrementa di 1 le ore
        ldi     a,24        ; carica in a il numero 24
        cp      a,ore       ; confronta a con variabile ore
        jrnz    fine        ; salto condizionato a fine
        clr     ore         ; azzera le ore
fine                        ; fine dell'incremento
```

Quando il numero dei **secondi** è **diverso** da **60**, numero caricato nell'**accumulatore**, si fa fare al programma un **salto condizionato**, cioè si **salta** alla riga di programma con l'**etichetta** chiamata **fine**.
Quando il numero dei **secondi** è **uguale** a **60** questo **salto** non avviene, quindi il programma passa alla riga successiva **aumentando** di **1** la **variabile** dei **minuti** ed **azzerando** quella dei **secondi**.
Fino a quando la **variabile** dei **minuti** non avrà raggiunto il numero **60**, si fa fare un **salto condizionato** sull'etichetta **fine**.
Quando il numero dei **minuti** è **uguale** a **60** questo **salto** non avviene, quindi il programma passa alla riga successiva **aumentando** di **1** la **variabile** delle **ore** ed **azzerando** quella dei **minuti**.
Il programma **confronta** il numero presente in questa **variabile** con quello presente nell'**accumulatore**, che è **24**, e quando nella **variabile** è presente il numero **24**, il programma passa alla riga successiva, vedi **clr ore**, per azzerare la variabile **ore**.
Se il microprocessore ripete questo programma **ogni secondo**, compiendo nel frattempo altre istruzioni di programma, avrete ottenuto la funzione orologio.

CPI

Questa istruzione si differenzia dalla precedente perché **confronta** il numero contenuto nell'**accumulatore** con un **numero** direttamente scritto da voi.
Da questo confronto il microprocessore ricava sempre queste tre sole **condizioni**:

- il numero dell'**accumulatore** è **minore** rispetto a quello della **variabile**.

- il numero dell'**accumulatore** è **uguale** a quello della **variabile**.

- il numero dell'**accumulatore** è **maggiore** rispetto a quello della **variabile**.

Questo confronto è utile quando occorre far compiere dei **salti condizionati** al programma, per eseguire le operazioni che desiderate.

Ad esempio, se volete realizzare un **termostato** che **disecciti** un **relè** quando la temperatura ha raggiunto i **20 gradi**, un modo per scrivere le istruzioni potrebbe risultare il seguente:

	ld	**a,gradi**	; carica in **a** i gradi
	cpi	**a,20**	; compara i **gradi** con numero **20**
	jrc	**funz**	; salta a **funz** se **minore** di **20**
	res	**7,port_b**	; diseccita il relè
funz			**; non** diseccitare il **relè**

La **temperatura** prelevata da una sonda viene messa nell'**accumulatore** per essere poi **comparata** con il numero **20.**
Se la temperatura è **minore** di **20**, il programma salta all'**etichetta** siglata **funz** e quindi il relè non si diseccita.
Quando la **temperatura** ha raggiunto i **20 gradi**, il microprocessore passa ad eseguire l'istruzione presente nella **quarta** riga, cioè si **resetta,** portando a **livello logico 0** il **piedino 7** della porta **B.** In questo modo il relè collegato su questo piedino si **diseccita**.

DEC

Questa istruzione serve per **decrementare di 1** il numero presente nella **variabile** specificata di seguito.
Ad esempio, se volete che trascorsi **40 minuti** si disecciti un **relè**, dovete scrivere il programma nel seguente modo:

	ldi	**tempo,40**	; carica in **tempo** il numero **40**
	.....	**.......**	;
	dec	**tempo**	; decrementa la **variabile** tempo
	ld	**a,tempo**	; carica in **a** la **variabile** tempo
	cpi	**a,0**	; confronta **a** con il numero **0**
	jrnz	**funz**	; se è diverso da **0** salta in **funz**
	res	**7,port_b**	; diseccita relè
funz			**; non** diseccitare il **relè**

Dopo avere caricato il numero **40** nella **variabile tempo**, dovete **completare** il programma (spazio indicato con **puntini**) per far eseguire al programma un decremento ogni **60 secondi.**

INC

Questa istruzione serve per **aumentare di 1** il numero presente nella **variabile** specificata di seguito.
Ritornando all'esempio scritto per l'istruzione **CP**, la prima riga conteneva l'istruzione:

	inc	**secondi**	; incrementa la **variabile secondi**

quindi il numero presente nella **variabile secondi** viene **aumentato** di **1.**

NOTA importante per le istruzioni DEC e INC

Quando utilizzate queste istruzioni dovete tenere presente quanto segue:
- un ulteriore **decremento** (istruzione **DEC**) quando la **variabile** è arrivata al numero **decimale 0** porta il valore della **variabile** a **255.**
- un ulteriore **incremento** (istruzione **INC**) quando la **variabile** è arrivata al numero **decimale 255** porta il valore della **variabile** a **0.**

JP

Questa istruzione consente di effettuare un **salto incondizionato** in un punto qualsiasi del programma marcato da un'**etichetta.**
Ad esempio, se dovete far **lampeggiare** un **diodo led** con una cadenza di **1 secondo**, dovete scrivere:

inizio			**; etichetta**
	com	**a**	; se acceso spegni o viceversa
	ld	**port_b,a**	; sposta su **b** quello che c'è in **a**
	call	**ritardo**	; chiama funzione **ritardo**
	jp	**inizio**	; ripeti funzione dall'inizio

Vi ricordiamo che **ritardo** è un'**etichetta** (vedi istruzione **CALL**).

JRC

Questa istruzione viene sempre inserita nei programmi **dopo** un'istruzione **CP** o **CPI** per effettuare un salto **condizionato.**
Se dalla **comparazione** il microprocessore rileva che il numero presente nell'**accumulatore** è **minore** di quello presente nella **variabile**, viene effettuato un **salto** sull'**etichetta.**
Nella funzione **CPI**, in cui vi abbiamo presentato un esempio di programma per **termostato**, avete trovato utilizzata l'istruzione **JRC**:

	ld	**a,gradi**	; carica nell'**accumul.** i gradi
	cpi	**a,20**	; compara i gradi con numero **20**
	jrc	**funz**	; salta a **funz** se minore di **20**
	res	**7,port_b**	; diseccita il relè
funz			**; non** diseccitare il **relè**

In questo caso il **salto** viene effettuato sull'**etichetta** siglata **funz** fino a quando la temperatura non raggiunge i **20 gradi.**

Nota importante: Il salto **jrc** riesce a raggiungere un'**etichetta** solo se questa si trova ad una distanza pari a circa **8 righe** di **programma.**
Se eseguite un salto **jrc** su un'**etichetta** che dista più di **8 righe**, quando **assemblerete** il programma, vi verrà segnalato **errore** con la scritta "**5-bit displacement overflow**".

JRNC

Questa istruzione viene sempre inserita **dopo** un'istruzione **CP** o **CPI** per effettuare un salto **condizionato.**
Se dalla **comparazione** il microprocessore rileva che il numero presente nell'**accumulatore** è **maggiore** oppure **uguale** a quello presente nella **variabile**, viene effettuato un **salto** sull'**etichetta.**
Nella funzione **CPI** il relè si **diseccitava** quando la temperatura **superava** i **20 gradi**; se ora volete che il relè si **disecciti** quando la temperatura **scende** sotto i **20 gradi**, dovete modificare nella **terza** riga l'istruzione **jrc funz** con la scritta **jrnc funz**, come qui sotto riportato:

	ld	**a,gradi**	; carica nell'**accumul.** i gradi
	cpi	**a,20**	; compara i gradi con numero **20**
	jrnc	**funz**	; salta a **funz** se **maggiore** di **20**
	res	**7,port_b**	; diseccita il relè
funz			; **non** diseccitare il **relè**

In questo caso il **salto** viene effettuato sull'**etichetta** siglata **funz** solo se la temperatura è **maggiore** o **uguale** a **20 gradi.**
In pratica si ottiene la funzione **opposta** quella che si otteneva con l'istruzione **jrc.**

Nota importante: Il salto **jrnc** riesce a raggiungere un'**etichetta** solo se questa si trova ad una distanza pari a circa **8 righe** di **programma.**
Se eseguite un salto **jrnc** su un'**etichetta** che dista più di **15 righe**, quando **assemblerete** il programma, vi verrà segnalato **errore** con la scritta "**5-bit displacement overflow**".

JRNZ

Questa istruzione viene sempre inserita nel programma **dopo** un'istruzione **CP** o **CPI** per effettuare un salto **condizionato.**
Se dalla **comparazione** il microprocessore rileva che il numero presente nell'**accumulatore** è **diverso** da quello presente nella **variabile**, viene effettuato un **salto** sull'**etichetta.**

Nell'**istruzione DEC** avevamo riportato un esempio per far **diseccitare** un **relè** dopo **40 minuti.**

	ldi	**tempo, 40**	; carica in **tempo** il numero **40**
	......	....	
	dec	**tempo**	; decrementa la **variabile** tempo
	ld	**a, tempo**	; carica in **a** la **variabile** tempo
	cpi	**a, 0**	; confronta **a** con il numero **0**
	jrnz	**funz**	; se è **diverso** da **0** salta in **funz**
	res	**7, port_b**	; diseccita relè
funz			; **non** diseccitare il **relè**

Poiché nella **variabile** abbiamo messo il numero **40** ed il microprocessore **decrementa** questo numero di **1**, avremo via via **39 - 38 - 37 ecc.**
Dopo ogni decremento il numero presente nella variabile viene **comparato** con il numero **0** e fino a quando il numero nella variabile è diverso da **0,** viene effettuato il **salto** nella riga **funz** ed il programma non esegue la successiva istruzione che **diseccita** il relè.
Solo quando il numero presente nella **variabile** è **uguale** a **0**, il microprocessore passa ad eseguire l'istruzione successiva e **diseccita** il relè.

Nota importante: Il salto **jrnz** riesce a raggiungere un'**etichetta** solo se questa si trova ad una distanza pari a circa **8 righe** di **programma.**
Se eseguite un salto **jrnz** su un'**etichetta** che dista più di **8 righe**, quando **assemblerete** il programma, vi verrà segnalato **errore** con la scritta "**5-bit displacement overflow**".

JRZ

Questa istruzione viene sempre inserita in un programma **dopo** un'istruzione **CP** o **CPI** per effettuare un salto **condizionato.**
Se dalla **comparazione** il microprocessore rileva che il numero presente nell'**accumulatore** è **uguale** a quello presente nella **variabile**, viene effettuato un **salto** sull'**etichetta.**

L'esempio riportato nell'**istruzione DEC** permetteva di **diseccitare** un **relè** dopo **40 minuti.**
Di seguito potete vedere le modifiche che abbiamo apportato al programma per usare l'istruzione **JRZ.**

	ldi	**tempo,0**	; carica in **tempo** il numero **0**
	.....	.....	
	inc	**tempo**	; incrementa la **variabile** tempo
	ld	**a,tempo**	; carica in **a** la **variabile** tempo
	cpi	**a,40**	; confronta **a** con il numero **40**
	jrz	**funz**	; se è **uguale** salta in **funz**
	jp	**fine**	; se **non** è uguale salta in **fine**
funz			; prosegui alla riga dopo
	res	**7,port_b**	; diseccita il relè
fine			; **non** diseccita il **relè**

In questo caso abbiamo messo nella **variabile** il numero **0**, poi avendo dato l'istruzione per **incre-**

mentare questo numero di **1**, avremo via via **0 - 1 - 2 - 3 ecc.**
Tutte le volte che viene effettuato un **incremento**, il numero presente nella **variabile tempo** viene **comparato** con il numero **40** e fino a quando questi due numeri sono **diversi** viene effettuato il **salto** sull'etichetta **fine** ed il relè non si diseccita.
Solo quando il numero presente nella variabile è uguale a **40**, il microprocessore compie un **salto** sull'etichetta **funz** ed esegue l'istruzione successiva **diseccitando** il relè.

Nota importante: Il salto **jrz** riesce a raggiungere un'**etichetta** solo se questa si trova ad una distanza pari a circa **8 righe** di **programma**.
Se effettuate un salto **jrz** su un'**etichetta** che dista più di **8 righe**, quando **assemblerete** il programma, vi verrà segnalato **errore** con la scritta "**5-bit displacement overflow**".

JRR

Questa istruzione serve per controllare se una **cifra** di un **numero binario** si trova a **livello logico 0** e quando si rileva questa condizione viene effettuato un **salto**.
L'istruzione **JRR** può risultare utile per controllare se il **piedino d'ingresso** di una qualsiasi porta **A-B-C** si trova a **livello logico 0** o a **livello logico 1** (vedi figg.10-11).
Come sapete ogni porta da **8 bit** è numerata **0A - 1A- 2A - 3A** ecc. **0B - 1B - 2B - 3B** ecc.
Per **controllare** quando l'interruttore posto sulla **porta 6B** è **chiuso** (vedi fig.12), e fare in modo che quando si riscontra questa condizione si ecciti un **relè** di allarme posto sulla porta **d'uscita 2A**, dovete scrivere l'istruzione in questo modo:

```
        jrr   6,port_b,eccita   ; controlla porta 6B
        jp    fine              ; va a fine se 6B è a 1
eccita                          ; etichetta per proseguire
        set   2,port_a          ; eccita il relè su 2A
fine                            ; non eccitare il relè
```

In questa istruzione è necessario fare un **doppio salto**: il primo serve ad eccitare il relè se il **l'interruttore** applicato sulla **porta 6B** è chiuso, il secondo (**jp fine**) serve a **non eccitare** il relè nel caso in cui l'interruttore non risulti chiuso.
Se non avessimo inserito l'istruzione **jp fine**, il microprocessore avrebbe proseguito con le istruzioni successive ed avrebbe ugualmente **eccitato** il relè anche se l'interruttore non fosse stato **chiuso**.

Nota importante: Il salto **jrr** riesce a raggiungere un'**etichetta** solo se questa si trova ad una distanza pari a circa **60 righe** di **programma**.

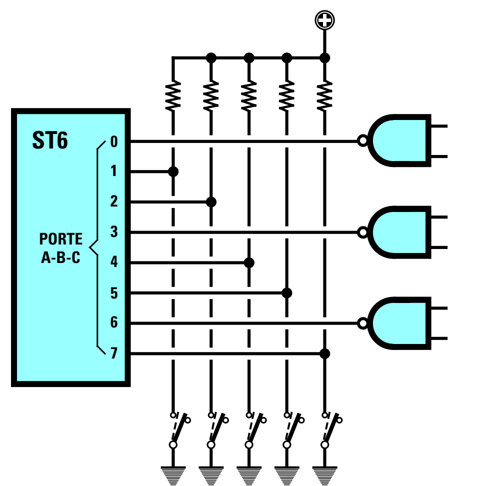


Fig.10 Le istruzioni JRR e JRS possono essere utili per controllare se le porte A-B-C utilizzate come ingressi sono a "livello logico 1" oppure a "livello logico 0". Usando l'istruzione JRR avviene un "salto" se sulla porta d'ingresso è presente un "livello logico 0", mentre usando l'istruzione JRS il "salto" avviene se sulla porta d'ingresso è presente un "livello logico 1".

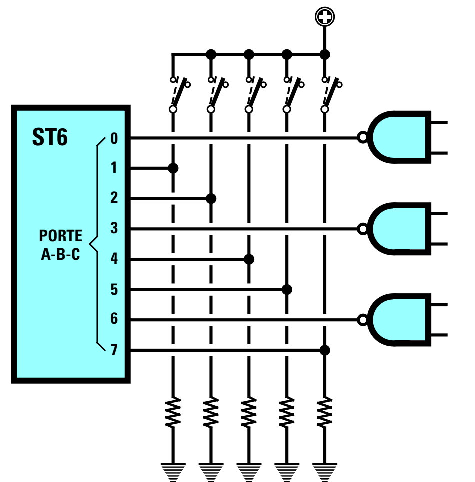


Fig.11 Dopo che vi avremo insegnato come predisporre una porta come Entrata, potrete collegarle degli interruttori o dei pulsanti rivolti verso "massa" (vedi fig.10) o verso il "positivo" (vedi fig.11) oppure l'uscita di una porta Nand - Nor - Or ecc. per controllare il loro livello logico. Nota: Sui vari ingresso dovrete collegare delle resistenze con valori compresi tra 3.300 - 100.000 ohm.

JRS

Questa istruzione serve per controllare se una **cifra** di un **numero binario** si trova a **livello logico 1** e quando si rileva questa condizione viene effettuato un **salto**.
L'istruzione **JRS** può essere utile per controllare se il **piedino d'ingresso** di una qualsiasi porta **A-B-C** si trova a **livello logico 0** o a **livello logico 1** (vedi figg.10-11).
Se volete **controllare** che l'interruttore posto sulla **porta 6B** risulti **chiuso** (vedi fig.13) e quando si riscontra questa condizione, eccitare un **relè** di allarme posto sulla porta d'**uscita 2A**, dovete scrivere l'istruzione in questo modo:

	jrs	**6,port_b,eccita**	;controlla porta **6B**
	jp	**fine**	; va a fine se **6B** è a **0**
eccita			; etichetta per proseguire
	set	**2,port_a**	; eccita il relè su **2A**
fine			; non eccitare il relè

In questa istruzione è necessario compiere un **doppio salto**: il primo serve ad eccitare il relè se l'interruttore applicato sulla **porta 6B** è chiuso, il secondo (**jp fine**) serve a **non eccitare** il relè nel caso in cui l'interruttore non risulti chiuso.
Se non avessimo inserito l'istruzione **jp fine**, il microprocessore avrebbe proseguito con le istruzioni successive ed avrebbe ugualmente **eccitato** il relè anche se l'interruttore non fosse stato **chiuso**.

Nota importante: Il salto **jrs** riesce a raggiungere un'**etichetta** solo se questa si trova ad una distanza pari a circa **60 righe** di **programma**.

LD

Questa istruzione serve per **caricare** il **numero** contenuto in una **variabile** nell'**accumulatore** o viceversa.
Nella funzione **CPI**, in cui abbiamo riportato un esempio per far **diseccitare** un relè quando la temperatura **supera** i **20 gradi**, la **comparazione** viene effettuata solo con il **numero** presente nell'**accumulatore**, quindi abbiamo dovuto inserire all'interno dell'accumulatore il **numero** che era presente nella **variabile** chiamata **gradi**:

	ld	**a,gradi**	; carica nell'**accumul.** i gradi
	cpi	**a,20**	; compara i gradi con numero **20**
	jrnc	**funz**	; salta a **funz** se **maggiore** di **20**
	res	**7,port_b**	; diseccita il relè
funz			; **non** diseccitare il **relè**

Se la **variabile gradi** contiene il numero **15**, dopo l'istruzione **ld a,gradi** anche il numero presente nell'**accumulatore** avrà un valore di **15**.

LDI

Questa istruzione serve per **caricare** in una **variabile** oppure nell'**accumulatore** un qualsiasi **numero** da voi **prescelto** e compreso tra **0** e **255**.
Nell'**istruzione DEC** abbiamo riportato un esempio per **diseccitare** un **relè** dopo **40 minuti**. Questo numero va quindi inserito nella **variabile tempo** come qui sotto riportato:

	ldi	**tempo,40**	; carica in **tempo** il numero **40**
	......	**......**	
	dec	**tempo**	; decrementa la **variabile** tempo
	ld	**a,tempo**	; carica in **a** la **variabile** tempo
	cpi	**a,0**	; confronta **a** con il numero **0**
	jrnz	**funz**	; se è **diverso** salta in **funz**
	res	**7,port_b**	; diseccita relè
funz			; **non** diseccitare il **relè**

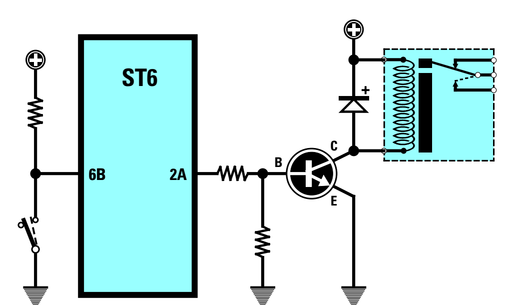


Fig.12 Nel programma riportato al paragrafo JRR il relè si eccita ogni volta che cortocircuitate verso "massa" l'interruttore posto sulla porta 6B. In questo modo sulla porta di uscita 2A ritroverete un livello logico 1 che polarizzerà il transistor.

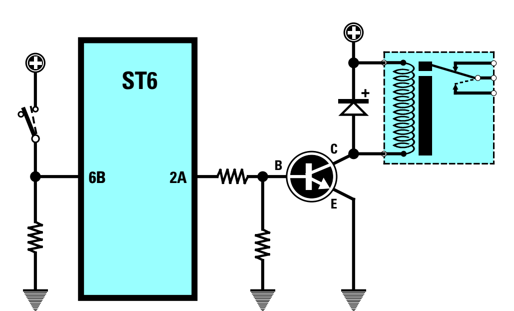


Fig.13 Nel programma riportato al paragrafo JRS il relè si eccita ogni volta che collegate verso il "positivo" l'interruttore posto sulla porta 6B. In questo modo sulla porta di uscita 2A ritroverete un livello logico 1 che polarizzerà il transistor.

Dopo l'istruzione **ldi tempo,40** scritta nella prima riga, la **variabile tempo** è uguale a **40.**

NOP

Questa istruzione viene usata pochissimo, perché serve solamente per ottenere un **ritardo** di qualche **microsecondo**. Infatti fa eseguire al microprocessore **2 cicli macchina** a **vuoto**.
Per eseguire questa funzione è sufficiente scrivere, dopo aver lasciato **uno spazio**, la parola **NOP**; scrivendola più volte aumenterete il **ritardo.**
Se nel microprocessore avete utilizzato un quarzo da **8 MHz**, che esegue **1 ciclo macchina** in un tempo di **1,625 microsecondi**, e scrivete:

```
        nop             ; ritardo 3,25 microsec.
        nop             ; ritardo 3,25 microsec.
        nop             ; ritardo 3,25 microsec.
```

otterrete un ritardo totale di **9,75 microsecondi**.

RES

Questa istruzione serve per **forzare** ad un **livello logico 0** il **bit** di una **variabile**.
Nella funzione **CPI** abbiamo riportato un esempio per **diseccitare** un relè quando la temperatura **supera** i **20 gradi**. Questa operazione è compiuta dalla istruzione **res** a cui bisogna specificare di seguito quale **porta** deve resettare (nel nostro esempio è la **7,port_b**).

```
        ld      a,gradi         ; carica nell'accumul. i gradi
        cpi     a,20            ; compara i gradi con numero 20
        jrnc    funz            ; salta a funz se maggiore di 20
        res     7,port_b        ; diseccita il relè
funz                            ; non diseccitare il relè
```

Ammettendo che tutte le uscite della **porta B** siano a **livello logico 1**, cioè:

11111111

Il programma modificherà il solo bit **7** della **porta B**, cioè il primo bit a sinistra, quindi vi ritroverete con:

01111111

Se voleste azzerare oltre il **piedino 7** della porta **B** anche i **piedini 6 - 5**, dovreste scrivere:

```
        res     7,port_b
        res     6,port_b
        res     5,port_b
```

Vi ricordiamo che il numero di **bit** per ogni porta va da **0** a **7**.

SET

Questa istruzione serve per **forzare** ad un **livello logico 1** il **bit** di una **variabile**.
Nell'istruzione **JRS** abbiamo riportato un esempio per **eccitare** un relè ogni volta che il **pulsante** posto sulla **porta 6B** viene **pigiato**.
Nella riga in cui è posta la funzione **set 2,port_a**, viene forzata l'uscita **2** della **porta A** a **livello logico 1**, in modo che provveda ad eccitare il relè.

```
        jrs     6, port_b,eccita   ; controlla porta 6B
        jp      fine               ; va a fine se 6B è a 0
eccita                             ; etichetta per proseguire
        set     2,port_a           ; eccita il relè su 2A
fine                               ; non eccitare il relè
```

In pratica l'istruzione **SET** compie l'operazione inversa all'istruzione **RES.**

RET

Questa istruzione viene posta alla fine di una **subroutine** per comunicare al microprocessore di ritornare nel punto del programma in cui questa **subroutine** è stata chiamata.
Nell'istruzione **CALL** abbiamo riportato una **subroutine** per ottenere un **ritardo**, cioè:

```
ritardo ldi     x,103           ; assegna 103 a x
ripeti  dec     x               ; sottrai 1 a x
        jrnz    ripeti          ; ripeti se x non è a 0
        ret                     ; ritorna al programma
```

Volendo ottenere un **ritardo** dovete richiamare la **subroutine** chiamata **ritardo** scrivendo:

```
        call    ritardo         ; chiama subroutine ritardo
```

Alla fine della **subroutine** abbiamo posto l'istruzione **RET** per far tornare il programma alla riga posta di seguito all'istruzione **call ritardo.**
Vi ricordiamo che al termine della **subroutine** occorre sempre mettere l'istruzione **RET**, diversamente il programma **non** ritornerà nel punto in cui abbiamo chiamato la subroutine, ma proseguirà con le righe successive, senza segnalare nessun **errore**, ma causando anomalie nel funzionamento del programma.

RETI

Questa istruzione viene utilizzata alla fine di particolari tipi di **subroutine** che si chiamano **interrupt** e che sono identificati con le **etichette**:

```
ad_int                          ; serve per il convertitore A/D
tim_int                         ; serve per il timer
BC_int                          ; serve per le porte B-C
A_int                           ; serve per la porta A
nmi_int                         ; serve per il piedino nmi
```

Dopo una di queste etichette, si **interrompono** momentaneamente le funzioni del programma **principale** e vengono eseguite le istruzioni che si trovano dopo l'**interrupt**.
Alla fine di questa **subroutine** di **interrupt** dovrete inserire l'istruzione **RETI** per ritornare al programma principale, nel punto in cui si trovava prima dell'**interruzione forzata**.
Come noterete, nel microprocessore **ST6** c'è un piedino chiamato appunto **nmi** (corrisponde al piedino **5**) sul quale potete collegare un pulsante (vedi fig.14) che potrete usare per **interrompere** forzatamente una funzione.
Ammettiamo che abbiate scritto un programma per l'automazione di un **trapano** e mentre questo sta eseguendo la foratura, **si spezzi la punta**.
In questo caso dovete **immediatamente** interrompere il programma e togliere la tensione di alimentazione dal trapano per sostituire la punta.
Il programma va scritto nel seguente modo:

```
nmi_int                      ; etichetta nmi di interrupt
            res    5,port_b  ; resetta la porta 5B
            reti             ; termine subroutine
```

La **porta 5B** è quella che eccita o diseccita il **teleruttore** che alimenta il trapano.

RLC

Questa istruzione serve per **spostare** verso sinistra tutte le **cifre** di un **numero binario** presente nell'**accumulatore**.
Sulla **destra** di tale numero entrerà un **1** o uno **0**, che risulta parcheggiato in una particolare cella di memoria **RAM** chiamata **CARRY**, presente all'interno del microprocessore.
Il valore del **carry** è **1** solo quando l'ultima operazione eseguita ha un riporto, in altre parole quando il risultato è superiore a **255**.
Ad esempio, se il valore nell'accumulatore è **250**, dopo aver eseguito l'istruzione:

```
        addi    a,100
```

il valore del **carry** è **1**.

Tornando all'istruzione **RLC**, se all'interno dell'**accumulatore** è presente il numero **binario 00000100** e nel **carry** è presente un **1**, scrivendo l'istruzione:

```
        rlc     a
```

ritroverete nell'accumulatore questo numero **binario**:

00001001

Se nel **carry** fosse parcheggiato uno **0**, nell'accumulatore ritrovereste questo numero **binario**:

00001000

La funzione **RLC** potrebbe risultarvi utile per accendere in sequenza a **ciclo continuo** dei diodi led. Infatti il numero che **perdete** sulla sinistra entra nel **carry** per rientrare poi sulla **destra**.

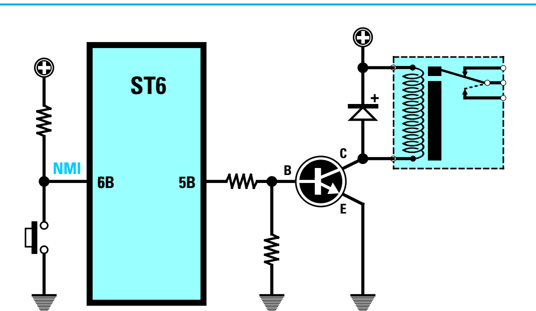


Fig.14 Se scrivete il programma con l'etichetta "nmi_int" riportato nel paragrafo RETI (vedi sopra), quando il piedino 5 dell'NMI viene posto a livello logico 0, anche sull'uscita 5B ritroverete un livello logico 0 ed il relè si disecciterà.

SLA

Anche questa istruzione serve per **spostare** verso sinistra tutte le **cifre** di un **numero binario** presente nell'**accumulatore**, con la sola differenza che, non utilizzando il **carry**, sulla destra entra sempre uno **0** ed il numero che fuoriesce a **sinistra** va **perso**.
L'istruzione **SLA** può essere utile per ottenere una **moltiplicazione** per **2**.
Ad esempio, se nell'**accumulatore** è presente questo numero **binario**

00010111

che corrisponde al numero **decimale 23**, scrivendo:

```
        sla     a
```

nell'accumulatore troverete questo diverso numero **binario**:

00101110

che corrisponde al numero **decimale 46** (equivalente a **23 x 2**).
Il numero massimo che potete **moltiplicare** per 2 è **127**.

STOP

Questa istruzione serve per interrompere l'esecuzione di un programma e per spegnere lo stadio oscillatore del **clock**.
E' un'istruzione che si usa raramente.

WAIT

Questa istruzione serve per interrompere l'esecuzione di un programma, con la sola differenza rispetto alla precedente, che non si spegne lo stadio oscillatore del **clock**.

SUB

Questa istruzione viene utilizzata per **sottrarre** dal numero presente nell'**accumulatore** il numero contenuto in una **variabile**.
Se nell'**accumulatore** è presente il numero **183** e volete sottrargli il numero decimale **53**, dovrete inserire in una variabile, chiamata ad esempio **pippo**, il numero **53** scrivendo questa istruzione:

```
      ldi    a, 183       ; numero nell'accumulatore
      ldi    pippo, 53    ; numero inserito nella variabile
      sub    a, pippo     ; sottrai da a il valore di pippo
```

Eseguita questa operazione nell'accumulatore avrete **183-53 = 130**.

SUBI

Questa istruzione viene utilizzata per **sottrarre** dal numero presente nell'**accumulatore** un numero da voi inserito nella riga di questa istruzione.
Se nell'**accumulatore** è presente il numero **183** e gli volete sottrarre il numero **53**, dovrete scrivere l'istruzione in questo modo:

```
      ldi    a,183        ; numero nell'accumulatore
      subi   a,53         ; sottrai da a il numero 53
```

Il risultato presente nell'accumulatore sarà dopo questa istruzione **130**.
Come per la precedente, anche con questa istruzione non potete sottrarre un numero **maggiore** a quello presente nell'**accumulatore**.

CONTINUA nel PROSSIMO NUMERO

Sebbene con gli articoli e gli esempi pubblicati sulle riviste N.172/173 e 174 siate già riusciti a **programmare** e a **cancellare** senza difficoltà il microprocessore **ST6**, questo non è ancora sufficiente per passare alla **fase** più propriamente **pratica**, iniziare cioè a scrivere dei programmi, perché occorre prima conoscere come vanno correttamente utilizzati:

- la funzione **watchdog**
- i piedini delle **porte** come **ingressi**
- i piedini delle **porte** come **uscite**
- la funzione **interrupt**
- la funzione **A/D converter**
- la funzione **timer**

IMPARARE a programmare

NOTA IMPORTANTE

Negli articoli precedenti (vedi riviste N.172/173 e 174) vi abbiamo consigliato di scrivere tutte le istruzioni in **minuscolo** sottolineando nel frattempo che non cambia assolutamente nulla se utilizzate la forma **maiuscola**.
Abbiamo tuttavia notato che la tipografia quando trascrive questi testi in **minuscolo** compie più frequentemente **errori tipografici**; infatti confonde la lettera **i** con la lettera **l** e viceversa, la lettera **r** per la **n**, e spesso trascrive così vicino le due lettere **r n** da farle sembrare una **m**.
Proprio per evitare questo tipo di errori da adesso in poi scriveremo le righe dei programmi un po' in **maiuscolo** ed un po' in **minuscolo**.
Ripetiamo nuovamente che ogni **riga** di programma è composta da **quattro blocchi** chiamati **Etichetta - Istruzione - Operando ; Commento** (vedi fig.1), che dovrete tenere distanziati gli uni dagli altri con un carattere di **tabulazione**.
In questo modo avrete tutte le istruzioni perfettamente incolonnate ed in caso di necessità potrete controllarle meglio.
Non è obbligatorio inserire l'ultimo **blocco**, quello del **commento**, ma se lo aggiungete dovete ricordarvi di farlo precedere sempre da un **punto e virgola**.
Nei programmi che troverete di seguito ogni **blocco** è stato evidenziato da un rettangolo **colorato** in azzurro.
Così se ad esempio in una riga di programma **manca** l'**etichetta**, troverete al suo posto un **rettangolo** colorato senza alcuna **scritta** al suo interno.
Questo per ricordarvi che per avere tutte le istruzioni incolonnate dovete lasciare un carattere di **tabulazione** prima di scrivere il secondo blocco, quello cioè dell'**istruzione**.

WATCHDOG

Il **watchdog** è un contatore pilotato dalla frequenza di **clock** del quarzo, che, iniziando dal numero che avete inserito nel registro **wdog** (è sempre bene inserire un numero alto, ad esempio **255**), conta a rovescio fino ad arrivare a **0**.
Quando raggiunge lo **0**, il watchdog **resetta** automaticamente il microprocessore, che in questo modo non potrà più **proseguire** con il programma che stava eseguendo.
Il **watchdog** impedisce che eventuali **disturbi** presenti sulla tensione di rete dei **220 volt** o generati da altri fonti, entrando involontariamente nel microprocessore lo predispongano per eseguire funzioni **non previste**.
Tanto per portare un esempio, supponiamo di aver programmato un **ST6** per far eseguire ad un **tornio automatico** delle viti **filettate** lunghe **10 millimetri**.
Può verificarsi il caso in cui un **disturbo**, entrando nel micro, lo predisponga per **togliere** la filettatura

ETICHETTA	ISTRUZIONE	OPERANDO	;	COMMENTO RIGA

Fig.1 Ogni riga di programma è composta da quattro blocchi che occorre tenere distanziati da uno "spazio". Potrete anche non inserire l'ultimo blocco del COMMENTO, ma se lo utilizzate, dovrete farlo precedere da un "punto e virgola". Se nella riga di un programma mancasse l'ETICHETTA, dovrete lasciare uno SPAZIO prima di scrivere l'ISTRUZIONE.

i microprocessori ST6

Poiché ci siamo ripromessi di insegnarvi a programmare correttamente un ST6, è nostra intenzione spiegarvi in modo molto semplice e con numerosi esempi tutti i passi che bisogna compiere, per evitare che commettiate quei comuni ed involontari piccoli errori, che potrebbero impedire il regolare funzionamento del microprocessore.

o per fare delle viti lunghe **10 centimetri**.
Per **evitare** che, in assenza di **disturbi**, il contatore del **watchdog** possa raggiungere lo **0** e quindi **resettare** il microprocessore mentre sta eseguendo le istruzioni del programma, dovrete sempre ricordarvi di **inserire** ogni **20 - 30 - 40 righe** di programma questa scritta:

```
        LDI   wdog,255
```

Conviene inserire questa istruzione dopo un'**etichetta** che faccia ripetere alcune righe di programma diverse volte, perché è in questi casi che il contatore del **watchdog** può più facilmente **scaricarsi**, cioè raggiungere lo **0**.
Se ciò avviene, il microprocessore si **resetta**, in altre parole non può più proseguire con le successive istruzioni ed il programma riparte dall'inizio.
Molti programmatori principianti non trovando un'esauriente spiegazione e nemmeno nessun esempio su come utilizzarla, non inserivano questa istruzione, e quando il programma si **bloccava** perché il **watchdog** arrivava a **0**, non sapendo trovare un'altra spiegazione, cambiavano l'**ST6** ritenendolo **difettoso** oppure cercavano nel programma un **errore**, che in realtà non esisteva.
A chi ci ha interpellato per queste **anomalie** abbiamo chiesto se vicino ad un'etichetta **ripeti** o ad una funzione che il programma eseguiva diverse volte era stata riportata la riga **LDI wdog,255**, e quasi sempre abbiamo ricevuto una risposta negativa.
Dopo aver spiegato che il **watchdog** si può paragonare ad una **pila ricaricabile** e, come tale, ogni tanto occorre **ricaricarla** per evitare che arrivi a **0 volt**, tutti hanno capito l'importanza di questa funzione e dopo averla inserita nel loro programma gli inconvenienti lamentati sono spariti.

Negli **esempi** che trovate in questo articolo troverete spesso questa istruzione:

```
ripeti  LDI   wdog,255   ; carica il watchdog
        .....            ; programma
        JP    ripeti     ; salta a ripeti
```

che in pratica serve a **ricaricare** questa "pila" sul numero massimo che possiamo inserire, cioè **255**.

TABELLA N.1 per ST62/E10 - ST62/T10 - ST62/E20 - ST62/T20

porta	A0	A1	A2	A3	B0	B1	B2	B3	B4	B5	B6	B7
piedino	19	18	17	16	15	14	13	12	11	10	9	8

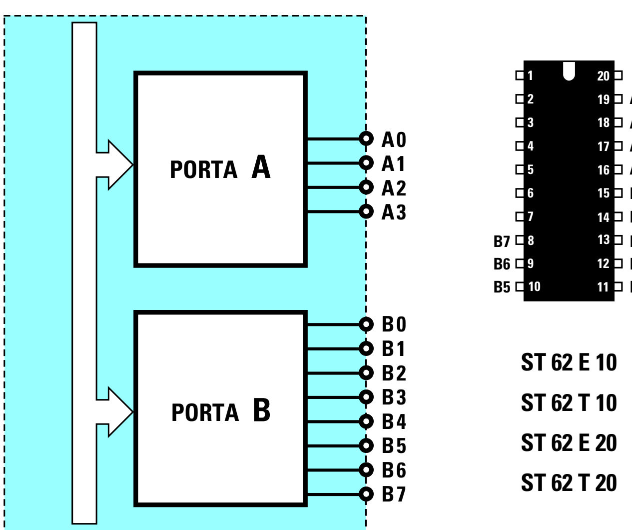


Fig.2 All'interno dei microprocessori ST62 della serie 10-20 sono presenti due PORTE chiamate A-B. Nella porta A troviamo solo quattro piedini Ingressi/Uscita siglati A0-A1-A2-A3, mentre nella porta B troviamo otto piedini siglati da B0 a B7.
Nella Tabella sopra riportata abbiamo indicato a quale piedino dell'ST6 corrispondono i piedini di queste due PORTE.

TABELLA N.2 per ST62/E15 - ST62/T15 - ST62/E25 - ST62/T25

porta	A0	A1	A2	A3	A4	A5	A6	A7	B0	B1	B2	B3	B4	B5	B6	B7	C4	C5	C6	C7
piedino	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	9	8	7	6

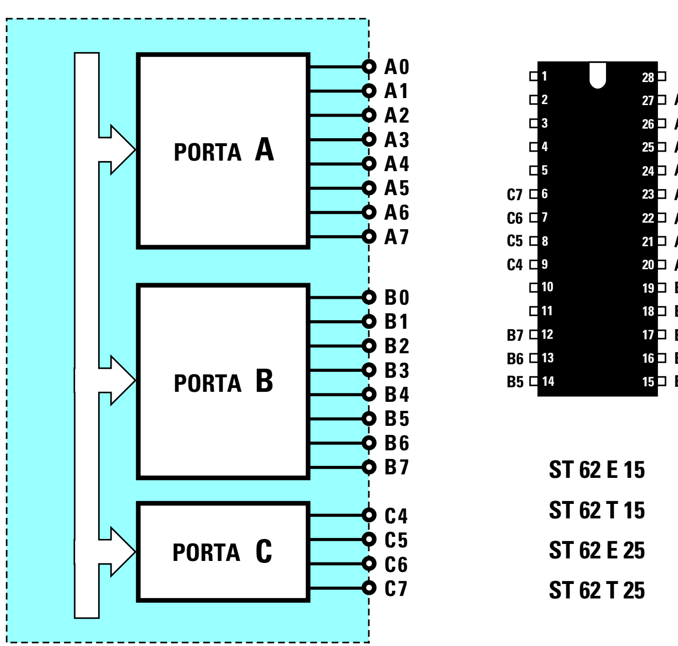


Fig.3 All'interno dei microprocessori ST62 della serie 15-25 sono presenti tre PORTE chiamate A-B-C.
Nelle due porte A-B sono presenti otto piedini Ingressi/Uscita siglati da A0 a A7 e da B0 a B7, mentre nella porta C troviamo solo quattro piedini siglati C4-C5-C6-C7.
Nella Tabella sopra riportata abbiamo indicato a quale piedino dell'ST6 corrispondono i piedini di queste tre PORTE.

LE PORTE sui piedini del MICROPROCESSORE

All'interno di tutti i microprocessori **ST6** sono presenti due oppure tre **porte** siglate **A-B-C** i cui terminali, contraddistinti dalle sigle **A0 - A1 - A2** ecc. e **B0 - B1 - B2** ecc., fanno capo ai piedini del microprocessore (vedi figg.2-3).
Nelle **Tabelle N.1** e **N.2** riportiamo i numeri dei piedini a cui sono collegati i terminali delle **porte** presenti nei diversi integrati **ST6**.
Conoscere a quale **piedino** risultano collegati i terminali di queste **porte** è molto importante, perché nella **riga** del programma non viene mai indicato il **numero** del piedino, ma la sola **sigla** della **porta**, cioè **A0 - A2 - A3** o **B5 - B6 - B7** ecc.

COME SETTARE LE PORTE

Per **settare** il piedino prescelto come **ingresso senza pull-up** - **ingresso con interrupt - ingresso analogico** - **ingresso pull-up** oppure come **uscita open collector - uscita push-pull**, dobbiamo inserire nei registri **pdir - popt - port** degli **0** o degli **1**, disponendoli come abbiamo riportato nella **Tabella N.3**.

Esempio: Ammesso di voler predisporre tutti i piedini della **porta A** come **ingressi** tipo **pull-up**, dovremmo necessariamente scrivere queste tre righe:

```
LDI  pdir_a,00000000B
LDI  popt_a,00000000B
LDI  port_a,00000000B
```

Se volessimo invece predisporre tutti i piedini della **porta A** come **uscita** in **push-pull** dovremmo necessariamente scrivere queste tre righe.

```
LDI  pdir_a,11111111B
LDI  popt_a,11111111B
LDI  port_a,00000000B
```

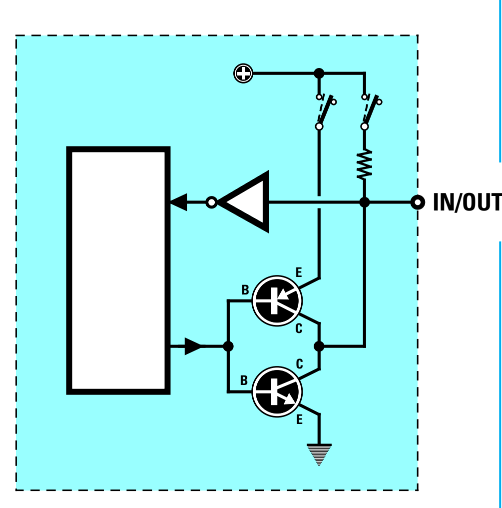


Fig.4 Inserendo nei tre registri PDIR - POPT - PORT degli 0 o degli 1 (vedi Tabella N.3), noi possiamo predisporre il piedino di una qualsiasi Porta a funzionare come Ingresso oppure come Uscita.

Ricordatevi che dopo il nome dei tre registri, **pdir - popt - port**, bisogna sempre indicare il tipo di **porta** che si vuole utilizzare scrivendo:

_**a** se la porta è la **A**
_**b** se la porta è la **B**
_**c** se la porta è la **C**

Non scrivete **-a**, in altre parole non utilizzate il segno della **sottrazione**, ma sempre il segno del **sottolineato**, cioè **_a**.
La lettera **B**, posta dopo l'ultima cifra di destra di ogni riga, serve ad indicare che si tratta di un numero **binario**.

TABELLA N.3 per predisporre gli ingressi e le uscite

Registri	INGRESSI				USCITE	
	con pull-up	senza pull-up	con interrupt	per segnali analogici	open collector	uscita push-pull
pdir	0	0	0	0	1	1
popt	0	0	1	1	0	1
port	0	1	0	1	0	0

Fig.5 Per predisporre il piedino di una Porta a funzione come Ingresso oppure come Uscita, bisogna scrivere nei tre registri PDIR - POPT - PORT gli 0 o gli 1 nell'ordine riportato in questa Tabella. Nell'articolo trovate molti esempi su come settare queste Porte.

NOTA IMPORTANTE: Non tutti i piedini delle **porte** si possono settare come **ingresso** per **segnali analogici**, quindi per evitare **errori** ricordatevi che di tutti i microprocessori riportati sia nella **Tabella N.1** sia nella **Tabella N.2**, non potrete mai utilizzare come **ingressi analogici** i piedini **A0 - A1 - A2 - A3**.
Queste porte potranno invece essere **settate** per tutti gli altri tipi di ingresso, cioè **con pull-up - senza pull-up - con interrupt**.
Al contrario tutti i piedini di tutte le porte possono essere **settati** come **uscite**, tenendo presente che l'**uscita** più comunemente utilizzata è il **push-pull**, che si ottiene ponendo a **1-1-0** i tre registri **pdir**, **popt** e **port**.

Per concludere facciamo presente che l'**ingresso** più comunemente utilizzato è quello **con pull-up** che si ottiene ponendo a **0-0-0** i tre registri **pdir**, **popt** e **port**.
E' inoltre sempre consigliabile **settare** come **ingressi** tutti i piedini che non vengono usati, perché in questo modo non si corre il rischio di provocare involontariamente dei **cortocircuiti** che potrebbero mettere fuori uso il microprocessore.
Infatti se un piedino che non viene utilizzato viene **settato** come **uscita** e poi viene collegato involontariamente a **massa**, anche tramite una resistenza di basso valore, può capitare che per un **errore** nel programma questo piedino cambi il suo stato logico da **0** a **1**, ed in questo caso in uscita si potrebbe ritrovare una tensione **positiva** di **5 volt** che potrebbe provocare dei **cortocircuiti**.
Per questo motivo, come noterete dagli esempi, **tutti** i piedini delle **porte A - B - C** che non vengono utilizzati li abbiamo settati come **ingressi** in **pull-up**.

Posizione dei PIEDINI nel NUMERO BINARIO

Per poter predisporre i piedini delle **porte** come **ingressi** o come **uscite** si utilizza un numero **binario** composto da otto cifre.
Molti tra voi si staranno chiedendo come si fa a capire qual è il piedino **A0 - A1 - A2** ecc. oppure **B0 - B1 - B2** ecc., in quanto anche se abbiamo indicato il tipo di porta con **_a**, con **_b** o con **_c**, non abbiamo mai precisato il suo terminale **0-1-2-3-4-5-6-7**.
Inoltre non abbiamo ancora specificato come ci si deve comportare con i microprocessori della **Tabella N.1**, dove la **porta A** ha solo quattro piedini siglati **A0 - A1 - A2 - A3** o con quelli della **Tabella N.2**, dove la **porta C** ha invece quattro piedini siglati **C4 - C5 - C6 - C7**.

Ogni **terminale** di queste **porte** viene definito da uno degli **otto** numeri posti dopo la **virgola**, ricordando che il terminale **7** è la prima cifra a **sinistra** ed il terminale **0** è l'ultima cifra a **destra**, come qui sotto riportato:

TABELLA N.4

cifra	1°	2°	3°	4°	5°	6°	7°	8°
piedino	A7	A6	A5	A4	A3	A2	A1	A0
piedino	B7	B6	B5	B4	B3	B2	B1	B0
piedino	C7	C6	C5	C4				

Quindi se volessimo caricare un **1** nel registro **pdir** di **A7** dovremmo scrivere:

```
LDI    pdir_a,10000000B
```

mentre se volessimo caricare un **1** nel registro **pdir** di **A4** dovremmo scrivere:

```
LDI    pdir_a,00010000B
```

ed ancora se volessimo caricare un **1** nel registro **pdir** di **A0** dovremmo scrivere:

```
LDI    pdir_a,00000001B
```

Per completare il **settaggio** di ogni porta si devono aggiungere le altre **due** righe di programma, dove compaiono i registri **popt** e **port**, mettendo degli **0** o degli **1** come riportato nella **Tabella N.3**.

Ammesso che si voglia **settare** il piedino **5** della **porta B** come **uscita** in **push-pull** si dovrà scrivere:

```
LDI    pdir_b,00100000B
LDI    popt_b,00100000B
LDI    port_b,00000000B
```

Infatti la sequenza per settare un piedino in **uscita push-pull** è **1-1-0**, e, come abbiamo riportato nella **Tabella N.4**, il piedino **B5** è la terza cifra a partire da sinistra.
Come abbiamo già avuto modo di dire, **tutti** i piedini delle **porte A - B - C** che non vengono utilizzati devono comunque essere settati come **ingressi** in **pull-up (0 - 0 - 0)**.

Nei microprocessori che hanno solo quattro piedini per la **porta A** (**A0 - A1 - A2 - A3**), si devono comunque riportare sempre tutte le otto cifre del **numero binario**.
Quindi se volessimo predisporre come **uscite pu-**

sh-pull i piedini della porta **A** di un **ST62E10**, dovremmo scrivere:

```
LDI  pdir_a,00001111B
LDI  popt_a,00001111B
LDI  port_a,00000000B
```

Allo stesso modo, se volessimo predisporre come **uscite push-pull** tutti i quattro piedini dei microprocessori che hanno anche la **porta C** (**C4 - C5 - C6 - C7**) dovremmo scrivere:

```
LDI  pdir_c,11110000B
LDI  popt_c,11110000B
LDI  port_c,00000000B
```

Ogni piedino di ogni **porta** può essere singolarmente **settato** come **ingresso** o come **uscita** e persino in maniera differente.
Ad esempio, possiamo **settare** il piedino **B1** della **porta B** come **uscita push-pull**, il piedino **B2** come **ingresso pull-up**, il piedino **B3** come **ingresso analogico** ed il piedino **B4** come **uscita open collector**.

ESEMPI di SETTAGGIO piedini

Esempio N.1 = Disponendo di un **ST6** del tipo riportato nella **Tabella N.1**, ad esempio un **ST62E10**, vorremmo programmare i piedini **A0 - A1** come **ingressi senza pull-up** ed i piedini **B4 - B5 - B6 - B7** come **uscite** in **push-pull**.

Soluzione: Per scrivere queste istruzioni esaminiamo prima di tutto la **Tabella N.3**, per sapere qual è la cifra, se **0** o **1**, che dobbiamo mettere nei tre registri **pdir - popt - port**.
Sapendo che per ogni riga che definisce il settaggio dei piedini dobbiamo sempre mettere **8 cifre** e che il piedino **7** è definito dalla prima cifra a **sinistra** e lo **0** dall'ultima cifra a **destra**, possiamo scrivere le nostre istruzioni:

```
LDI  pdir_a,00000000B
LDI  popt_a,00000000B
LDI  port_a,00000011B
```

```
LDI  pdir_b,11110000B
LDI  popt_b,11110000B
LDI  port_b,00000000B
```

legenda:

LDI = significa **carica** sul registro.

pdir - popt - port = sono i tre registri necessari per **settare** una porta.

_a e **_b** = sono le **porte A** e **B** ed il numero che segue dopo la **virgola** indica quale degli **otto** piedini, **7-6-5-4-3-2-1-0**, vogliamo **settare** come **ingresso** o come **uscita**.

B = questa lettera posta sull'estrema destra indica che il **numero** a **otto** cifre è un **Binario**.

Esempio N.2 = Disponendo di un **ST6** del tipo riportato nella **Tabella N.2**, ad esempio un **ST62E15**, vorremmo programmare tutti i piedini delle **porte A-B-C** come **ingressi pull-up**.

Soluzione: Controlliamo nella **Tabella N.3** se dobbiamo mettere uno **0** o un **1** nei tre registri **pdir - popt - port** per poterli settare come **ingressi pull-up**, quindi poiché in tutti va inserito uno **0** scriviamo:

```
LDI  pdir_a,00000000B
LDI  popt_a,00000000B
LDI  port_a,00000000B
```

```
LDI  pdir_b,00000000B
LDI  popt_b,00000000B
LDI  port_b,00000000B
```

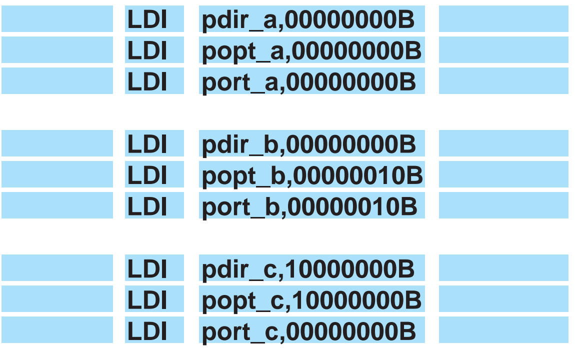
```
LDI  pdir_c,00000000B
LDI  popt_c,00000000B
LDI  port_c,00000000B
```

Come avrete notato, anche sulla porta **C** abbiamo messo tutti **0** sebbene questa abbia solo quattro piedini siglati **C4 - C5 - C6 - C7**.

Esempio N.3 = Disponendo di un **ST6** del tipo riportato nella **Tabella N.2** vorremmo programmare il piedino **B2** come **ingresso pull-up**, il piedino **B1** come **ingresso analogico** ed il piedino **C7** come **uscita push-pull**.

Soluzione: Controlliamo innanzitutto nella **Tabella N.3** come vanno settati i tre registri **pdir - popt - port** per avere un **ingresso pull-up**, un **ingresso analogico** ed una **uscita push-pull**, quindi sapendo che per ogni riga di programma dobbiamo

sempre mettere **8 cifre** e che il piedino **7** è definito dalla prima cifra a **sinistra** e lo **0** dall'ultima cifra a **destra**, scriviamo:



	LDI	**pdir_a,00000000B**	
	LDI	**popt_a,00000000B**	
	LDI	**port_a,00000000B**	
	LDI	**pdir_b,00000000B**	
	LDI	**popt_b,00000010B**	
	LDI	**port_b,00000010B**	
	LDI	**pdir_c,10000000B**	
	LDI	**popt_c,10000000B**	
	LDI	**port_c,00000000B**	

Come indicato dalla **Tabella N.3**, nei tre registri relativi alla porta **B** abbiamo messo **0-0-0** per il piedino **B2** e **0-1-1** per il piedino **B1**, mentre nelle tre righe relative alla **porta C**, per il piedino **C7** abbiamo posto **1-1-0**.

Qualcuno si starà già chiedendo in quale applicazione pratica possiamo utilizzare un piedino **settato** come **ingresso**. Anche se in seguito troverete alcuni **esempi** su questo argomento, possiamo anticiparvi subito che potete usarlo per vedere se sull'**ingresso** entra una tensione positiva oppure per stabilire se questa cambia di stato logico da **0** a **1** e viceversa, o ancora per **convertire** una tensione **analogica** in una **digitale** ecc.

Esempio N.4 = Abbiamo realizzato lo schema di fig.6 da utilizzare per un antifurto, quindi vorremmo che si **eccitasse** un relè ogni volta che l'interruttore viene pigiato.

Soluzione: La prima operazione da compiere è quella di **settare** il piedino della porta che vogliamo utilizzare come **ingresso** ed il piedino della porta che vogliamo utilizzare come **uscita**.
Come **ingresso** abbiamo deciso di scegliere il piedino **A1** e come **uscita** il piedino **A2**.

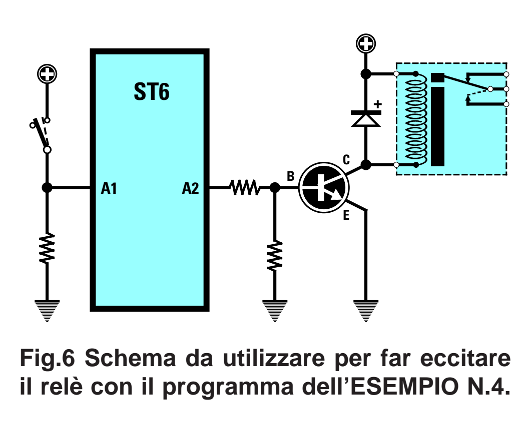


Fig.6 Schema da utilizzare per far eccitare il relè con il programma dell'ESEMPIO N.4.

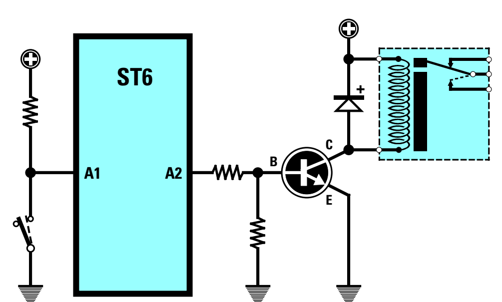


Fig.7 Schema da utilizzare per far eccitare il relè con il programma dell'ESEMPIO N.5.

Dalla **Tabella N.3** controlliamo come settare i registri **pdir - popt - porta** per far diventare questi piedini **ingressi** o **uscite**, dopodiché possiamo scrivere tutte le righe del programma indicato a fine pagina.
Nella riga **ripeti** abbiamo ricaricato il **watchdog**, poi con l'istruzione **JRS** abbiamo detto al programma di saltare all'etichetta **eccita** solo quando sul piedino **A1** riscontra una tensione di **5 volt**.
Se non riscontra questa condizione, il microprocessore passa ad eseguire la riga **JRR** e da qui salta all'etichetta **spegni**, che **diseccita** il relè se prima risultava eccitato, o lo lascia diseccitato se si trova già in questa condizione.

	LDI	**pdir_a,00000100B**	; con queste tre righe abbiamo settato la porta **A1** come
	LDI	**popt_a,00000100B**	; **ingresso** e la porta **A2** come **uscita**
	LDI	**port_a,00000000B**	;
ripeti	**LDI**	**wdog,255**	; ricarica il **watchdog**
	JRS	**1,port_a,eccita**	; se in **A1** ci sono **5 Volt** salta a **eccita**
	JRR	**1,port_a,spegni**	; se in **A1** ci sono **0 Volt** salta a **spegni**
eccita	**SET**	**2,port_a**	; setta a **5 volt** l'uscita di **A2**
	JP	**ripeti**	; salta all'etichetta **ripeti**
spegni	**RES**	**2,port_a**	; setta a **0 volt** l'uscita di **A2**
	JP	**ripeti**	; salta all'etichetta **ripeti**

Esempio N.5 = Abbiamo realizzato lo schema di fig.7 e vorremmo che, quando l'interruttore **corto-circuita** verso massa il piedino d'ingresso **A1**, si **eccitasse** il relè collegato sull'uscita **A2**.
Soluzione: La prima operazione da compiere è quella di **settare** il piedino **A1** come **ingresso** ed il piedino **A2** come **uscita.**
In questo caso, dovendo ottenere una funzione opposta a quella dell'**esempio N.4**, dovremo scrivere il programma come indicato di seguito:

```
        LDI   pdir_a,00000100B   ; con queste tre righe abbiamo settato la porta A1 come
        LDI   popt_a,00000100B   ; ingresso e la porta A2 come uscita
        LDI   port_a,00000000B   ;
ripeti  LDI   wdog,255           ; ricarica il watchdog
        JRR   1,port_a,eccita    ; se in A1 ci sono 0 Volt salta a eccita
        JRS   1,port_a,spegni    ; se in A1 ci sono 5 Volt salta a spegni
eccita  SET   2,port_a           ; setta a 5 volt l'uscita di A2
        JP    ripeti             ; salta all'etichetta ripeti
spegni  RES   2,port_a           ; setta a 0 volt l'uscita di A2
        JP    ripeti             ; salta all'etichetta ripeti
```

Esempio N.6 = Ammettiamo di voler predisporre tutti i piedini della porta **A** e della porta **B** come **uscite** in **push-pull**.

Soluzione: Per ottenere questa condizione dobbiamo scrivere solo **1** per i due registri **pdir - popt** e solo **0** per il terzo registro **port** (vedi **Tabella N.3**).

```
        LDI   pdir_a,11111111B
        LDI   popt_a,11111111B
        LDI   port_a,00000000B

        LDI   pdir_b,11111111B
        LDI   popt_b,11111111B
        LDI   port_b,00000000B
```

Se la porta **A** disponesse di solo **4 piedini** (vedi microprocessori della **Tabella N.1**) dovremmo scrivere **1** solo sui piedini delle porte utilizzate.

Esempio N.7 = Supponiamo di voler predisporre i piedini **A0 - A1 - A2 - A3** e **B0 - B1** come **ingressi pull-up** e i piedini **A4 - A5 - A6 - A7** e **B2 - B3 - B4 - B5 - B6 - B7** come **uscite push-pull**.
In questo caso dovremo scrivere:

```
        LDI   pdir_a,11110000B
        LDI   popt_a,11110000B
        LDI   port_a,00000000B

        LDI   pdir_b,11111100B
        LDI   popt_b,11111100B
        LDI   port_b,00000000B
```

Esempio N.8 = Abbiamo un microprocessore del tipo riportato nella **Tabella N.2**, con a disposizione le tre porte **A-B-C**, e vorremmo predisporre tutte i piedini della porta **A-B** come **ingressi pull-up** e quelli della porta **C** come **uscite push-pull**.
In questo caso dovremo scrivere:

```
        LDI   pdir_a,00000000B
        LDI   popt_a,00000000B
        LDI   port_a,00000000B

        LDI   pdir_b,00000000B
        LDI   popt_b,00000000B
        LDI   port_b,00000000B

        LDI   pdir_c,11110000B
        LDI   popt_c,11110000B
        LDI   port_c,00000000B
```

INTERRUPT

L'**Interrupt** serve per "interrompere" momentaneamente l'esecuzione delle istruzioni principali affinché il microprocessore possa eseguire altre istruzioni, che si trovano inserite tra una delle cinque **etichette** qui sotto riportate e la scritta **reti.**

```
ad_int              ; etichetta
        .....       ; programma da eseguire
        reti        ; fine interrupt

tim_int             ; etichetta
        .....       ; programma da eseguire
        reti        ; fine interrupt

BC_int              ; etichetta
        .....       ; programma da eseguire
        reti        ; fine interrupt
```

```
A_int                ; etichetta
        .....        ; programma da eseguire
        reti         ; fine interrupt

nmi_int              ; etichetta
        .....        ; programma da eseguire
        reti         ; fine interrupt
```

Nota: la funzione di **interrupt** viene abilitata ed eseguita dal microprocessore solo se nel programma è stata scritta la seguente istruzione:

```
        SET   4,ior
```

Dopo aver scritto le righe di programma da eseguire con la funzione **interrupt** è necessario completarle con la scritta **reti**, perché quando il microprocessore arriva ad eseguire l'istruzione **reti** ritorna al programma principale nel punto in cui era stato **momentaneamente** interrotto.
L'**interrupt** viene utilizzato per far eseguire al microprocessore un'istruzione che in quel momento è **più importante** di quella che stava eseguendo.
Ad esempio, se avessimo programmato un microprocessore per **aprire** e **chiudere** il cancello scorrevole del nostro giardino, dovremmo utilizzare la funzione **interrupt** applicata alla **fotocellula** per fare in modo che in fase di **chiusura** se un bambino o il nostro cane attraversa improvvisamente la **fotocellula**, il cancello si blocchi.

Etichetta ad_int

Questa etichetta serve per riconoscere quando il convertitore **A/D** ha terminato la conversione. Normalmente non si usa **mai** perché in sua sostituzione si preferisce scrivere questa riga di programma.

```
attendi   JRR   6,adcr,attendi
```

Etichetta tim_int

Questa etichetta viene utilizzata nelle funzioni **timer**. In pratica quando il registro **tcr** (vedi paragrafo **Timer**) arriva a **0**, il microprocessore esegue tutte le istruzioni che sono state scritte tra **tim_int** e **reti**.

Etichetta BC_int

Questa etichetta viene utilizzata per fare eseguire tutte le istruzioni che abbiamo scritto tra **BC_int** e **reti** quando su uno dei piedini da noi scelto delle porte **B** o **C** la tensione cambia di stato, in altre parole quando passa dal **livello logico 0** al **livello logico 1** (fronte di **salita**) o quando passa dal **livello logico 1** al **livello logico 0** (fronte di **discesa**).

Nota: per la funzione **ingresso con interrupt** possiamo abilitare **un solo piedino** di una delle due porte **B** o **C**.

Esempio N.9 = Supponiamo di voler **settare** il piedino **B2** come ingresso **con interrupt** che intervenga sul **fronte di salita** e tutti gli altri piedini della **porta B** come ingresso in **pull-up**. Come prima operazione controlliamo nella **Tabella N.3** come vanno **settati** i registri **pdir - popt - port** per l'ingresso **con interrupt**, dopodiché possiamo scrivere il programma indicato "**Esempio n.9**".

PROGRAMMA per Esempio n.9

```
        LDI   pdir_b,00000000B   ; abbiamo settato il piedino B2 come ingresso interrupt
        LDI   popt_b,00000100B   ; e tutti gli altri come ingressi pull-up
        LDI   port_b,00000000B   ;
        SET   4,ior              ; serve per abilitare l'interrupt
        SET   5,ior              ; serve per sentire il fronte di salita
BC_int                           ; etichetta d'inizio interrupt
        .....                    ; istruzioni da eseguire con l'interrupt
        RETI                     ; istruzione fine interrupt
```

PROGRAMMA per Esempio n.10

```
        LDI   pdir_b,00000000B   ; abbiamo settato il piedino B7 come ingresso interrupt
        LDI   popt_b,10000000B   ; e tutti gli altri come ingressi pull-up
        LDI   port_b,00000000B   ;
        SET   4,ior              ; serve per abilitare l'interrupt
        RES   5,ior              ; serve per sentire il fronte di discesa
BC_int                           ; etichetta d'inizio dell'interrupt
        ....                     ; istruzioni da eseguire con l'interrupt
        RETI                     ; istruzione fine interrupt
```

Esempio N.10 = Supponiamo di voler **settare** il piedino **B7** come ingresso **con interrupt** che intervenga sul **fronte di discesa** e tutti gli altri piedini della **porta B** come ingresso in **pull-up**. Come prima operazione controlliamo nella **Tabella N.3** come vanno **settati** i registri **pdir - popt - port** per l'ingresso **con interrupt**, dopodiché possiamo scrivere il programma indicato "**Esempio n.10**".

Etichetta A_int

Questa etichetta viene utilizzata per fare eseguire tutte le istruzioni che abbiamo scritto tra **A_int** e **reti** quando sul piedino della porta **A** da noi scelto la tensione passa dal **livello logico 1** al **livello logico 0** (fronte di **discesa**).

Nota: per la funzione **ingresso con interrupt** possiamo abilitare **un solo piedino** della porta **A**.

Esempio N.11 = Ammettiamo di voler **settare** il piedino **A5** come ingresso **con interrupt** che intervenga quando il **livello logico 1** cambia a **livello logico 0**. Come prima operazione controlliamo nella **Tabella N.3** come dobbiamo **settare** i registri **pdir - popt - port** per l'ingresso **con interrupt**, dopodiché possiamo scrivere il programma indicato "**Esempio n.11**".

Etichetta nmi_int

Questa etichetta viene utilizzata per fare eseguire tutte le istruzioni che abbiamo scritto tra **nmi_int** e **reti** quando sul piedino siglato **NMI** (piedino **5** di tutti gli **ST6**) la tensione passa dal **livello logico 1** al **livello logico 0** (fronte di **discesa**). Poiché questo piedino è sempre **abilitato**, non è necessario scrivere nel programma l'istruzione:

```
        SET   4,ior
```

Esempio N.12 = Vogliamo che pigiando il pulsante **P1** (vedi fig.8) si accendano i **diodi led** applicati sui piedini **A0 - A1 - A2 - A3**.

Come prima operazione **settiamo** i piedini **A0 - A1 - A2 - A3** come **uscite push-pull** prelevando dalla **Tabella N.3** i dati da inserire nelle righe **pdir - popt - port** e a questo punto possiamo scrivere il programma indicato "**Esempio n.12**".

NOTA IMPORTANTE: se nelle righe del programma dell'**interrupt** fossero presenti delle istruzioni che utilizzano l'accumulatore **A**, ad esempio:

```
        CPI    a,10   ; confronta A con il numero 10
        ADDI   a,10   ; somma ad A il numero 10
        LDI    a,10   ; metti in A il numero 10
```

bisognerà inserire in una **variabile**, che potremo chiamare **salva**, il valore dell'accumulatore **A** subito dopo l'etichetta dell'**interrupt** e, prima di terminare con l'istruzione **RETI**, lo dovremo reinserire nell'accumulatore **A.**
In questo modo quando il microprocessore tornerà ad eseguire il programma principale, nell'accumulatore si avrà lo stesso valore che c'era prima dell'interrupt.

PROGRAMMA per Esempio n.11

```
         LDI    pdir_a,00000000B   ; abbiamo settato il piedino A5 come ingresso interrupt
         LDI    popt_a,00100000B   ; e tutti gli altri come ingressi pull-up
         LDI    port_a,00000000B   ;
         SET    4,ior              ; serve per abilitare l'interrupt
A_int                              ; etichetta d'inizio dell'interrupt
         ....                      ; istruzioni da eseguire con l'interrupt
         RETI                      ; istruzione fine interrupt
```

PROGRAMMA per Esempio n.12

```
         LDI    pdir_a,00001111B   ; con queste tre righe abbiamo settato come uscite
         LDI    popt_a,00001111B   ; push-pull i piedini A0 - A1 - A2 - A3
         LDI    port_a,00000000B   ;
mni_int                            ; etichetta d'inizio dell'interrupt
         SET    0,port_a           ; accendi il led sul piedino A0
         SET    1,port_a           ; accendi il led sul piedino A1
         SET    2,port_a           ; accendi il led sul piedino A2
         SET    3,port_a           ; accendi il led sul piedino A3
         RETI                      ; istruzione fine interrupt
```

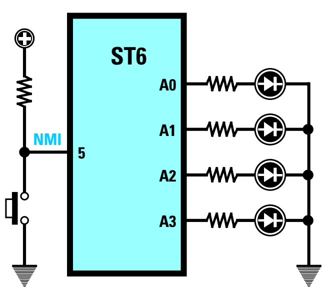


Fig.8 Affinché, pigiando il pulsante posto sul piedino "NMI", si accendano tutti i led applicati sui piedini A0-A1-A2-A3, potrete utilizzare il programma riportato nell'Esempio N.12 o quello dell'Esempio N.13.

Esempio N.13 = Nell'esempio N.12 abbiamo riportato un programma che accendeva contemporaneamente **quattro diodi led** pigiando il pulsante collegato sul piedino **NMI** (vedi fig.8). Usando l'accumulatore **A** si ottiene lo stesso risultato, ma bisogna modificare il programma come indicato nell'"**Esempio n.13**".

A/D CONVERTER

All'interno di ogni microprocessore **ST6** è presente un **A/D converter**, cioè un circuito in grado di convertire una tensione **analogica** compresa tra **0** e **5 volt** in un **numero decimale** compreso tra **0** e **255**.
Per ottenere questa condizione occorre **settare** il piedino della porta in cui viene applicata questa **tensione** come **ingresso analogico**.
Ricordatevi che in tutti i **microprocessori** della serie **ST6** (vedi **Tabelle N.1 - N.2**) **non possono** essere mai utilizzati come **ingressi** per **segnali analogici** i piedini **A0 - A1 - A2 - A3**.
La **massima tensione** che si può applicare su questi ingressi non deve mai **superare** i **5 volt positivi**.
Il valore **numerico** che otteniamo applicando una tensione al piedino che abbiamo settato come **ingresso analogico** si può calcolare con questa formula:

numero decimale = (volt x 255) : 5

Pertanto se applichiamo sul piedino d'ingresso una tensione di **4 volt** otteniamo un **numero decimale** di:

(4 x 255) : 5 = 204

Ricordatevi che l'**A/D converter** fornisce in uscita soltanto dei **numeri interi** quindi se nel risultato sono presenti dei decimali, questi vengono arrotondati al numero intero più prossimo.
Ad esempio, se la tensione di **4 volt** dovesse scendere a **3,98 volt**, sull'uscita non otterremmo:

(3,98 x 255) : 5 = 202,98

ma il numero più prossimo, cioè **203**.

Se la tensione di **4 volt** dovesse salire a **4,04 volt**, sull'uscita non otterremmo:

(4,04 x 255) : 5 = 206,04

ma il numero più prossimo, cioè **206**.

Anche con l'arrotondamento del numero si ottiene sempre un'elevata precisione in quanto la differenza risulta di sole poche **decine** di **millivolt**.
Ad esempio, prendendo sempre la tensione di **4 volt**, per ogni variazione di **0,01 volt** otterremo questi **numeri decimali**:

3,96 volt = 202
3,97 volt = 202
3,98 volt = 203
3,99 volt = 203
4,00 volt = 204
4,01 volt = 205
4,02 volt = 205
4,03 volt = 206
4,04 volt = 206

PROGRAMMA per Esempio n.13

	LDI	**pdir_a,00001111B**	; con queste tre righe abbiamo settato come **uscite**
	LDI	**popt_a,00001111B**	; push-pull i piedini **A0 - A1 - A2 - A3**
	LDI	**port_a,00000000B**	;
mni_int			; etichetta d'inizio dell'**interrupt**
	LD	**salva,a**	; copia nella variabile **salva** il valore di **A**
	LDI	**a,00001111B**	; carica in **A** il numero binario **00001111**
	LD	**port_a,a**	; copia il valore **A** nel registro della **porta a**
	LD	**a,salva**	; copia nell'accumulatore **A** il valore di **salva**
	RETI		; istruzione fine **interrupt**

Conoscendo il **numero decimale** è possibile calcolare il valore della tensione in **volt** utilizzando questa formula:

volt = (decimale x 5) : 255

Quindi il **numero decimale 203** corrisponde ad un valore di tensione pari a:

(203 x 5) : 255 = 3,98 volt

con una differenza di **0,01 volt** in più o in meno.
Come abbiamo già detto, un ingresso **settato** per **segnali analogici** può servire soltanto per misurare delle **tensioni continue** che non superino i **5 volt**.
Se la tensione risultasse maggiore, occorrerà ridurla con dei **partitori resistivi**, come in pratica accade in tutti i **tester analogici** che, pur disponendo di uno strumento da **1 volt fondo scala**, possono misurare tensioni anche di **250 - 300 volt**.
E' inoltre possibile misurare delle **tensioni alternate**, se si provvede prima a **raddrizzarle**.
Un ingresso **analogico** può servire per misurare delle **temperature**, delle variazioni di **luce**, degli **ohm** oppure la **reattanza** dei condensatori o delle impedenze, ed anche la **corrente** assorbita da un circuito o la **potenza** di un amplificatore.

IMPORTANTE: Poiché all'interno dei microprocessori **ST6** è presente un **solo A/D converter**, solo **un piedino** può essere adibito a questa funzione.
Se per errore vengono settati come **ingressi** per **segnali analogici due piedini**, questi verranno posti in **cortocircuito** ed in questo modo verrà danneggiato il microprocessore.

Esempio N.14 = Vogliamo realizzare un circuito che **accenda** un **diodo led** quando la tensione applicata sul piedino prescelto supera i **3 volt**. Per l'**ingresso** si potrebbe decidere di utilizzare il piedino **B1** e come **uscita** il piedino **A0** (vedi fig.9).

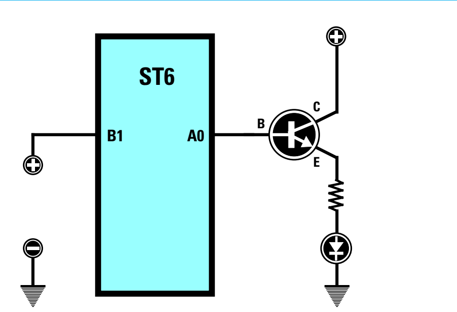


Fig.9 Per accendere il diodo led applicato sul piedino A0 quando la tensione sul piedino B1 supera i 3 Volt, dovrete scrivere il programma posto a fine pagina. Leggere attentamente la soluzione dell'Esempio N.14.

Come si deve scrivere il programma perché il micro esegua questa funzione?
Soluzione: Dalla **Tabella N.3** controlliamo come dobbiamo settare i registri **pdir - popt - port** per predisporre **B1** come **ingresso analogico** e **A0** come **uscita** in **push-pull**.
Successivamente calcoliamo il **numero decimale** di **3 volt** che risulta:

(3 x 255) : 5 = 153

A questo punto possiamo scrivere il nostro programma.

PROGRAMMA per Esempio n.14

```
        LDI   pdir_a,00000001B   ; in queste prime tre righe abbiamo settato la porta
        LDI   popt_a,00000001B   ; A0 come uscita in push-pull
        LDI   port_a,00000000B   ;
        LDI   pdir_b,00000000B   ; in queste tre righe abbiamo settato la porta
        LDI   popt_b,00000010B   ; B1 come ingresso analogico
        LDI   port_b,00000010B   ;
ripeti  LDI   wdog,255           ; carichiamo il watchdog
        LDI   adcr,00110000B     ; provvedi a convertire da analogico a digitale
attendi JRR   6,adcr,attendi     ; attendere che avvenga la conversione A/D
        LD    a,addr             ; carica nell'accumulatore A il numero digitale
        CPI   a,153              ; compara il valore di A con il numero 153
        JRNC  accendi            ; se A è maggiore di 153 salta all'etichetta accendi
        JRC   spegni             ; se A è minore di 153 salta all'etichetta spegni
        JP    ripeti             ; salta all'etichetta ripeti del watchdog
accendi SET   0,port_a           ; setta l'uscita del piedino A0 a 5 volt
        JP    ripeti             ; salta all'etichetta ripeti del watchdog
spegni  RES   0,port_a           ; setta l'uscita del piedino A0 a 0 volt
        JP    ripeti             ; salta all'etichetta ripeti del watchdog
```

Esempio N.15 = Vogliamo accendere una fila di **5 diodi led**, ma in modo che con **1 volt** si accenda un **solo led**, con **2 volt** si accendano **2 led**, con **3 volt** si accendano **3 led** ecc., fino a far accendere tutti i **5 diodi led** quando la tensione raggiunge i **5 volt**.
Come **ingresso analogico** abbiamo deciso di scegliere il piedino **A7** e come **uscite** i piedini da **B0** a **B4**.

Soluzione: La prima operazione che dobbiamo compiere è quello di calcolare il **numero decimale** corrispondente ai valori di tensione di **1 - 2 - 3 - 4 - 5 volt** usando la formula che già conosciamo:

(1 x 255) : 5 = 51 numero decimale di 1 Volt
(2 x 255) : 5 = 102 numero decimale di 2 Volt
(3 x 255) : 5 = 153 numero decimale di 3 Volt
(4 x 255) : 5 = 204 numero decimale di 4 Volt
(5 x 255) : 5 = 255 numero decimale di 5 Volt

A questo punto possiamo programmare il piedino **A7** come **ingresso analogico** ed i piedini **B0 - B1 - B2 - B3 - B4** come **uscite push-pull** (vedi **Tabella N.3**).

Se anziché accendere tutta la fila dei **diodi led** volessimo accendere **un solo** diodo led per volta, cioè prima quello su **B0**, poi quello su **B1 - B2 - B3 - B4**, dovremmo modificare tutte le righe delle **etichette LED** mettendo un **1** solo sul piedino a cui è collegato il led che vogliamo accendere e degli **0** sui piedini a cui sono collegati i led che vogliamo spegnere.

Nell'esempio che si trova a fine pagina dovremmo riscrivere le sole righe **LED2 - LED3 - LED4 - LED5** in questo modo:

```
LED2   LDI   port_b,00000010B
LED3   LDI   port_b,00000100B
LED4   LDI   port_b,00001000B
LED5   LDI   port_b,00010000B
```

PROGRAMMA per Esempio n.15

```
        LDI    pdir_a,00000000B   ; in queste tre righe abbiamo settato
        LDI    popt_a,10000000B   ; il piedino A7 come ingresso analogico
        LDI    port_a,10000000B   ;
        LDI    pdir_b,00011111B   ; in queste righe abbiamo settato
        LDI    popt_b,00011111B   ; i piedini da B0 a B4 come uscite
        LDI    port_b,00000000B   ;
ripeti  LDI    wdog,255           ; carichiamo il watchdog
        LDI    adcr,00110000B     ; provvedi a convertire da analogico a digitale
attendi JRR    6,adcr,attendi     ; attendere che avvenga la conversione A/D
        LD     a,addr             ; carica nell'accumulatore A, il numero digitale
        CPI    a,255              ; compara il valore di A con il numero 255
        JRNC   LED5               ; se A è uguale a 255 salta all'etichetta LED5
        CPI    a,204              ; compara il valore di A con il numero 204
        JRNC   LED4               ; se A è maggiore di 204 salta all'etichetta LED4
        CPI    a,153              ; compara il valore di A con il numero 153
        JRNC   LED3               ; se A è maggiore di 153 salta all'etichetta LED3
        CPI    a,102              ; compara il valore di A con il numero 102
        JRNC   LED2               ; se A è maggiore di 102 salta all'etichetta LED2
        CPI    a,51               ; compara il valore di A con il numero 51
        JRNC   LED1               ; se A è maggiore di 51 salta all'etichetta LED1
        JP     LED0               ; se A è minore di 51 salta all'etichetta LED0
LED0    LDI    port_b,00000000B   ; non accendere nessun diodo led
        JP     ripeti             ; salta all'etichetta ripeti del watchdog
LED1    LDI    port_b,00000001B   ; accendi il led sul piedino B0
        JP     ripeti             ; salta all'etichetta ripeti del watchdog
LED2    LDI    port_b,00000011B   ; accendi i led sui piedini B0 - B1
        JP     ripeti             ; salta all'etichetta ripeti del watchdog
LED3    LDI    port_b,00000111B   ; accendi i led sui piedini B0 - B1 - B2
        JP     ripeti             ; salta all'etichetta ripeti del watchdog
LED4    LDI    port_b,00001111B   ; accendi i led sui piedini B0 - B1 - B2 - B3
        JP     ripeti             ; salta all'etichetta ripeti del watchdog
LED5    LDI    port_b,00011111B   ; accendi i led sui piedini B0 - B1 - B2 - B3 - B4
        JP     ripeti             ; salta all'etichetta ripeti del watchdog
```

Questa modifica potrebbe risultare utile se invece di accendere dei **diodi led** volessimo **eccitare** cinque diversi **relè** per ogni diverso valore di tensione, ad esempio, il **relè1** quando la tensione raggiunge **1 volt**, il **relè 2** quando la tensione raggiunge i **2 volt** ecc.

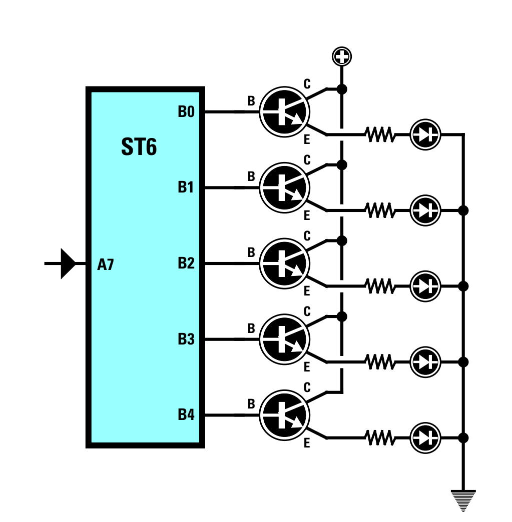


Fig.10 Volendo realizzare un Level-Meter che accenda una fila di 4 led all'aumentare della tensione continua, applicata sul piedino A7 settato come Ingresso, dovrete utilizzare il programma dell'Esempio N.15 riportato sulla pagina in basso a sinistra.

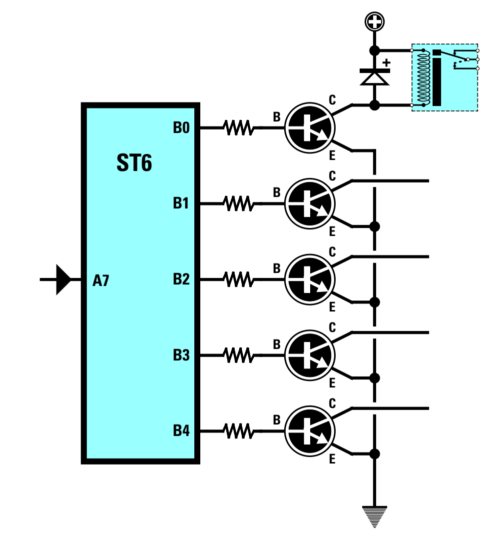


Fig.11 Se vorrete eccitare dei relè dovrete modificare lo schema come qui sopra riportato. Per motivi di spazio abbiamo inserito il relè solo sul primo transistor in alto. Modificando il programma potrete utilizzare come uscite anche tutti gli 8 piedini della Porta B.

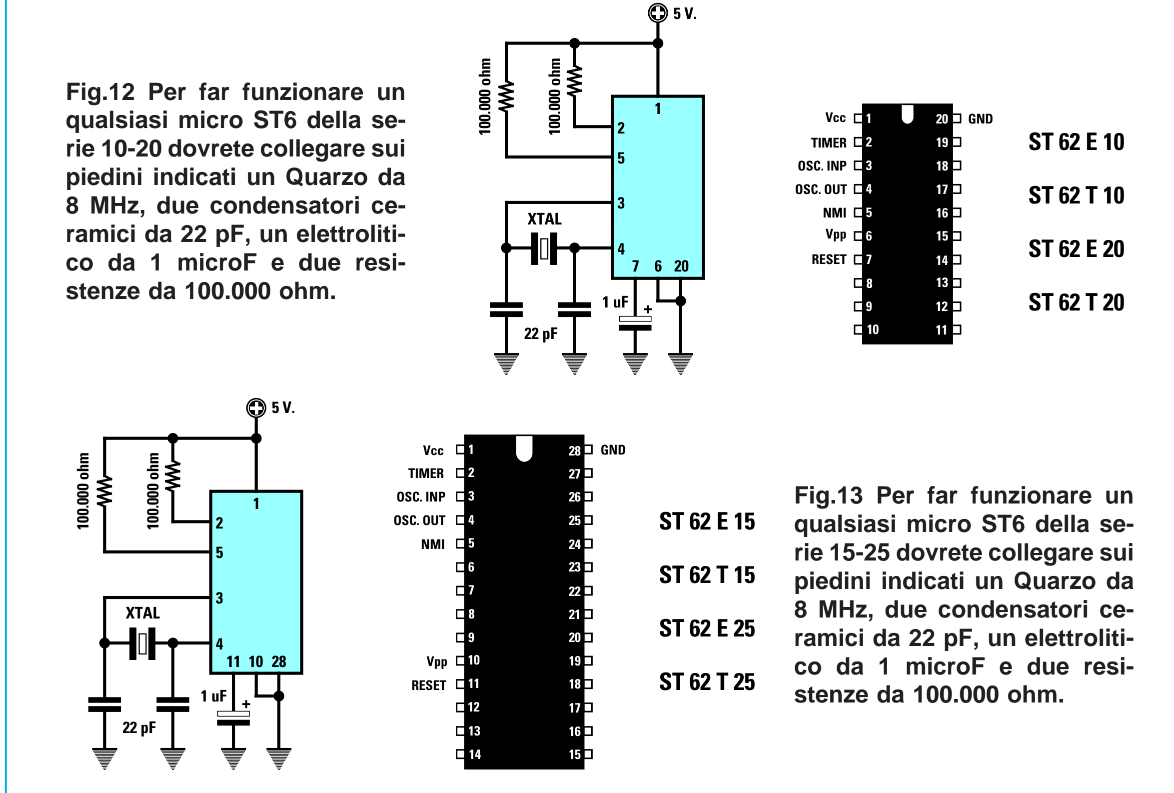


Fig.12 Per far funzionare un qualsiasi micro ST6 della serie 10-20 dovrete collegare sui piedini indicati un Quarzo da 8 MHz, due condensatori ceramici da 22 pF, un elettrolitico da 1 microF e due resistenze da 100.000 ohm.

Fig.13 Per far funzionare un qualsiasi micro ST6 della serie 15-25 dovrete collegare sui piedini indicati un Quarzo da 8 MHz, due condensatori ceramici da 22 pF, un elettrolitico da 1 microF e due resistenze da 100.000 ohm.

TIMER

Il **timer** è un **contatore** collegato al **quarzo** del microprocessore tramite un **prescaler** ed un **divisore x 12** (vedi fig.14).
Per la funzione **timer** dobbiamo settare due registri, uno chiamato **tcr** e l'altro **tscr**.
Nel registro **tcr** (timer counter register) dovremo inserire un **numero decimale** compreso tra **1** e **255**.
Il **tcr** partendo da questo numero **conterà all'indietro** e quando arriverà al numero **0** automaticamente eseguirà tutte le istruzioni comprese tra l'etichetta **tim_int** e l'istruzione **reti**.
Ammesso che si voglia inserire nel registro **tcr** il numero **255** dovremmo scrivere questa riga di programma:

```
LDI    tcr,255
```

Nel registro **tscr** (timer status control register), che controlla il **prescaler**, dovremo inserire un **numero binario** come riportato nella **Tabella N.5**.

TABELLA N.5 Registro tscr

01011000	divide x	1
01011001	divide x	2
01011010	divide x	4
01011011	divide x	8
01011100	divide x	16
01011101	divide x	32
01011110	divide x	64
01011111	divide x	128

Ammesso di voler far dividere il **prescaler** per **128** dovremo scrivere questa riga di programma:

```
LDI    tscr,01011111B
```

Alla fine della riga non dobbiamo dimenticarci di inserire una **B**, perché questo è un **numero binario**.
Per calcolare il **tempo** in **secondi** possiamo usare questa formula:

secondi = (12 x tscr x tcr) : Xtal in Hz

Ammesso che si usi un quarzo da **8 MHz** (pari a **8.000.000 Hz**), che nella riga **tscr** si sia inserito il numero **128** e nella riga **tcr** il numero **255**, otterremo un tempo **massimo** di:

(12 x 128 x 255) : 8.000.000 = 0,0489 secondi

che corrispondono a **48,9 millisecondi**.

Tempi così ridotti potrebbero servire soltanto per realizzare dei **generatori di onde quadre**, ma certo non dei **timer** dove normalmente occorrere raggiungere dei tempi di **minuti** o **ore**.
Per ottenere dei **tempi molto lunghi** possiamo usare degli accorgimenti come per esempio ricorrere all'uso di altre variabili.

Esempio N.16 = Vorremmo prelevare dal piedino **A7** degli **impulsi** di **1 millisecondo**, quindi vorremmo sapere come impostare il programma.

Soluzione: Come prima operazione convertiamo i **millisecondi** in **secondi** dividendoli per **1.000** e così otteniamo:

1 : 1.000 = 0,001 secondo

Quindi calcoliamo quale numero dobbiamo mettere nel **registro tcr** con la formula:

tcr = [(Xtal Hz : 12) : tscr] x secondi

Tenete presente che il numero del **tcr** non deve mai risultare **maggiore** di **255** quindi se questo si

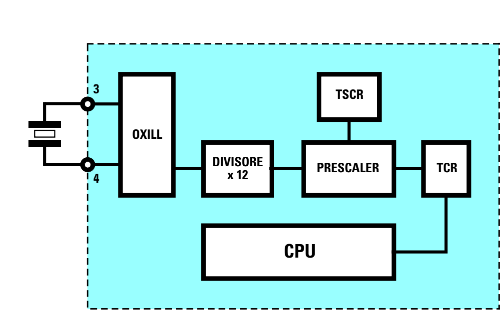


Fig.14 All'interno di ogni microprocessore c'è un Timer che preleva dall'oscillatore quarzato la frequenza generata e la DIVIDE subito x 12. Per ricavare tempi diversi si deve agire solo sui due registri chiamati TCSR e TCR. Nell'articolo trovate due Esempi, uno ha il numero 16 e l'altro il 17.

verificasse dovrete **aumentare** il numero di divisione del **prescaler**.
Questo numero, che possiamo prelevare dalla **Tabella N.5**, è **1 - 2 - 4 - 8 - 16 - 32 - 64 - 128**.
Ad esempio se scegliamo per **tscr** il numero **2** otteniamo un **tcr** di:

[(8.000.000 : 12) : 2] x 0,001 = 333,33 tcr

poiché questo numero è **maggiore** di **255** dovremo scegliere per **tscr** un numero **maggiore**, ad esempio **4** e così otterremo:

[(8.000.000 : 12) : 4] x 0,001 = 166,66 tcr

Poiché il microprocessore lavora solo con **numeri interi**, dovremo arrotondarlo sul numero più prossimo che nel nostro caso è **167**.
Noi abbiamo scelto per **tscr** il numero **4**, ma potevamo anche scegliere **8 - 16 - 32 ecc.** tenendo comunque presente che più alto è il numero del **tscr** minore risulterà la **precisione** sul tempo.
Disponendo dei valori richiesti cioè:

tscr = 4
tcr = 167

possiamo calcolare il **tempo** con la formula:

secondi = (12 x tscr x tcr) : Xtal in Hz

ottenendo:

(12 x 4 x 167) : 8.000.000 = 0,001002 secondi

che corrispondono a:

0,001002 x 1.000 = 1,002 millisecondi

Il programma che dobbiamo scrivere per prelevare dal piedino **A7** questi impulsi è visibile in fondo alla pagina.

Esempio N.17 = Vogliamo prelevare dal piedino **A7** delle **onde quadre** che abbiano una frequenza di **1.200 Hz**, quindi vogliamo sapere come impostare il programma.

Soluzione: Come prima operazione dobbiamo calcolare il **tempo** in **secondi** corrispondente alla frequenza di **1.200 Hz** e per ottenere questo dato usiamo la formula:

secondi = (1 : Hz) : 2

Nel nostro caso otteniamo:

(1 : 1.200) : 2 = 0,0004 secondi

Quindi calcoliamo qual è il numero che dobbiamo mettere nel **registro tcr** con la formula:

tcr = [(Xtal Hz : 12) : tscr] x secondi

Poiché nella formula manca il valore del **tscr**, consultiamo la **Tabella N.5** scegliendo uno di questi numeri **1 - 2 - 4 - 8 - 16 - 32 - 64 - 128**.
Facciamo presente che il valore di **tcr** che ricaveremo da questa formula non dovrà mai superare il numero **255** quindi se risultasse maggiore dovremo usare un **tscr** maggiore, cioè **4 - 8 - 16** ecc. (vedi **Tabella N.5**).
Nel nostro esempio abbiamo scelto per **tscr** il numero **2** perché otteniamo un valore **minore** di **255**.

[(8.000.000 : 12) : 2] x 0,0004 = 133,33 tcr

PROGRAMMA per Esempio n.16

```
         LDI   pdir_a,10000000B   ; queste tre righe servono per settare
         LDI   popt_a,10000000B   ; il piedino A7 come uscita in push-pull
         LDI   port_a,00000000B   ;
         SET   4,ior              ; abilita l'interrupt quando il tcr diventa 0
         LDI   tcr,167            ; numero 167 calcolato per il tcr
         LDI   tscr,01011010B     ; numero binario per un fattore di divisione di 4 (Tabella N.5)
main     LDI   wdog,255           ; ricarichiamo il watchdog
         JP    main               ; salta all'etichetta main
tim_int  LDI   wdog,255           ; ricarichiamo il watchdog
         LDI   tcr,167            ; ricarichiamo 167 nel tcr per ripetere gli impulsi
         LDI   tscr,01011010B     ; questa riga fa ripartire il contatore
         SET   7,port_a           ; fa uscire dal piedino A7 un impulso a 5 volt
         RES   7,port_a           ; riporta il piedino A7 a 0 volt
         RETI                     ; ritorna al programma main
```

Poiché il microprocessore lavora solo con **numeri interi** dovremo arrotondarlo sul numero più prossimo che nel nostro esempio è **133.**
Disponiamo così di tutti i dati richiesti:

tscr = 2
tcr = 133

Per prelevare dal piedino **A7** delle **onde quadre** che abbiano una frequenza di **1.200 Hz**, dovremo scrivere il programma come visibile qui sotto.

PROGRAMMA per Esempio n.17

	LDI	**pdir_a,10000000B**	; queste tre righe ci servono per settare
	LDI	**popt_a,10000000B**	; il piedino **A7** come uscita in **push-pull**
	LDI	**port_a,00000000B**	;
	SET	**4,ior**	; abilita l'**interrupt**
	LDI	**tcr,133**	; carica nel **tcr** il numero **133**
	LDI	**tscr,01011001B**	; numero binario per un fattore di divisione di **2** (Tabella N.5)
main	**LDI**	**wdog,255**	; ricarica il **watchdog**
	JP	**main**	; salta all'etichetta **main**
tim_int			; etichetta dell'**interrupt**
	LDI	**wdog,255**	; ricarica il **watchdog**
	LDI	**tcr,133**	; ricarica **133** nel **tcr** per continuare
	LDI	**tscr,01011001B**	; questa riga fa ripartire il contatore
	JRR	**7,port_a,salita**	; se **A7** è a **0** salta all'etichetta **salita**
	RES	**7,port_a**	; se **A7** è a **1** cambia il livello logico a **0**
	JP	**continua**	; salta all'etichetta **continua**
salita	**SET**	**7,port_a**	; metti il piedino **A7** a **livello logico 1**
continua	**RETI**		; fine dell'**interrupt**

Con il microprocessore ST6 si possono realizzare un'infinità di circuiti, come ad esempio orologi, contasecondi, timer, antifurto, controlli numerici per macchine utensili, termostato, piccoli robot, comandi per luci, termometri, inoltre si possono scrivere delle parole sui display LCD, convertire dei calcoli ecc., e qui ci fermiamo perché volendo elencare tutto ci vorrebbero non poche pagine della rivista.
Anche se nelle riviste precedenti vi abbiamo spiegato le istruzioni per scrivere un programma, per chi è all'inizio queste informazioni potrebbero risultare ancora **insufficienti**.
Infatti chi volesse realizzare un orologio a **display** potrebbe trovarsi in difficoltà nel far apparire i numeri delle **ore** e dei **minuti**. Se poi in sostituzione di un display a sette segmenti si volesse utilizzare un display **LCD**, non si saprebbe quali modifiche apportare al software.
Chi vuole **eccitare** un relè ad una determinata **ora** si chiederà invece quale istruzione gli permette di ottenere questa condizione.
Anche ammesso di aver scritto tutte le istruzioni richieste, il lettore desidera giustamente sapere se il **suo programma** è in grado di eseguire senza errori tutte le funzioni per cui è stato scritto, ha cioè bisogno di **testarlo** e per questo gli servono delle **schede di test** universali.
Sono proprio queste che prenderemo in esame con

BUS per

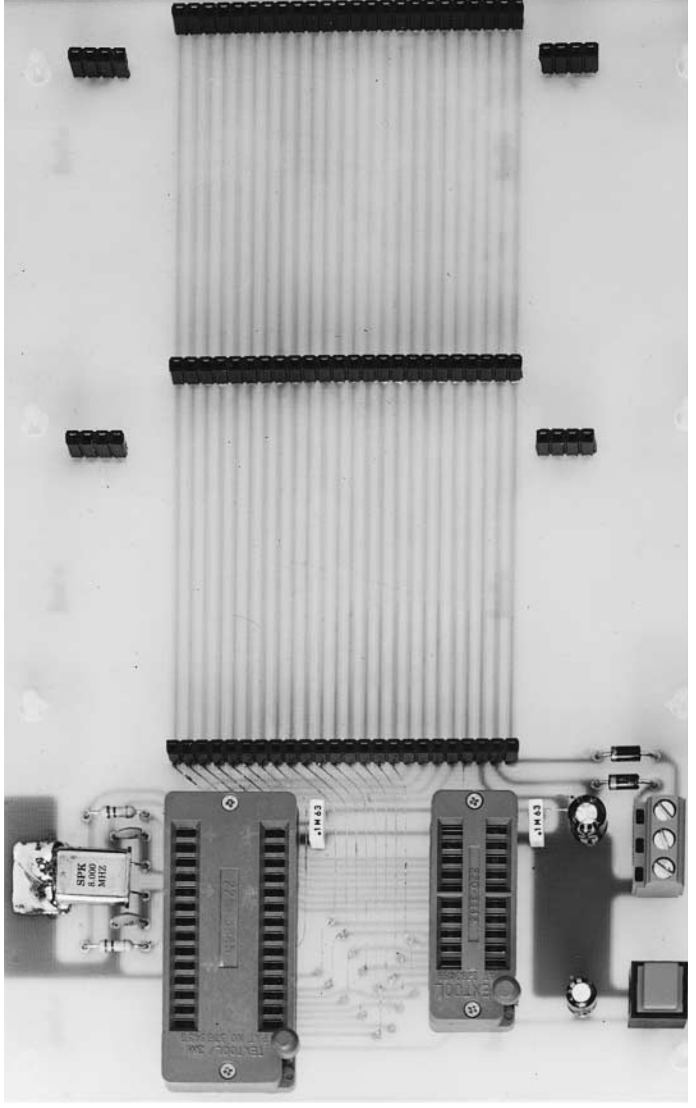


Fig.1 Foto del bus progettato per testare i micro ST6.
I due costosi zoccoli textool possono essere sostituiti con due economici zoccoli per integrati, come visibile nel disegno pratico di fig.3.

TESTARE i micro ST6

Il primo problema che si presenta a quanti desiderano iniziare a scrivere del software personalizzato per i microprocessori ST6 della SGS, è quello di poter controllare il programma per verificare che esegua le funzioni richieste. Per aiutarvi abbiamo progettato delle schede sperimentali, che saranno particolarmente utili agli Istituti Tecnici se usate come supporto al consueto materiale didattico.

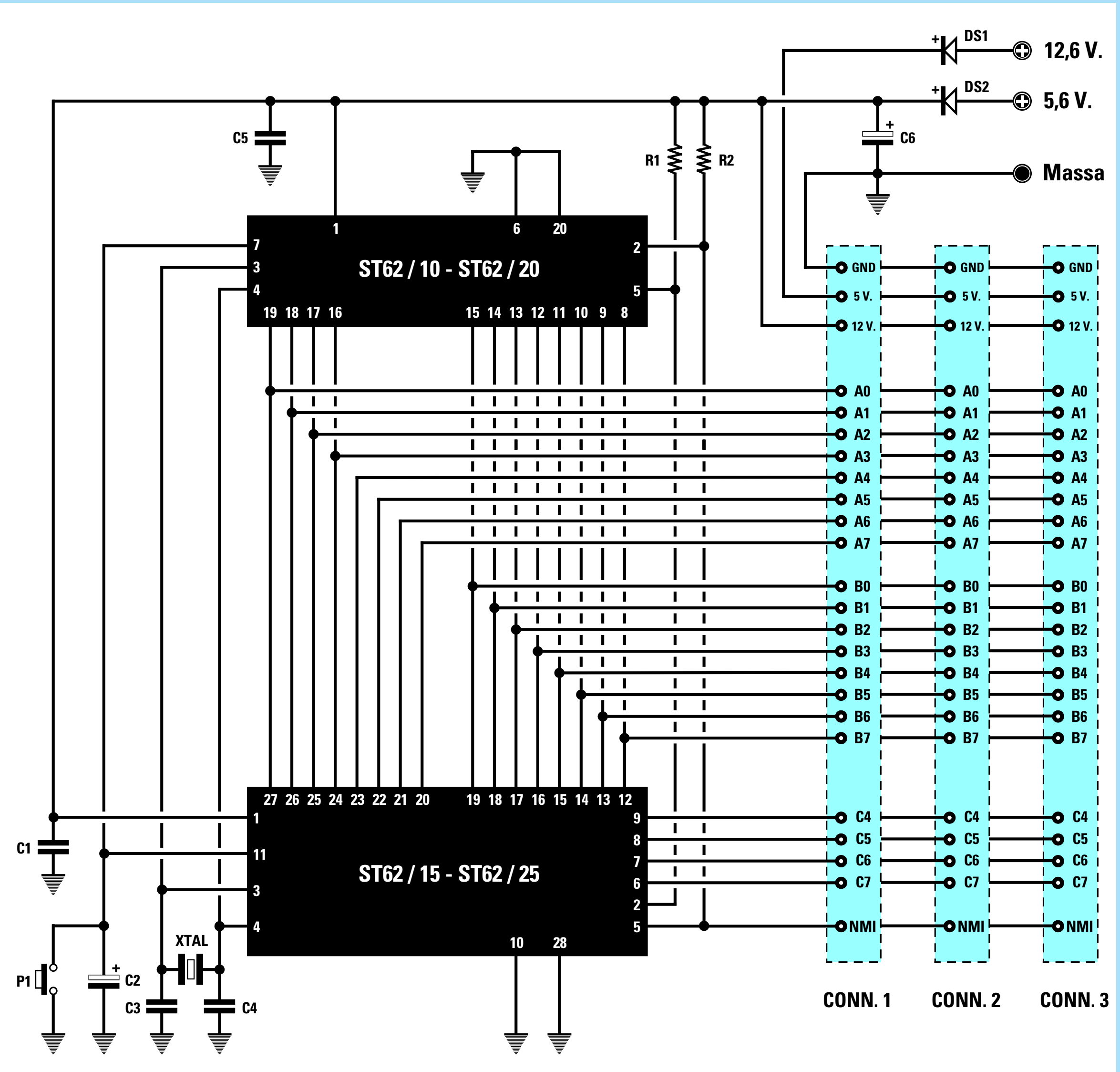
il nostro articolo.
A parte vi forniamo un **dischetto** contenente alcuni programmi completi di esaurienti **commenti**, che potranno servirvi per realizzare orologi, timer, contasecondi, antifurto ecc., e con riportate tutte le modifiche che si possono apportare.
Come sempre i maligni penseranno che il nostro obiettivo sia solo quello di vendere al lettore un **dischetto**, ma essi non considerano che riempire **10 pagine** della rivista con sole righe di programma **non risulta** per nulla gradito a coloro ai quali **non** interessa l'**ST6**.
Inoltre non pensano che nel trascrivere i programmi sulla rivista si possono verificare degli **errori di stampa**, ed altri errori possono commetterli gli stessi lettori nel ricopiare le istruzioni.
Disponendo di un **dischetto** con programmi già testati, il lettore potrà subito metterli in funzione e poi modificarli secondo le proprie esigenze.
Se con le modifiche apportate il programma darà qualche **errore**, sarà sempre possibile fare un controllo con il **programma originale** per verificare dove è stato commesso l'errore.

IMPORTANTE

La **SGS** ci ha comunicato che cesserà di produrre i microprocessori **riprogrammabili** tipo **ST62E10** e tipo **ST62E15**, entrambi da **2 K** di **memoria**, perché tutte le Industrie chiedono e preferiscono utilizzare i **riprogrammabili** da **4 K** di **memoria** anche se più costosi.
I microprocessori **non riprogrammabili** tipo **ST62T10 - ST62T15** con **2 K** di memoria rimarranno invece sempre in produzione.
Pertanto **fuori produzione** andranno i tipi:

ST62E10 che verranno sostituiti dagli **ST62E20**
ST62E15 che verranno sostituiti dagli **ST62E25**

Poiché le dimensioni e la piedinatura dei due mo-



ELENCO COMPONENTI LX.1202

R1 = 100.000 ohm 1/4 watt
R2 = 100.000 ohm 1/4 watt
C1 = 100.000 pF poliestere
C2 = 1 mF elettr. 63 volt
C3 = 22 pF ceramico
C4 = 22 pF ceramico
C5 = 100.000 pF poliestere
C6 = 100 mF elettr. 35 volt
XTAL = quarzo 8 MHz
DS1 = diodo 1N.4007
DS2 = diodo 1N.4007
CONN.1-2-3 = connettori 24 poli
P1 = pulsante

Fig.2 Schema elettrico del circuito bus progettato per testare i programmi per i microprocessori ST6. Anche se nel bus sono presenti due zoccoli, dovrete sempre utilizzarne uno SOLO alla volta, quindi prima di inserire un micro in uno zoccolo dovrete togliere quello presente sull'altro zoccolo.
Il bus andrà alimentato con lo stadio di alimentazione visibile nelle figg.4-7 cercando di non invertire i due fili dei 5,6 e 12,6 volt.

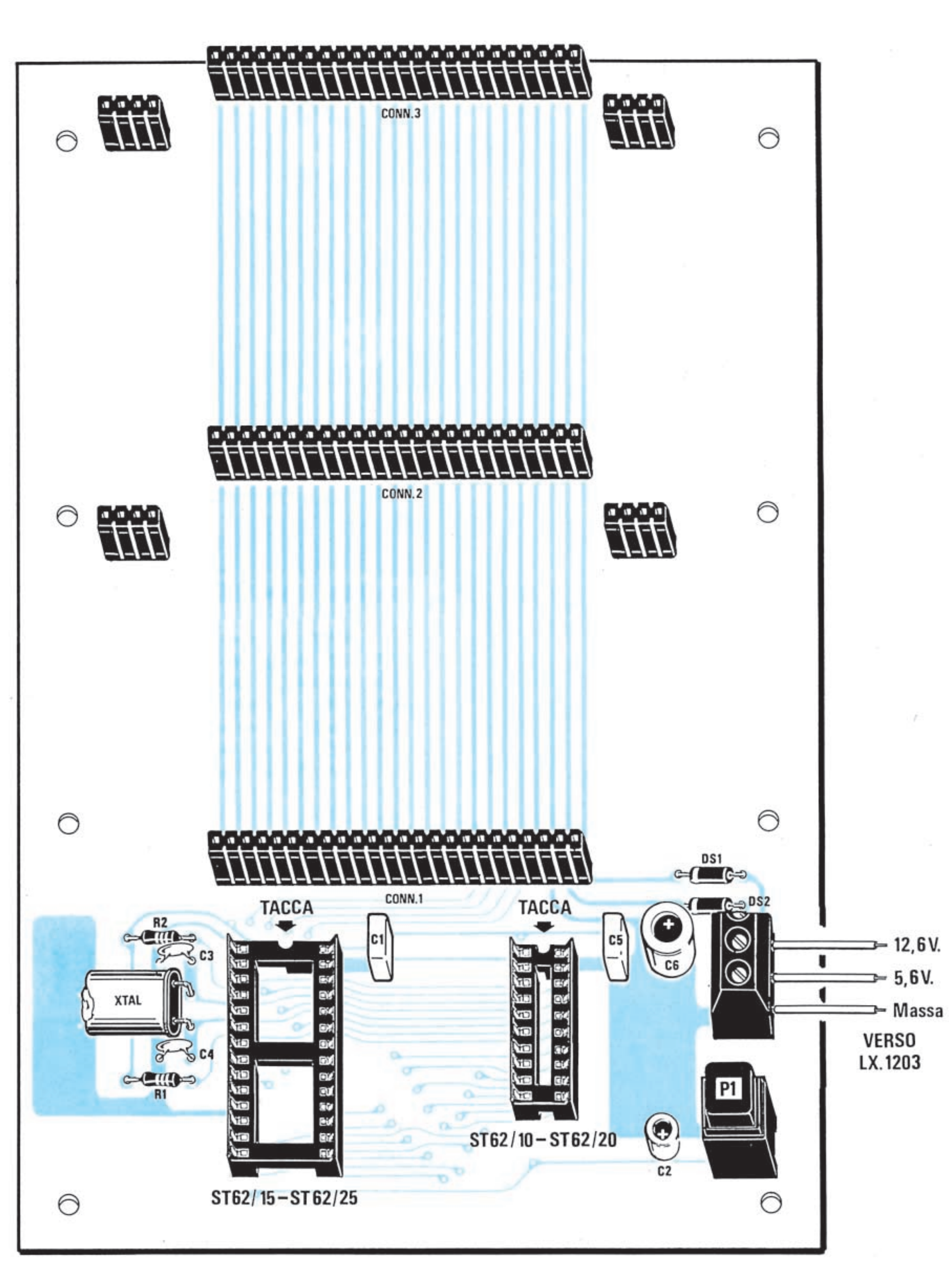


Fig.3 Schema pratico di montaggio del bus. Potete sostituire i due comuni zoccoli per i microprocessori ST6 con i più comodi, ma costosi textool (vedi fig.1).

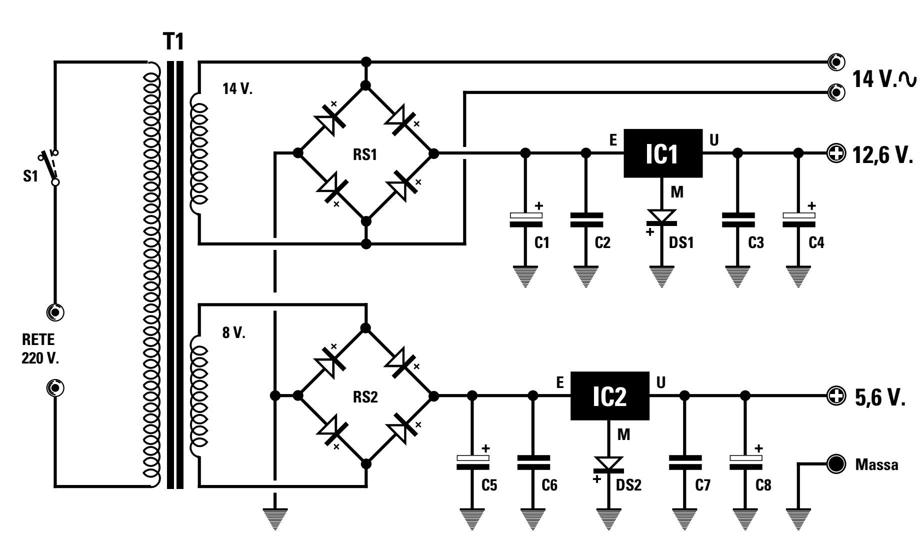


Fig.4 Schema elettrico dello stadio di alimentazione da utilizzare per il bus.

delli risultano identiche sarà possibile sostituirli senza problemi.
Il motivo per cui tutti preferiscono i microprocessori **riprogrammabili** da **4 K** è ovvio.
I microprocessori **riprogrammabili** servono e vengono utilizzati unicamente per **testare** i programmi e quindi possono essere **cancellati** e riutilizzati per provare altri programmi.
Dopo aver verificato che il programma funziona, si può definitivamente trasferirlo sui microprocessori **non riprogrammabili** tipo **ST62T10 - ST62T20** o **ST62T15 - ST62T25**.
Poiché un microprocessore **riprogrammabile** viene riutilizzato un **centinaio** di volte, si preferisce acquistarne uno da **4 K**, perché può essere usato sia per i programmi che occupano **1 - 1,5 - 2 K** sia per quelli che occupano **2,5 - 3 - 4 K**.

ELENCO COMPONENTI LX.1203

C1 = 2.200 mF elettr. 35 volt
C2 = 100.000 pF poliestere
C3 = 100.000 pF poliestere
C4 = 100 mF elettr. 35 volt
C5 = 2.200 mF elettr. 35 volt
C6 = 100.000 pF poliestere
C7 = 100.000 pF poliestere
C8 = 100 mF elettr. 35 volt
DS1 = diodo 1N.4007
DS2 = diodo 1N.4007
RS1 = ponte raddr. 100 V. 1 A.
RS2 = ponte raddr. 100 V. 1 A.
IC1 = uA.7812
IC2 = uA.7805
T1 = trasformatore 25 watt (T025.01)
sec. 14 V. 1 A. - 8 V. 1 A.
S1 = interruttore

SCHEMA ELETTRICO scheda BUS

Per testare i programmi abbiamo realizzato una scheda in cui si possono utilizzare sia i microprocessori da 20 piedini sia quelli da 28 piedini.
Per questa scheda abbiamo inoltre realizzato un **bus** che porta tutti i segnali del microprocessore ai connettori femmina sui quali potrete inserire diversi tipi di schede, ad esempio con dei **display** a **7 segmenti**, oppure con un **display LCD** o con dei **relè** o ancora con dei **Triac**.

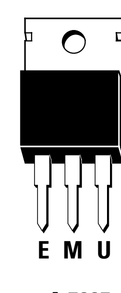


Fig.5 Connessioni dei due integrati stabilizzatori di tensione.

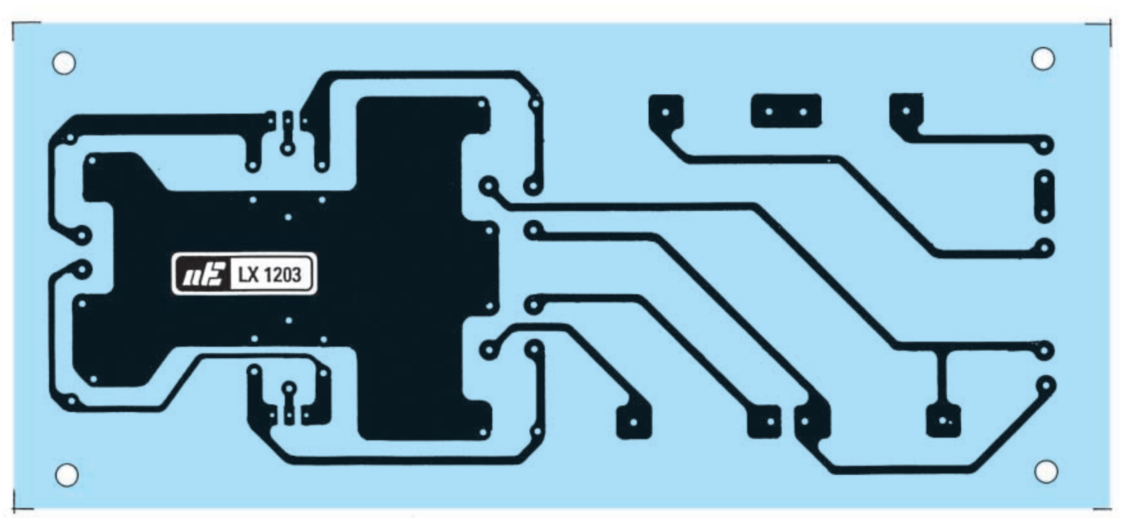


Fig.6 Disegno del circuito stampato dello stadio di alimentazione siglato LX.1203 visto dal lato rame. Nota: il disegno è stato leggermente ridotto per farlo rientrare nella pagina; le sue misure reali sono Lunghezza = mm 160 ed Altezza = mm 70.

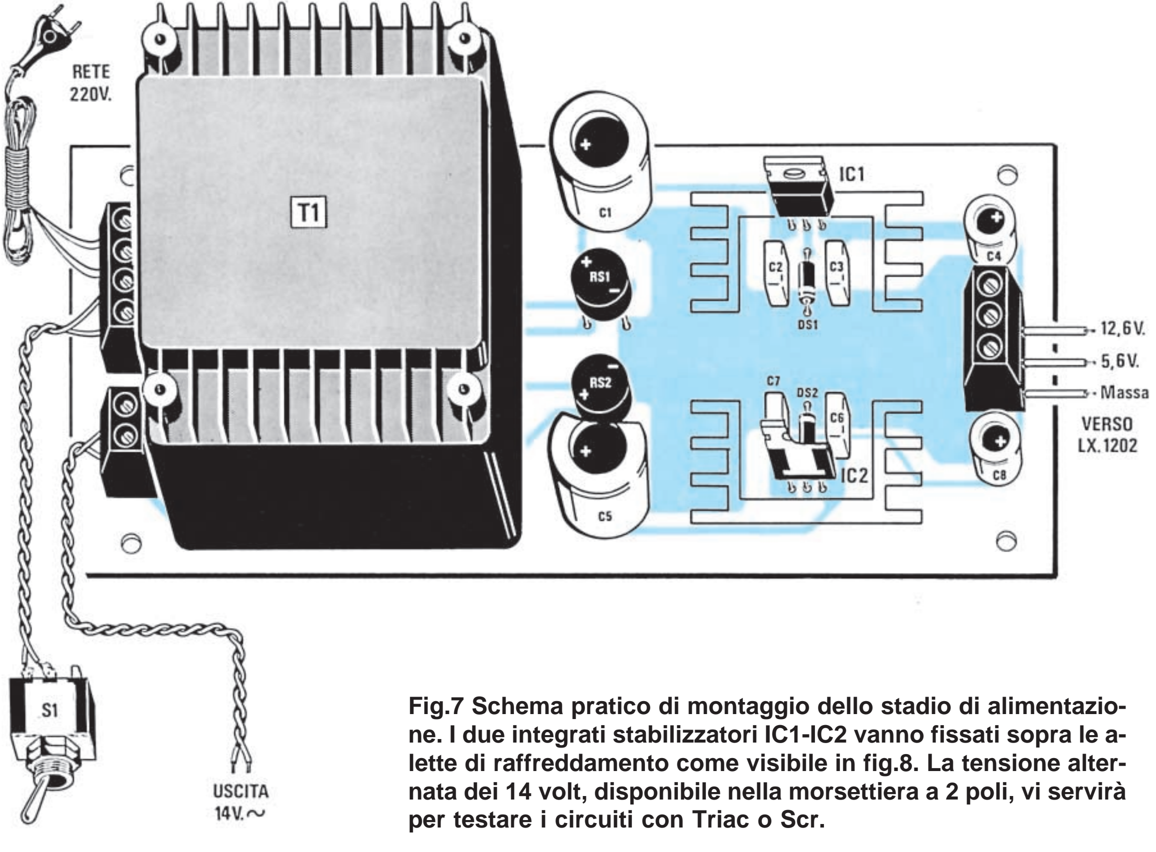


Fig.7 Schema pratico di montaggio dello stadio di alimentazione. I due integrati stabilizzatori IC1-IC2 vanno fissati sopra le alette di raffreddamento come visibile in fig.8. La tensione alternata dei 14 volt, disponibile nella morsettiera a 2 poli, vi servirà per testare i circuiti con Triac o Scr.

Per la descrizione dello schema elettrico, visibile in fig.2, iniziamo proprio dai due zoccoli, che, ovviamente, dovranno essere utilizzati **solo uno** per volta.

Dunque se avete già inserito un microprocessore in uno zoccolo, per inserirne un altro nel secondo zoccolo dovrete **togliere** il primo.
Tutti i terminali dei due zoccoli sono collegati tra loro in modo da sfruttare per entrambi lo stesso **quarzo** per il **clock**, lo stesso pulsante di **reset** ed ovviamente la stessa alimentazione.
Anche tutte le porte d'ingresso/uscita sono collegate in **parallelo** e qui è necessario ricordare che nello zoccolo che riceverà i microprocessori **ST62E10 - ST62E20** la **porta A** inizia da **A0** e termina ad **A3**, la **porta B** inizia da **B0** e termina a **B7**, e che manca la **porta C**.
Nello zoccolo che riceverà i microprocessori **ST62E15 - ST62E25** risultano presenti tutte le **porte** da **A0** ad **A7** e da **B0** a **B7**, e la **porta C** inizia da **C4** e termina a **C7**.

Tutti gli **ingressi/uscite** raggiungono i connettori femmina **CONN.1 - CONN.2 - CONN.3** nei quali andranno inserite le **schede sperimentali**.
Sempre sugli stessi **connettori** risultano presenti le piste di alimentazione, cioè **5,6 volt positivi - 12,6 volt positivi** e la **massa**.
I diodi al silicio **DS1 - DS2**, posti in serie sui due ingressi di alimentazione, sono stati inseriti per evitare di danneggiare il **micro** nel caso venisse applicata su questi terminali una polarità opposta a quella richiesta.
Poiché questi diodi introducono una tensione di circa **0,6 volt**, applicando sull'ingresso una tensione di **5,6 volt** e **12,6 volt**, in uscita si otterranno esattamente **5 volt** e **12 volt**.

REALIZZAZIONE PRATICA

Sul circuito stampato siglato **LX.1202** dovete montare i pochi componenti visibili in fig.3.
Come noterete, nel kit abbiamo inserito due zoccoli, uno da **20** e l'altro da **28** piedini, ma qui dobbiamo aprire una piccola parentesi.
Poiché questo progetto verrà utilizzato anche da piccole e medie Industrie per **testare** i loro microprocessori, noi consigliamo di utilizzare, in sostituzione degli zoccoli inseriti nel kit, degli zoccoli **textool** provvisti di una levetta di bloccaggio (vedi fig.1).
Usando questi zoccoli risulterà più semplice e veloce inserire e togliere i microprocessori, ma questo vantaggio costerà 55.000 lire in più.

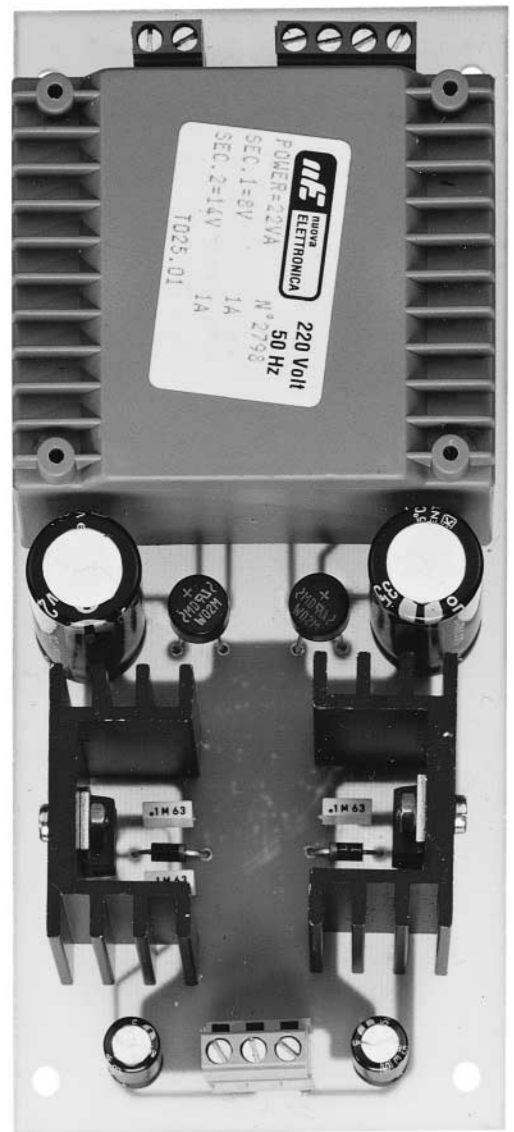


Fig.8 Come si presenta a montaggio ultimato lo stadio di alimentazione che vi fornisce tutte le necessarie tensioni per alimentare il nostro bus.
Vi consigliamo di racchiudere l'alimentatore dentro un mobile plastico e a questo proposito vi indichiamo il mobile codificato MTK06.22 del costo di € 6,97

Come visibile in fig.3, sullo stampato dovrete montare i due zoccoli, il pulsante di **reset**, il **quarzo**, i **connettori** per le schede sperimentali ed i pochi componenti passivi, cioè resistenze, condensatori ed i due diodi al silicio, rispettando il verso della **fascia bianca** riportata sul loro corpo.
I quattro connettori femmina a **4 terminali**, che sullo stampato risultano isolati dal circuito, serviranno come **punto di appoggio** per le schede sperimentali che inserirete nel **bus**.
Questi connettori vi serviranno anche per evitare di inserire le schede sperimentali in senso inverso al richiesto.

ALIMENTAZIONE

Le tensioni richieste per alimentare questo **bus** devono risultare, come già anticipato, di **5,6** e di **12,6 volt**.
Come visibile in fig.4, le due tensioni prelevate dai due secondari del trasformatore **T1** vengono raddrizzate da **RS1** ed **RS2**, poi stabilizzate a **5,6 volt** dall'integrato **uA.7805** (vedi **IC2**) e a **12,6 volt** dall'integrato **uA.7812** (vedi **IC1**).
Per prelevare sull'uscita di questi due integrati una tensione di **5,6 volt** anziché di **5 volt** ed una tensione di **12,6 volt** anziché di **12 volt**, abbiamo inserito tra il terminale **M** e la **massa** un diodo al silicio (vedi **DS1 - DS2**).
Dallo stesso trasformatore si preleva anche una **tensione alternata** di circa **14 volt**, che potrà servire per testare i circuiti che utilizzano dei **Triac**.
Anche se per queste prove è possibile applicare sulla loro uscita una tensione di **220 volt alternati**, noi ve lo **sconsigliamo**, perché se inavvertitamente toccaste con le mani le **piste** del circuito stampato potrebbe risultare **molto pericoloso**.

REALIZZAZIONE PRATICA ALIMENTATORE

Sul circuito stampato monofaccia siglato **LX.1203** monterete tutti i componenti visibili in fig.7 cercando come sempre di rispettare la polarità dei terminali dei diodi al silicio, dei ponti raddrizzatori e dei condensatori elettrolitici.
Come potete osservare anche dalle foto, i due integrati stabilizzatori vanno fissati sopra due alette di raffreddamento.
Per le tensioni d'uscita inserite una **morsettiera** a **3 poli** dalla quale potrete prelevare le tensioni **stabilizzate** di **12,6 volt - 5,6 volt** più il filo della **massa**, ed una **morsettiera** a **2 poli** dalla quale potrete prelevare una tensione **alternata** di circa **14 volt** che vi servirà per collaudare i programmi che "pilotano" i diodi **Triac**.
Vi conviene racchiudere l'alimentatore dentro un qualsiasi mobile e a tal proposito vi consigliamo il mobile siglato **MTK06.22.**
Per le tensioni d'uscita utilizzate dei fili di diverso **colore** così da poter subito stabilire il valore della tensione presente e non correre il rischio di invertirli quando li collegherete alla morsettiera del **bus**.
Tanto per fare un esempio, per la **massa** potrete scegliere il colore **nero**, per i **5,6 volt** il colore **giallo** o **marrone** e per i **12,6 volt** il colore **rosso** o **arancio**.
Per i due fili dell'**alternata** potrete usare due fili **bianchi** oppure di un colore completamente diverso da quello scelto per le altre uscite.

Per le schede **sperimentali** vi rimandiamo all'articolo pubblicato su questa stessa rivista.

COSTO DI REALIZZAZIONE

Il solo Bus siglato LX.1202 completo di circuito stampato, quarzo, zoccoli normali, connettori, pulsante, cioè tutti i componenti visibili in fig.3 (sensa textool) ....€ 25,80

Costo del solo stampato LX.1202 ....€ 17,04
Costo di uno zoccolo textool a 20 piedini € 19,63
Costo di uno zoccolo textool a 28 piedini € 28,41

Il solo stadio di alimentazione siglato LX.1203 completo di circuito stampato, due integrati stabilizzatori, alette di raffreddamento, due ponti raddrizzatori, cordone di alimentazione e relativo trasformatore ....€ 25,80

Costo del solo stampato LX.1203 ....€ 4,34

Vi consigliamo di racchiudere lo stadio di alimentazione dentro un mobile plastico e a tale scopo vi proponiamo il modello MTK06.22.

Costo del mobile MTK06.22 ....€ 6,97

Ai prezzi riportati andranno aggiunte le sole spese di spedizione a domicilio.

Inserendo queste due schede nel **bus** siglato **LX.1202**, riportato su questo numero, e collocando nel suo zoccolo un **microprocessore ST6**, che voi stessi potrete programmare **copiando** uno dei programmi di **esempio** presenti nel dischetto che vi forniremo, potrete ottenere **orologi - contasecondi - timer - cronometri - contaimpulsi** ecc.
Poiché i nostri programmi di **esempio** si possono facilmente modificare, riuscirete in breve tempo a capire come scriverne altri per ottenere funzioni che noi attualmente non abbiamo previsto.
Queste schede risulteranno quindi utilissime per **testare** tutti i programmi che scriverete, perché vedrete dal **vivo** se appaiono i **numeri** desiderati e se i **relè** si eccitano o si diseccitano nei tempi previsti.
A queste schede ne seguiranno via via altre che utilizzeranno i display **LCD** ed i diodi **TRIAC**.
Entrerete così in possesso di un valido banco di **test** per tutti i tipi di programmi per **microprocessori ST6**.

Sul piedino **22** di **data** vanno inviati dei **bit seriali**, che l'integrato **M.5450** convertirà in **bit paralleli** necessari per accendere i **segmenti** dei quattro display.
La sequenza dei **bit** necessari per accendere i **segmenti** dei display deve essere preceduta da un **bit di start** (vedi fig.1).
In pratica all'integrato giunge una sequenza di **37 bit** come qui sotto riportato:

1 bit di **Start**
32 bit per accendere i **display**
2 bit per accendere i **diodi led**
2 bit di fine **caricamento**

I segmenti dei quattro **display** e dei due **diodi led** si accendono soltanto quando all'integrato sono giunti tutti i **37 bit**, vale a dire tutta la sequenza sopra riportata compresi gli ultimi **2 bit** di fine **caricamento**.

SCHEDA TEST per ST6

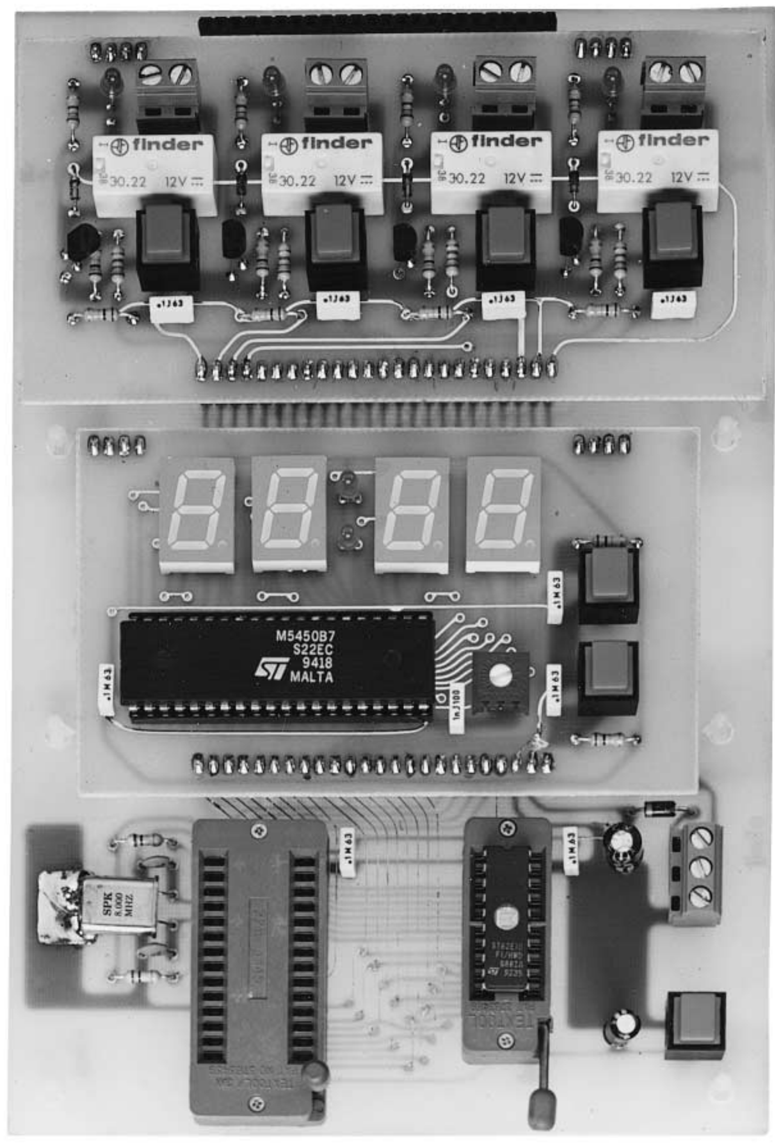
SCHEDA DISPLAY

Lo schema elettrico di questa scheda, riportato in fig.6, è molto semplice, perché utilizza il solo integrato **M.5450** (vedi **IC1**) necessario per pilotare **4 display**.
Nel caso realizzaste un **orologio**, i due **pulsanti** presenti nel circuito vi potranno servire per mettere a punto le **ore** ed i **minuti**, mentre i due **diodi led** potrebbero visualizzare i **secondi** oppure potreste impiegarli per altre funzioni da assegnare tramite software.
Il trimmer siglato **R1** serve soltanto per variare la **luminosità** dei display.
Poiché desideriamo che il lettore sappia come si riesca a far accendere un qualsiasi numero sui **4 display** collegando due soli fili all'integrato **M.5450** (vedi piedini **22-21**), dobbiamo a questo punto spiegare come vanno gestiti questi piedini.
Sul piedino **21** di **clock** va applicata una **frequenza** ad onda quadra che faremo generare dall'ST62 inserendo nel programma le due istruzioni **Set-Res** (vedi righe 119-120 nel programma **DISPLAY.ASM** presente nel dischetto).

Se guardate la fig.1, in cui sono riportati tutti i **7 segmenti** di un display contrassegnati da una **lettera**, potrete ricavare dalla **Tabella N.1** i **bit** che devono giungere al **piedino 22** dell'integrato per accendere i vari **segmenti**.
Il **bit**, come già sapete, è una cifra **binaria** che può assumere un **livello logico 0** oppure un **livello logico 1**.
Per accendere i **segmenti** interessati, i **bit** che entrano sul **piedino 22** devono avere un **livello logico 1**.
E' poi l'integrato che provvede a commutare le sue uscite a **livello logico 0** in modo che i **segmenti** ed i **diodi led** risultino alimentati.

Quindi se volessimo accendere il numero **7** sul display **N.4**, sull'ingresso di **IC1** dovremmo far giungere la sequenza di bit visibile in fig.2.
Se invece volessimo accendere il numero **7** sul display **N.1**, dovremmo far giungere sull'ingresso di **IC1** la sequenza di bit visibile in fig.3.
Nel dischetto che vi forniremo abbiamo inserito un programma chiamato **DISPLAY** per mostrarvi come si possa accendere qualsiasi **numero** in uno dei **quattro display**.

Potrete utilizzare queste due schede come orologio, contasecondi, timer, cronometro, contaimpulsi, e se questo non bastasse potrete eccitare, nei tempi da voi desiderati, dei relè per pilotare una cicalina o per alimentare una qualsiasi apparecchiatura elettrica.



SCHEDA RELÈ

Lo schema elettrico di questa scheda, riportato in fig.9, utilizza solo **4 relè** pilotati da altrettanti **transistor**.
In teoria avremmo potuto inserire ben **20 relè** utilizzando così tutte le porte **A - B - C**, ma poiché questa scheda viene inserita nel **bus** assieme alla scheda dei **display**, che già utilizza le **porte B0 - B1 - B2 - B3**, non potevamo servirci della stessa **porta** per accendere un display ed eccitare un relè.
I vari relè si eccitano quando sulle **porte B4 - B5 - B6 - B7** è presente un **livello logico 1**, che, polarizzando le Basi dei transistor, li portano in conduzione.
A relè **eccitato** si accende il **diodo led** applicato

TABELLA N. 1

DISPLAY N. 4

Bit	piedino IC1	segmento display
9	10	A
10	9	B
11	8	C
12	7	D
13	6	E
14	5	F
15	4	G
16	3	punto

DISPLAY N. 3

Bit	piedino IC1	segmento display
1	18	A
2	17	B
3	16	C
4	15	D
5	14	E
6	13	F
7	12	G
8	11	punto

DISPLAY N. 2

Bit	piedino IC1	segmento display
25	33	A
26	32	B
27	31	C
28	30	D
29	29	E
30	28	F
31	27	G
32	26	punto

DISPLAY N. 1

Bit	piedino IC1	segmento display
17	2	A
18	40	B
19	39	C
20	38	D
21	37	E
22	36	F
23	35	G
24	34	punto

ai suoi capi, quindi se sulla **morsettiera d'uscita** applicheremo una lampadina o un motorino, alimentati con una qualsiasi tensione esterna sia in **continua** sia in **alternata**, la lampada si **accenderà** ed il **motorino** inizierà a **girare**.
I pulsanti **P1 - P2 - P3 - P4**, collegati alle **porte A0 - A1 - A2 - A3**, sono stati inseriti per diseccitare o eccitare **manualmente** uno dei quattro relè.

REALIZZAZIONE PRATICA DISPLAY

Sul circuito stampato siglato **LX.1204** monterete i pochi componenti visibili in fig.7.
Per iniziare consigliamo di inserire lo **zoccolo** per l'integrato **IC1**, poi, dal lato opposto dello stampato, inserite il connettore **maschio** ad 1 fila provvisto di **24 terminali** e gli altri due connettori maschi, sempre ad 1 fila, provvisti di **4 terminali**, che in seguito vi serviranno per innestare questa scheda sui **connettori femmina** della scheda **bus** siglata **LX.1202**.
Dopo aver stagnato questi componenti, potete inserire le due resistenze, i quattro condensatori, il trimmer **R1** ed i due pulsanti **P1 - P2**.
Proseguendo nel montaggio inserite i due **diodi led** rivolgendo il terminale **più lungo**, cioè l'**Anodo**, verso sinistra.
Se invertirete questo terminale, i diodi led non si accenderanno.
Per completare il montaggio dovrete saldare direttamente sul circuito i quattro **display**, controllando che il lato in cui è presente il **punto decimale** risulti rivolto verso il basso, cioè verso l'integrato **IC1**.
Dopo aver inserito tutti i componenti, dovete inserire nel suo zoccolo l'integrato **M.5450** rivolgendo la sua tacca di riferimento ad **U** verso sinistra, come risulta visibile in fig.7.

REALIZZAZIONE PRATICA RELÈ

Sul circuito stampato siglato **LX.1205** monterete tutti i componenti visibili in fig.10.
Per il montaggio vi consigliamo di iniziare inseren-

DIODO LED

Bit	piedino IC1	diodi led
34	24	DL2
33	25	DL1

Con queste tabelle potrete sapere quali bit devono giungere sul piedino 22 dell'integrato M.5450 (vedi fig.4) per accendere i sette segmenti dei quattro display ed i due diodi led.

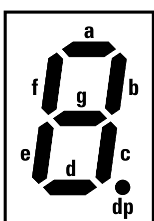


Fig.1 Sul piedino 22 dell'integrato M.5450 giunge un primo bit di Start. A questo seguono 32 bit per accendere i segmenti dei display, 2 bit per accendere i led e 2 bit di fine caricamento. I sette segmenti del display sono identificati da una lettera (vedi A-B-C-D-E-F-G) quindi per visualizzare il numero 7 dovrete alimentare i segmenti A-B-C e per visualizzare il numero 3 dovrete alimentare i segmenti A-B-G-C-D.

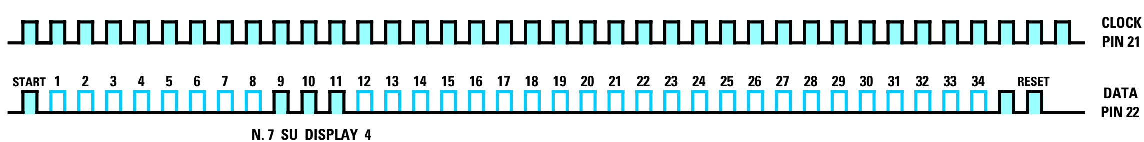


Fig.2 Per accendere il numero 7 sul display 4 dovrete far giungere sull'integrato M.5450 questa sequenza di bit seriali. Con i bit 9-10-11 verranno alimentati i soli segmenti A-B-C del display n.4 (vedi Tabella posta sulla sinistra).

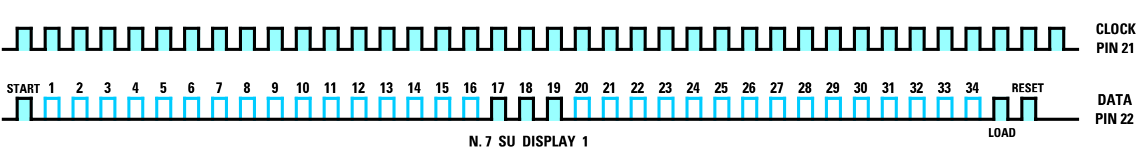


Fig.3 Per accendere il numero 7 sul display 1 dovrete far giungere sull'integrato M.5450 questa sequenza di bit seriali. Con i bit 17-18-19 verranno alimentati i soli segmenti A-B-C del display n.1 (vedi Tabella posta sulla sinistra).

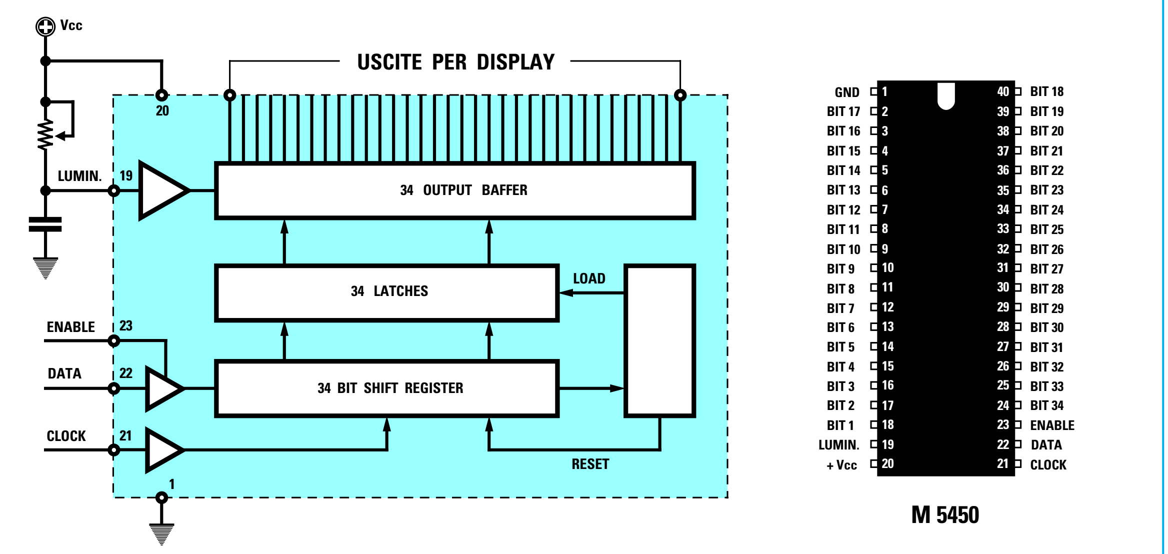


Fig.4 Schema a blocchi dell'integrato M.5450 e connessioni dei piedini sullo zoccolo viste da sopra. Entrando con un segnale seriale nel piedino 22 di questo integrato voi potrete accendere i segmenti dei 4 display (vedi fig.6).

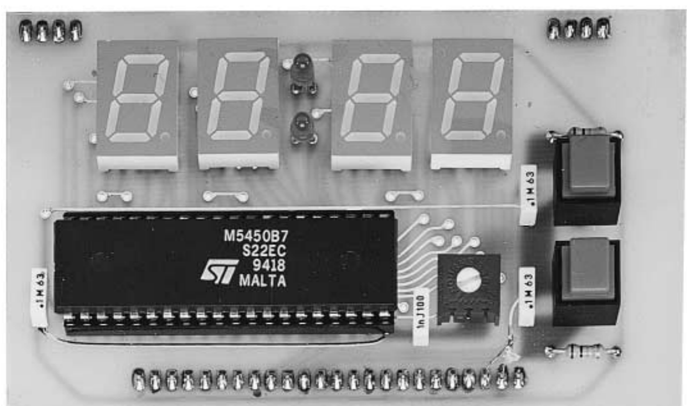


Fig.5 Foto della scheda display notevolmente rimpicciolita per motivi di spazio.

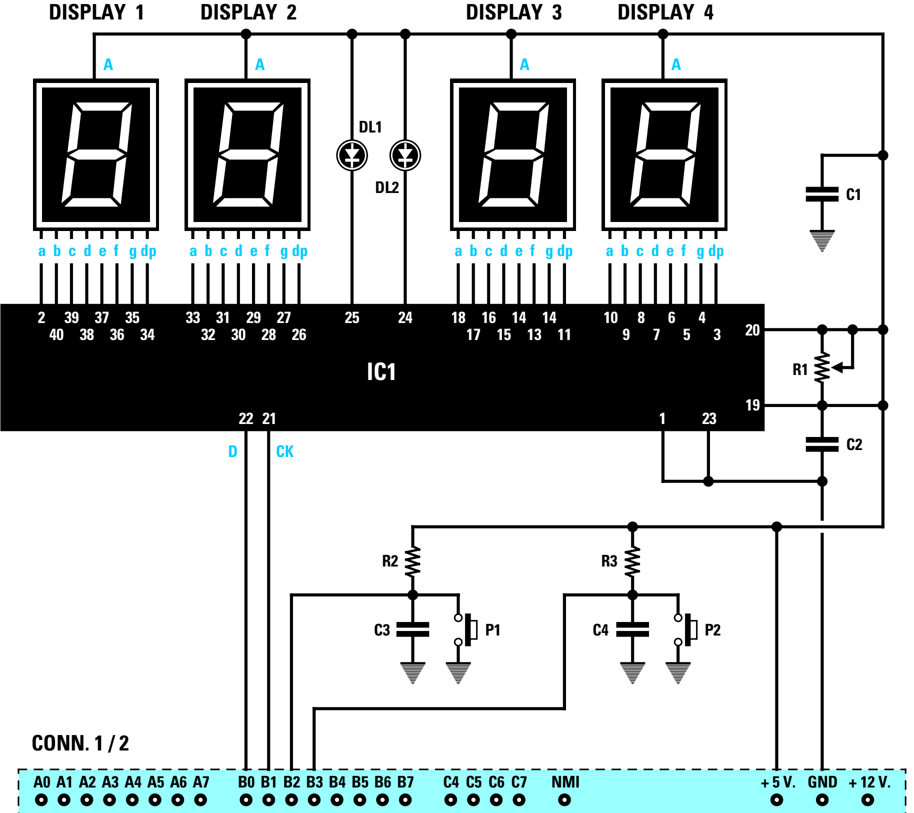


Fig.6 Schema elettrico della scheda display. I segmenti dei display si accendono quando il piedino d'uscita dell'integrato si porta a livello logico 0.

ELENCO COMPONENTI LX.1204

R1 = 50.000 ohm trimmer
R2 = 10.000 ohm 1/4 watt
R3 = 10.000 ohm 1/4 watt
C1 = 100.000 pF poliestere
C2 = 1.000 pF poliestere
C3 = 100.000 pF poliestere
C4 = 100.000 pF poliestere
DL1 = diodo led
DL2 = diodo led
DISPLAY1-4 = display tipo BS.A501
IC1 = M.5450
P1-P2 = pulsanti
CONN.1/2 = connettore 24 poli

do, dal lato opposto a quello dei componenti, il connettore **maschio** ad 1 fila provvisto di **24 terminali** e gli altri due connettori maschi, sempre ad 1 fila, provvisti di **4 terminali**, che in seguito vi serviranno per innestare questa scheda sui **connettori femmina** della scheda **bus** siglata **LX.1202**.
Ora voltate lo stampato e sulla parte inferiore del circuito inserite tutte le **resistenze**, poi i **diodi al silicio** non dimenticando di rivolgere il lato contornato da una **fascia nera** verso le morsettiere di uscita.
Proseguendo nel montaggio inserite tutti i **condensatori**, poi tutti i **transistor** rivolgendo la parte **piatta** del loro corpo verso sinistra come visibile nello schema pratico di fig.10.
Completata questa operazione, potete inserire i quattro **pulsanti**, i quattro **relè** e le quattro **morsettiere** a **2 poli**.
Per ultimi montate i **diodi led** non dimenticando di rivolgere il terminale **più lungo** dell'**Anodo** verso i relè.
Le quattro morsettiere presenti nello stampato sono collegate ai **contatti** dei relè, perciò quando il relè si **ecciterà** il contatto si chiuderà e pertanto lo potrete utilizzare come **interruttore** per accendere **lampadine**, alimentare **motorini** o **trasformatori** oppure dei servorelè a **220 volt**.

INSERIMENTO SCHEDE nel BUS

Potete inserire queste schede nel **bus** siglato **LX.1202** indifferentemente su uno dei due connettori **femmina** presenti sullo stampato.

L'ultimo connettore **femmina**, posto sulla parte superiore dello stampato **LX.1202**, è stato previsto nell'eventualità che vogliate collegare un vostro personale circuito stampato oppure per prolungare il **bus**.

I PROGRAMMI

Il nuovo dischetto che vi forniremo è in pratica lo stesso che abbiamo distribuito in precedenza (vedi rivista N.172/173), con la differenza che oltre ai programmi:

CONTA.ASM
LED.ASM
LOTTO.ASM
STANDARD.ASM

abbiamo aggiunto questi nuovi files:

RELE.ASM
DISPLAY.ASM
OROLOGIO.ASM
CRONOMET.ASM
TEMPOR.ASM
TIMER.ASM

Ogni riga di programma è stata completata da un **commento** che spiega la funzione dell'istruzione e quindi permette di sapere come modificare il programma per fargli compiere una funzione diversa da quella per cui era stato scritto.

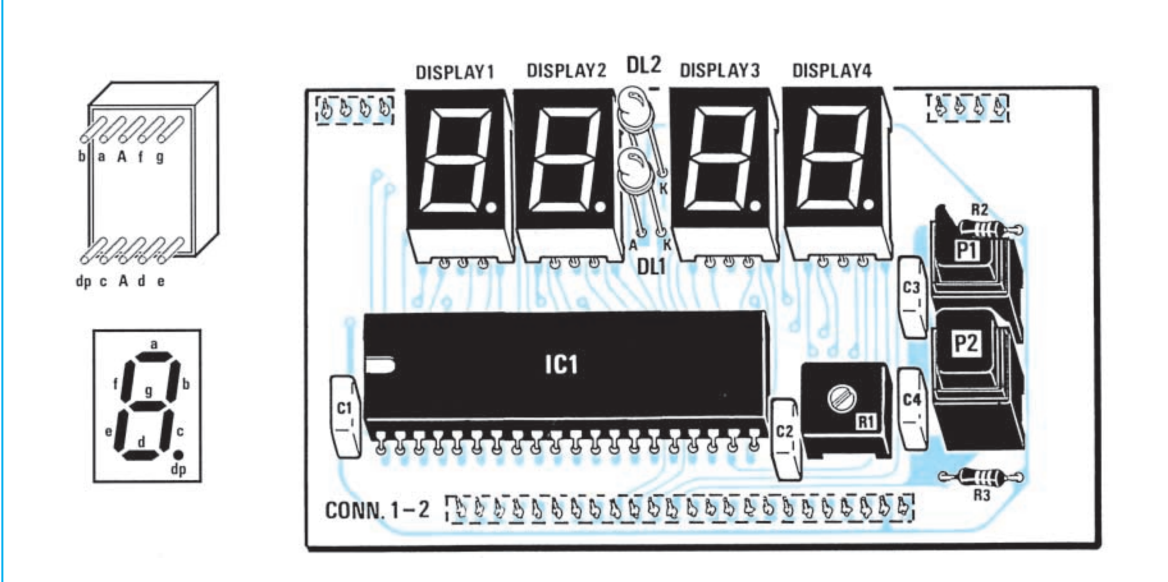


Fig.7 Schema pratico di montaggio della scheda display siglata LX.1204. Quando inserite i display rivolgete il loro "punto decimale" verso l'integrato IC1.

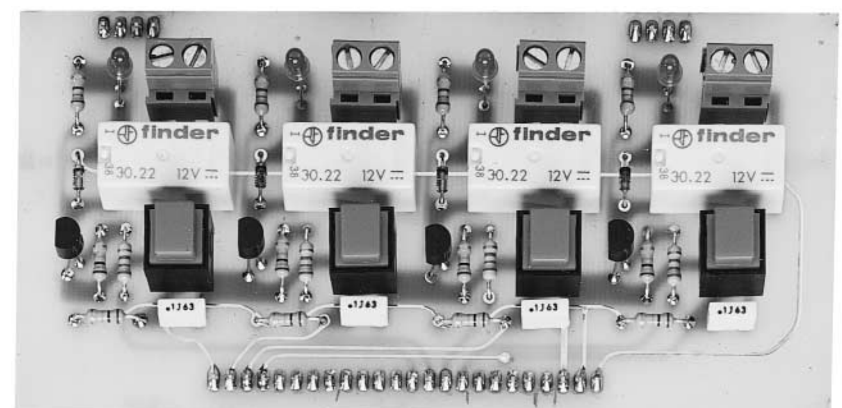


Fig.8 Foto ridotta della scheda relè.

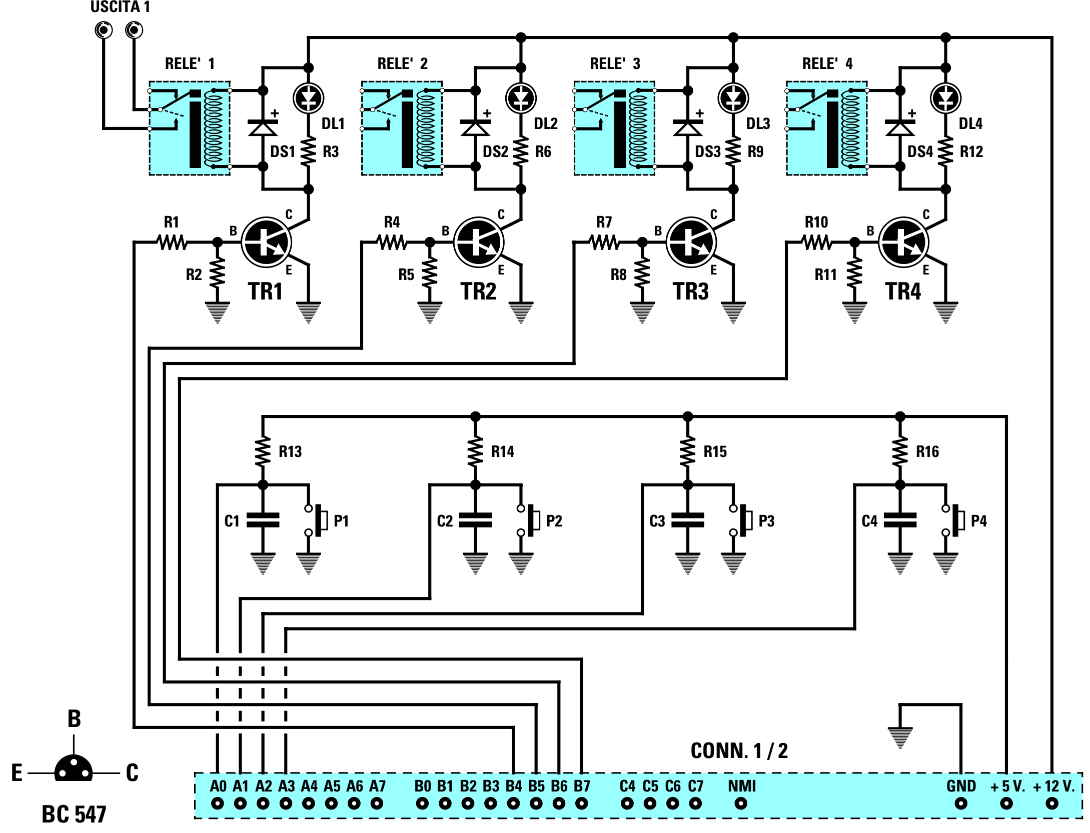


Fig.9 Schema elettrico della scheda relè siglata LX.1205 e connessioni del transistor BC.547 viste da sotto. Potrete utilizzare i relè per alimentare motorini o lampade.

ELENCO COMPONENTI LX.1205

R1 = 2.200 ohm 1/4 watt
R2 = 10.000 ohm 1/4 watt
R3 = 680 ohm 1/4 watt
R4 = 2.200 ohm 1/4 watt
R5 = 10.000 ohm 1/4 watt
R6 = 680 ohm 1/4 watt
R7 = 2.200 ohm 1/4 watt
R8 = 10.000 ohm 1/4 watt
R9 = 680 ohm 1/4 watt
R10 = 2.200 ohm 1/4 watt
R11 = 10.000 ohm 1/4 watt
R12 = 680 ohm 1/4 watt
R13 = 10.000 ohm 1/4 watt
R14 = 10.000 ohm 1/4 watt
R15 = 10.000 ohm 1/4 watt
R16 = 10.000 ohm 1/4 watt
C1-C4 = 100.000 pF poliestere
DS1-DS4 = diodi 1N.4150
DL1-DL4 = diodi led
TR1-TR4 = NPN tipo BC.547
RELE'1-4 = relè 12 volt
P1-P4 = pulsanti
CONN.1/2 = connettore 24 poli

Tanto per fare un esempio, se nel programma **TIMER** volete variare i tempi di eccitazione dei **relè**, troverete spiegato in quale riga va modificata l'istruzione e quale numero occorre inserire.
Se volete che il programma **TIMER** ecciti un **relè** per far **suonare** un campanello o per accendere una **caldaia** ad una precisa ora, vi verrà spiegato quale riga modificare per migliorare in base alle vostre personali esigenze la funzionalità del programma.

TRASFERIMENTO file nell'HARD-DISK

Una volta inserito il dischetto nel drive floppy, per trasferire tutti i suoi files nell'Hard-Disk dovete scrivere:

C:\>A: poi Enter
A:\>INSTALLA poi Enter

Non usate mai l'istruzione **Copy** del **Dos** o altri comandi analoghi del **PCshell - PCtools - Norton** o il **File Manager** di **Windows**, perché il programma **INSTALLA** presente nel dischetto provvede a **scompattare** automaticamente i files inseriti.
Dopo aver pigiato Enter apparirà la scritta:

Directory C:\ST6

Se il computer vi comunica che questa directory **esiste già**, non preoccupatevi e premete due volte il tasto **S**.
A questo punto inizia la **scompattazione** dei files (vedi fig.11) ed al termine dell'operazione vedrete apparire sul monitor la scritta:

Buon divertimento

Premendo un tasto qualsiasi, sul monitor apparirà la scritta:

C:\ST6>

Nel dischetto che vi abbiamo preparato, abbiamo incluso un semplice **Editor** che vi sarà molto utile per **visualizzare** tutte le righe di ogni programma, e, se lo desiderate, per **modificarle**.
Potrete usare lo stesso **Editor** per **scrivere** nuovi programmi, ed anche per **assemblarli** prima di **trasferirli** nella memoria del microprocessore **ST6** tramite il nostro programmatore siglato **LX.1170** presentato sulla rivista **N.172/173**.

Tutte le istruzioni per utilizzare l'**Editor** sono state già descritte nella rivista N.172/173, in particolare nell'articolo sul **Circuito Test** a pag.56, quindi vi consigliamo di rileggere attentamente questo articolo.

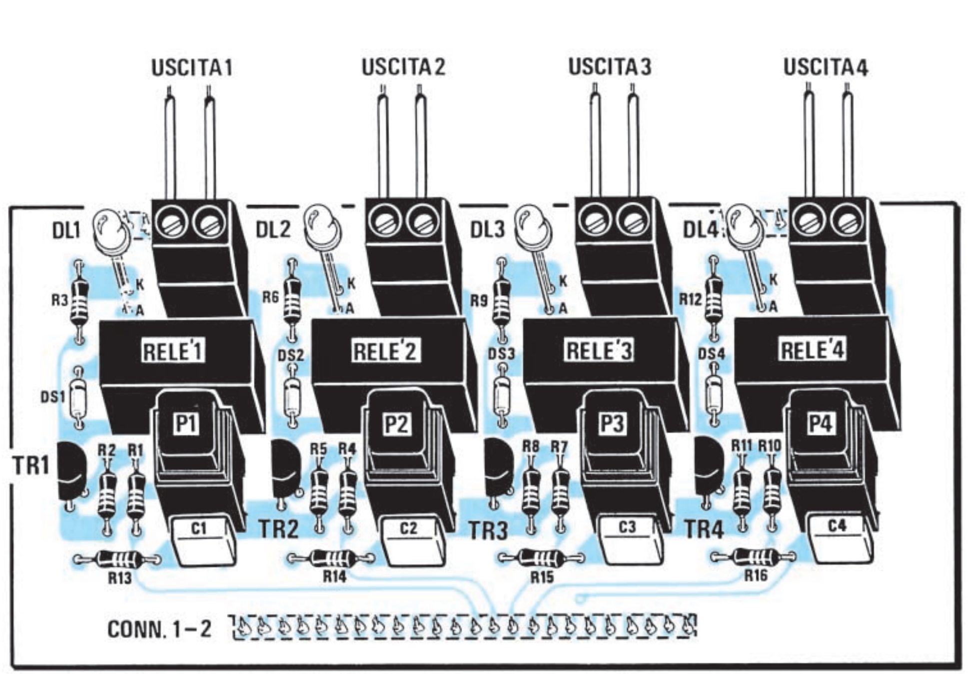


Fig.10 Schema pratico di montaggio della scheda LX.1205. Potrete modificare il programma di base che troverete nel dischetto in modo da adattarlo alle vostre esigenze.

Nelle pagine seguenti ci limiteremo infatti a descrivervi le istruzioni più importanti.

DUPLICARE un FILE con NOME diverso

Anziché **modificare** i files che trovate nel dischetto, vi consigliamo di ricopiarli nell'**Hard-Disk** con un nome **diverso**, e di apportare su questo **nuovo** file le modifiche che riterrete opportune.
In questo modo avrete sempre a disposizione, in caso di bisogno, il programma originale.
Ammesso che vogliate **duplicare** il file **DISPLAY**, dovrete innanzitutto scegliere un **nome** che non abbia più di **8 caratteri**, ad esempio **DPLPROVA**.
Quando siete nel **menu** principale del nostro **Editor**:

premete **ALT+F**
poi pigiate **D**

apparirà la scritta **C:\ST6>**
A questo punto potete scrivere l'istruzione:

C:\ST6>Copy DISPLAY.ASM DPLPROVA.ASM

poi premete Enter.
Dopo pochi secondi apparirà la scritta:

1 file copiato

Per ritornare al **menu** principale dovrete scrivere:

C:\ST6>EXIT poi Enter

Ora con il tasto funzione **F3** potrete richiamare il file **DPLPROVA** sul quale potrete apportare tutte le modifiche che riterrete opportune.

PER MODIFICARE un PROGRAMMA

Premendo il tasto funzione **F3**, quando siete nel **menu** principale, appare sul monitor una finestra con l'intero elenco dei **programmi**.
Premendo prima il tasto **TAB** e poi i tasti **freccia** potrete portare il cursore sui nomi dei programmi.
Quando il **cursore** si trova sul programma che volete modificare, premete **Enter** ed apparirà il **listato** completo del programma selezionato.

Per **cancellare una sola parola**
pigiate i tasti **CTRL+T**.

Per **cancellare un'intera riga**
pigiate i tasti **CTRL+Y**.

Dopo aver **corretto**, **modificato**, **cancellato** una **riga** o una **parola** dovrete **ricordarvi** di premere il tasto funzione **F2** per **memorizzare** tutte le modifiche che avete effettuato.

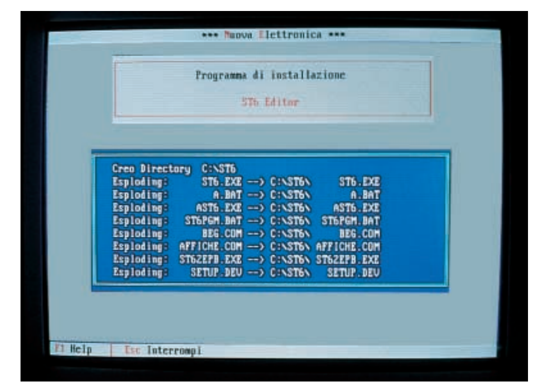


Fig.11 Tutti i files presenti nel dischetto verranno automaticamente scompattati quando li memorizzerete nell'Hard-Disk.

ASSEMBLAGGIO del PROGRAMMA

Un programma **corretto**, **modificato** o **riscritto**, si può trasferire in un micro **ST6** solo dopo averlo **assemblato.**
Dopo aver salvato un programma premendo il tasto funzione **F2**, per **assemblarlo** dovrete procedere come segue:

Premete **ALT+T**
quindi premete **A**

Se nel programma che avete modificato o riscritto non è presente nessun **errore**, dopo pochi secondi comparirà sul monitor la scritta:

SUCCESS

Se avete commesso qualche **errore**, apparirà in basso a sinistra sul monitor un **numero**, che corrisponde alla riga in cui c'è un **errore**.
Dovrete allora riaprire il file ed andare sulla **riga** che vi è stata segnalata per scoprire quale errore avete commesso.
Terminata la correzione, dovrete nuovamente **salvare** il programma premendo il tasto **F2**, dopodiché potrete ripetere i comandi già descritti per **assemblare** il vostro programma.

PER TRASFERIRE il programma sull'ST6

Dopo aver **assemblato** il programma ed aver ottenuto la scritta *****success*****, potrete trasferirlo nella memoria del microprocessore **ST6**.

Quando siete nel **menu** principale dovrete:

premere **ALT+P**

e dopo pochi secondi comparirà l'intestazione del software delle **SGS** in lingua inglese.
A questo punto, prendete la rivista **N.172/173** (se non l'avete potrete sempre richiederla) poi rileggete quanto riportato nelle **pagg.39-41**, che non riscriviamo perché sarebbe un'inutile ripetizione di quanto abbiamo già spiegato.

TRASFERIRE un FILE sul DISCHETTO

Nel caso voleste **trasferire** un programma già **assemblato** dall'Hard-Disk in un dischetto, ad esempio per darlo ad un amico, dovrete, sempre partendo dal **menu** principale:

premere **ALT+F**
poi premere **D**

Apparirà la scritta **C:\ST6>**
A questo punto inserite il **dischetto** nel **drive**, ed ammesso che abbiate chiamato il programma che volete trasferire **DPLPROVA**, dovrete scrivere questa istruzione:

C:\ST6>Copy DPLPROVA.ASM A:\DPLPROVA.ASM

poi Enter
Quando apparirà la scritta **1 file copiato** dovrete scrivere:

C:\ST6>EXIT poi Enter

TRASFERIRE dal FLOPPY all'HARD-DISK

Per **trasferire** un programma dal dischetto all'Hard-Disk dovrete, quando vi trovate nel **menu** principale**:**

premere **ALT+F**
poi premere **D**

Apparirà la scritta **C:\ST6>**
A questo punto inserite il **dischetto** nel **drive**, ed ammesso che il programma da trasferire si chiami **DPLPROVA**, dovrete scrivere questa istruzione:

C:\ST6>Copy A:\DPLPROVA.ASM DPLPROVA.ASM
poi Enter

Quando comparirà la scritta **1 file copiato** dovrete scrivere:

C:\ST6>EXIT poi Enter

NOTA IMPORTANTE

All'inizio di tutti i programmi che vi forniamo troverete questa istruzione:

.ORG 880h

che serve per i soli microprocessori **ST6** con **2K** di memoria, cioè i:

ST62/E10 - ST62/E15 - ST62/T10 - ST62/T15

Se utilizzerete dei microprocessori **ST6** con **4K** di memoria, cioè i:

ST62/E20 - ST62/E25 - ST62/T20 - ST62/T25

dovrete sostituire il numero **880h** con il numero **080h**, quindi dovrete scrivere:

.ORG 080h

Pertanto se usate un **ST6** da **2K** di memoria, dovrete necessariamente scrivere all'inizio del programma **880h**, perché se scriverete **080h**, non risultando presenti in questo **ST6** queste celle di memoria, non riuscirete a trasferire nessun programma, ed il programmatore lo segnalerà.
Se usate un **ST6** da **4K** di memoria dovrete scrivere all'inizio del programma **080h** per poter utilizzare tutta la sua memoria.
Facciamo presente che se avete un programma da **2K**, che ovviamente inizierà con l'indirizzo di memoria **880h**, lo potrete tranquillamente trasferire con lo stesso indirizzo anche in un **ST6** da **4K.**
In questo caso partendo dall'indirizzo **880h** rimarranno inutilizzate tutte le celle di memoria da **080h** a **87Fh**.

COSTO DI REALIZZAZIONE

Tutti i componenti per realizzare la scheda Display siglata LX.1204 visibile nelle figg.6-7 completa di circuito stampato, integrato M.5450 e 4 display ....€ 18,60

Tutti i componenti per realizzare la scheda Relè siglata LX.1205 visibile nelle figg.9-10 completa di circuito stampato e 4 relè ....€ 19,10

Costo del solo stampato LX.1204 ....€ 4,34
Costo del solo stampato LX.1205 ....€ 5,06

Ai prezzi riportati andranno aggiunte le sole spese di spedizione a domicilio.

Quando un lettore ci scrive che, montato un nostro progetto non riesce a farlo funzionare, presumiamo che abbia commesso un **errore**, poichè prima di pubblicare nella rivista un qualsiasi circuito, è nostra consuetudine farne montare una **decina** di esemplari e se constatiamo che uno di questi risulta **critico** o presenta qualche **anomalia** lo riportiamo in laboratorio per ricercarne le cause e per eliminarle.
Questo modo di procedere lo adottiamo per ridurre al **minimo** le **riparazioni** e per assicurare al lettore il sicuro ed immediato funzionamento di ogni nostro progetto.
A volte si verificano anche delle **strane anomalie** che fanno **arrabbiare** i lettori ed **impazzire** i tecnici della consulenza.

per **capire** per quale motivo apparisse un simile **errore**, abbiamo chiesto a **2 lettori** residenti a **Ravenna** e a **Ferrara** se fossero disposti a venire a Bologna portando il loro **computer**, perchè volevamo cercare di scoprire la causa di questa anomalia.
Infatti se su **4.738** kit che funzionano in modo perfetto solo **16** si rifiutano di farlo, il difetto può essere causato solo dal **computer** ed infatti grazie a questi **2 lettori** siamo riusciti ad individuarlo.
Abbiamo scoperto che nei loro computer il segnale che entrava nel piedino **4** del **CONN.1** (vedi nello schema elettrico riportato a pag.31 della rivista N.172/173 il segnale **D2** che, tramite la **R7**, giunge al **piedino 1** della porta **IC1/E**) era **strettissimo** oppure aveva un'ampiezza **ridotta**.

NOTA per il programmatore

Se voi foste un tecnico che da anni usa questo **programmatore per ST6** senza mai riscontrare nessun inconveniente, che conosce tanti amici (esattamente **4.738**) che lo hanno realizzato con successo, usandolo con diversi tipi di computer, e poi trovaste solo **16** lettori che si lamentano perchè il loro montaggio non funziona, cosa rispondereste loro ?
Senz'altro che hanno commesso un errore nel montaggio, ed è questa anche la nostra risposta.
Ciò che ci ha stupito è constatare che a tutti e **16** questi lettori sul monitor appariva lo stesso messaggio, cioè:

Target chip not present or defective!

Subito abbiamo pensato che avessero acquistato dei **microprocessori ST6 difettosi** già all'origine.
Per risolvere tale problema ci siamo fatti inviare questi **ST6** insieme al **programmatore** per controllare entrambi in laboratorio.
Appena arrivati, li abbiamo provati su 8 diversi computer e tutti gli **ST6** che ci sono stati inviati si sono regolarmente programmati, senza **errori**.
Rispediti i **programmatori** a questi lettori, tutti e **16** ci hanno risposto che appariva nuovamente il medesimo **errore**.
Se non ci chiamassimo **Nuova Elettronica**, a questo punto avremmo abbandonato questi **16 lettori** con i loro ST6 difettosi, ma per **serietà** ed anche

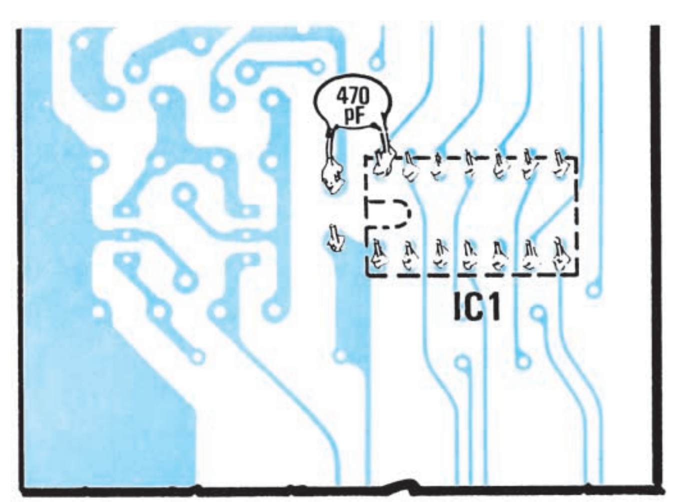
Scartata l'idea di manomettere il computer, abbiamo risolto il problema allargando l'impulso con un piccolo condensatore da **470 picoFarad** ceramico posto tra la resistenza **R7** e la **massa** come visibile nelle figg.1-2.
Immediatamente abbiamo comunicato ai **16 lettori** che non riuscivano a programmare gli **ST6**, di aggiungere sul loro **programmatore** questo condensatore da **470 picoFarad** e questi ci hanno risposto che la scritta:

Target chip not present or defective!

non appare più e con questa semplice modifica ora riescono a programmare qualsiasi **ST6**.

Importante = Se il vostro **programmatore** funziona correttamente non è necessario che inseriate questo condensatore, comunque se qualche volta vi capita di non riuscire a programmare un **ST6**, provate a collocare questo condensatore da **470 pF** tra la **R7** e la **massa** ed il difetto sparirà.
Come potete constatare, quando ci imbattiamo in qualche **anomalia**, facciamo tutto il possibile per eliminarla, ma se i due lettori di **Ravenna** e **Ferrara** non ci avessero portato il loro computer, forse questo caso sarebbe rientrato negli **insoluti**, perchè nessuno poteva supporre che il segnale che usciva dalla loro **presa parallela** fosse **fuori standard**.

Fig.1 Se constatate che il vostro programmatore LX.1170 non sempre riesce a programmare un ST6, potrete risolvere questo problema collegando tra il piedino 1 dell'integrato IC1-E e la massa dello stampato, un condensatore da 470 picoFarad.



LX.1170 per micro ST6

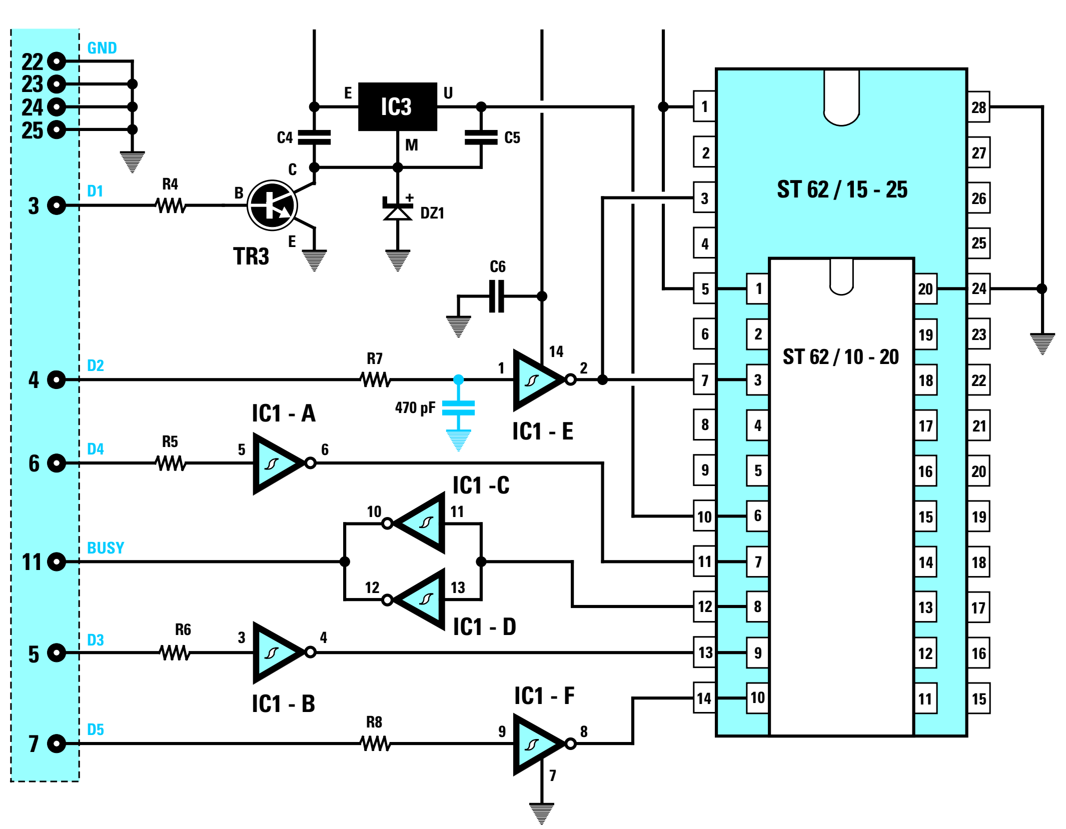


Fig.2 Questo condensatore, collegato dopo la resistenza R7, serve per allargare quegli impulsi che giungono troppo stretti sull'ingresso dell'inverter IC1-E.

A pag.124 della rivista N.175/176 abbiamo riportato il programma N.15 che consente di accendere 5 diodi led variando la tensione d'ingresso da 1 a 5 volt.
Molti lettori sulla base di questo esempio hanno utilizzato i micro **ST62E15** o **ST62E25**, poi hanno cercato di modificare il programma per far accendere 8 diodi led e quando sono andati ad assemblarlo, il computer ha segnalato questo errore:

5-bit displacement overlow

Con tale indicazione il microprocessore segnala che non può fare un **salto** all'**etichetta** richiesta perchè sono troppe le righe che lo separano da essa.
Nel nostro esempio avevamo usato soltanto **5 diodi led**, quindi utilizzando degli altri **diodi led** si sono dovute aggiungere delle altre righe di programma e poichè le istruzioni **JRC - JRNC - JRZ - JRNZ** possono fare solo dei **salti** limitati ad un certo numero di righe, se queste sono maggiori del richiesto appare il messaggio di **errore** menzionato.
In presenza di **salti** molto **lunghi** è necessario modificare l'istruzione presente con quella **inversa**, poi scrivere nella riga successiva l'istruzione **JP** che è in grado di fare un **salto** in qualsiasi punto del programma anche se molto distante.

Le istruzioni andranno quindi modificate:

da **JRC** a **JRNC**
da **JRNC** a **JRC**
da **JRZ** a **JRNZ**
da **JRNZ** a **JRZ**

PROGRAMMA per Esempio n.15

	LDI	pdir_a,00000000B	; in queste tre righe abbiamo settato
	LDI	popt_a,10000000B	; il piedino **A7** come **ingresso analogico**
	LDI	port_a,10000000B	;
	LDI	pdir_b,00011111B	; in queste righe abbiamo settato
	LDI	popt_b,00011111B	; i piedini da **B0** a **B4** come **uscite**
	LDI	port_b,00000000B	;
ripeti	LDI	wdog,255	; carichiamo il **watchdog**
	LDI	adcr,00110000B	; provvedi a convertire da **analogico** a **digitale**
attendi	JRR	6,adcr,attendi	; attendere che avvenga la conversione **A/D**
	LD	a,addr	; carica nell'accumulatore A, il numero **digitale**
	CPI	a,255	; compara il valore di A con il numero **255**
	JRNC	LED5	; se A è **uguale** a 255 salta all'etichetta **LED5**
	CPI	a,204	; compara il valore di A con il numero **204**
	JRNC	LED4	; se A è **maggiore** di 204 salta all'etichetta **LED4**
	CPI	a,153	; compara il valore di A con il numero **153**
	JRNC	LED3	; se A è **maggiore** di 153 salta all'etichetta **LED3**
	CPI	a,102	; compara il valore di A con il numero **102**
	JRNC	LED2	; se A è **maggiore** di 102 salta all'etichetta **LED2**
	CPI	a,51	; compara il valore di A con il numero **51**
	JRNC	LED1	; se A è **maggiore** di 51 salta all'etichetta **LED1**
	JP	LED0	; se A è **minore** di 51 salta all'etichetta **LED0**
LED0	LDI	port_b,00000000B	; non accendere **nessun** diodo led
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED1	LDI	port_b,00000001B	; accendi il led sul piedino **B0**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED2	LDI	port_b,00000011B	; accendi i led sui piedini **B0 - B1**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED3	LDI	port_b,00000111B	; accendi i led sui piedini **B0 - B1 - B2**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED4	LDI	port_b,00001111B	; accendi i led sui piedini **B0 - B1 - B2 - B3**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED5	LDI	port_b,00011111B	; accendi i led sui piedini **B0 - B1 - B2 - B3 - B4**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**

PROGRAMMA SALTO Esempio n. 15 bis

	LDI	pdir_a,00000000B	; in queste tre righe abbiamo settato
	LDI	popt_a,10000000B	; il piedino **A7** come **ingresso analogico**
	LDI	port_a,10000000B	;
	LDI	pdir_b,11111111B	; in queste righe abbiamo settato
	LDI	popt_b,11111111B	; i piedini della porta **B** come **uscite**
	LDI	port_b,00000000B	;
ripeti	LDI	wdog,255	; carichiamo il **watchdog**
	LDI	adcr,00110000B	; provvedi a convertire da **analogico** a **digitale**
attendi	JRR	6,adcr,attendi	; attendere che avvenga la conversione **A/D**
	LD	a,addr	; carica nell'accumulatore A il numero **digitale**
	CPI	a,255	; compara il valore di A con **255**
	JRC	etich1	; se A è **minore** di 255 salta a **etich1**
	JP	LED8	; salta a **LED8**
etich1	CPI	a,224	; compara il valore di A con **224**
	JRC	etich2	; se A è **minore** di 224 salta a **etich2**
	JP	LED7	; salta a **LED7**
etich2	CPI	a,192	; compare il valore di A con **192**
	JRC	etich3	; se A è **minore** di 192 salta a **etich3**
	JP	LED6	; salta a **LED6**
etich3	CPI	a,160	; compara il valore di A con **160**
	JRC	etich4	; se A è **minore** di 160 salta a **etich4**
	JP	LED5	; salta a **LED5**
etich4	CPI	a,128	; compara il valore di A con **128**
	JRC	etich5	; se A è **minore** di 128 salta a **etich5**
	JP	LED4	; salta a **LED4**
etich5	CPI	a,96	; compara il valore di A con **96**
	JRC	etich6	; se A è **minore** di 96 salta a **etich6**
	JP	LED3	; salta a **LED3**
etich6	CPI	a,64	; compara il valore di A con **64**
	JRC	etich7	; se A è minore di 64 salta a **etich7**
	JP	LED2	; salta a **LED2**
etich7	CPI	a,32	; compara il valore di A con **32**
	JRC	etich8	; se A è minore di 32 salta a **etich8**
	JP	LED1	; salta a **LED1**
etich8	JP	LED0	; salta a **LED0**
LED0	LDI	port_b,00000000B	; non accendere **nessun** diodo led
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED1	LDI	port_b,00000001B	; accendi il led sul piedino **B0**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED2	LDI	port_b,00000011B	; accende i led sui piedini **B0 - B1**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED3	LDI	port_b,00000111B	; accende i led sui piedini **B0 - B1 - B2**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED4	LDI	port_b,00001111B	; accende i led sui piedini **B0 - B1 - B2 - B3**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED5	LDI	port_b,00011111B	; accende i led sui piedini **B0 - B1 - B2 - B3 - B4**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED6	LDI	port-b,00111111B	; accende i led sui piedini da **B0** a **B5**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED7	LDI	port-b,01111111B	; accende i led sui piedini da **B0** a **B6**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**
LED8	LDI	port-b,11111111B	; accende i led sui piedini da **B0** a **B7**
	JP	ripeti	; salta all'etichetta **ripeti** del **watchdog**

Nel programma presentato a **pag.124** come "**Esempio N.15**" abbiamo scritto nella **12° riga** questa istruzione:

JRNC LED5 ; se A è uguale a 255 salta all'etichetta LED5

Nel caso in cui si desiderino aggiungere degli altri led, sarà necessario **modificare** il programma come segue:

JRC etich1; se A è minore di 255 salta a etich1
JP LED8; salta a LED8
Etich1; etichetta 1 che proseguirà con il programma

Se ancora tutto questo non vi risulta chiaro, confrontate il primo programma **Esempio N.15** con il secondo programma modificato, che abbiamo chiamato **Esempio N.15 BIS**.

CONVERTITORE A/D

Nella rivista N.175/176 a pag.123 abbiamo scritto che, risultando presente all'interno dell'**ST6** un **solo A/D converter**, potevamo utilizzare come ingresso per **segnali analogici** un **solo** piedino.
In pratica è possibile utilizzare anche **più piedini** come ingressi **analogici**, sempre che si scriva un programma che vada a leggere in **multiplexer** le tensioni presenti su tutti i piedini d'ingresso che abbiamo prescelto per questa funzione.
Per farvi comprendere come un **solo A/D converter** possa leggere le tensioni poste su **diversi** piedini, vi proponiamo qui di seguito un esempio.
Ammesso di possedere un **solo voltmetro** e di voler leggere con questo valori di **tensione** presenti su punti diversi, potremo farlo se sull'ingresso del **voltmetro** applicheremo il **cursore** di un **commutatore rotativo** e sui terminali di commutazione le diverse tensioni che vorremo leggere.
Ruotando il commutatore sulle diverse posizioni, potremo leggere **più tensioni** pur disponendo di un **solo voltmetro**.
Vogliamo comunque ricordarvi che come **ingressi analogici** potremo usare qualsiasi piedino ad eccezione dei soli piedini **A0-A1-A2-A3**.
Nel programma che qui riportiamo come **Programma A/D** vi facciamo vedere come bisognerà scrivere le istruzioni per poter leggere le tensioni presenti sui piedini d'ingresso di **B5-B6-B7**.

PROGRAMMA A/D

```
        LDI   pdir_b,00000000B   ; nelle prime cinque righe
        LDI   popt_b,00000000B   ; abbiamo settato il piedino B7
        LDI   port_b,00000000B   ; come ingresso analogico
        LDI   port_b,10000000B   ;
        LDI   popt_b,10000000B   ;
        LDI   adcr,00110000B     ; provvedi a convertire da analogico a digitale
AD1     JRR   6,adcr,AD1         ; attendere che avvenga la conversione A/D
        LD    a,ADDR             ; copia in A il valore dell'A/D
        LD    VOLT1,a            ; copia in VOLT1 il valore di a
        LDI   pdir_b,00000000B   ; nelle prime cinque righe
        LDI   popt_b,00000000B   ; abbiamo settato il piedino B6
        LDI   port_b,00000000B   ; come ingresso analogico
        LDI   port_b,01000000B   ;
        LDI   popt_b,01000000B   ;
        LDI   adcr, 00110000B    ; provvedi a convertire da analogico a digitale
AD2     JRR   6,adcr,AD2         ; attendere che avvenga la conversione A/D
        LD    a,ADDR             ; copia in A il valore dell'A/D
        LD    VOLT2,a            ; copia in VOLT2 il valore di a
        LDI   pdir_b,00000000B   ; nelle prime cinque righe
        LDI   popt_b,00000000B   ; abbiamo settato il piedino B5
        LDI   port_b,00000000B   ; come ingresso analogico
        LDI   port_b,00100000B   ;
        LDI   popt_b,00100000B   ;
        LDI   adcr,00110000B     ; provvedi a convertire da analogico a digitale
        JRR   6,adcr,AD3         ; attendere che avvenga la conversione A/D
        LD    a,ADDR             ; copia in A il valore dell'A/D
        LD    VOLT3,a            ; copia in VOLT3 il valore di a
```

SCHEDA con 4 TRIAC

Se avete già acquistato il kit per **testare** gli ST6 siglato **LX.1202** e le due schede, una con quattro **display** siglata **LX.1204** e l'altra con quattro **relè** siglata **LX.1205** pubblicate nella rivista **N.179**, avrete ricevuto anche un dischetto con codice **DF.1202/3 = DF.1170/3** contenente diversi programmi formato "**.ASM**" che, caricati nell'hard-disk, vi serviranno per gestire le due schede sperimentali apparse nella rivista **N.179** e quella che appare su questo numero con quattro **triac** siglata **LX.1206**.
Prossimamente vi forniremo altre due schede per display **LCD alfanumerici** ed altri nuovi programmi.
Anche se lo abbiamo già precisato negli articoli precedenti, vi ricordiamo che i programmi **.ASM** li potrete trasferire **singolarmente** nella **memoria** di un microprocessore **ST6** solo dopo averli **assemblati**, cioè convertiti in files formato **.HEX**.

I tre programmi **CONTA-LED-LOTTO** che sono **assemblati** in **.HEX** sono già pronti per essere caricati all'interno della memoria dell'ST6.
Tutti gli altri programmi che terminano con **.ASM** li potrete tranquillamente modificare, ampliare e, come già accennato, prima di passarli nella memoria di un **ST6** li dovrete **assemblare** per convertirli in files **.HEX**.

Le modifiche in questi programmi sono sempre necessarie per adattarli alle vostre esigenze. Ad esempio, noi abbiamo predisposto i programmi **TIMER.ASM** e **TEMPOR.ASM** per eccitare un **relè** o un **Triac** in un tempo di **3 minuti** sia contando all'indietro (TEMPOR.ASM) che in avanti (TIMER.ASM) e poichè questo tempo non vi servirà per nessuna delle vostre applicazioni, basterà leggere all'interno del programma i vari **commenti** per sapere quale riga dovrete **correggere** e quale **numero** inserire per modificarla.
Anche in tutti gli altri programmi troverete di lato ad ogni riga un **commento** che vi spiegherà se potete modificarla, sostituirla, o cancellarla.
Le modifiche non fatele mai sul nostro **file**, ma su un identico file che **duplicherete** attribuendogli un **nome** diverso, in modo da avere sempre a disposizione il **file originale** per poterlo consultare o confrontare per scoprire eventuali errori sui files modificati.

Tutte le istruzioni richieste per poter **duplicare** un **file** le troverete in questo articolo.
Non dovrete **mai modificare** i programmi presenti all'interno del dischetto che terminano con **BAT - EXE - COM - DEV - HEX**.
Sul disco che vi forniremo, oltre ai programmi **.ASM**, è presente anche un **EDITOR** che vi servirà per scrivere dei programmi e per **ASSEMBLARE** i files prima di caricarli sui microprocessori **ST6** tramite il programmatore **LX.1170** (leggere la rivista **N.172-173**).

Una volta comprese le funzioni dei vari **blocchi**, potrete **ampliarli**, **modificarli** oppure **trasferirli** su un vostro programma per poter gestire, secondo la vostra fantasia, queste ed altre **schede sperimentali**.

Nel dischetto **DF.1202/3** identico al **DF.1170/3** che da oggi forniamo, troverete questi **18 programmi**:

1^ CONTA.ASM
2^ LED.ASM
3^ LOTTO.ASM
4^ STANDARD.ASM

5^ CRONOMET.ASM
6^ DISPLAY.ASM
7^ LM093.ASM
8^ OROLOGIO.ASM
9^ RELE.ASM
10^ TEMPOR.ASM
11^ TIMER.ASM
12^ TRIAC.ASM
13^ CLOCK.ASM
14^ TIME90.ASM
15^ TEMP90.ASM

16^ CONTA.HEX
17^ LED.HEX
18^ LOTTO.HEX

Nota = I primi **4** programmi ve li avevamo già forniti con il **primo** disco floppy assieme al programmatore per **ST6** (vedi rivista **n.172-173**).

per microprocessori ST6

Sul precedente numero della rivista vi abbiamo presentato due schede per ST6, una per accendere dei normali display a 7 segmenti ed un'altra per eccitare dei relè. In questo numero vi presentiamo una scheda per eccitare quattro diodi Triac, spiegandovi anche come si possono modificare i programmi da noi forniti.

UN PROMEMORIA

Anche se nel numero **179** della rivista abbiamo spiegato che occorre necessariamente inserire queste schede sperimentali nel **bus** siglato **LX.1202**, molti ci chiedono se e su quale tipo di computer occorra collegarle e se per le prove convenga usare un **ST6 cancellabile** o **non cancellabile**.

- La scheda **bus** siglata **LX.1202** risulta progettata per ricevere tutte le schede **sperimentali** che vi abbiamo presentato e anche le future con display **LCD.** Questa scheda **non andrà** collegata a nessun computer, ma serve per inserirvi l'**ST6** che abbiamo programmato.

- Oltre alla scheda **bus** vi servirà anche il **Programmatore per micro ST6** siglato **LX.1170** pubblicato nella rivista **N.172/173**, che sarà **indispensabile** per poter trasferire il **programma** che sceglierete o che avrete scritto dal computer alla memoria del **microprocessore ST6**.
Dopo aver trasferito il programma nel microprocessore, dovrete togliere quest'ultimo dallo zoccolo Textool del programmatore **LX.1170** ed inserirlo nella scheda **bus** siglata **LX.1202** per poter gestire le schede sperimentali che applicherete su questo **bus**.
Vi ricordiamo che il solo programmatore **LX.1170** andrà collegato alla **porta parallela** del vostro computer, purchè questo sia un **IBM** o un **compatibile** serie **XT - AT - SX - DX** tipo **8088 - 286 - 386 - 486 - Pentium** con qualsiasi **frequenza di clock**, compresi anche i **portatili** con installato il sistema operativo **DOS** dal **3** al **6.2**.
Ripetiamo nuovamente che il **software** per **ST6** non è assolutamente compatibile per computer tipo Commodore, Apple, Amiga, ecc.

Fig.1 Per duplicare il programma STANDARD.ASM dovrete prima richiamare l'Editor, scrivendo C:\>ST6 poi Enter. Dopodichè scriverete C:\ST6>ST6 e a questo punto potrete premere il tasto Enter.

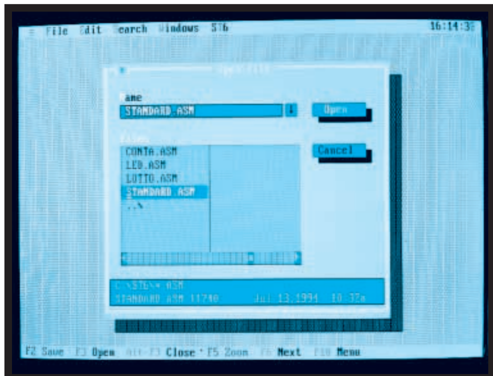


Fig.2 Premete i tasti ALT F poi F3 e, così facendo, vi appariranno tutti i files ASM. Portate il cursore sulla riga STANDARD.ASM, premete Enter e, in tal modo, vi apparirà la finestra di fig.3.

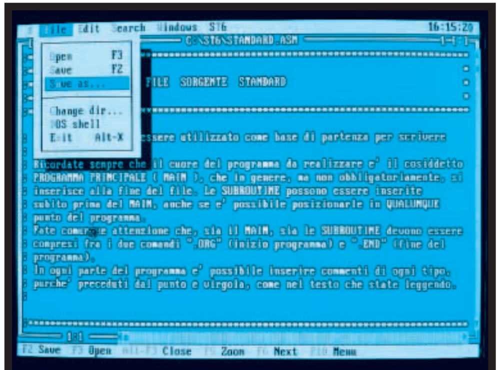


Fig.3 Portate il cursore sulla riga SAVE as ... poi premete Enter. Per duplicare questo file con un nome "diverso" ripremete i due tasti ALT F e, in tal modo, vi apparirà la finestra di fig.4.

- Per le prove **sperimentali** conviene scegliere un microprocessore tipo **ST62E20** con **4 K di Rom** (il micro **ST62E10** è stato messo fuori produzione dalla **SGS**), perchè, anche se risulta molto **costoso**, lo potrete **cancellare** e riutilizzare diverse centinaia di volte per **memorizzare** degli altri nuovi e diversi programmi.

- Per **cancellare** uno di questi microprocessori potrete usare la lampada ad **ultravioletti** siglata **LX.1183** presentata sulla rivista **N.174.**

- Se volete usare i più economici microprocessori tipo **ST62T10** o **ST62T20** potete farlo, ma poichè questi **non sono cancellabili**, se **sbaglierete** nello scrivere un programma, dovrete buttarli ed acquistarne degli **altri**.
Normalmente i microprocessori **non cancellabili** vengono utilizzati solo dopo aver **testato** più di una volta il micro **cancellabile** tipo **ST62E20**, per avere la certezza che nel **vostro programma** non vi siano degli **errori**.

TABELLA N.1 micro NON CANCELLABILI

Sigla Micro	memoria utile	Ram utile	zoccolo piedini	piedini utili per i segnali
ST62T.10	2 K	64 byte	20 pin	12
ST62T.15	2 K	64 byte	28 pin	20
ST62T.20	4 K	64 byte	20 pin	12
ST62T.25	4 K	64 byte	28 pin	20

TABELLA N.2 micro CANCELLABILI

Sigla Micro	memoria utile	Ram utile	zoccolo piedini	piedini utili per i segnali
ST62E.20	4 K	64 byte	20 pin	12
ST62E.25	4 K	64 byte	28 pin	20

- Quando vorrete scrivere dei **nuovi** programmi dovrete sempre caricare il file **STANDARD.ASM**, perchè questo è il file **sorgente** che definisce la locazione dei **5 registri** del micro e fa il **settaggio** delle periferiche, cioè una inizializzazione completa.

IL SORGENTE STANDARD.ASM

Ritenevamo di aver spiegato abbastanza bene a cosa serva il file **STANDARD.ASM**, ma leggendo i quesiti che ci sono pervenuti in proposito abbiamo capito di non essere stati sufficientemente esaurienti.

Per riparare, ve lo rispiegheremo proponendovi anche qualche esempio.
Quando si scrive un **programma**, occorre sempre iniziare con dei **dati ripetitivi** che non varino mai da un programma ad un altro, quindi per non **riscriverli** ogni volta con il rischio di commettere degli **errori**, ve li ritroverete già tutti **impostati** nel programma **STANDARD.ASM** con una **nota** relativa a cosa dovrete **modificare**.
Ad esempio, quando arriverete al paragrafo **SETTAGGIO INIZIALE**, subito dopo **INIZIO PROGRAMMA** troverete tra le prime righe l'istruzione:

```
        .org    0880h
```

Se usate un micro da **2K** di memoria, cioè un **ST62E10 - ST62T10 - ST62E15 - ST62T15**, non dovrete **modificare** questo numero.

Se usate un micro da **4K**, cioè un **ST62E20 - ST62T20 - ST62E25 - ST62T25**, dovrete invece **modificare** questo numero come segue:

```
        .org    080h
```

Ammettiamo ad esempio di voler **creare** un nuovo programma chiamato **ALIBABA.**
La prima operazione da effettuare sarà quella di **uscire** da qualsiasi programma in utilizzo, come ad esempio **Windows**, **Pcshell**, **Norton**, ecc., in modo da vedere in alto a sinistra del vostro schermo il solo **Prompt** dei comandi, cioè **C:\>**
A questo punto potrete richiamare il programma **ST6** scrivendolo indifferentemente sia in maiuscolo che in minuscolo:

C:\>CD ST6 poi premete Enter

e in questo modo vi apparirà:

C:\ST6>

poi scrivete **ST6** come sotto riportato:

C:\ST6>ST6 poi premete Enter

e subito vedrete apparire sul monitor del computer la finestra dell'**EDITOR** (vedi fig.1).

A questo punto premete contemporaneamente i tasti **ALT F**, poi **F3** e vi apparirà una seconda finestra con gli elenchi di tutti i **files .ASM** (vedi fig.2).
Premete il tasto **Enter** e vedrete che il cursore andrà sul primo file **.ASM** colorandolo di **verde**.

Utilizzando i tasti **freccia** presenti sulla tastiera, portate il **cursore** sulla riga **STANDARD.ASM**, poi premete **Enter** e vedrete apparire sul monitor il **li-**

Fig.4 Per cambiare il nome del file STANDARD.ASM con ALIBABA dovrete scrivere C:\ST6\ALIBABA.ASM e poi premere ALT F3. Se desiderate cambiare il listato del file TIMER dovrete scrivere C:\ST6\TIMER.ASM.

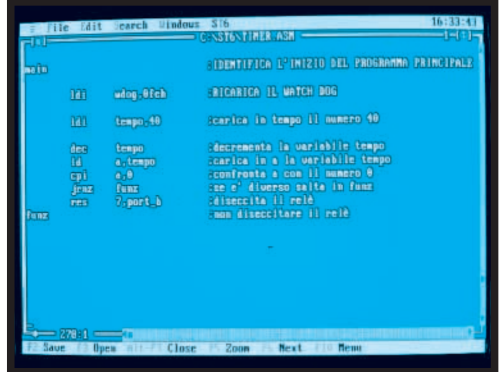


Fig.5 In tutti i listati troverete di lato un "commento condensato" che vi aiuterà a capire quale funzione esplicano le varie righe. Con un po' di esperienza riuscerete molto facilmente a modificarle.

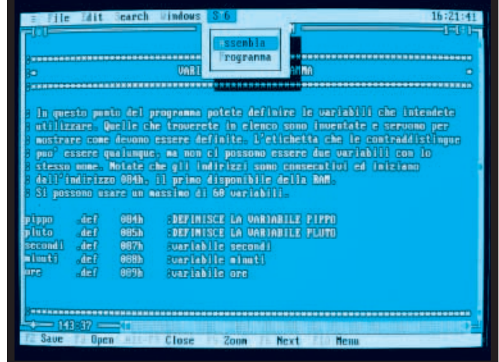


Fig.6 Corretto o riscritto, dovrete "salvare" il nuovo programma premendo il tasto F2, dopodichè lo potrete ASSEMBLARE premendo i tasti ALT T poi A. Se non avete commesso "errori" apparirà SUCCESS.

stato di questo programma che potrete leggere dall'inizio fino alla fine.
A questo punto se premerete contemporaneamente i due tasti **ALT F** vedrete apparire una nuova maschera (vedi fig.3).
Con il tasto freccia giù andate alla riga dove è scritto:

SAVE as.... poi premete Enter

Così facendo vi apparirà una riga con scritto:

C:\ST6\STANDARD.ASM

Poichè intendiamo chiamare il **nuovo** programma **ALIBABA**, questa riga la dovrete riscrivere come segue:

C:\ST6\ALIBABA.ASM poi premete Enter

Nota = I nomi dei **files** non debbono mai avere più di **8 caratteri** escluso ovviamente **.ASM**
Corretta questa riga, premete i tasti **ALT F3** e vi apparirà nuovamente la maschera dell'**EDITOR**.
A questo punto dovrete premere i due tasti **ALT F**, poi **F3** e nella lista dei **files** troverete il nuovo file denominato **ALIBABA.ASM**.

Dopo aver premuto **Enter**, con i tasti delle **frecce** portate il **cursore** sul file **ALIBABA.ASM** e, in questo modo, vi apparirà il **listato duplicato** del file **STANDARD.ASM**, che potrete tranquillamente modificare perchè ora lavorerete sul file **ALIBABA.ASM.**
Se, per ipotesi, tutte le modifiche che apporterete sul file **ALIBABA.ASM** non lo faranno funzionare per qualche **errore** da voi commesso, lo potrete **cancellare** e nuovamente **ricopiare**, utilizzando il file originale **STANDARD.ASM** come vi abbiamo appena spiegato.
Nel programma che in precedenza si chiamava **STANDARD.ASM** e che ora avete chiamato **ALIBABA.ASM** dovrete ricordarvi che la riga:

```
        .org    0880h
```

non va modificata se userete dei micro con **2K** di memoria, vale a dire se userete degli **ST62E10**, **ST62E15**, **ST62T10**, **ST62T15**.

Se invece userete dei micro con **4K**, vale dire degli **ST62E20**, **ST62E25**, **ST62T20**, **ST62T25**, questa riga va **modificata** come segue:

```
        .org    080h
```

A questo punto dovrete modificare il settaggio di tutte le **porte A-B-C**, cioè dovrete impostarle come **ingressi** o **uscite** a seconda delle esigenze del vostro programma.
Nel caso non sappiate come si faccia a settare le porte, vi consigliamo di leggere l'articolo **"Imparare a programmare i microprocessori ST6"** pubblicato nella rivista n.**175-176**.

Terminate tutte le modifiche e scritto il **nuovo programma** che si chiama **ALIBABA.ASM**, lo dovrete **salvare** premendo il tasto **F2**.
Ricordatevi di premere il tasto funzione **F2** tutte le

ELENCO COMPONENTI LX.1206

R1 = 1.000 ohm 1/4 watt
R2 = 100 ohm 1/4 watt
R3 = 1.000 ohm 1/4 watt
R4 = 100 ohm 1/4 watt
R5 = 1.000 ohm 1/4 watt
R6 = 100 ohm 1/4 watt
R7 = 1.000 ohm 1/4 watt
R8 = 100 ohm 1/4 watt
R9 = 220 ohm 1/4 watt
R10 = 220 ohm 1/4 watt
R11 = 220 ohm 1/4 watt
R12 = 220 ohm 1/4 watt
R13 = 10.000 ohm 1/4 watt
R14 = 10.000 ohm 1/4 watt
R15 = 10.000 ohm 1/4 watt
R16 = 10.000 ohm 1/4 watt
C1 = 47.000 pF pol. 400 V.
C2 = 47.000 pF pol. 400 V.
C3 = 47.000 pF pol. 400 V.
C4 = 47.000 pF pol. 400 V.
C5 = 100.000 pF poliestere
C6 = 100.000 pF poliestere
C7 = 100.000 pF poliestere
C8 = 100.000 pF poliestere
TRC1 = triac tipo 500 V. 5 A.
TRC2 = triac tipo 500 V. 5 A.
TRC3 = triac tipo 500 V. 5 A.
TRC4 = triac tipo 500 V. 5 A.
OC1 = fototriac tipo MOC.3020
OC2 = fototriac tipo MOC.3020
OC3 = fototriac tipo MOC.3020
OC4 = fototriac tipo MOC.3020
P1 = pulsante
P2 = pulsante
P3 = pulsante
P4 = pulsante

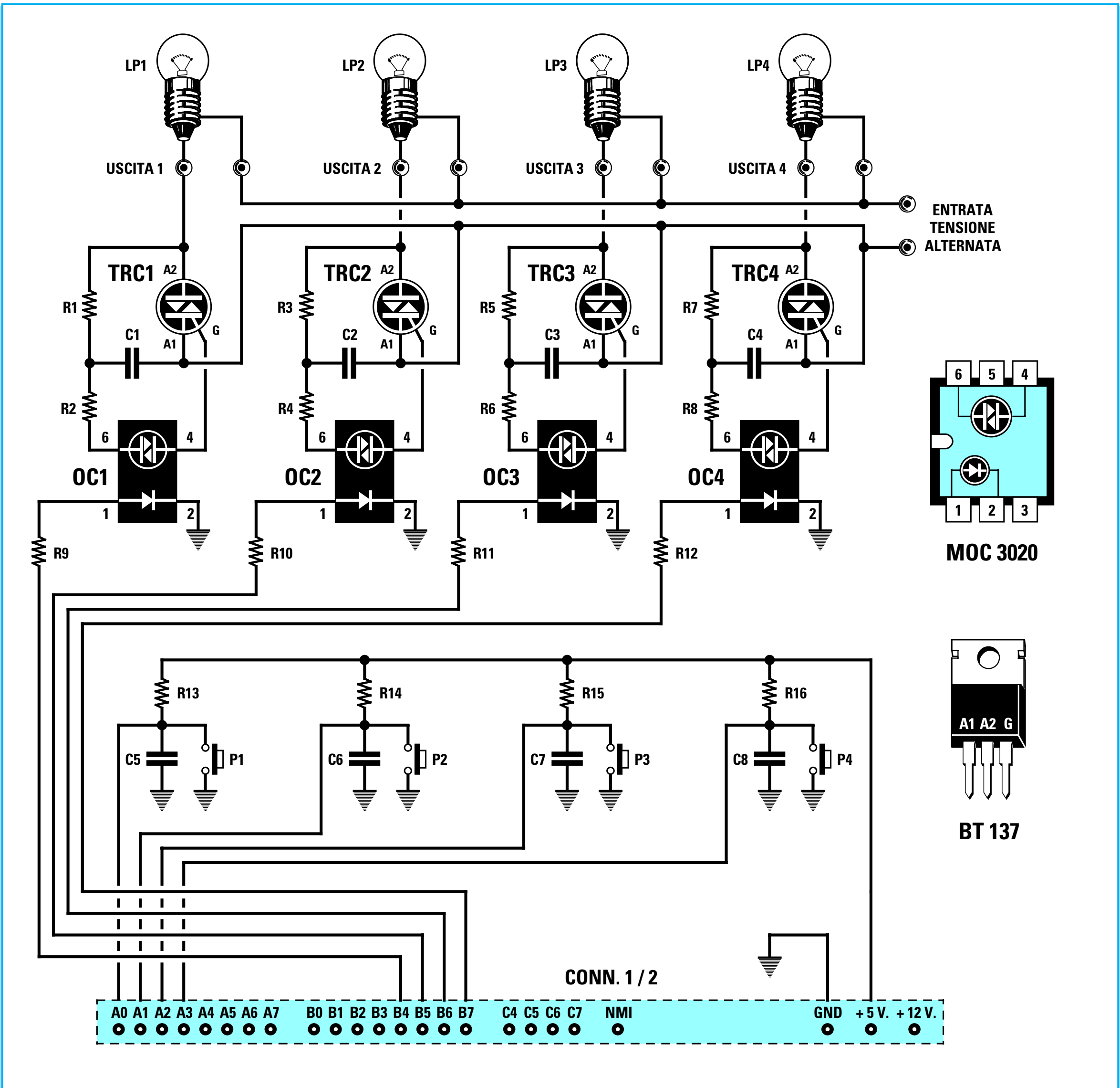


Fig.7 Schema elettrico della scheda Triac siglata LX.1206 e connessioni del fototriac visto da sopra e del triac BT.137. Sulle "uscite" dei Triac dovrete collegare delle lampadine o altre apparecchiature elettriche che funzionino con il valore della tensione alternata che applicherete sulla morsettiera d'ingresso (vedi fig.8).

volte che eseguirete una **variazione** o farete un'**aggiunta** nel programma, diversamente queste non verranno **memorizzate.**
Una volta **memorizzato**, lo dovrete anche **assemblare** premendo i tasti:

ALT T poi il tasto **A** (vedi fig.6)

Se non avrete commesso degli **errori** nello scrivere un'istruzione, sul monitor vi apparirà la scritta:

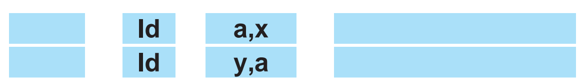
success

Se, in sostituzione di questa scritta, vi apparirà un **numero** in basso a sinistra sul monitor, significa che nella **riga** del programma che avete modificato o variato avete commesso un **errore**, ad esempio avete scritto **lp** anzichè **ld.**

Un **errore** che molti commettono è quello di caricare direttamente nel **registro Y** il contenuto del **registro X** scrivendo:

	ld	y,x	

Per caricare il contenuto del registro **X** nel registro **Y** dovrete **prima** caricare il registro **X** nell'accumulatore **A**, poi caricare su questo il registro **Y**, quindi questa istruzione andrà scritta su due righe:



	ld	**a,x**	
	ld	**y,a**	

Tutti i files una volta assemblati diventeranno dei **.HEX**, cioè convertiti in **esadecimale**, perchè questo è il solo linguaggio che il microprocessore è in grado di interpretare.
Vi ricordiamo che i files **.HEX** non potranno più essere modificati.

SCHEMA ELETTRICO

Lo schema elettrico di questa scheda per **Triac** è visibile in fig.7.
In teoria, potevamo sfruttare tutte le porte **A-B-C** del microprocessore **ST6** come **uscite** ed in tal modo potevamo inserire in questa scheda ben **20 Triac** con il micro da **28** piedini e **12 Triac** con il micro da **20** piedini.

In pratica, non potremo mai farlo, perchè dobbiamo tenere impegnate diverse porte **A** e **B** per gestire anche le altre schede che inseriremo nel **bus** assieme alla scheda Triac o Relè.

Quando sui piedini della porta **B** siglati **B4-B5-B6-B7** (vedi fig.7) apparirà un **livello logico 1**, vale a dire una **tensione positiva**, questa polarizzerà il **diodo emittente** presente all'interno dei fotoaccoppiatori siglati **OC1-OC2-OC3-OC4** e di conseguenza si **ecciterà** il Triac ad essi collegato.
Questi quattro **fotoaccoppiatori** li abbiamo utilizzati per separare elettricamente l'uscita del **microprocessore** dalla tensione che applicheremo sulle due boccole visibili a destra indicate con la scritta **Entrata Tensione Alternata**.
Per alimentare i Triac potremo usare qualsiasi tensione **alternata** partendo da un minimo di **4,5 volt** per arrivare ad un massimo di **220 volt**.
Ovviamente sulle **morsettiere** indicate **uscita 1 - uscita 2 - uscita 3 - uscita 4** dovremo applicare delle lampadine, dei motorini in alternata, o qualsiasi altra apparecchiatura elettrica che funzioni con il valore di **tensione** utilizzata per alimentare i **Triac**.
Non utilizzate una tensione **continua** per alimentare i Triac. Se volete usare una tensione **continua** dovrete necessariamente servirvi della **scheda RELÈ** siglata LX.1205 presentata nella rivista N.179.

REALIZZAZIONE PRATICA

Sul circuito stampato siglato **LX.1206** dovrete montare tutti i componenti visibili in fig.8.
Per il montaggio vi consigliamo di saldare sul lato opposto di questo stampato il connettore **maschio** a **1 fila** provvisto di **24 terminali** (vedi **CONN 1-2**) e vicino alle due morsettiere **Uscita 1** e **Uscita 4** i due connettori a **1 fila** provvisti di soli **4 terminali**, che vi serviranno per innestare i **connettori femmina** presenti sulla scheda **bus** siglata **LX.1202**.
Nella parte di stampato visibile in fig.8 inserirete i quattro zoccoli per i **fotoaccoppiatori**, tutte le **resistenze**, poi **condensatori** e le **5 morsettiere** a due poli.
Proseguendo nel montaggio, inserirete i **4 Triac** siglati **TRC1-TRC2-TRC3-TRC4**, rivolgendo la parte **metallica** del loro corpo verso **destra** come visibile nello schema pratico di fig.8.

Completata questa operazione, potrete inserire negli zoccoli i quattro **fotoaccoppiatori** rivolgendo verso **sinistra** il lato del loro corpo provvisto del **piccolo foro** di riferimento.

INSERIMENTO SCHEDA nel BUS

Questa scheda la potrete inserire nel **bus** siglato **LX.1202**, indifferentemente su un qualsiasi connettore **femmina** presente su questo stampato e lo stesso dicasi per la **scheda** dei **display** che utilizzerete per visualizzare i relativi **tempi**.
Sulle morsettiere di destra indicate **Entrata Tensione Alternata** dovrete applicare la tensione che servirà per alimentare i **motorini** o le **lampade** che applicherete sulle quattro **morsettiere d'uscita**.
Per le prime prove vi consigliamo di utilizzare una **tensione alternata** di **14 volt** fornita dall'apposito alimentatore **LX.1203**, presentato sulla rivista **N.179**, collegando alle uscite dei Triac delle normali lampadine da **12 volt** massimo **3 watt.**
Se lo ritenete opportuno potrete anche entrare nella morsettiera di destra con una tensione **alternata** di **220 volt** collegando all'uscita dei Triac delle lampadine da **220 volt**, ma **attenzione**, se userete la tensione di rete ricordatevi di **non toccare mai** le parti metalliche dei Triac e le uscite dei fotoaccoppiatori per evitare che la tensione dei **220 volt** si scarichi sul vostro corpo.
Con questa scheda potrete utilizzare tutti i programmi che abbiamo usato per la **scheda RELÈ** siglata LX.1025. A fine articolo abbiamo riportato come si possono modificare i tempi e le funzioni.

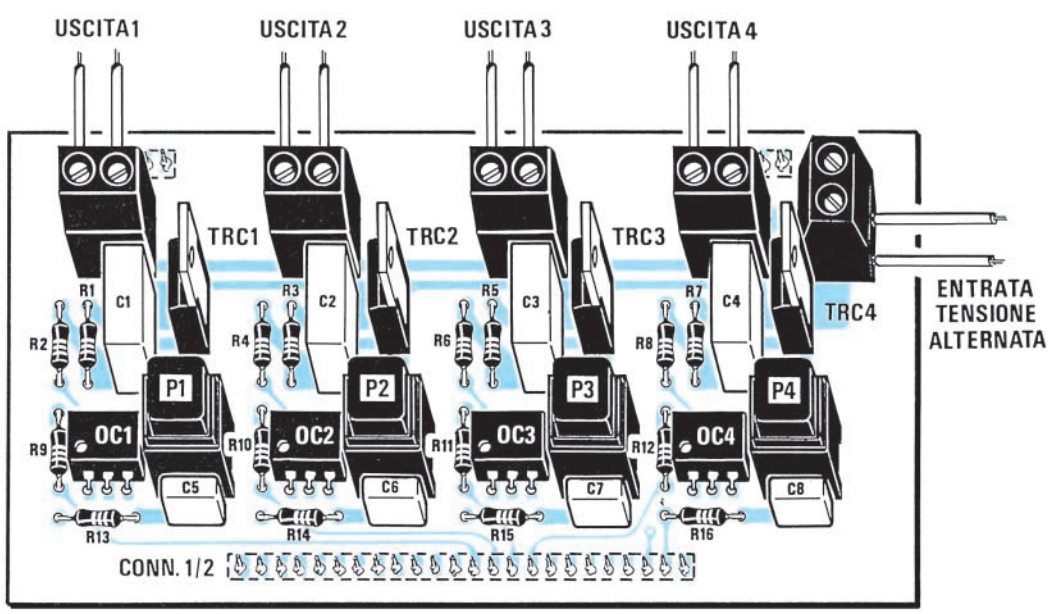


Fig.8 Schema pratico di montaggio della scheda LX.1206. Il circuito stampato che vi forniremo è un "doppia faccia" con fori metallizzati, quindi non cercate mai di allargare questi fori con una punta da trapano perchè, così facendo, asportereste il sottile strato di rame applicato per via galvanica al loro interno e che è necessario per collegare le piste presenti sotto al circuito stampato con quelle presenti sopra.

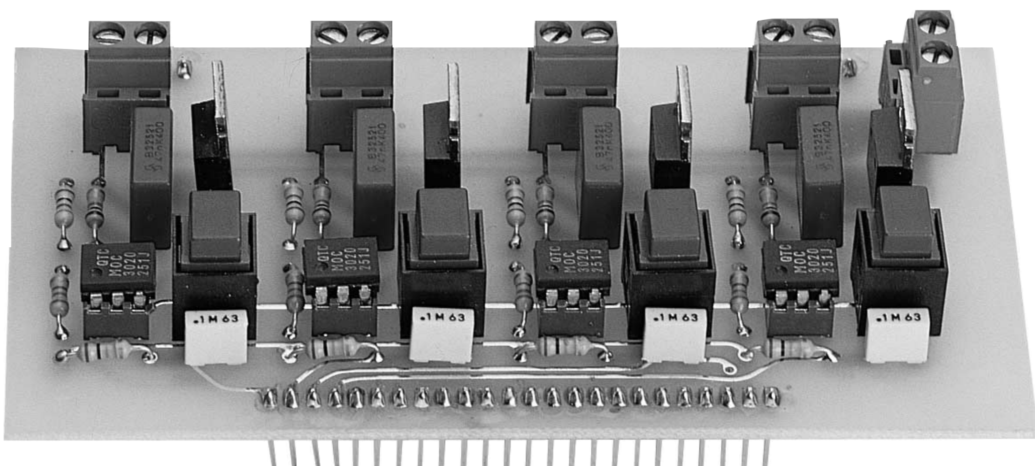


Fig.9 Foto della scheda LX.1206 come si presenterà a montaggio ultimato. Questa scheda andrà posta sul Bus siglato LX.1202 (vedi fig.10) assieme al microprocessore ST6 che avrete già programmato con uno dei programmi necessari per gestirlo. Consigliamo di rileggere le riviste N.172/173 - 174 - N.175/176 - N.179 per sapere come si deve procedere per memorizzare un micro e per settare tutte le PORTE.

INSTALLAZIONE programmi nell'HARD-DISK

Riportiamo in forma condensata quanto già scritto nelle riviste numero **172/173 - 174 - 175/176 - 179**, perchè ad alcuni lettori potrebbe essere sfuggito uno o più di questi numeri in cui sono apparsi i seguenti articoli:

Programmatore per micro ST6
Circuito test per microprocessore ST62E10
Impariamo a programmare i micro ST6
Lampada per cancellare Eprom
Impariamo a programmare i micro ST6
Bus per testare i micro ST6
Scheda test per Relè
Scheda test per display

Facciamo presente che sono ancora reperibili presso la nostra Sede alcune centinaia di copie delle riviste sopra elencate, quindi chi ne fosse sprovvisto e volesse avere la serie completa di articoli inerenti il **microprocessore ST6**, potrà richiedercele fino al loro esaurimento.

Per copiare il dischetto relativo all'**ST6** nell'Hard-Disk di un computer dovrete procedere come segue:

1 - Uscite da qualsiasi programma tipo **Windows - Pcshell - Norton**, ecc.

2 - Quando sul monitor apparirà il prompt **C:\>**, inserite nel drive floppy **A** il dischetto contenente i programmi, poi digitate:

C:\>A: poi premete il tasto Enter

e vi apparirà **A:\>**

A questo punto potrete scrivere:

A:\>installa poi premete Enter

Subito vi apparirà sul monitor la richiesta su quale **directory** volete installare il contenuto del disco. La directory da noi predefinita è **ST6**, quindi se digiterete **Enter** (vedi fig.11) il programma creerà u-

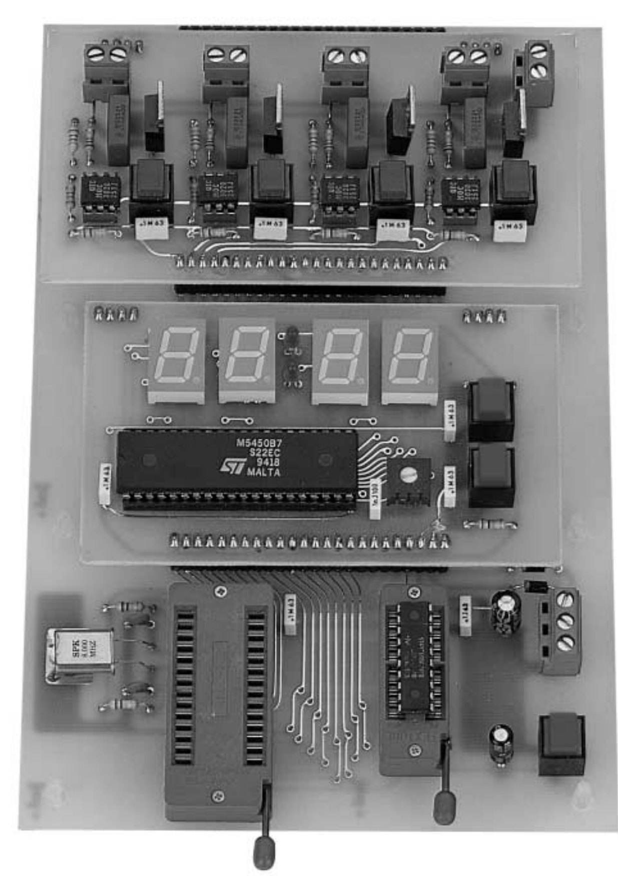


Fig.10 A quanti desiderano seguire questo corso sugli ST6 consigliamo di procurarsi il kit LX.1170 che servirà per trasferire i programmi che abbiamo assemblato da .ASM in .HEX, più il BUS siglato LX.1202 completo del suo alimentatore LX.1203. Questo BUS LX.1202 servirà, prossimamente, per controllare la scheda con un display LCD numerico e la scheda con un display LCD alfanumerico.

na directory con questo nome e copierà nell'**Hard-Disk** tutto il contenuto del dischetto **scompattando** i files.

Se volete trasferire il contenuto su una **diversa** directory, ad esempio **LX1202**, quando sul monitor vi apparirà:

C:\ST6

dovrete **cancellare** il nome della **directory** precedente, cioè **ST6** (vedi fig.11), scrivendo in sua sostituzione **LX1202** come qui sotto riportato:

C:\LX1202 poi premete Enter

Ricordatevi che qualsiasi **nome** di directory sceglierete non dovrete mai superare gli **8 caratteri** e nemmeno utilizzare dei caratteri che il sistema operativo **DOS** non accetta, quali ad esempio **barre**, **punto interrogativo**, **segno di uguale**, ecc.
Se farete questo **errore** il **DOS** ve lo segnalerà immediatamente con la scritta **ERROR** su uno sfondo rosso.

Ad alcuni lettori che ci avevano segnalato di non riuscire a trasferire il contenuto del dischetto nell'**Hard-Disk** perchè sul computer appariva subito la scritta **error** o la **mancanza** di un file, abbiamo consigliato di creare prima una **directory** con il nome **ST6**, poi di trasferire su questa tutto il contenuto del dischetto e da ultimo di **scompattare** il programma procedendo come segue:

Quando sul monitor appare il prompt **C:\>** dovrete scrivere:

C:\>MD ST6 poi premete Enter

In questo modo riapparirà il prompt **C:\>** e a questo punto dovrete inserire nel driver il disco che vi abbiamo fornito con la sigla **DF1170.3** o **DF1202.3** (il contenuto dei due dischetti è identico) e scrivere quanto segue:

C:\>COPY A: *.* C:\ST6 poi premere Enter

Nota = Rispettate gli **spazi**: per agevolarvi abbiamo colorato in **azzurro** le scritte che il computer farà apparire sul monitor, senza colore quelle che dovrete scrivere voi inserendo come **spaziatura** un rettangolo in **azzurro**.

Terminato di copiare il contenuto del dischetto nell'Hard-disk, apparirà nuovamente la scritta **C:\>**, ma non potrete ancora usare il programma **ST6**

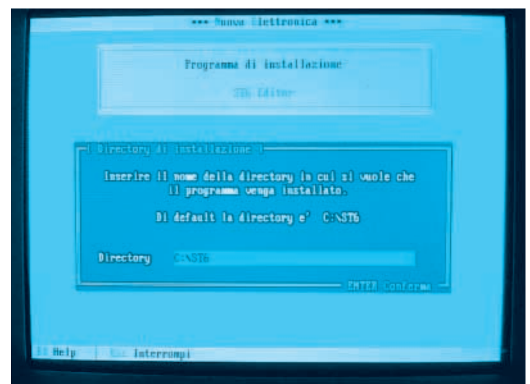


Fig.11 Quando trasferite il programma dal dischetto nell'Hard-Disk, se volete chiamare la directory ST6 in LX1202, quando vi appare questa riga sostituite C:\ST6 con la scritta C:\LX1202 e poi premete Enter.

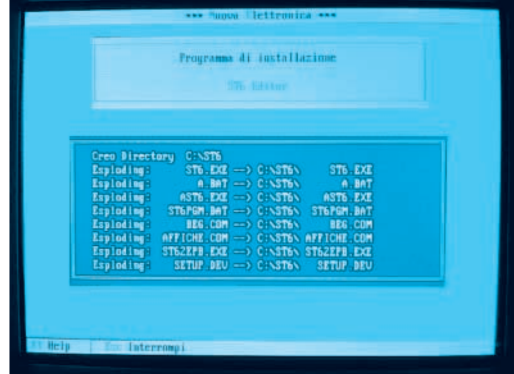


Fig.12 Immediatamente tutti i files "compattati" si scompatteranno. Ricordatevi che il programma occupa circa 1 MEGA di memoria. Non premete nessun tasto durante la fase di scompattazione.

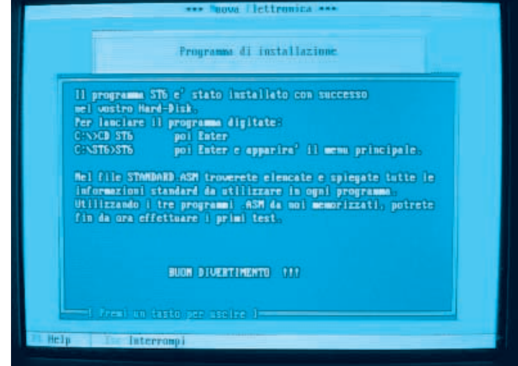


Fig.13 Solo quando sul monitor apparirà la scritta BUON DIVERTIMENTO potrete premere un qualsiasi tasto per uscire. Per richiamare il programma dovrete scrivere C:\>ST6 poi C:\ST6>ST6 ed Enter.

perchè non l'avete ancora **scompattato**.
Per farlo dovrete scrivere:

C:\>CD ST6 poi premete Enter

Quando vi apparirà **C:\ST6>**, scrivete:

C:\ST6>installa poi premete Enter

Non appena premerete il tasto **Enter**, sul monitor vi apparirà la scritta che questa **directory esiste già**, ma di ciò non preoccupatevi e **premete nuovamente Enter** e quando vi sarà chiesta la **conferma** premete per una **seconda** volta il tasto **Enter**.

Subito vedrete sul monitor tutti i nomi dei files che, abbastanza velocemente, si stanno **scompattando** (vedi fig.12).

I PROGRAMMI

Prima di trasferire tutti i files con l'estensione **.ASM** nella memoria di un micro **ST6**, li dovrete **assemblare** per ottenere un file in estensione **.HEX**
Eseguita questa operazione, nel computer troverete sempre due identici **files**, uno **ASM** ed uno **HEX**.
Se tenterete di trasferire dall'Hard-disk alla memoria di un micro **ST6** un file **ASM**, il computer segnalerà **errore** con questa scritta in inglese:

error can't open file

oppure con:

enter name of source HEX file

Per sapere se il file richiesto risulta convertito nell'estensione **HEX**, potrete procedere come segue.
Quando sul monitor del computer apparirà la finestra dell'**Editor** (vedi fig.1) premete i tasti **ALT F**, poi **F3** e quando sul monitor vi apparirà la fascia con la scritta:

***.ASM**

sostituitela con:

***.HEX** poi premete Enter

e, in questo modo, vi appariranno tutti i files in **.HEX**.

I **kit** che vi serviranno per provare tutte le nostre **schede sperimentali** sono i seguenti:

LX.1170 = Questo kit, pubblicato nella rivista **N.172/173**, serve per trasferire i programmi che abbiamo **assemblato** e convertito da **.ASM** in **.HEX** dall'Hard-disk del computer alla memoria di un **ST6 vergine**. Questo kit va collegato alla presa uscita **parallela** del computer.

LX.1202 = Questo kit, pubblicato nella rivista **N.179**, serve per ricevere tutte le **schede sperimentali** che abbiamo già pubblicato e quelle che pubblicheremo in seguito. Questa scheda, che **non va** collegata al computer, andrà alimentata con il kit **LX.1203**.

LX.1203 = Questo kit, pubblicato nella rivista **N.179**, serve per alimentare la scheda **LX.1202** e tutte le **schede sperimentali** che inserirete in questa stessa scheda.

LX.1204 = Questo kit, pubblicato nella rivista **N.179**, provvisto di **4 display** a **sette segmenti** serve per realizzare dei cronometri-orologi-timer, ecc. Questa scheda va inserita nei connettori presenti nella scheda **LX.1202**.

LX.1205 = Questo kit, pubblicato nella rivista **N.179**, provvisto di **4 relè** serve per alimentare lampade-motorini o accendere qualsiasi apparecchiatura elettronica. Questa scheda va inserita nei connettori presenti nella scheda **LX.1202**.

LX.1206 = Questo kit, pubblicato nella rivista **N.180**, provvisto di **4 Triac** serve per alimentare delle lampade - motorini o altre apparecchiature elettroniche che funzionano con **tensioni alternate**. Questa scheda va inserita nei connettori presenti nella scheda **LX.1202**.

CRONOMET.HEX

Questo programma è un semplice **cronometro**, quindi per visualizzare i tempi occorre inserire nel **bus LX.1202** la **sola** scheda dei display siglata **LX.1204**.
Se nel bus inserirete le schede dei **relè** o dei **triac**, non potrete renderle attive perchè nel programma non è presente nessuna istruzione per gestirle.
Una volta caricato su un micro **ST6** vergine il programma **CRONOMET.HEX** ed inserito nello zoccolo presente sulla scheda bus **LX.1202**, appena

alimenterete il circuito sui **4 display** apparirà il numero:

00:00

Premendo il pulsante **P1** il micro comincerà a contare in avanti ad intervalli di tempo di un **secondo**, quindi sui display vedrete apparire i numeri:

00:01 - 00:02 - 00:03 ecc.

Sui primi due display di sinistra vedrete i **minuti** e sui display di destra i **secondi**.
I **due led** che separano i display dei minuti e dei secondi lampeggeranno con una cadenza di un secondo.
Come noterete, quando si è raggiunto un tempo di **00:59 secondi**, subito dopo si passerà al tempo successivo di **01:00**, cioè **1 minuto** e **00 secondi**.
Il massimo numero che potrete visualizzare sarà quindi di **99 minuti** e **59 secondi**, dopodichè apparirà **00:00**.
Se in fase di conteggio premerete **P1**, il conteggio si **bloccherà** sul tempo raggiunto e premendolo nuovamente questo ripartirà dal numero sul quale si era fermato.
Se invece premerete il pulsante **P2**, il conteggio ripartirà da **zero**, cioè il tempo visualizzato si **azzererà**.

DISPLAY.HEX

Questo programma serve solo per far capire come si possa **visualizzare** il numero desiderato sui **4 display** della scheda siglata **LX.1204**.
Una volta caricato su un micro **ST6** vergine il programma **DISPLAY.HEX** ed inserito nello zoccolo presente sulla scheda bus **LX.1202**, non appena alimenterete il circuito sui **4 display** apparirà il numero:

12:34

e nient'altro.
Questo programma l'abbiamo composto soltanto per farvi capire come si deve scrivere una istruzione per gestire in modo seriale l'integrato **M.5450**.
Le righe da utilizzare per cambiare questo **numero** sono quelle numerate dalla numero **159** alla numero **162** del listato **DISPLAY.ASM**:

```
    ldi   bcd3,1   ;159 accende 1 sul display 1
    ldi   bcd4,2   ;160 accende 2 sul display 2
    ldi   bcd1,3   ;161 accende 3 sul display 3
    ldi   bcd2,4   ;162 accende 4 sul display 4
```

Quindi basta cambiare il numero **dopo la virgola** in questi registri per modificare il numero visualizzato.
Poichè questo programma fa molto poco, vi consigliamo di visualizzarlo solo sul monitor e di non memorizzarlo su un **ST6**.

OROLOGIO.HEX

Questo programma è un semplice **orologio.**
Per poter visualizzare le **ore** ed i **minuti** dovrete inserire nel **bus LX.1202** la **sola** scheda dei display siglata **LX.1204**.
Se nel bus inserirete le schede dei **relè** o dei **triac**, non potrete renderle attive, perchè nel programma non è presente nessuna istruzione per gestirle.
Una volta caricato su un micro **ST6** vergine il programma **OROLOGIO.HEX** ed inserito nello zoccolo presente nella scheda bus **LX.1202**, non appena alimenterete il circuito sui **4 display** apparirà il numero:

00:00

I primi due display di **sinistra** segneranno le **ore**, mentre i due di **destra** i **minuti.**
I due **diodi led** che separano i due display lampeggeranno con una cadenza di **1 secondo.**
Come noterete, raggiunte le **ore 23** ed i **59 minuti**, dopo **1 minuto** si passerà alle **24 ore** che verranno visualizzate con **00:00**.
Per mettere a **punto** le **ore** dell'orologio si utilizzerà il pulsante **P2** e, per mettere a punto i **minuti**, il pulsante **P1**.
Facciamo presente che potrete solo **far avanzare** i **numeri** e non **indietreggiare**.

RELE.HEX

Questo programma serve solo per far eccitare dei **relè** o dei **triac**, quindi nella scheda **bus LX.1202** potrete inserire la **sola** scheda dei Relè siglata **LX.1205** o la **sola** scheda dei Triac siglata **LX.1206**.
Se inserirete nel bus anche la scheda dei **display** siglata **LX.1204** non potrete renderla attiva, perchè nel programma non è presente nessuna istruzione per gestirla.
Una volta caricato su un micro **ST6** vergine il programma **RELE.HEX** ed inserito nello zoccolo presente sulla scheda bus **LX.1202**, non appena ali-

menterete il circuito tutti i relè risulteranno **diseccitati**.
Non appena premerete uno o più dei pulsanti da **P1** a **P4** si ecciterà il relè o il triac corrispondente.
Premendolo una **seconda** volta, il relè o il triac si **disecciterà**.

TEMPOR.HEX

Questo programma è un semplice **temporizzatore** con conteggio all'**indietro**.
Nel **bus LX.1202** dovrete inserire, oltre alla scheda display siglata **LX.1204**, anche la scheda Relè siglata **LX.1205**, oppure la scheda Triac siglata **LX.1206**.
Una volta caricato su un micro **ST6** vergine il programma **TEMPOR.HEX** ed inserito nello zoccolo presente nella scheda bus **LX.1202**, non appena alimenterete il circuito, tutti i relè o i triac risulteranno **diseccitati** e sui **4 display** vedrete apparire il numero:

03:00

che indica **03 minuti** e **00** secondi.
Immediatamente, partendo da questo **numero**, il conteggio inizierà a contare all'**indietro** con una cadenza di **un secondo**, quindi sui display vedrete i numeri:

02:59 - 02:58 - 02:57 ecc.

Quando apparirà il numero **00:00**, si ecciterà il solo relè **1** presente nella scheda **LX.1205** oppure il solo triac **1** presente nella scheda **LX.1206**.
Per ricominciare il ciclo basterà premere il pulsante **P1** presente sulla scheda bus **LX.1202**.
Tutti i pulsanti presenti sulla scheda del relè o del triac non risultano **attivati**.

PER cambiare i TEMPI

Per cambiare il tempo da noi prefissato **03:00** basterà modificare i valori impostati sulle righe **239** e **240** nel listato del programma **TEMPOR.ASM**.
Come tempo **massimo** potrete partire da **99 minuti** e **59 secondi**.
Attualmente sulla riga **240** che è la riga dei **minuti** troverete riportato il numero **3**:

```
      ldi   minuti,3    ;240 carica 3 minuti
```

quindi se volete partire da **12** minuti basterà semplicemente scrivere **12** al posto di **3** come qui sotto riportato:

```
      ldi   minuti,12   ;240 carica 12 minuti
```

Se volete cambiare anche i **secondi** dovrete modificare il numero sulla riga **239** che attualmente è **0**:

```
      ldi   secondi,0   ;239 carica 0 secondi
```

Ammesso che ai **12 minuti** già presenti vogliate sommare **28 secondi**, dovrete scrivere nella riga **239** il numero **28** come qui sotto riportato:

```
      ldi   secondi,28  ;239 carica 28 secondi
```

Così facendo il conteggio partirà da **12:28** e quando raggiungerà lo **00:00** si **ecciterà** il relè **1** o il triac **1**.

Nota = Anche se nella riga **241** troverete:

```
      ldi   ore,0       ;241 carica 0 ore
```

questo parametro **ore** non viene utilizzato, pertanto questa istruzione non dovrete **mai modificarla**.

Se in sostituzione del relè **1** volete eccitare un altro relè ad esempio il relè **2**, o un corrispondente triac, dovrete modificare la riga **262**:

```
loop3  set  4,port_b   ;262 accende RL1
```

sostituendo dopo il **set** il numero **4** con il numero **5** come qui sotto riportato:

```
loop3  set  5,port_b   ;262 eccita RL2
```

Se volete eccitare il relè **RL3**, dovrete sostituire il numero **4** con il numero **6**:

```
loop3  set  6,port_b   ;262 eccita RL3
```

Se volete eccitare il relè **RL4** dovete sostituire il numero **4** con il numero **7**:

```
loop3  set  7,port_b   ;262 eccita RL4
```

Se volete eccitare contemporaneamente tutti e **4** i relè, dovrete aggiungere tutte queste righe:

```
loop3  set  4,port_b   ;262 eccita RL1
       set  5,port_b   ;262.1 eccita RL2
       set  6,port_b   ;262.2 eccita RL3
       set  7,port_b   ;262.3 eccita RL4
```

In pratica, se vi interessa eccitare i soli relè **RL1**, **RL3** ed **RL4** dovrete scrivere queste tre righe:

```
loop3   set   4, port_b   ;262 eccita RL1
        set   6, port_b   ;262.1 eccita RL3
        set   7, port_b   ;262.2 eccita RL4
```

Vi ricordiamo che tutte le volte che modificherete un programma, lo dovrete **salvare** digitando il tasto **F2**, poi lo dovrete **riassemblare** premendo i tasti **ALT T**, poi il tasto **A**, dopodichè lo potrete trasferire nella **memoria** di un **ST6**.

TIMER.HEX

Questo programma è un semplice **temporizzatore** con conteggio in **avanti**.
Nel **bus LX.1202** dovrete inserire oltre alla scheda display siglata **LX.1204**, anche la scheda relè siglata **LX.1205**, oppure la scheda triac siglata **LX.1206**.
Una volta caricato su un micro **ST6** vergine il programma **TIMER.HEX** ed inserito nello zoccolo presente sulla scheda bus **LX.1202**, non appena alimenterete il circuito, tutti i relè o i triac risulteranno **diseccitati** e sui **4 display** vedrete apparire il numero:

00:00

Immediatamente, partendo da questo **numero**, il conteggio inizierà a contare in **avanti** con una cadenza di **un secondo**, quindi sui display vedrete i numeri salire:

00:01 - **00:02** - **00:03** ecc.

e quando raggiungerete il numero **03:00**, corrispondente a **3 minuti** e **00 secondi**, subito si ecciterà il relè **1** presente nella scheda **LX.1205** oppure il solo triac **1** presente nella scheda **LX.1206**.
Per ricominciare il ciclo basterà premere il pulsante **P1** presente sulla scheda bus **LX.1202**.
Tutti i pulsanti presenti sulla scheda dei relè o dei triac non risultano **attivati**.

PER cambiare i TEMPI

Per cambiare il tempo da noi prefissato **03:00** basterà modificare i valori impostati sulle righe **248** e **252** nel listato del programma **TEMPOR.ASM**.
Come tempo massimo potrete raggiungere i **99 minuti** e **59 secondi**.

Attualmente sulla riga **248**, che è la riga dei **secondi**, troverete riportato il numero **0**:

```
        cpi   a,0      ;248 compara con zero
```

e nella riga **252**, che è quella dei **minuti**, troverete riportato:

```
        cpi   a,3      ;252 compara con 3
```

Se volete ad esempio eccitare il relè **RL1** dopo **15 secondi** dall'accensione, basterà inserire nella riga **248**:

```
        cpi   a,15     ;248 compara con 15 secondi
```

ed inserire il numero **0** nella riga **252** dei **minuti**:

```
        cpi   a,0      ;252 compara con 0 minuti
```

Con queste modifiche, quando sul display apparirà il numero **00:15** il relè **RL1** si ecciterà.
Se volete far eccitare il relè dopo **30 minuti** e **5 secondi** basterà inserire il numero **5** nella riga **248**:

```
        cpi   a,5      ;248 compara con 5 secondi
```

ed inserire il numero **30** nella riga **252** dei **minuti**:

```
        cpi   a,30     ;252 compara con 30 minuti
```

Con queste modifiche, quando sul display apparirà il numero **30:05** il relè **RL1** si ecciterà.

Nota = Anche se nella riga **256** troverete riportato:

```
        ldi   ore,0    ;256 carica 0 ore
```

questo parametro **ore** non viene utilizzato, pertanto non dovrete **mai modificare** questa istruzione.

Se in sostituzione del relè **1** volete eccitare un altro relè, ad esempio il relè **2**, o un corrispondente triac, dovrete modificare la riga **259**:

```
loop3   set   4,port_b    ;259 eccita RL1
```

sostituendo dopo il **set** il numero **4** con il numero **5** come qui sotto riportato:

```
loop3   set   5,port_b    ;259 eccita RL2
```

Se volete eccitare il relè **RL3** dovrete sostituire il numero **4** con il numero **6**:

```
loop3  set  6,port_b    ;259 eccita RL3
```

Per eccitare il relè **RL4** dovrete sostituire il numero **4** con il numero **7**:

```
loop3  set  7,port_b    ;259 eccita RL4
```

Per eccitare contemporaneamente tutti e **4** i relè dovrete aggiungere le seguenti righe:

```
loop3  set  4,port_b    ;259 eccita RL1
       set  5,port_b    ;259.1 eccita RL2
       set  6,port_b    ;259.2 eccita RL3
       set  7,port_b    ;259.3 eccita RL4
```

Se vi interessa eccitare i soli relè **RL1**, **RL3** ed **RL4**, dovrete scrivere queste tre righe:

```
loop3  set  4,port_b    ;259 eccita RL1
       set  6,port_b    ;259.1 eccita RL3
       set  7,port_b    ;259.2 eccita RL4
```

Vi ricordiamo che tutte le volte che modificherete un programma, lo dovrete **salvare** premendo il tasto **F2**, poi lo dovrete **riassemblare** premendo i tasti **ALT T**, poi il tasto **A**, dopodichè lo potrete trasferire nella **memoria** di un **ST6**.

TRIAC.HEX

Questo programma è una dimostrazione di come si possano **eccitare** in sequenza dei **Triac** o dei **Relè**.

Nel **bus LX.1202** dovrete inserire la **sola** scheda siglata **LX.1206** o, in sua sostituzione, quella dei relè siglata **LX.1205**.

Una volta caricato su un micro **ST6** vergine il programma **TRIAC.HEX** ed inserito nello zoccolo presente sulla scheda bus **LX.1202**, non appena alimenterete il circuito vedrete eccitarsi in **sequenza**, con un intervallo di **1 secondo**, i quattro **triac** o i quattro **relè** se avrete inserito la scheda **LX.1205**.

Nota = In questo programma i pulsanti **P1-P2-P3-P4** presenti nella scheda dei triac o dei relè non risultano **attivati**, quindi anche se li premerete non accadrà **nulla.**

Questo programma può essere modificato per eccitare i triac in senso inverso a quello indicato, modificando anche i **tempi** oppure facendo accendere delle coppie di triac, ecc., modificando semplicemente il **numero binario** riportato nelle righe **74-76-78-80-82-84-86-88** .

```
       ldi  port_b,10000000b   ;74 eccita il triac TRC4
       ldi  port_b,01000000b   ;76 eccita il triac TRC3
       ldi  port_b,00100000b   ;78 eccita il triac TRC2
       ldi  port_b,00010000b   ;80 eccita il triac TRC1
       ldi  port_b,00100000b   ;82 eccita il triac TRC2
       ldi  port_b,01000000b   ;84 eccita il triac TRC3
       ldi  port_b,10000000b   ;86 eccita il triac TRC4
       ldi  port_b,00000000b   ;88 diseccita tutti i triac
```

Come noterete, dopo la dicitura **port_b** ci sono **otto numeri**, ma quelli che dovrete modificare sono solo i primi **quattro**, cioè **1000 - 0100 - 0010 - 0001**.

Se nella riga **88** sostituirete i quattro **0000** con **1111**, tutti i quattro triac si **ecciteranno** anzichè **diseccitarsi.**

Per far **lampeggiare** per una infinità di volte il solo **TRC4**, dovrete scrivere in tutte le righe **74-76-78-80-82-84-86** solo **1000**.

Per accendere contemporaneamente **TRC4-TRC3** dovrete scrivere nelle righe **74-76** il numero **1100.**

PER cambiare i TEMPI

Per modificare i **tempi** di intervallo tra l'accensione di un triac e quella del successivo, occorre modificare la **subroutine** chiamata **delay**.

Anche se prima non le abbiamo riportate, tra una riga e l'altra delle **74-76-78-80-82-84-86-88** troverete una istruzione **call delay** che chiama una **subroutine**:

```
       ldi   port_b,10000000b   ;74 eccita il triac TRC4
       call  delay              ;75 esegue ritardo
       ldi   port_b,01000000b   ;76 eccita il triac TRC4
```

questa **subroutine** la troverete nelle righe **59 - 60**.

```
       ldi  x,255     ;59 carica in x 255
del1   ldi  y,255     ;60 carica in y 255
```

Il **massimo** ritardo che potete ottenere è di circa **1 secondo**, perchè disponendo di registri **X-Y** ad **8 bit** non potrete mettere un numero **maggiore** di **255.**

Volendo **ridurre** il ritardo, dovrete inserire dei numeri **minori**, ad esempio per ottenere all'incirca **1/2 secondo** dovrete caricare nella **riga 59** il numero **127** come qui sotto riportato:

```
       ldi  x,127     ;59 carica in x 127
del1   ldi  y,255     ;60 carica in y 255
```

Se volete ottenere **1/4** di **secondo** dovrete cambiare le due righe come segue:

```
      ldi  x,127      ;59 carica in x 127
del1  ldi  y,127      ;60 carica in y 127
```

Se anzichè **ridurli** li voleste **aumentare**, potrete farlo utilizzando questo "trucchetto".
Tutte le istruzioni che trovate inserite tra le righe delle istruzioni dal numero **74** alla **88** con la parola **call delay**, le dovrete scrivere **due-tre-quattro** o più volte, ad esempio:

```
     ldi   port_b,10000000b   ;74 eccita TRC4
     call  delay              ;75 esegue ritardo
     call  delay              ;raddoppia tempo
     call  delay              ;triplica tempo
     ldi   port_b,01000000b   ;76 eccita TRC3
```

Quindi inserendo più o meno righe di **call delay** tra una riga e l'altra potrete variare i tempi di eccitazione tra un triac e l'altro.

CLOCK.HEX

Questo programma **CLOCK** anche se funziona come **orologio** è totalmente diverso dal precedente programma **OROLOGIO**, perchè oltre a visualizzare le **ore** e i **minuti** permette di **eccitare** un **relè** o un **triac** ad un'**ora** prestabilita e di **diseccitarlo** dopo un tempo che voi stessi potrete prefissare modificando alcune righe del programma.
Per farlo funzionare occorre inserire nel **bus LX.1202** la scheda dei display siglata **LX.1204** e quella dei relè siglata **LX.1205**, oppure quella dei triac siglata **LX.1206.**
Prima di spiegarvi quali righe dovrete modificare, consigliamo ai meno esperti di **leggere attentamente** tutto l'articolo, dopodichè potranno modificare i **parametri** nelle sole righe che noi indicheremo.
Come abbiamo accennato, il programma **CLOCK.HEX** ci dà la possibilità di **eccitare** o **diseccitare** uno o più **relè** anche contemporaneamente, su orari che noi stessi potremo stabilire, purchè non si superino più di **8 cicli** o **periodi** nell'arco delle **24 ore.**
Questo programma potrà servire per **accendere** o **spegnere** una o più caldaie, delle insegne luminose ad orari prestabiliti, ecc.
Per mettere a **punto** le **ore** dell'orologio si utilizzerà il pulsante **P2** e per mettere a punto i **minuti** il pulsante **P1**.

Facciamo presente che è possibile soltanto far **avanzare** i **numeri** e non **indietreggiare**.
Appena accenderete l'orologio **tutti** i **4 relè** o **triac** partiranno **eccitati**.
Se volete che all'accensione dell'orologio tutti i relè risultino **diseccitati**, dovrete andare alla riga **N.59** dove troverete questa istruzione:

```
     ldi   port_b,11110011b
```

e modificarla inserendo in sostituzione degli **1** degli **0** come qui sotto riportato:

```
     ldi   port_b,00000011b
```

Nota = Anche se questa riga è composta da **8 numeri**, dovrete modificare solo i primi **4** di sinistra.

Se volete far **eccitare** all'accensione il solo relè **RL4**, dovrete mettere un **1** in corrispondenza della prima cifra di sinistra come qui sotto riportato:

```
     ldi   port_b,10000011
```

A questo punto vi spieghiamo che cosa s'intende per **8 cicli** o **periodi** da utilizzare nell'arco delle **24 ore** che troverete riportati in queste righe:

1° periodo = righe 309 - 310 - 311
2° periodo = righe 316 - 317 - 318
3° periodo = righe 323 - 324 - 325
4° periodo = righe 330 - 331 - 332
5° periodo = righe 337 - 338 - 339
6° periodo = righe 344 - 345 - 346
7° periodo = righe 351 - 352 - 353
8° periodo = righe 358 - 359 - 360

Ogni ciclo è composto da **3 righe** d'istruzioni, quindi nel **1° ciclo** o **periodo** del nostro programma troverete:

```
     .byte  02          ;309 riga delle ore
     .byte  30          ;310 riga dei minuti
     .byte  11100000b   ;311 riga per comando relè
```

Attualmente il **1° ciclo** inizia alle ore **2,30 di notte.**
Per modificare l'orario basterà mettere nella **prima** riga l'**ora** che vi interessa, ad esempio **05-06-10**, e nella **seconda** riga i relativi **minuti**, ad esempio **00 - 10 - 30 - 50**.
Nella **terza** riga sono riportati i relè che desiderate **eccitare** e quelli che **non** desiderate eccitare all'orario da noi prestabilito.
Mettendo un **1** il relè si **ecciterà**, mettendo uno **0** si **disecciterà**.
Ciò che dovrete modificare in questa terza riga so-

no **solo** i primi **quattro numeri** di sinistra posti dopo la parola **byte.**
Tenete presente che il **primo** numero di sinistra **piloterà** il relè **RL4** e l'**ultimo** numero di destra il relè **RL1**, quindi avrete in ordine:

RL4-RL3-RL2-RL1

Per farvi capire come modificare tutti questi numeri vi faremo un semplice esempio.
Ammesso che alle **06,10** desideriate **eccitare** i relè **RL2-RL1**, scriverete nelle righe **309 - 310 - 311** (1° ciclo) questi numeri:

```
.byte   06          ;309 riga delle ore
.byte   10          ;310 riga dei minuti
.byte   00110000b   ;311 riga per comando relè
```

Se alle **09,30** vorrete **diseccitare** ed **eccitare** il solo relè **RL4**, dovrete scrivere nelle righe **316 - 317 - 318** (2° ciclo) questi numeri:

```
.byte   09          ;316 riga delle ore
.byte   30          ;317 riga dei minuti
.byte   10000000b   ;318 riga per comando relè
```

Se alle **12,00** vorrete **diseccitare** anche il relè **RL4**, dovrete scrivere nelle righe **323 -324 -325** (3° ciclo) questi numeri:

```
.byte   12          ;323 riga delle ore
.byte   00          ;324 riga dei minuti
.byte   00000000b   ;325 riga per comando relè
```

Se alle **18,45** vorrete **eccitare** tutti i relè, dovrete scrivere nelle righe **330 - 331 - 332** (4° ciclo) questi numeri:

```
.byte   18          ;330 riga delle ore
.byte   45          ;331 riga dei minuti
.byte   11110000b   ;332 riga per comando relè
```

Se alle **22,30** vorrete **diseccitare** i relè **RL4-RL3**, dovrete scrivere nelle righe **337 - 338- 339** (5° ciclo) questi numeri:

```
.byte   22          ;337 riga delle ore
.byte   30          ;338 riga dei minuti
.byte   00110000b   ;339 riga per comando relè
```

Se alle **23,40** vorrete lasciare **eccitato** il solo relè **RL1**, dovrete scrivere nelle righe **344 - 345 - 346** (6° ciclo) questi numeri:

```
.byte   23          ;344 riga delle ore
.byte   40          ;345 riga dei minuti
.byte   00010000b   ;346 riga per comando relè
```

Se alle **24,00** vorrete **diseccitare** anche questo relè, dovrete scrivere nelle righe **351 - 352 - 353** (7° ciclo) questi numeri:

```
.byte   00          ;351 riga delle ore
.byte   00          ;352 riga dei minuti
.byte   00000000b   ;353 riga per comando relè
```

Avendo utilizzato solo **7 cicli** degli **8** disponibili, se l'ultimo non vi interessa lo **potrete cancellare** oppure inibire, mettendo davanti alle righe **358 - 359 - 360** un **punto** e **virgola** o mettendo sulla terza riga **000000b.**
Potrete **aggiungere** altri cicli se **8** risultassero insufficienti.

Facciamo presente che questi **cicli** si **ripeteranno** automaticamente all'**infinito** agli stessi **orari** tutti i giorni.

TIME90.HEX

Questo programma è un **timer** che, contando in **avanti**, ecciterà un relè o un triac quando raggiungerà i **minuti** e i **secondi** da noi prefissati.
Per farlo funzionare occorre inserire nel **bus LX.1202** la scheda dei display siglata **LX.1204** e quella dei relè siglata **LX.1205**, oppure quella dei triac siglata **LX.1206.**
Non appena alimenterete il circuito, il conteggio partirà da **00:00** e inizierà a contare in **avanti**; a questo punto potrete utilizzare i pulsanti **P1** e **P2** presenti sulla scheda display **LX.1204.**

Premendo **P1** il conteggio si **ferma**.
Premendo nuovamente **P1** il conteggio riparte dal **numero** sul quale si era fermato.
Premendo **P2** il contatore si **resetta.**
Premendo **P1** il contatore riparte da **00:00.**

Nota = Il pulsante **P2** di **reset** sarà attivo solamente se avrete **fermato** il conteggio con **P1.** Se premerete **P2** mentre è **attivo** il conteggio, questo non si azzererà.
I pulsanti presenti sulle schede Triac e Relè non risultano **attivati.**
Il conteggio del display arriva ad un massimo di **89 minuti** e **59 secondi.**

Il programma **TIME90.ASM**, come potrete notare, dispone di **4 cicli** perchè **quattro** sono i **relè** e i **triac** presenti sulle schede sperimentali.

1° ciclo = Dopo **20 secondi** dall'accensione si ecciterà il solo relè **RL1**.

Ovviamente sui display vedrete apparire **19** e, quando questo numero raggiungerà **00:00**, il relè si **ecciterà.**

2° ciclo = Passando al **secondo ciclo**, questo relè rimarrà **eccitato** per un tempo da noi prefissato in **1 minuto** e **30 secondi** e raggiunto questo **tempo** il relè **RL1** si **disecciterà** e automaticamente si **ecciterà** il relè **RL2**.
Il relè **RL2** si ecciterà un secondo dopo che sui display sarà apparso il numero **01:29** che cambierà in **00:00**.

3° ciclo = Dopo **47 secondi**, cioè quando sul display il numero **46** passerà sullo **00**, il relè **RL2** si **disecciterà** e si **ecciterà** il terzo relè **RL3**.

4° ciclo = Il conteggio continuerà ed allo scoccare dei **3 minuti** e **00 secondi** (tempo da noi prefissato) si **disecciterà** il relè **RL3** e si **ecciterà** il relè **RL4**, cioè si ritornerà al **1° ciclo** per ripetere all'infinito i **quattro cicli**.

Per **variare** i **tempi** che noi abbiamo prefissato dovrete variare queste righe:

1° ciclo = righe 289 - 290
2° ciclo = righe 295 - 296
3° ciclo = righe 301 - 302
4° ciclo = righe 307 - 308

Se volete che il **1° ciclo** abbia una durata di **1 minuto** e **30 secondi**, dovrete inserire nelle sue righe questi numeri:

```
    ldi   stsex,30    ;289 secondi per RL1
    ldi   stmix,1     ;290 minuti per RL1
```

Se volete che il **2° ciclo** abbia una durata di **50 secondi**, dovrete inserire nelle sue righe questi numeri:

```
    ldi   stsex,50    ;295 secondi per RL1
    ldi   stmix,00    ;296 minuti per RL1
```

Se volete che il **3° ciclo** abbia una durata di **15 minuti** e **20 secondi**, dovrete inserire nelle sue righe questi numeri:

```
    ldi   stsex,20    ;301 secondi per RL1
    ldi   stmix,15    ;302 minuti per RL1
```

Se volete che il **4° ciclo** abbia una durata di **2 minuti** e **10 secondi**, dovrete inserire nelle sue righe questi numeri:

```
    ldi   stsex,10    ;307 secondi per RL1
    ldi   stmix,2     ;308 minuti per RL1
```

Nelle righe **292/293 - 298/299 - 304/305 - 310/311** sono riportate le sigle dei **relè** che volete **eccitare** e di quelli che volete rimangano **diseccitati.**
Guardando l'esempio riportato nel programma **CLOCK.ASM** saprete già che scrivendo questa istruzione:

```
    ldi   port_b,11110011b
```

potrete eccitare ad ogni **ciclo** anche **più relè** a vostra scelta.
Nei primi **quattro** numeri di sinistra (vedi **1111**) dovrete mettere un **1** sul relè che volete far **eccitare** ed uno **0** se **non** lo volete eccitare.

TEMP90.HEX

Questo programma è un **timer** che fa esattamente l'**inverso** del programma **TIME90**, cioè **conta all'indietro** e quando raggiunge lo **00:00** eccita i relè.
I relè, come per il programma precedente, li ecciterete in **4 cicli** e come tempo **massimo** di partenza potrete impostare **90 minuti** e **00 secondi.**
Non appena alimenterete il circuito, il conteggio partirà da **00:20** (questo tempo lo abbiamo prescelto noi, ma poi vi spiegheremo come modificarlo) e procederà all'**indietro.**
Dopo che avrà avuto inizio il conteggio, potrete utilizzare i pulsanti **P1** e **P2** presenti sulla scheda display **LX.1204**.

Premendo **P1** il conteggio si **ferma**.
Premendo nuovamente **P1** il conteggio riparte dal **numero** sul quale si era fermato.
Premendo **P2** il contatore si **resetta**.
Premendo **P1** il contatore riparte dal tempo che avete impostato come **partenza** per il conteggio all'**indietro**.

Nota = Il pulsante **P2** di **reset** sarà attivo solamente se avrete **fermato** il conteggio con **P1**. Se premerete **P2** mentre è **attivo** il conteggio, questo non si azzererà. Premendo **P2** per **resettarlo**, è intuitivo che contando all'**indietro** sul display ritorni il tempo di **partenza**, cioè **00:20**.

Nei **4 cicli** impostati otterrete queste condizioni:

1° ciclo = All'accensione si ecciterà il solo relè **RL1** e sui display apparirà **00:20** e a questo punto avrà inizio il conteggio alla **rovescia** che si fermerà sullo **00:00**.

2° ciclo = Dopo un secondo si ecciterà il relè **RL2** e a questo punto inizierà il **secondo ciclo**, che farà apparire sui display **01:30** (tempo **1 minuto** e **30 secondi**) che, secondo per secondo, decrementerà fino ad arrivare a **00:00**.

3° ciclo = A questo punto si **ecciterà** il relè **RL3** e sui display apparirà **00:47** che decrementerà fino ad arrivare allo **00:00**.

4° ciclo = L'ultimo ciclo farà eccitare il relè **RL4** e farà apparire sui display il numero **03:00** (**3 minuti**). Quando con il conteggio alla **rovescia** si arriverà al numero **00:00**, questo relè si **disecciterà** e contemporaneamente si **disecciterà** il relè **RL1**, cioè si ritornerà al **1° ciclo** per ripetere all'infinito i **quattro cicli**.

Per **variare** i **tempi** prefissati dovrete variare queste righe:

1° ciclo = righe 290 - 291
2° ciclo = righe 296 - 297
3° ciclo = righe 302 - 303
4° ciclo = righe 308 - 309

Se volete che il **1° ciclo** abbia una durata di **1 minuto** e **30 secondi**, dovrete inserire nelle sue righe questi numeri:

```
    ldi    stsex,30    ;290 secondi per RL1
    ldi    stmix,1     ;291 minuti per RL1
```

Se volete che il **2° ciclo** abbia una durata di **50 secondi**, dovrete inserire nelle sue righe questi numeri:

```
    ldi    stsex,50    ;296 secondi per RL1
    ldi    stmix,00    ;297 minuti per RL1
```

Se volete che il **3° ciclo** abbia una durata di **15 minuti** e **20 secondi**, dovrete inserire nelle sue righe questi numeri:

```
    ldi    stsex,20    ;302 secondi per RL1
    ldi    stmix,15    ;303 minuti per RL1
```

Se volete che il **4° ciclo** abbia una durata di **2 minuti** e **10 secondi**, dovrete inserire nelle sue righe questi numeri:

```
    ldi    stsex,10    ;308 secondi per RL1
    ldi    stmix,2     ;309 minuti per RL1
```

Nelle righe **292/293 - 298/299 - 304/305 - 310/311** sono riportate le sigle dei **relè** che si **ecciteranno** e di quelli che si **disecciteranno**.
Anche in questo programma possiamo sostituire le righe sopra menzionate con questa sola riga d'istruzione:

```
    ldi    port_b,11110011b
```

Nei primi **quattro** numeri di sinistra (vedi **1111**) dove metterete **1** il relè si **ecciterà**, dove metterete **0** si **disecciterà**.

NOTA

I programmi che vi abbiamo fornito e che in seguito vi forniremo sono a sfondo **didattico** e servono per far capire ai principianti come si debba scrivere un'istruzione per ottenere una specifica funzione e per questo abbiamo aggiunto, di fianco ad ogni riga di programma, un **commento**.
Blocchi di un programma si possono prelevare e trasferire su un altro programma, cosa che potrete fare quando avrete già acquisito una certa esperienza.

COSTO DI REALIZZAZIONE

Tutti i componenti necessari per la realizzazione di questa scheda, compresi circuito stampato, connettore, triac da 500 Volt 5 Amper, fotoaccopiatori, zoccoli, pulsanti, morsettiere, condensatori e resistenze (vedi fig.8) ....€ 18,60

Il solo circuito stampato **LX.1206** ....€ 4,91

Nota = Nel kit **non è incluso** il dischetto dei programmi **DF1202.3** perchè già fornito agli acquirenti del kit **LX.1202**.
A chi non avesse acquistato questo kit e volesse il solo dischetto dei programmi, possiamo inviarlo a € 6,20 più le spese postali.

Ai prezzi riportati, già comprensivi di IVA, andranno aggiunte le sole spese di spedizione a domicilio.

SCHEDA con DISPLAY

Dopo avervi presentato una scheda per eccitare dei relè, una per pilotare dei triac e una terza per pilotare dei normali display a 7 segmenti, oggi vogliamo proporvi una scheda per display numerici LCD dopodichè passeremo ai display alfanumerici.

Chissà quanti di voi essendo in possesso di un display **LCD** avranno tentato di far apparire un numero senza riuscirci, pur avendolo alimentato correttamente, collegato i vari **segmenti** e controllato più volte il montaggio per verificare di non aver commesso involontariamente qualche errore.
Il motivo per il quale non siete riusciti ad accenderlo ve lo spiegheremo in questo articolo, quindi ammesso che non vi interessi sapere come si programmi un microprocessore **ST6** per pilotare un display **LCD**, vi consigliamo ugualmente di leggerlo.

GESTIRE un DISPLAY a 7 SEGMENTI

Se volessimo pilotare **4 display** a **7 segmenti** con un integrato **driver**, questo dovrebbe avere come minimo **32 piedini** d'uscita, infatti oltre ai **7 segmenti** presenti su ogni display per un totale di **4 x 7 = 28** terminali, essendo presente su ognuno di questi il **punto decimale**, dovremo aggiungere altri **4** terminali e, in tal modo, otterremo un totale di **32 piedini**.
Come già saprete, per accendere uno o più **segmenti** basterà alimentarli con una tensione che potremo prelevare da una normale pila.
Per pilotare **4 display** esistono degli integrati provvisti di **34** piedini d'**uscita** necessari per accendere tutti i vari segmenti.
Per l'**ingresso** troviamo invece solo **3 piedini** perchè questi lavorano in modo **seriale**.
Questi **3** piedini d'**ingresso** sono chiamati:

Enable
Data
Clock

Nell'integrato **driver** tipo **M.5450** in grado di pilotare **4 display** a **7 segmenti**:

- il piedino **23** dell'**Enable** lo dovremo collegare a **massa**.
- il piedino **22** del **Data** serve per ricevere i **dati seriali** necessari per far accendere i vari segmenti.
- il piedino **21** del **Clock** serve per ricevere un treno di onde quadre per **sincronizzare** i segnali.
I due piedini supplementari d'uscita corrispondenti ai bit **33-34** potrebbero risultare utili per far lampeggiare ad una cadenza di **1 secondo** due diodi led interposti tra i due numeri delle **ore** e dei **minuti**, se realizzeremo un orologio.
I piedini d'uscita di questo integrato, come visibile in fig.1, vengono indicati **BIT1 - BIT2 - BIT3**, ecc.
Quando tutti i segmenti dei display sono **spenti**, nel piedino del **clock** entrerà un treno di onde quadre e nel piedino **data** il solo impulso di **Start** (bit **0**) e due impulsi, uno di **Load** (bit **35**) ed uno di **Reset** (bit **36**)(vedi fig.2).
A questo punto potremo assegnare ciascuno di questi **34 bit** ad un singolo **segmento** presente in ogni display come visibile in fig.2 e come riportato nella Tabella N.1).

Se in questi **4 display** volessimo accendere il numero **1 0 3 2** tramite il **programma software**, dovremmo portare a **livello logico 1** tutti i bit corrispondenti ai segmenti che desideriamo accendere.

Sul **display N.1**, dovendo accendere i segmenti **B-C**, dovremo portare a **livello logico 1** i **bit 2-3**.
Sul **display N.2**, dovendo accendere i segmenti **A-B-C-D-E-F**, dovremo portare a **livello logico 1** i **bit 9-10-11-12-13-14**.
Sul **display N.3**, dovendo accendere i segmenti **A-B-C-D-G**, dovremo portare a **livello logico 1** i **bit 17-18-19-20-23**.
Sul **display N.4**, dovendo accendere i segmenti **A-B-G-E-D**, dovremo portare a **livello logico 1** i **bit 25-26-31-29-28** (vedi fig.3).

LCD pilotata con un ST6

TABELLA N.1

BIT comando	segmento DISPLAY 1	piedino INTEGRATO
1	A	18
2	B	17
3	C	16
4	D	15
5	E	14
6	F	13
7	G	12
8	punto	11

BIT comando	segmento DISPLAY 2	piedino INTEGRATO
9	A	10
10	B	9
11	C	8
12	D	7
13	E	6
14	F	5
15	G	4
16	punto	3

Per accendere un numero su un normale display a 7 segmenti è sufficiente portare a "livello logico 1" i bit che pilotano i segmenti interessati (vedi figg.2-3).

BIT comando	segmento DISPLAY 3	piedino INTEGRATO
17	A	2
18	B	40
19	C	39
20	D	38
21	E	37
22	F	36
23	G	35
24	punto	34

BIT comando	segmento DISPLAY 4	piedino INTEGRATO
25	A	33
26	B	32
27	C	31
28	D	30
29	E	29
30	F	28
31	G	27
32	punto	26

BIT comando	diodi LED	piedino INTEGRATO
33	led 1	24
34	led 2	25

Come avrete intuito, basta portare a **livello logico 1** uno di questi **34 bit** per far apparire un **numero** oppure anche una **lettera**, ad esempio una **L**, una **C** o una **H**.
Tramite software, il **display 1** potremo farlo diventare il **4°** oppure il **3°** in modo da adattarlo al disegno del circuito stampato.
Un altro particolare **molto importante** da sottolineare è quello di non confondere il **livello logico** del **bit** con il livello logico che in pratica ci ritroveremo sul **piedino d'uscita** dell'integrato che, come potrete constatare, risulta **invertito**, quindi portando ad esempio i **bit 25-26** a **livello logico 1**, sui corrispondenti piedini **33-32** dell'integrato **M.5450** (vedi Tabella N.1) ci ritroveremo un **livello logico 0**.
Quindi se qualcuno andasse a controllare con un tester la tensione presente sui piedini d'uscita che **accendono** un segmento, troverebbe **0 volt** e non una tensione **positiva**.

GESTIRE un DISPLAY LCD

Come già spiegato, se volessimo pilotare un **display a led** con **4 cifre** con un integrato **driver**, questo dovrebbe avere **34 piedini** d'uscita.
Per quanto riguarda i display **LCD** dobbiamo far presente che il **punto decimale** che dovrebbe risultare presente sul **4° display** spesso viene sostituito da **due punti** posti tra le prime due e le ultime due cifre.
Questi **due punti** vengono spesso utilizzati negli orologi per dividere le **ore** dai **minuti**.
A differenza del **drive M.5450**, utilizzato per pilotare i display a **7 segmenti**, che dispone di tre **ingressi** chiamati **Enable-Data-Clock**, il **drive M.8438/AB6**, utilizzato per i display **LCD**, dispone di tre **ingressi** chiamati:

Strobe
Data
Clock

- Il piedino **2** dello **Strobe** si porta a **livello logico 1** ogni volta che deve inviare al display il treno dei **dati**.
- Il piedino **34** del **Data** serve per ricevere i **dati seriali** necessari per far accendere i vari segmenti.
- Il piedino **40** del **Clock** invia un treno di onde quadre per **sincronizzare** i segnali (vedi fig.5).

Come potrete notare, il **drive** per un display **LCD** ha l'ingresso **Strobe** che mancava nel **drive** per display a **7 segmenti a led** ed ha in più un piedino chiamato **Back/Plane** (piedino **30**) che viene utilizzato per alimentare il display con un'onda quadra di circa **80 Hertz**.
Quindi i quattro display **LCD** non vengono alimentati da una **tensione continua**, ma da una tensione che cambia in continuità il suo **livello logico** da **1** a **0** e questo, come vedremo, è indispensabile per poter **accendere** i vari segmenti.

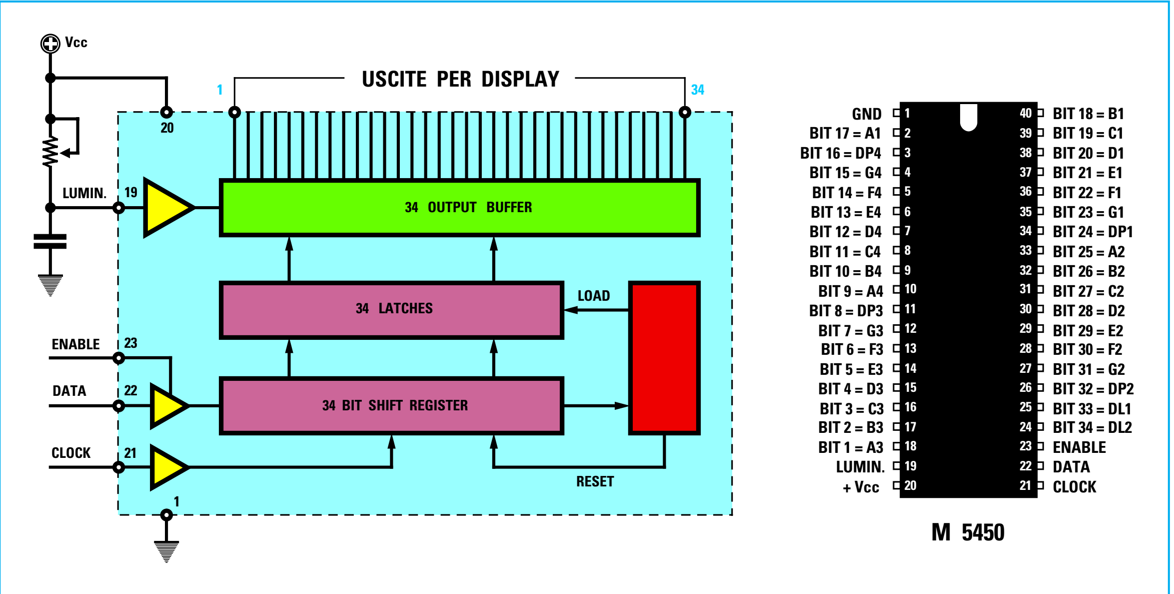


Fig.1 Per pilotare 4 display a 7 segmenti occorre un integrato drive tipo M.5450 o altri equivalenti. Dei tre piedini d'ingresso, quello indicato Enable viene collegato a massa, e quelli indicati Data e Clock vengono utilizzati per ricevere i dati seriali e i segnali di sincronismo.

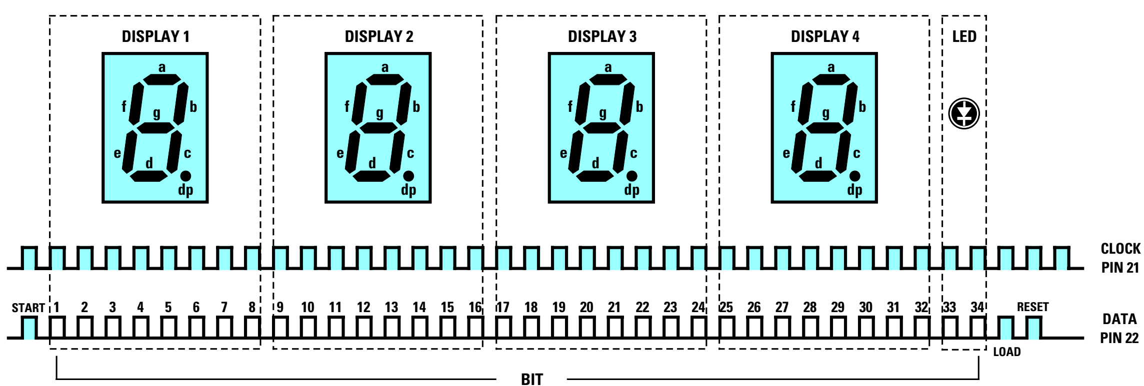


Fig.2 Poiché per ogni display sono necessari 8 bit (uno serve per il punto decimale - vedi Tabella N.1), l'integrato drive M.5450 convertirà i dati seriali applicati sugli ingressi in dati paralleli a 34 bit. Oltre a questi 34 bit l'integrato invia in uscita un bit di Start, uno di Load (carica dati) ed uno di Reset.

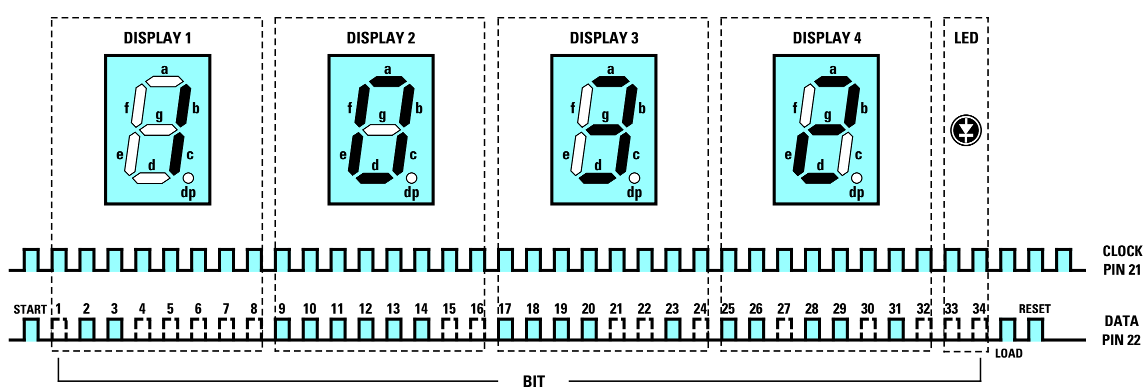


Fig.3 Per accendere il numero "1032", sul primo display dovremo portare a livello logico 1 i bit 2-3, sul secondo display i bit 9-10-11-12-13-14, sul terzo display i bit 17-18-19-20-23 e sull'ultimo display i bit 25-26-28-29-31. Non confondete il livello logico del "bit" con quello che apparirà sul piedino d'uscita che risulta invertito.

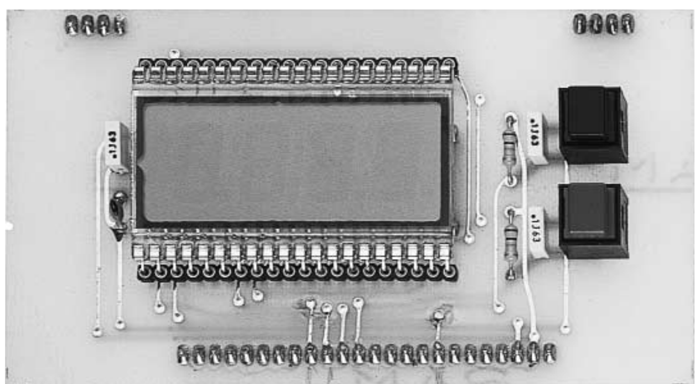


Fig.4 Per pilotare un display LCD dovremo sempre portare a "livello logico 1" i bit dei segmenti che vogliamo accendere, ma come potete vedere in fig.7 i segmenti che rimangono spenti non vengono portati a "livello logico 0".

I piedini d'uscita di questo integrato, come visibile in fig.5, indicati **A1 - B1 - C1**, ecc., andranno collegati ad ogni singolo **segmento** presente in ciascun display (vedi fig.6 e Tabella N.2).

TABELLA N.2

BIT comando	segmento DISPLAY 1	piedino INTEGRATO
1	A	10
2	B	9
3	C	8
4	D	7
5	E	6
6	F	5
7	G	4
8	punto	11

BIT comando	segmento DISPLAY 2	piedino INTEGRATO
9	A	18
10	B	17
11	C	16
12	D	15
13	E	14
14	F	13
15	G	12
16	punto	19

BIT comando	segmento DISPLAY 3	piedino INTEGRATO
17	A	26
18	B	25
19	C	24
20	D	23
21	E	22
22	F	21
23	G	20
24	punto	27

BIT comando	segmento DISPLAY 4	piedino INTEGRATO
25	A	39
26	B	38
27	C	37
28	D	33
29	E	32
30	F	29
31	G	28
32	punto	3

Anche in un **display LCD** se volessimo accendere i numeri **1 0 3 2** tramite il **programma software**, dovremmo portare a **livello logico 1** tutti i segmenti che desideriamo accendere.

Sul **display N.1**, dovendo accendere i segmenti **B-C**, dovremo portare a **livello logico 1** i **bit 2-3**.
Sul **display N.2**, dovendo accendere i segmenti **A-B-C-D-E-F**, dovremo portare a **livello logico 1** i **bit 9-10-11-12-13-14**.
Sul **display N.3**, dovendo accendere i segmenti **A-B-C-D-G**, dovremo portare a **livello logico 1** i **bit 17-18-19-20-23**.
Sul **display N.4**, dovendo accendere i segmenti **A-B-G-E-D**, dovremo portare a **livello logico 1** i **bit 25-26-31-29-28**.

A questo punto si potrebbe pensare che un **drive** per pilotare un display a **7 segmenti** possa servire anche per pilotare un display **LCD**, ma purtroppo questo non è possibile perchè i suoi **segmenti** non vengono pilotati da una **tensione continua**, ma da un treno di **onde quadre** che lo stesso integrato **M.8438/AB6** invia al segmento da **accendere**, sfasandolo di **180°** rispetto alle onde quadre che giungono sui segmenti che debbono rimanere **spenti**.

Quindi se controllassimo con un oscilloscopio tutti i **32 piedini** d'uscita di questo integrato, troveremmo in **tutti** delle onde quadre, perciò potrebbe risultare alquanto complesso capire come si riescano ad accendere questi **segmenti**.

Per spiegarvelo in modo che possiate comprenderlo perfettamente vi proponiamo questo semplice esempio.
Ammesso di voler accendere in un display i segmenti **B-C** in modo da far apparire il numero **1**, in questi due **soli** segmenti giungeranno delle onde quadre **invertite** rispetto alle onde quadre che giungono invece sul **Back/Plane** (vedi fig.7).
Di questa inversione del segnale ad onda quadra non dovremo preoccuparci, perchè sarà l'integrato **M.8438/AB6** che provvederà ad **invertirlo** quando, tramite software, gli diremo di accendere i segmenti **B-C**.

Il software per i display **LCD** in pratica è molto similare a quello per i display a **7 segmenti**, sempre che si rispettino le connessioni d'uscita dell'integrato **M.8438/AB6** con i piedini che fanno capo ai segmenti dei quattro **display** (vedi Tabelle **N.1** e **N.2**).

Nelle connessioni del display tipo **S.5126** oppure **LC.513040** riportate in fig.8, abbiamo indicato con **A1-B1-C1**, ecc., tutti i segmenti del display **N.1**, con **A2-B2-C2**, ecc., tutti i segmenti del display **N.2** e così dicasi per il display **N.3** e **N.4**.
Se abbiamo un display con una sigla diversa, do-

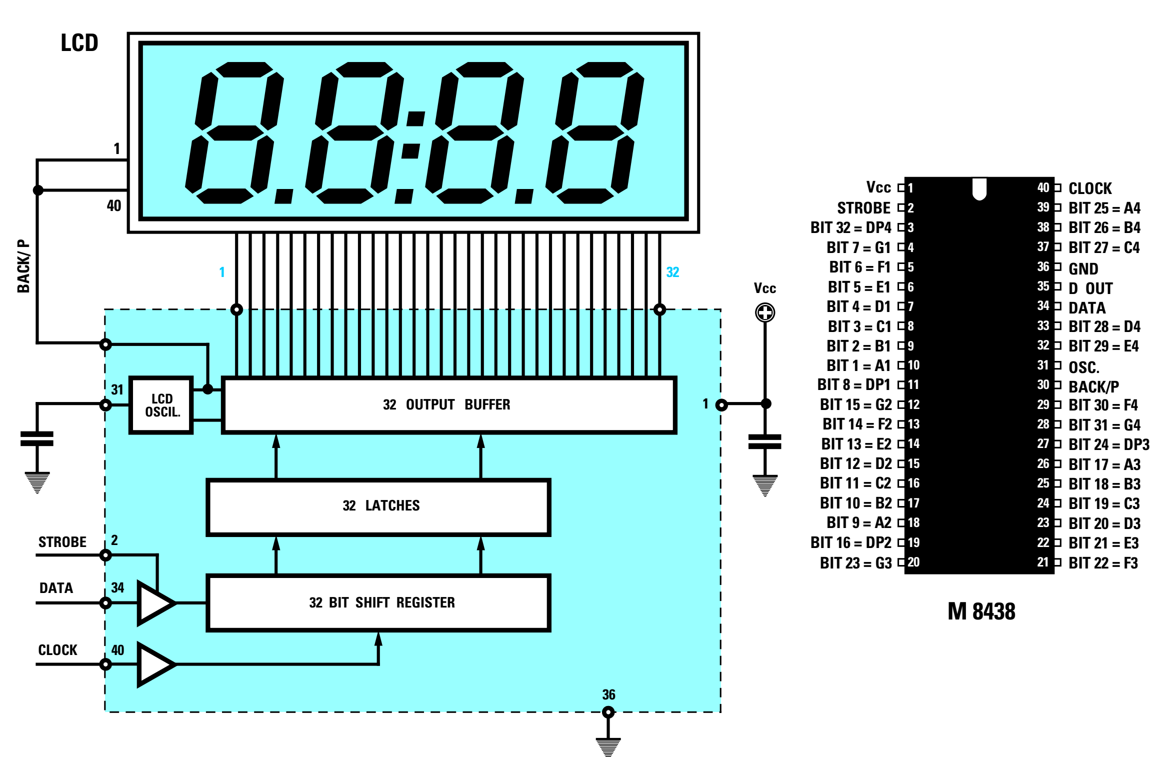


Fig.5 Per pilotare 4 display LCD occorre un integrato drive tipo M.8438 o altri equivalenti. Dei tre piedini d'ingresso, quello indicato Strobe si porta a livello logico 1 quando deve caricare i dati, quelli indicati Data e Clock vengono utilizzati per ricevere i dati seriali e i segnali di sincronismo. Il display LCD viene alimentato direttamente dall'integrato M.8438 tramite il piedino Back/Plane.

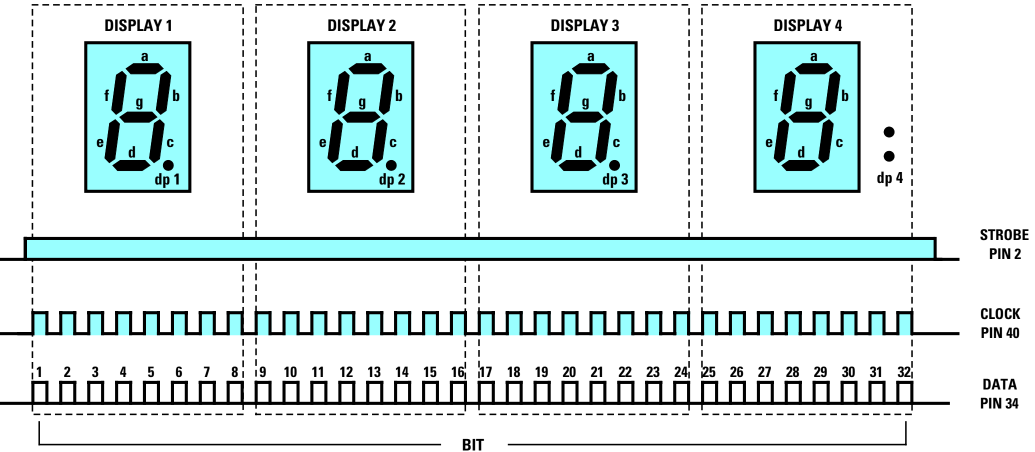


Fig.6 Ogni bit corrisponde ad un singolo segmento di ogni display (vedi Tabella N.2) e per agevolarvi abbiamo riportato su ogni piedino dell'integrato M.8438 (vedi fig.5) sia il numero del bit sia quale segmento si accenderà sui display 1-2-3-4.

vremo controllare a quali piedini fanno capo i segmenti **A-B-C-D-E-F-G** del primo, del secondo, del terzo e del quarto display, per collegarli in ordine ai piedini d'uscita dell'integrato **drive.**

SCHEMA ELETTRICO

Dopo avervi spiegato la differenza che esiste tra un display a **7 segmenti a led** ed uno a **LCD** possiamo passare a presentarvi lo schema elettrico.
Guardando la fig.9 è possibile notare che dal connettore siglato **CONN. 1/2** preleviamo:

- dal piedino **B0** i **dati** da inviare al piedino **34** dell'integrato **M.8438/AB6.**
- dal piedino **B1** il segnale del **Clock** da inviare al piedino **40** di **IC1**.
- dal piedino **C4** il segnale di **Strobe** da inviare al piedino **2** di **IC1**.
- dal piedino **+5V** la tensione di alimentazione per l'integrato.

Gli altri due piedini **B2-B3** sono collegati ai pulsanti **P1-P2** che ci servono, realizzando un orologio, per mettere a punto le ore e i minuti.
Poichè l'integrato **M.8438/AB6** ha bisogno del segnale di **Strobe**, dovremo necessariamente utilizzare dei microprocessori **ST6** con **28** piedini, cioè:

ST62E25 (da 4K **cancellabile**)
ST62T25 (da 4K **non cancellabile**)
ST62T15 (da 2K **non cancellabile**)

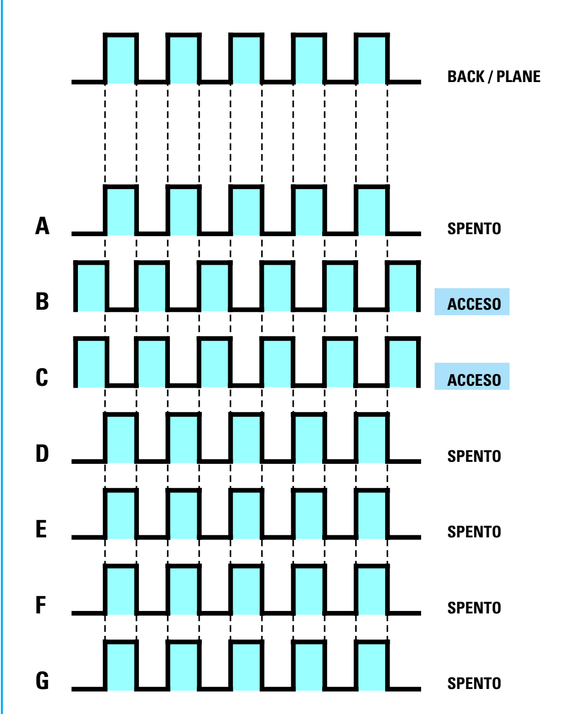
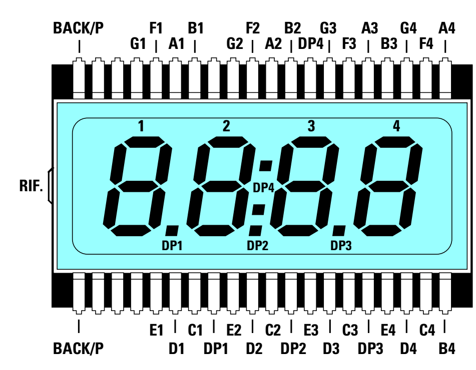
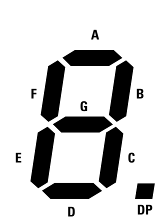


Fig.7 Per accendere un segmento in un display LCD l'integrato IC1 invierà sul piedino corrispondente un'onda quadra sfasata di 180° rispetto a quella del Back/Plane. In questo esempio risultano accesi i soli segmenti B-C.



LC 513040 o S 5126



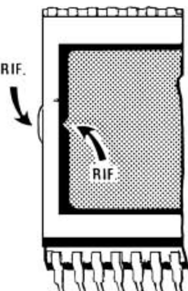


Fig.8 Connessioni del display LCD. Su ogni terminale abbiamo riportato la lettera dei sette segmenti A-B-C ecc. seguita dai numeri del display cioè 1-2-3-4. La tacca di riferimento è costituita da una "goccia" di vetro o dal segno ">" posto da un lato del corpo.

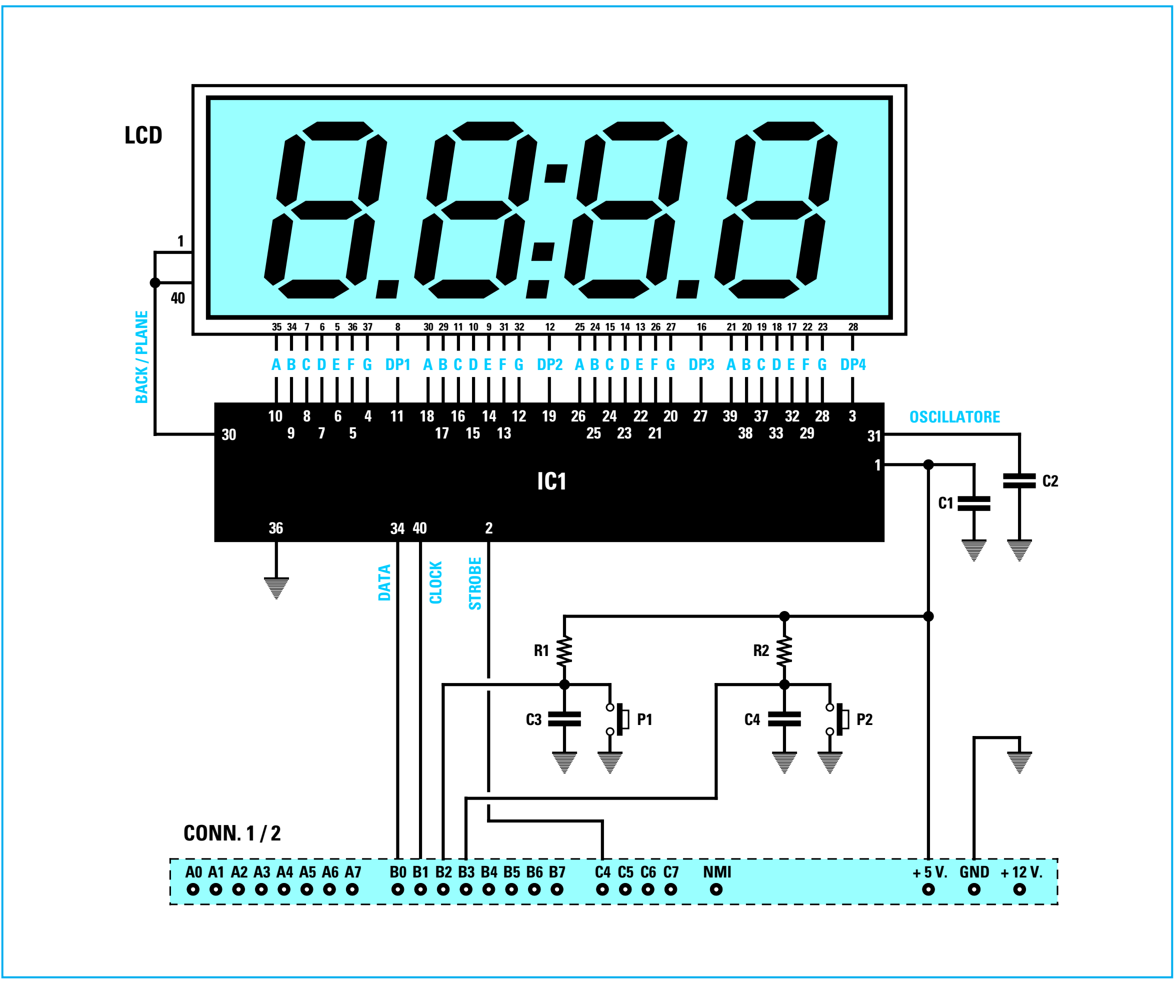


Fig.9 Schema elettrico da noi utilizzato per pilotare un display LCD.

ELENCO COMPONENTI LX.1207

R1 = 10.000 ohm 1/4 watt
R2 = 10.000 ohm 1/4 watt
C1 = 100.000 pF poliestere
C2 = 22 pF ceramico
C3 = 100.000 pF poliestere
C4 = 100.000 pF poliestere
IC1 = M.8438-AB6 o M.8438-B6
LCD = display LCD tipo S.5126
CONN.1/2 = connettore 24 poli
CONN. = 2 connettori 4 poli
P1 = pulsante
P2 = pulsante

Il motivo per il quale si è obbligati ad utilizzare dei micro **ST6** a **28 piedini** è dovuto al fatto che ci serve la porta **C4** per il segnale di **Strobe**.
A questo punto qualcuno ci potrà far osservare che il segnale di **Strobe** potevamo benissimo prelevarlo da una bit libero della porta **A** oppure **B** e, in tal modo, potevamo ancora utilizzare un micro **ST6** con soli **20** piedini.
Purtroppo nel **bus** siglato **LX.1202** (vedi rivista **N.179**) se oltre alla scheda di questo display volessimo inserire anche la scheda **LX.1205** che pilota dei **relè**, oppure la scheda **LX.1206** che pilota dei **triac**, constateremmo che tutti i **bit** delle porte **A-B** risultano occupati.
Poichè in questa scheda utilizziamo anche la **porta C**, il programma che vi forniremo per questo **display LCD** sarà completamente diverso rispetto al software che vi avevamo già fornito per la scheda display **LX.1204** e questo potrà risultarvi molto utile perchè, confrontando i due programmi, potrete notare le differenze.
Comunque questo programma per **display LCD** svolgerà le stesse funzioni che svolge tutt'ora il programma per il display a **7 segmenti**.

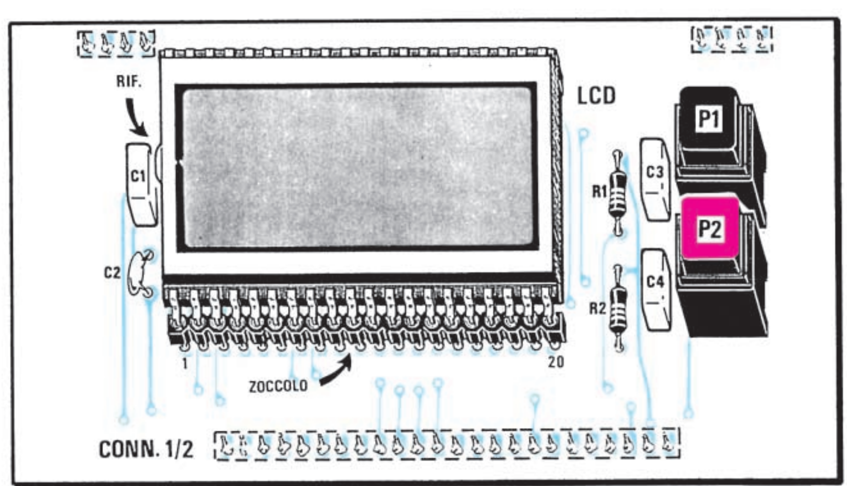
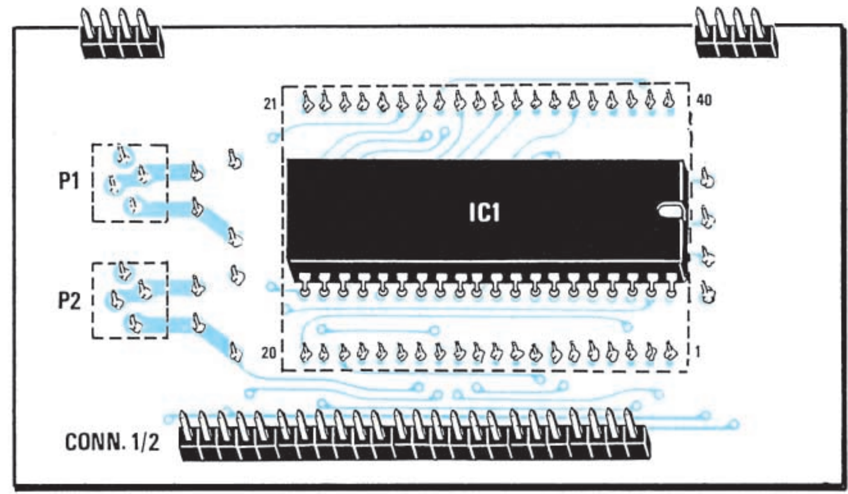


Fig.10 Scheda vista dal lato del display.



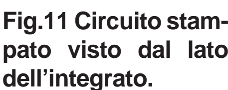


Fig.11 Circuito stampato visto dal lato dell'integrato.

Ritornando al nostro schema elettrico, non dovrete preoccuparvi delle connessioni dell'integrato **IC1** con il **display LCD**, perchè queste vengono automaticamente effettuate tramite le **piste** in rame presenti sul circuito stampato.
In questo schema di critico c'è il solo condensatore **C2** collegato tra il piedino **31** e la **massa**.
Questo condensatore, come potrete leggere nell'elenco componenti, deve risultare da **22 picoFarad** e tale valore serve per generare, tramite uno stadio oscillatore interno, una frequenza di circa **20.500 Hz** che, divisa internamente per **256**, farà uscire dal piedino **30** di **IC1** un'onda quadra di circa **80 Hz**.
Questa frequenza chiamata **Back/Plane**, è quella che ci servirà per alimentare i piedini **1-40** del display **LCD**.

REALIZZAZIONE PRATICA

Realizzare questa scheda per display **LCD** che abbiamo siglato **LX.1207** è molto semplice.
Nel circuito stampato a doppia faccia con fori metallizzati dovrete inserire tutti i componenti richiesti e per iniziare vi consigliamo di saldare lo zoccolo per l'integrato **IC1**.
Dopo aver saldato tutti i piedini sulle piste dello stampato, potrete inserire i due connettori maschi da **4 piedini** e da **24 piedini**, che vi serviranno per innestare questa scheda nel **bus** siglato **LX.1202** (vedi rivista **N.179**).
Terminata questa operazione, capovolgete lo stampato e da questo lato inserite i due connettori femmina da **20** piedini che vi serviranno come zoccolo per il display **LCD**.

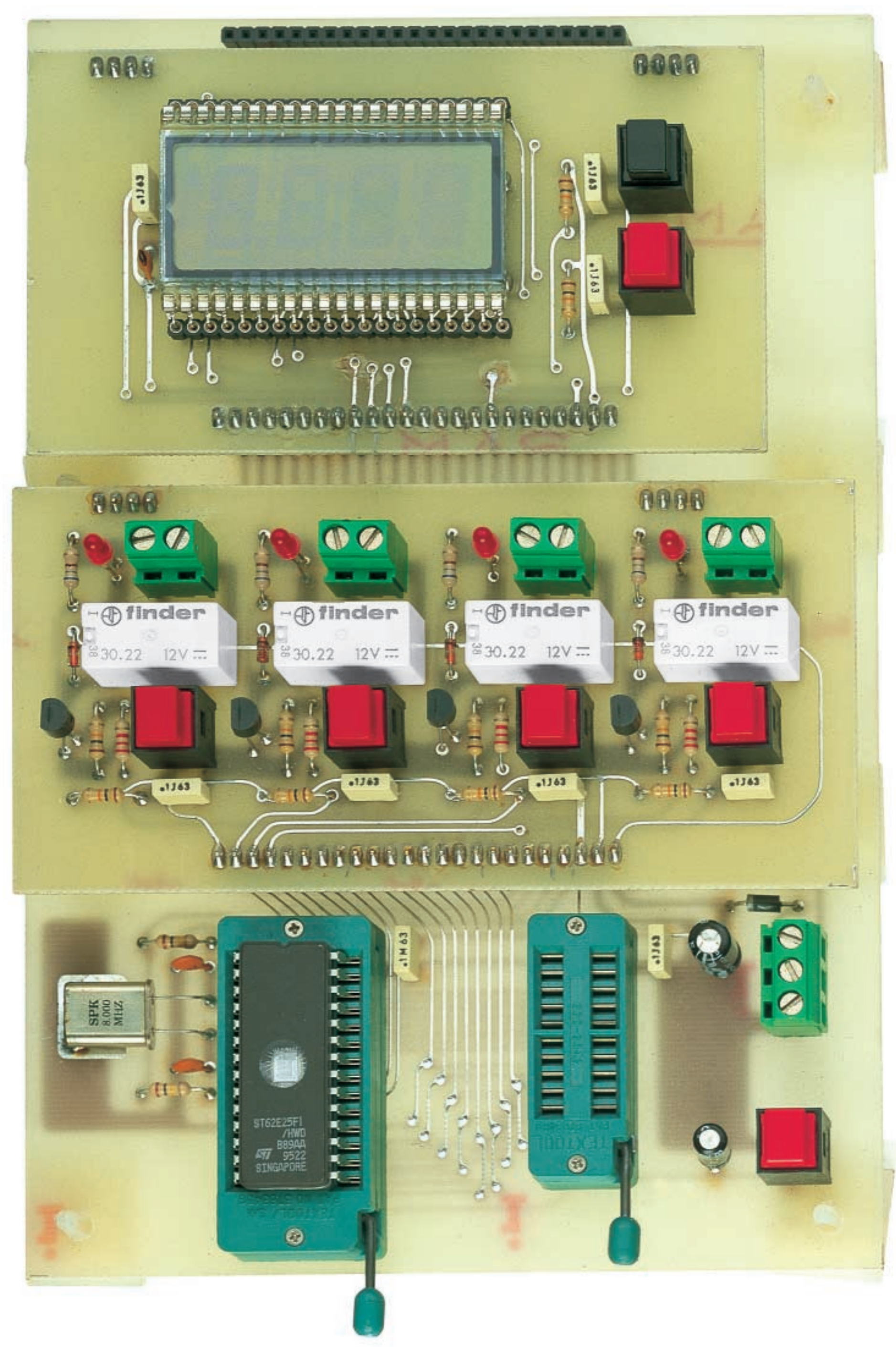


Fig.12 La scheda del display LCD siglata LX.1207 andrà inserita nel Bus LX.1202 congiunta anche alla scheda dei Relè o dei Triac. Come microprocessore ST6 dovrete necessariamente utilizzare quelli a 28 piedini, cioè l'ST62.E25 che è riprogrammabile o i tipi ST62.T15 e ST62.T25 che non sono riprogrammabili.

Per completare il montaggio dovrete inserire le resistenze **R1-R2**, i due pulsanti **P1-P2**, i tre condensatori poliestere ed il ceramico da **22 picoFarad** ed innestare nello zoccolo l'integrato **M.8438/AB6**, orientando la tacca a forma di **U** impressa sul suo corpo come visibile nella fig.10.
Dal lato opposto dovrete inserire nei due connettori femmina tutti i piedini presenti nel display **LCD** e qui forse incontrerete qualche difficoltà, perchè spesso i piedini del display risultano troppo divaricati.
Per poterli restringere in modo uniforme potrete premere sul piano di un tavolo tutti i terminali.
Prima di inserire il display nei connettori, dovrete ricercare sul suo corpo la **tacca di riferimento**, perchè se lo inserirete in senso inverso non potrà funzionare.
In questi display questa tacca di riferimento non è molto visibile, perchè quasi sempre è costituita da una piccola **goccia** di vetro posta su una sola estremità.
Da questo stesso lato troverete spesso sulla cornice nera che contorna l'interno del display il segno **>** (vedi fig.8).
Questa tacca di riferimento va sempre rivolta verso i condensatori **C1-C2**.
Spingete il display in corrispondenza dei lati in cui sono presenti i terminali e mai del centro perchè potrebbe spezzarsi.

PROGRAMMI

Per far funzionare questa scheda display **LCD** abbiamo preparato **5 programmi** che troverete inseriti in un dischetto floppy da **3** pollici siglato **DF1207.3**.
Una volta in possesso del dischetto, per caricarlo nell'Hard-Disk dovrete procedere come di seguito spiegato.

Uscite da qualsiasi programma che stavate utilizzando, **Windows - Pcshell - Norton** ecc., e quando sul monitor del computer appare **C:\>** inserite il dischetto nel drive **A**, quindi digitate:

C:\>A poi premete Enter

Quando appare **A:\>** scrivete:

A:\>installa e premete Enter

Il programma vi chiederà subito in quale **directory** volete installare il contenuto del disco.
Noi abbiamo già definito una **directory** che abbiamo chiamato **LX1207**, quindi se premete Enter il

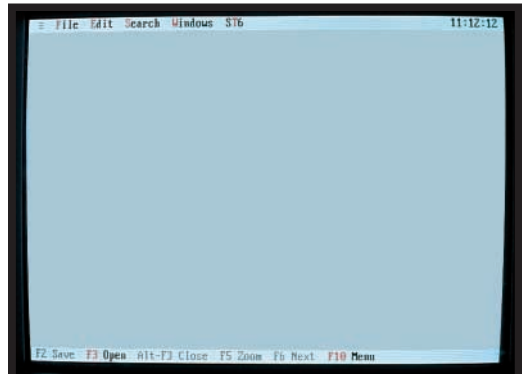


Fig.13 Quando sul monitor appare la finestra dell'Editor potete premere i tasti ALT+F poi F3 e vedrete apparire tutti i files con l'estensione .ASM.

Fig.14 Nel nuovo dischetto LX1207 non troverete nessuno dei vecchi programmi (vedi foto), ma i soli programmi da utilizzare per questa scheda con display LCD.

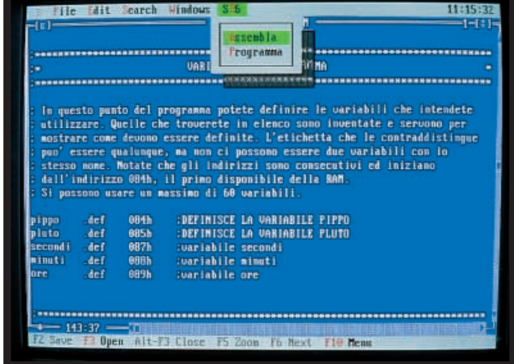


Fig.15 Potrete trasferire il programma scelto nella memoria di un ST6 solo dopo averlo assemblato. Per assemblarlo dovrete premere i tasti ALT+T poi Enter.

programma creerà automaticamente una directory con questo nome e, **scompattandolo**, copierà il contenuto del dischetto all'interno dell'Hard-Disk.
Se sul monitor dovesse apparire la scritta **error**, potrete ricaricare il dischetto con questo diverso sistema.

Quando appare **C:\>** dovrete creare la directory scrivendo:

C:\>MD LX1207 poi premete Enter

Quando riapparirà il prompt di **C:\>** inserite il dischetto floppy nel drive **A** poi scrivete:

C:\>COPY A:*.* C:\LX1207 poi premete Enter

Nota = Poiché nella digitazione è necessario rispettare la spaziatura, per agevolarvi abbiamo interposto una **barra** in colore che corrisponde allo **spazio** che occorre lasciare tra le lettere.

Quando il computer avrà terminato di copiare dal dischetto tutti i programmi, questi non saranno ancora stati **scompattati** quindi dovrete farlo voi scrivendo:

CD:\>CD LX1207 poi premete Enter

Quando appare **C:\LX1207>** digitate:

C:\LX1207>installa poi premete Enter

In questo modo vedrete via via comparire sul monitor i **nomi** dei files che si stanno **scompattando**.

CONVERTIRE i files .ASM in .HEX

Prima di trasferire un file nella memoria di un **ST6** occorre, come abbiamo precisato nei precedenti numeri, **assemblarlo** in modo da ottenere un secondo ed identico file, ma con l'estensione **.HEX**.
Se tenterete di trasferire un file **.ASM** nella memoria del micro vi verrà segnalato **error**.
Il micro da usare per i **display LCD** deve necessariamente avere **28 piedini**, quindi potrete adoperare un **ST62.E25** se volete riprogrammarlo e cancellarlo più volte, oppure un **ST62.T15** o un **ST62.T25** che come sapete **non sono** cancellabili.
Anche se nei precedenti articoli vi abbiamo spiegato come procedere per trasformare un file **.ASM** in un file **.HEX**, lo ripeteremo nuovamente.

Quando sul monitor appare **C:\LX1207>** dovete scrivere:

C:\LX1207>ST6 poi premere Enter

Con questo comando apparirà la finestra dell'**Editor** (vedi fig.13).
A questo punto dovrete premere prima i tasti **ALT+F** poi il tasto **F3** e sul monitor vedrete apparire tutti i files con estensione **.ASM** (vedi fig.14).

LCDCLOCK.ASM
LCDCRONO.ASM
LCDOROLO.ASM
LCDTEM90.ASM
LCDTIM90.ASM

Per posizionare il cursore su uno dei cinque files premete **ALT+F** poi premete i tasti freccia giù/su e quando sarete sul file che vi interessa premete **Enter**.
Entrerete così nell'editor del file selezionato e avrete in linea le istruzioni del programma.
Per **assemblare** il programma dovete premere i tasti **ALT+T** e poi Enter (vedi fig.15).
Ammesso che abbiate scelto il file **LCDCLOCK**, dopo pochi secondi apparirà sul monitor la scritta:

*** **success** ***

con il tempo di compilazione e di seguito la scritta:

C:\LX1207>

Premendo Enter rientrerete nel programma **LCDCLOCK**.
Per uscire dovrete premere **ALT+F3**.
Vedrete nuovamente la maschera dell'**Editor** e a questo punto premendo i tasti:

ALT+F poi **F3** poi Enter

apparirà nuovamente la maschera di tutti i files con estensione **.ASM**.
Nella riga in alto dovrete sostituire la scritta **.ASM** con la scritta **.HEX** poi premere Enter.
In questo modo apparirà l'elenco dei files convertiti in **.HEX**, e poiché è stato convertito il solo file **LCDCLOCK** comparirà:

LCDCLOCK.HEX

Vi ricordiamo che per modificare le righe di un programma dovrete sempre lavorare nell'estensione **.ASM**. Dopo aver fatto le modifiche le dovrete **salvare** premendo il tasto **F2**, poi dovrete **assemblare** il programma per convertirlo in un file **.HEX** come poc'anzi vi abbiamo spiegato.

I **5 programmi** che abbiamo inserito in questo dischetto hanno le stesse funzioni dei programmi che vi abbiamo presentato per i display a **7 segmenti**, ma sono stati riscritti e adattati per pilotare in **seriale** l'integrato **M.8438/B6**, quindi le righe che potrete modificare hanno un diverso numero.

LCDCRONO.HEX

Questo programma è un semplice **cronometro**, quindi per visualizzare i tempi occorre inserire nel **bus LX.1202** la **sola** scheda dei display siglata **LX.1207**.
Se nel bus inserirete le schede dei **relè** o dei **triac**, non potrete renderle attive perchè nel programma non è presente nessuna istruzione per gestirle.
Una volta caricato su un micro **ST62.E25 riprogrammabile** vergine il programma **LCDCRONO.HEX** ed inserito nello zoccolo presente sulla scheda bus **LX.1207**, appena alimenterete il circuito sui **4 display** apparirà il numero:

00:00

Premendo il pulsante **P1** il micro comincerà a contare in avanti ad intervalli di tempo di un **secondo**, quindi sui display vedrete apparire i numeri:

00:01 - 00:02 - 00:03 ecc.

Sui primi due display di sinistra vedrete i **minuti** e sui display di destra i **secondi**.
I **due punti** che separano i display dei minuti e dei secondi lampeggeranno con una cadenza di un secondo.
Come noterete, quando si è raggiunto un tempo di **00:59 secondi**, subito dopo si passerà al tempo successivo di **01:00**, cioè **1 minuto** e **00 secondi**.
Il massimo numero che potrete visualizzare sarà quindi di **99 minuti** e **59 secondi**, dopodichè apparirà **00:00**.
Se in fase di conteggio premerete **P1**, il conteggio si **bloccherà** sul tempo raggiunto e premendolo nuovamente ripartirà dal numero sul quale si era fermato.
Se invece premerete il pulsante **P2**, il conteggio ripartirà da **zero**, cioè il tempo visualizzato si **azzererà**.

LCDOROLO.HEX

Questo programma è un semplice **orologio.**
Per poter visualizzare le **ore** ed i **minuti** dovrete inserire nel **bus LX.1202** la **sola** scheda dei display siglata **LX.1207**.
Se nel bus inserirete le schede dei **relè** o dei **triac**, non potrete renderle attive, perchè nel programma non è presente nessuna istruzione per gestirle.
Una volta caricato su un micro **ST6** vergine il programma **LCDOROLO.HEX** ed inserito nello zoccolo presente nella scheda bus **LX.1202**, non appena alimenterete il circuito sui **4 display** apparirà il numero:

00:00

I primi due display di **sinistra** segneranno le **ore**, mentre i due di **destra** i **minuti**.
I due **punti** che separano i due display lampeggeranno con una cadenza di **1 secondo**.
Come noterete, raggiunte le **ore 23** ed i **59 minuti**, dopo **1 minuto** si passerà alle **24 ore** che verranno visualizzate con **00:00**.
Per mettere a **punto** le **ore** dell'orologio si utilizzerà il pulsante **P2** e, per mettere a punto i **minuti**, il pulsante **P1**.
Facciamo presente che potrete solo **far avanzare** i **numeri** e non **indietreggiare**.

LCDCLOCK.HEX

Questo programma è totalmente diverso dal precedente programma **LCDOROLO**, perchè oltre a visualizzare le **ore** e i **minuti** permette di **eccitare** un **relè** o un **triac** ad un'**ora** prestabilita e di **diseccitarlo** dopo un tempo che voi stessi potrete prefissare modificando alcune righe del programma.
Per farlo funzionare occorre inserire nel **bus LX.1202** la scheda dei display siglata **LX.1207** e quella dei relè siglata **LX.1205**, oppure quella dei triac siglata **LX.1206.**
Prima di spiegarvi quali righe dovrete modificare, consigliamo ai meno esperti di **leggere attentamente** tutto l'articolo, dopodichè potranno modificare i **parametri** nelle sole righe che noi indicheremo.
Come abbiamo accennato, il programma **LCDCLOCK.HEX** ci dà la possibilità di **eccitare** o **diseccitare** uno o più **relè** anche contemporaneamente, su orari che noi stessi potremo stabilire, purchè non si superino più di **8 cicli** o **periodi** nell'arco delle **24 ore**.
Questo programma potrà servire per **accendere** o **spegnere** una o più caldaie, delle insegne luminose ad orari prestabiliti, ecc.

Per mettere a **punto** le **ore** dell'orologio si utilizzerà il pulsante **P2** e per mettere a punto i **minuti** il pulsante **P1**.
Facciamo presente che è possibile soltanto far **avanzare** i **numeri** e non **indietreggiare**.
Appena accenderete l'orologio **tutti** i **4 relè** o **triac** partiranno **eccitati**.
Se volete che all'accensione dell'orologio tutti i relè risultino **diseccitati**, dovrete andare alla riga **N.57** dove troverete questa istruzione:

```
ldi    port_b,11110011b
```

e modificarla inserendo in sostituzione degli **1** degli **0** come qui sotto riportato:

```
ldi    port_b,00000011b
```

Nota = Anche se questa riga è composta da **8 numeri**, dovrete modificare solo i primi **4** di sinistra.

Se volete far **eccitare** all'accensione il solo relè **RL4**, dovrete mettere un **1** in corrispondenza della prima cifra di sinistra come qui sotto riportato:

```
ldi    port_b,10000011
```

A questo punto vi spieghiamo che cosa s'intende per **8 cicli** o **periodi** da utilizzare nell'arco delle **24 ore** che troverete riportati in queste righe:

1° periodo = righe 299 - 300 - 301
2° periodo = righe 306 - 307 - 308
3° periodo = righe 313 - 314 - 315
4° periodo = righe 320 - 321 - 322
5° periodo = righe 327 - 328 - 329
6° periodo = righe 334 - 335 - 336
7° periodo = righe 341 - 342 - 343
8° periodo = righe 348 - 349 - 350

Ogni ciclo è composto da **3 righe** d'istruzioni, quindi nel **1° ciclo** o **periodo** del nostro programma troverete:

```
.byte   02          ;299 riga delle ore
.byte   30          ;300 riga dei minuti
.byte   11100000b   ;301 riga per comando relè
```

Attualmente il **1° ciclo** inizia alle ore **2,30 di notte**.
Per modificare l'orario basterà mettere nella **prima** riga l'**ora** che vi interessa, ad esempio **05-06-10**, e nella **seconda** riga i relativi **minuti**, ad esempio **00 - 10 - 30 - 50**.
Nella **terza** riga sono riportati i relè che desiderate **eccitare** e quelli che **non** desiderate eccitare all'orario da noi prestabilito.

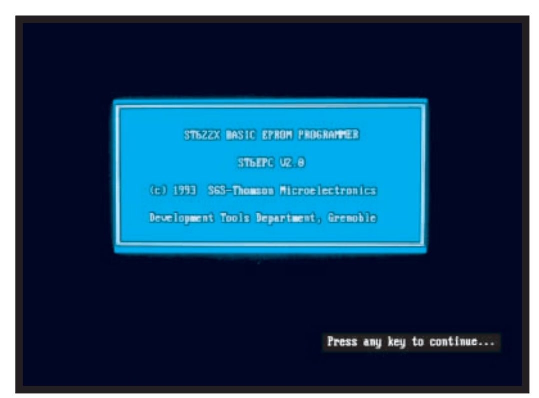


Fig.16 Dopo aver assemblato il programma prescelto (vedi fig.15) pigiate ALT+T poi la lettera P. Quando apparirà questa finestra dovrete premere un tasto qualsiasi.

Fig.17 Come vedrete, sullo schermo apparirà una finestra con tutti i tipi di ST62. Con i tasti freccia ricercate la sigla ST62E25 poi pigiate il tasto Enter.

Fig.18 Prima di programmare l'ST62E25 controllate attentamente che nel riquadro in basso appaia ST62E25. Se appare un'altra sigla dovrete ritornare alla fig.17.

Mettendo un **1** il relè si **ecciterà**, mettendo uno **0** si **disecciterà**.
Ciò che dovrete modificare in questa terza riga sono **solo** i primi **quattro numeri** di sinistra posti dopo la parola **byte.**
Tenete presente che il **primo** numero di sinistra **piloterà** il relè **RL4** e l'**ultimo** numero di destra il relè **RL1**, quindi avrete in ordine:

RL4-RL3-RL2-RL1

Per farvi capire come modificare tutti questi numeri vi faremo un semplice esempio.
Ammesso che alle **06,10** desideriate **eccitare** i relè **RL2-RL1**, scriverete nelle righe **299 - 300 - 301** (1° ciclo) questi numeri:

```
    .byte   06          ;299 riga delle ore
    .byte   10          ;300 riga dei minuti
    .byte   00110000b   ;301 riga per comando relè
```

Se alle **09,30** vorrete **eccitare** il solo relè **RL4**, dovrete scrivere nelle righe **306 - 307 - 308** (2° ciclo) questi numeri:

```
    .byte   09          ;306 riga delle ore
    .byte   30          ;307 riga dei minuti
    .byte   10000000b   ;308 riga per comando relè
```

Se alle **12,00** vorrete **disecciitare** anche il relè **RL4**, dovrete scrivere nelle righe **313 -314 -315** (3° ciclo) questi numeri:

```
    .byte   12          ;313 riga delle ore
    .byte   00          ;314 riga dei minuti
    .byte   00000000b   ;315 riga per comando relè
```

Se alle **18,45** vorrete **eccitare** tutti i relè, dovrete scrivere nelle righe **320 - 321 - 322** (4° ciclo) questi numeri:

```
    .byte   18          ;320 riga delle ore
    .byte   45          ;321 riga dei minuti
    .byte   11110000b   ;322 riga per comando relè
```

Se alle **22,30** vorrete **disecciitare** i relè **RL4-RL3**, dovrete scrivere nelle righe **327 - 328- 329** (5° ciclo) questi numeri:

```
    .byte   22          ;327 riga delle ore
    .byte   30          ;328 riga dei minuti
    .byte   00110000b   ;329 riga per comando relè
```

Se alle **23,40** vorrete lasciare **eccitato** il solo relè **RL1**, dovrete scrivere nelle righe **334 - 335 - 336** (6° ciclo) questi numeri:

```
    .byte   23          ;334 riga delle ore
    .byte   40          ;335 riga dei minuti
    .byte   00010000b   ;336 riga per comando relè
```

Se alle **24,00** vorrete **disecccitare** anche questo relè, dovrete scrivere nelle righe **341 - 342 - 343** (7° ciclo) questi numeri:

```
    .byte   00          ;341 riga delle ore
    .byte   00          ;342 riga dei minuti
    .byte   00000000b   ;343 riga per comando relè
```

Avendo utilizzato solo **7 cicli** degli **8** disponibili, se l'ultimo non vi interessa lo **potrete cancellare** oppure inibire, mettendo davanti alle righe **348 - 349 - 350** un **punto** e **virgola** o mettendo sulla terza riga **00000000b.**
Potrete **aggiungere** altri cicli se **8** risultassero insufficienti.

Facciamo presente che questi **cicli** si **ripeteranno** automaticamente all'**infinito** agli stessi **orari** tutti i giorni.

LCDTIM90.HEX

Questo programma è un **timer** che, contando in **avanti**, ecciterà un relè o un triac quando raggiungerà i **minuti** e i **secondi** da noi prefissati.
Per farlo funzionare occorre inserire nel **bus LX.1202** la scheda dei display LCD siglata **LX.1207** e quella dei relè siglata **LX.1205**, oppure quella dei triac siglata **LX.1206**.
Non appena alimenterete il circuito, il conteggio partirà da **00:00** e inizierà a contare in **avanti**; a questo punto potrete utilizzare i pulsanti **P1** e **P2** presenti sulla scheda display LCD **LX.1207**.

Premendo **P1** il conteggio si **ferma**.
Premendo nuovamente **P1** il conteggio riparte dal **numero** sul quale si era fermato.
Premendo **P2** il contatore si **resetta**.
Premendo **P1** il contatore riparte da **00:00**.

Nota = Il pulsante **P2** di **reset** sarà attivo solamente se avrete **fermato** il conteggio con **P1.** Se premerete **P2** mentre è **attivo** il conteggio, questo non si azzererà.
I pulsanti presenti sulle schede Triac e Relè non risultano **attivati**.
Il conteggio del display arriva ad un massimo di **89 minuti** e **59 secondi**.

Il programma **LCDTIM90.ASM**, come potrete no-

tare, dispone di **4 cicli** perchè **quattro** sono i **relè** e i **triac** presenti sulle schede sperimentali.

1° ciclo = Dopo **20 secondi** dall'accensione si ecciterà il solo relè **RL1**.
Ovviamente sui display vedrete apparire **19** e, quando questo numero raggiungerà **00:00**, il relè si **ecciterà**.

2° ciclo = Passando al **secondo ciclo**, questo relè rimarrà **eccitato** per un tempo da noi prefissato in **1 minuto** e **30 secondi** e raggiunto questo **tempo** il relè **RL1** si **disecciterà** e automaticamente si **ecciterà** il relè **RL2**.
Il relè **RL2** si ecciterà un secondo dopo che sui display sarà apparso il numero **01:29** che cambierà in **00:00**.

3° ciclo = Dopo **47 secondi**, cioè quando sul display il numero **46** passerà sullo **00**, il relè **RL2** si **disecciterà** e si **ecciterà** il terzo relè **RL3**.

4° ciclo = Il conteggio continuerà ed allo scoccare dei **3 minuti** e **00 secondi** (tempo da noi prefissato) si **disecciterà** il relè **RL3** e si **ecciterà** il relè **RL4**, cioè si ritornerà al **1° ciclo** per ripetere all'infinito i **quattro cicli**.

Per **variare** i **tempi** che noi abbiamo prefissato dovrete variare queste righe:

1° ciclo = righe 292 - 293
2° ciclo = righe 298 - 299
3° ciclo = righe 304 - 305
4° ciclo = righe 310 - 311

Se volete che il **1° ciclo** abbia una durata di **1 minuto** e **30 secondi**, dovrete inserire nelle sue righe questi numeri:

```
ldi    stsex,30    ;292 secondi per RL1
ldi    stmix,1     ;293 minuti per RL1
```

Se volete che il **2° ciclo** abbia una durata di **50 secondi**, dovrete inserire nelle sue righe questi numeri:

```
ldi    stsex,50    ;298 secondi per RL1
ldi    stmix,00    ;299 minuti per RL1
```

Se volete che il **3° ciclo** abbia una durata di **15 minuti** e **20 secondi**, dovrete inserire nelle sue righe questi numeri:

```
ldi    stsex,20    ;304 secondi per RL1
ldi    stmix,15    ;305 minuti per RL1
```

Se volete che il **4° ciclo** abbia una durata di **2 minuti** e **10 secondi**, dovrete inserire nelle sue righe questi numeri:

```
ldi    stsex,10    ;310 secondi per RL1
ldi    stmix,2     ;311 minuti per RL1
```

Nelle righe **295/296 - 301/302 - 307/308 - 313/314** sono riportate le sigle dei **relè** che volete **eccitare** e di quelli che volete rimangano **diseccitati**.
Guardando l'esempio riportato nel programma **LCDCLOCK.ASM** saprete già che scrivendo questa istruzione:

```
ldi    port_b,11110011b
```

potrete eccitare ad ogni **ciclo** anche **più relè** a vostra scelta.
Nei primi **quattro** numeri di sinistra (vedi **1111**) dovrete mettere un **1** sul relè che volete far **eccitare** ed uno **0** se **non** lo volete eccitare.

LCDTEM90.HEX

Questo programma è un **timer** che fa esattamente l'**inverso** del programma **LCDTIM90**, cioè **conta all'indietro** e quando raggiunge lo **00:00** eccita i relè.
I relè, come per il programma precedente, li ecciterete in **4 cicli** e come tempo **massimo** di partenza potrete impostare **90 minuti** e **00 secondi**.
Non appena alimenterete il circuito, il conteggio partirà da **00:20** (questo tempo lo abbiamo prescelto noi, ma poi vi spiegheremo come modificarlo) e procederà all'**indietro**.
Dopo che avrà avuto inizio il conteggio, potrete utilizzare i pulsanti **P1** e **P2** presenti sulla scheda display **LCD LX.1207**.

Premendo **P1** il conteggio si **ferma**.
Premendo nuovamente **P1** il conteggio riparte dal **numero** sul quale si era fermato.
Premendo **P2** il contatore si **resetta**.
Premendo **P1** il contatore riparte dal tempo che avete impostato come **partenza** per il conteggio all'**indietro**.

Nota = Il pulsante **P2** di **reset** sarà attivo solamente se avrete **fermato** il conteggio con **P1**. Se premerete **P2** mentre è **attivo** il conteggio, questo non si azzererà. Premendo **P2** per **resettarlo**, è intuitivo che contando all'**indietro** sul display ritorni il tempo di **partenza**, cioè **00:20**.

Nei **4 cicli** impostati otterrete queste condizioni:

1° ciclo = All'accensione si ecciterà il solo relè **RL1** e sui display apparirà **00:20** e a questo punto avrà inizio il conteggio alla **rovescia** che si fermerà sullo **00:00**.

2° ciclo = Dopo un secondo si ecciterà il relè **RL2** e a questo punto inizierà il **secondo ciclo**, che farà apparire sui display **01:30** (tempo **1 minuto** e **30 secondi**) che, secondo per secondo, decrementerà fino ad arrivare a **00:00**.

3° ciclo = A questo punto si **ecciterà** il relè **RL3** e sui display apparirà **00:47** che decrementerà fino ad arrivare allo **00:00**.

4° ciclo = L'ultimo ciclo farà eccitare il relè **RL4** e farà apparire sui display il numero **03:00** (**3 minuti**). Quando con il conteggio alla **rovescia** si arriverà al numero **00:00**, questo relè si **disecciterà** e contemporaneamente si **disecciterà** il relè **RL1**, cioè si ritornerà al **1° ciclo** per ripetere all'infinito i **quattro cicli**.

Per **variare** i **tempi** prefissati dovrete modificare queste righe:

1° ciclo = righe 295 - 296
2° ciclo = righe 301 - 302
3° ciclo = righe 307 - 308
4° ciclo = righe 313 - 314

Se volete che il **1° ciclo** abbia una durata di **1 minuto** e **30 secondi**, dovrete inserire nelle sue righe questi numeri:

```
      ldi    stsex,30     ;295 secondi per RL1
      ldi    stmix,1      ;296 minuti per RL1
```

Se volete che il **2° ciclo** abbia una durata di **50 secondi**, dovrete inserire nelle sue righe questi numeri:

```
      ldi    stsex,50     ;301 secondi per RL1
      ldi    stmix,00     ;302 minuti per RL1
```

Se volete che il **3° ciclo** abbia una durata di **15 minuti** e **20 secondi**, dovrete inserire nelle sue righe questi numeri:

```
      ldi    stsex,20     ;307 secondi per RL1
      ldi    stmix,15     ;308 minuti per RL1
```

Se volete che il **4° ciclo** abbia una durata di **2 minuti** e **10 secondi**, dovrete inserire nelle sue righe questi numeri:

```
      ldi    stsex,10     ;313 secondi per RL1
      ldi    stmix,2      ;314 minuti per RL1
```

Nelle righe **297/298 - 303/304 - 309/310 - 315/316** sono riportate le sigle dei **relè** che si **ecciteranno** e di quelli che si **disecciteranno**.
Anche in questo programma possiamo sostituire le righe sopra menzionate con questa sola riga d'istruzione:

```
      ldi  port_b,11110011b
```

Nei primi **quattro** numeri di sinistra (vedi **1111**) dove metterete **1** il relè si **ecciterà**, dove metterete **0** si **disecciterà**.

NOTA

Per imparare a programmare i microprocessori ST6 vi consigliamo di rileggere tutti i precedenti articoli riportati sulle riviste N.172/173 - 174 - 175/176 - 179 - 180, perché da oggi in avanti non ripeteremo più quello che vi abbiamo già spiegato.
Sul prossimo numero vi presenteremo un progetto completo dei relativi programmi per gestire un display LCD ALFANUMERICO a più righe, quindi proseguiremo spiegandovi come si dovrà procedere per ottenere dei programmi sempre più perfetti e funzionali.

KIT ESAURITO
perché l'integrato M.8438/AB6
è fuori produzione

COSTO DI REALIZZAZIONE

Tutti i componenti per realizzare questa scheda con display LCD, cioè circuito stampato, connettori maschi, pulsanti, integrato **M.8438/AB6**, display a cristalli liquidi tipo **S.5126** o **LC.513040** (escluso il solo software inserito nel dischetto **DF.1207/3**)..L.58.000

Costo del solo stampato **LX.1207**..............L.9.000

Nota = Per far funzionare questa scheda vi servono i **5 programmi** inseriti nel dischetto siglato **DF.1207/3** del costo di L.12.000

Se guardiamo il lato posteriore di un **normale** LCD vedremo il **vetro** del suo supporto, se guardiamo quello di un LCD **alfanumerico** vedremo un circuito stampato con sopra fissati due integrati in **SMD** provvisti di **62-80** piedini (vedi fig.2).
Se potessimo osservare anteriormente l'interno di un **normale** LCD vedremmo **quattro** caselle con i soliti **7 segmenti** che, accendendosi, ci permettono di far apparire un numero qualsiasi da **0** a **9**.
Poiché questo Display può visualizzare solo dei **numeri** viene definito **numerico**.
Se si potesse osservare, sempre anteriormente, l'interno di un Display **alfanumerico,** si vedrebbero tante **caselle** rettangolari che, anziché essere composte da 7 segmenti, presentano ben **40 punti** distribuiti **8** in senso **verticale** e **5** in senso **orizzontale** (vedi fig.3).

UNA SCHEDA per pilotare

Questi Display, conosciuti anche con il nome di **DMLCD** (vale a dire **D**ot **M**atrix **L**iquid **C**ristal **D**isplay che in italiano significa **D**isplay a **C**ristalli **L**iquidi con **M**atrice di **P**unti), sono chiamati **alfanumerici**.
Infatti, accendendo questi **40 punti** nelle varie combinazioni, potremo far apparire un qualsiasi **carattere alfabetico** maiuscolo o minuscolo, tutti i **numeri** da **0** a **9,** un qualsiasi **simbolo grafico,** come ad esempio **frecce,** $\sqrt{\ }$, Ω, Π e, volendo, anche caratteri **cinesi - arabi - greci - cirillici,** ecc.
Questi display **alfanumerici** li possiamo reperire con **1-2-3 righe,** ecc., anche se nel nostro caso ne abbiamo scelto uno con **16 caratteri 2 righe** per spiegarvi come si possa scrivere nella riga superiore ed in quella inferiore.
La definizione **16 caratteri** sta ad indicare che vi sono **16 caselle** per riga, quindi avendo scelto un Display con **2 righe** avremo un totale di **32 caselle** e poiché in ciascuna vi sono **40 punti** potremo accendere ben:

40 x 32 = 1.280 punti

Ammesso di voler visualizzare su un **normale** Display a **7 segmenti** il numero **3,** potremo risolvere il problema con estrema facilità alimentando i soli **segmenti a-b-g-c-d,** ma in un Display a **matrice** composto da **40 punti** le cose diverrebbero ben più complesse perché dovremmo alimentare, nella prima riga superiore i **5 punti** in orizzontale**,** nella seconda-terza-quarta-quinta riga **1 punto** nella posizione richiesta, nella sesta riga **2 punti,** uno all'inizio della riga ed uno alla fine e nella settima riga **3 punti** centrali.
Immaginatevi quindi quanto sarebbe complicato scrivere nelle **32 caselle** delle **frasi** o dei **numeri.**
In teoria lo si potrebbe fare con un microprocessore da **1.280 bit,** ma poiché non esiste, vi chiederete come si possano scrivere in tutte queste caselle **lettere - numeri - simboli**.

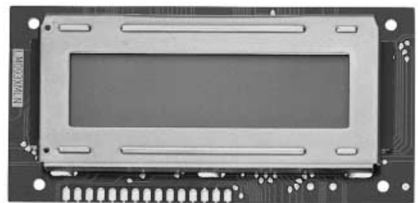


Fig.1 Le dimensioni di un display alfanumerico (vedi a sinistra) sono nettamente superiori a quelle del display numerico riprodotto a destra.

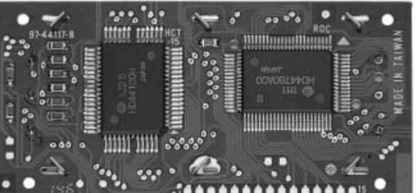
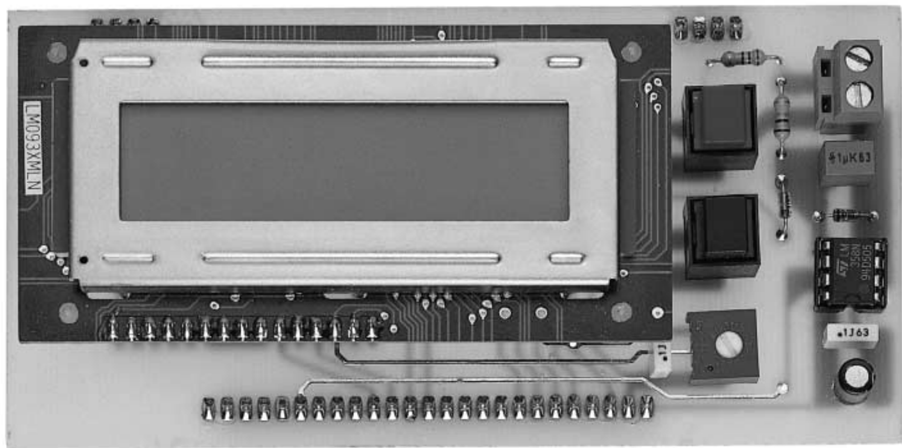


Fig.2 Dal lato opposto del solo display alfanumerico è presente un micro HD.44780 piu' un integrato siglato HD.44100.



un DISPLAY alfanumerico

Oltre ai normali display LCD a 7 segmenti presentati nella rivista N.181, in grado di visualizzare "4 numeri", esistono anche dei display LCD "alfanumerici" in grado di riprodurre un qualsiasi carattere grafico. In questo articolo vi spiegheremo come dovrete pilotarli per poter far apparire parole - numeri - simboli.

Per questi Display **alfanumerici** si sfrutta la stessa tecnica utilizzata per far apparire sul **monitor** del vostro computer tutte le **lettere** e i **numeri** presenti sulla **tastiera**.
Quando sulla tastiera digitiamo la lettera **A** generiamo un **codice** che, entrando in un integrato **generatore** di **caratteri,** viene trasformato in un **codice ASCII** che provvede a far accendere sul **monitor** tutti i punti richiesti per creare il simbolo **A**.
Lo stesso avviene in questi Display, i quali vengono gestiti da un codice di **8 bit** che, applicato sui piedini **DB0 - DB1 - DB2 - DB3 - DB4 - DB5 - DB6 - DB7** (piedini dal numero **7** al numero **14),** entrerà negli ingressi del microprocessore siglato **HD.44780** (presente sul retro del display) al cui interno è presente una **CGROM**.
La parola **CGROM** significa **C**haracters **G**enerator **R**ead **O**nly **M**emory, cioè lista di caratteri già **memorizzati** al suo interno.
All'interno di questa **CGROM** sono memorizzate tutte le lettere e i simboli visibili nella **Tabella N . 1**, quindi, se sui suoi piedini d'ingresso faremo giungere un **codice** composto da livelli logici **0-1,** selezioneremo nella sua **memoria** la lettera o il simbolo abbinati a questo **codice;** per poter accendere tutti i **punti** necessari per far apparire sul Display la lettera o il simbolo da noi prescelti, il microprocessore **HD.44780** attenderà una conferma dal secondo integrato siglato **HD.44100.**
Detto questo, molti potrebbero pensare che sia sufficiente applicare sui piedini **DB0 - DB1 - DB2 - DB3 - DB4 - DB5 - DB6 - DB7** dei livelli logici **1-0** per far apparire una lettera o un numero.
Chi tentasse di farlo **non vedrebbe** accendersi **nessun punto,** perché i due integrati **HD.44780** e **HD.44100** devono essere gestiti con un complesso **set di istruzioni** che potremo ottenere solo utilizzando un **microprocessore esterno** appositamente programmato.

- Di questo set di istruzioni una parte viene utilizzata per **inizializzare** il microprocessore **esterno,** cioè l'**ST6**.
Le rimanenti istruzioni sono necessarie al Display per prepararsi a ricevere tutti i nostri **dati**, cioè per **configurarsi** correttamente per ricevere i dati in **8 bit** oppure in **4+4 bit.**

Se non utilizzeremo questo **set** di **istruzioni** non riusciremo mai a visualizzare sul Display alcun **carattere**.

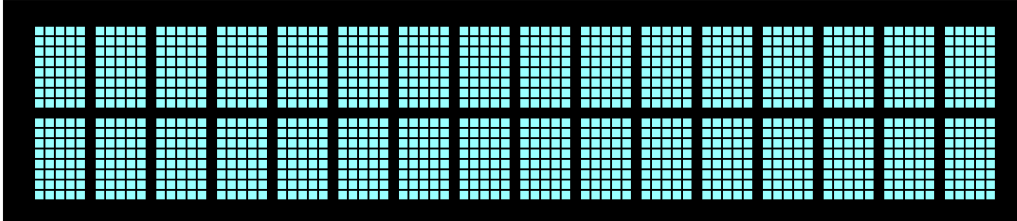


Fig.3 In un display 16 x 2 sono presenti 2 colonne di 16 caselle. In ogni casella vi sono 40 "punti" per accendere i quali occorrerebbe un microprocessore da 1.280 bit.

Per comunicare con il Display con **8 bit** si utilizzano tutti i piedini siglati da **DB0** a **DB7,** mentre per comunicare con **4+4 bit** si utilizzano i soli piedini siglati da **DB4** a **DB7** (gli altri piedini da **DB0** a **DB3** non vengono utilizzati).
Usando **4+4 bit,** verranno inviati al Display i **primi 4 bit,** poi i **successivi 4 bit**.
Nota = Nei nostri programmi abbiamo utilizzato il sistema dei **4+4 bit.**

- Come abbiamo detto**,** quando invieremo un **codice** all'**HD.44780** per far apparire un **carattere,** per poterlo visualizzare questo attenderà tutta una serie di **istruzioni**, ad esempio in quale delle **32 caselle** presenti nel Display vogliamo far apparire il segno **grafico**, se desideriamo utilizzare **entrambe** le **righe** del Display oppure **1 sola,** ecc.
Queste istruzioni verranno accettate solo quando sul **piedino 4** del **display,** denominato **R/S,** sarà presente un **livello logico 0**.

- Dopo aver inserito tutte le **istruzioni** richieste, dovremo mettere a **livello logico 1** il piedino **4** del **display** e solo a questo punto potremo inviare i **dati,** cioè la **lettera - numero - simbolo** che desideriamo far apparire.

- Ai due integrati **HD.44780 - HD.44100** occorre un certo **tempo** per eseguire tutte queste operazioni e questo **tempo** di **lavoro** lo dovremo considerare e **rispettare** anche se si tratta di pochi **millisecondi**, diversamente nella casella interessata potrebbero apparire dei **caratteri** strani e **non significativi.**

Nei programmi dei vari **esempi** che troverete nel dischetto **DF1208** troverete tutte queste istruzioni di **ritardo**, che dovrete necessariamente rispettare quando vi accingerete a scrivere dei vostri personali programmi.
Se non le rispetterete, non riuscirete mai a far funzionare un qualsiasi Display **alfanumerico**.

Questi Display vengono chiamati **intelligenti**, solo perché dispongono di una **memoria** con un **archivio** di caratteri, ma per poter funzionare necessitano sempre di un **microprocessore esterno** (nel nostro caso un **ST62/E25** con **28 piedini**) che indichi loro quali caratteri desideriamo far apparire nelle **32 caselle**.

TABELLA dei CARATTERI PREDEFINITI

Nella **Tabella N.1** abbiamo riprodotto tutti i **caratteri** presenti all'interno della **CGROM**.
Come potrete notare, sul lato **destro** sono presenti **4 bit** indicati con **x x x x** seguiti da altri **4 bit** predefiniti con **0** e **1**, ad esempio:

x x x x 0 0 0 1

In alto sono riportati altri **4 bit** predefiniti con **0** e **1,** ad esempio:

0 0 1 1

Questa Tabella si usa come una Tavola Pitagorica, quindi se volessimo far apparire sul display la lettera **A,** dovremmo sostituire le **x** presenti sul lato **destro** con i bit riportati nella casella **in alto**.
In questo esempio dovremo scrivere:

0 1 0 0 - 0 0 0 1

Nota = Abbiamo messo un segno **-** tra i **primi** quattro bit e i secondi quattro, solo per rendere l'esempio più chiaro, ma questo segno **non dovrete** mai inserirlo.

Se volessimo far apparire una **a** (minuscola) dovremmo scrivere:

0 1 1 0 - 0 0 0 1

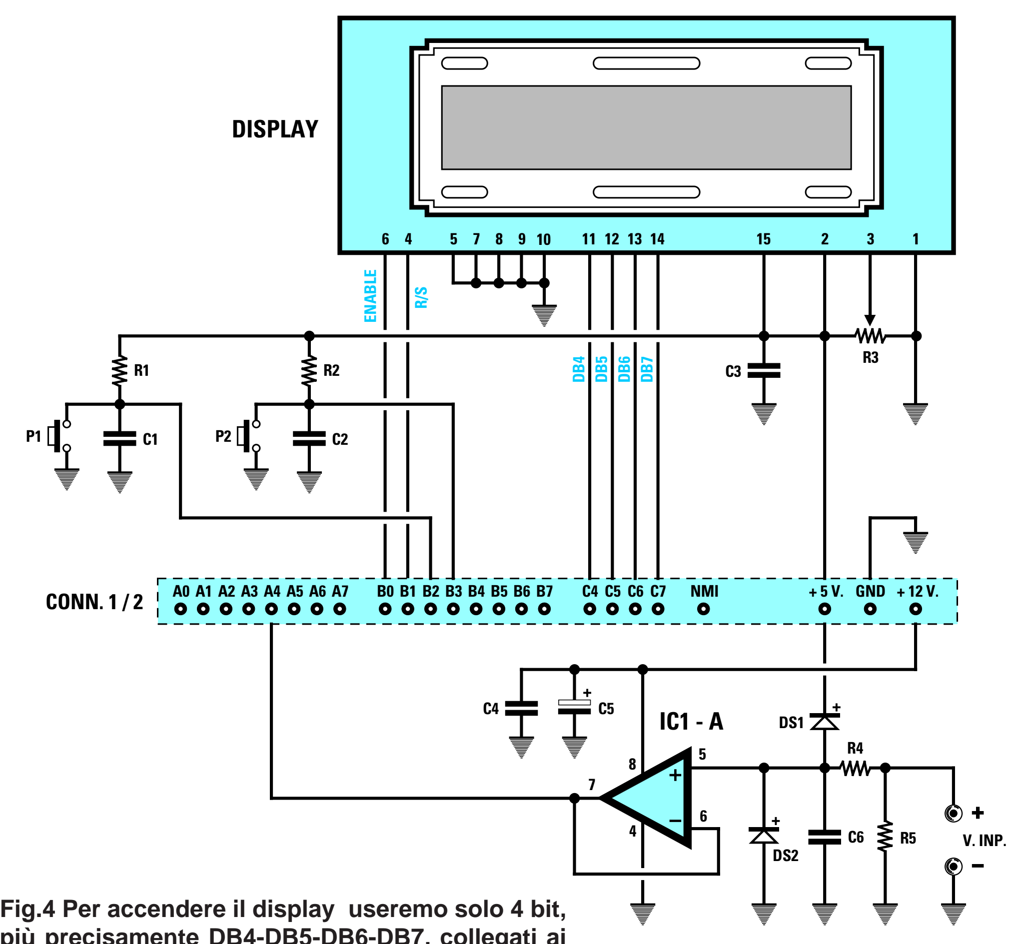


Fig.4 Per accendere il display useremo solo 4 bit, più precisamente DB4-DB5-DB6-DB7, collegati ai piedini C4-C5-C6-C7 del Connettore d'ingresso.

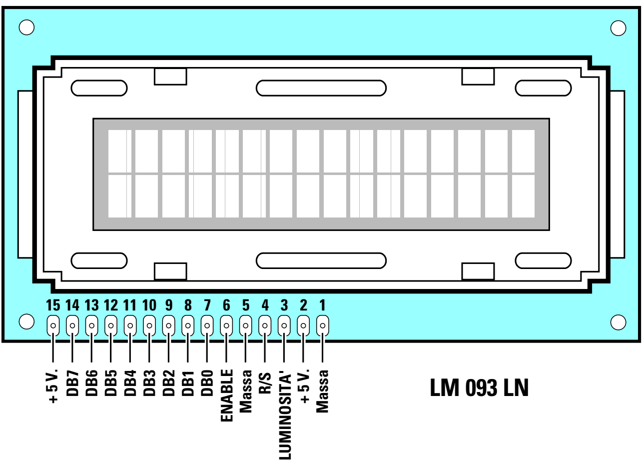


Fig.5 Gli altri bit che non vengono utilizzati, cioè DB0-DB1-DB2-DB3 che fanno capo ai piedini 7-8-9-10, andranno collegati a massa.

ELENCO COMPONENTI LX.1208

R1 = 10.000 ohm 1/4 watt
R2 = 10.000 ohm 1/4 watt
R3 = 10.000 ohm trimmer
R4 = 10.000 ohm 1/4 watt
R5 = 1 megaohm 1/4 watt
C1 = 100.000 pF poliestere
C2 = 100.000 pF poliestere
C3 = 100.000 pF poliestere
C4 = 100.000 pF poliestere
C5 = 10 mF elettr. 63 volt
C6 = 1 mF poliestere
DS1 = diodo tipo 1N.4150
DS2 = diodo tipo 1N.4150
IC1 = LM.358
DISPLAY = LCD tipo LM.093X
CONN.1/2 = connettore 24 poli
P1 = pulsante
P2 = pulsante

Se volessimo far apparire il segno grafico > dovremmo scrivere:

0 0 1 1 - 1 1 1 0

Anziché utilizzare questo **codice binario** per scrivere una **lettera** o un **carattere,** potremo usare anche un **codice decimale** e per questo motivo abbiamo inserito sotto ad ogni casella il rispettivo numero **decimale**.
Quindi scrivendo **65** sul Display apparirà la lettera **A maiuscola** e scrivendo **97** apparirà la lettera **a minuscola**.

Esempio in codice Binario

Per scrivere la lettera **A** in codice **binario** dovremo scrivere questa istruzione:

ldi car,01000001b

Esempio in codice Decimale

Per scrivere la lettera **A** in codice **decimale** dovremo scrivere questa istruzione:

ldi car,65

Esempio in codice ASCII

Anziché utilizzare un codice **binario** o **decimale,** potremo scrivere direttamente in **ASCII** ed in questo caso l'istruzione sarà la seguente:

car .ascii "A"

Tra i programmi dimostrativi riportati nel dischetto **DF1208** ne abbiamo inseriti diversi utilizzando questi tre diversi **codici**, quindi leggeteli attentamente perché con le istruzioni riportate comprenderete con estrema facilità quello che risulterebbe assai più complesso spiegare a parole.

OPERAZIONI MATEMATICHE

Molti si trovano in difficoltà con le operazioni **matematiche,** perché non pensano che il numero che desiderano far apparire è un **carattere grafico** che verrà prelevato all'interno della **CGROM.**
Nel caso della somma **3+2** che ci dà come risultato **5,** consultando la **Tabella N. 1** vedremo che sotto al numero **5** è indicato il numero **53**.

Per far apparire sul display il segno grafico **"5",** a questo numero dovremo sommare la **costante 48** e così facendo otterremo **5+48 = 53** e se andiamo a vedere nella **Tabella N. 1** noteremo che il numero **decimale 53** corrisponde effettivamente al carattere grafico **"5"**.
Pertanto, l'**istruzione** che dovremo scrivere per svolgere questa operazione sarà:

```
ldi a,3
addi a,2
addi a,48
ld ddata,a
call dsend
```

Se svolgiamo questa seconda operazione **9+7 = 16** otterremo un risultato di **due** cifre, quindi per far apparire questi due segni grafici dovremo sommare a **1** la costante **48** e, in tal modo, otterremo **49**; consultando la **Tabella N. 1** noteremo che il numero **49** corrisponde al segno grafico **"1"**.
Sommando al numero **6** la costante **48** otterremo **6 + 48 = 54** e sempre guardando la **Tabella N. 1** scopriremo che **54** corrisponde al segno **grafico "6"**.
Pertanto l'**istruzione** che dovremo scrivere per questa operazione sarà:

```
ldi a,1
addi a,48
ld ddata,a
call dsend

ldi a,6
addi a,48
ld ddata,a
call dsend
```

ISTRUZIONI di INIZIALIZZAZIONE

Come abbiamo già detto, quando scriverete dei nuovi programmi dovrete sempre **iniziare** con tutta una serie di istruzioni di inizializzazione.
Nei due programmi che troverete nel dischetto **DF1208** questo **set** di **istruzioni** sono riportate:

nel programma **DISP093** nelle **righe 77-109**
nel programma **TESTER** nelle **righe 81-113**

In questi due programmi questo **set** di **istruzioni** è posizionato nelle righe **77-109** e nelle righe **81-113** solo perché prima di queste abbiamo dovuto riportare due diverse serie di **variabili** necessarie per far funzionare i due programmi.

Come noterete questi due **set** di **istruzioni**, anche se posti in righe diverse, sono perfettamente identici.
Dovrete **sempre** riportare nei vostri programmi personalizzati tutte queste righe senza apportare alcuna modifica, dopo tutte le vostre **variabili.**

TABELLA n. 1

0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111	
32	48	64	80	96	112	160	176	192	208	224	240	xxxx0000
33	49	65	81	97	113	161	177	193	209	225	241	xxxx0001
34	50	66	82	98	114	162	178	194	210	226	242	xxxx0010
35	51	67	83	99	115	163	179	195	211	227	243	xxxx0011
36	52	68	84	100	116	164	180	196	212	228	244	xxxx0100
37	53	69	85	101	117	165	181	197	213	229	245	xxxx0101
38	54	70	86	102	118	166	182	198	214	230	246	xxxx0110
39	55	71	87	103	119	167	183	199	215	231	247	xxxx0111
40	56	72	88	104	120	168	184	200	216	232	248	xxxx1000
41	57	73	89	105	121	169	185	201	217	233	249	xxxx1001
42	58	74	90	106	122	170	186	202	218	234	250	xxxx1010
43	59	75	91	107	123	171	187	203	219	235	251	xxxx1011
44	60	76	92	108	124	172	188	204	220	236	252	xxxx1100
45	61	77	93	109	125	173	189	205	221	237	253	xxxx1101
46	62	78	94	110	126	174	190	206	222	238	254	xxxx1110
47	63	79	95	111	127	175	191	207	223	239	255	xxxx1111

PER PASSARE dalla 1° alla 2° RIGA

Poiché normalmente si scrive partendo dalla **1° riga** per poi passare alla **2° riga,** le prime istruzioni che dovrete scrivere saranno:

res 1,port_b "prepararsi per l'istruzione"
ldi ddata,00000010b "istruzione per la **1° riga**"
call dsend "subroutine per invio dati"

Continuerete quindi con le istruzioni che servono ad **incrementare** di una casella, cioè a far sì che la prima lettera o numero che vorrete far apparire si posizioni automaticamente nella **prima** casella, la seconda lettera nella **seconda** casella, ecc.
Ammesso di voler scrivere **A**, dovrete scrivere il suo numero **decimale**, quindi:

ldi ddata,00000110b "incrementa di una casella"
call dsend "subroutine per invio dati"
set 1,port_b "fine set istruzioni"
ldi ddata,65 "trasferisci A in ddata"
call dsend "subroutine per invio dati"

Per scrivere 16 caratteri in ogni **riga** si potrebbe scrivere 16 volte questa istruzione, mettendo nella riga **ldi ddata** il numero **decimale** che si desidera far apparire, ma poiché questa soluzione risulta **poco pratica,** vi consigliamo di andare a vedere nel programma **DISP093** come abbiamo risolto in modo più elegante il problema per far apparire sul Display la parola **N.ELETTRONICA**.

Per scrivere nella **2° riga** posta sotto la **1°,** dovrete scrivere queste istruzioni:

res 1,port_b "prepararsi per l'istruzione"
ldi ddata,11000000b "posizionamento in **2° riga**"
call dsend "subroutine per invio dati"
set 1,port_b "fine set istruzioni"

Ammesso di voler far apparire nella **seconda casella** la lettera **B,** dovrete scrivere:

ldi ddata,66 "trasferisci **B** in ddata"
call dsend "subroutine per invio dati"

Vorremmo aggiungere che anche se sul display sono **visibili** solo **16 caselle** per **riga,** in pratica ve ne sono per ogni riga altre **24 nascoste** e queste righe **nascoste** possono servire nel caso si desiderino far scorrere sul display delle **scritture** da destra verso sinistra o viceversa.
Nel programma **DISP093** che troverete nel dischetto **DF1208**, oltre a tutte le **sorgenti** abbiamo riportato anche degli esempi per ottenere questa funzione.

NOTA per l'EDIT dell'ST6

Dobbiamo precisare che l'**EDIT,** che vi avevamo fornito nei precedenti dischetti **LX.1207** con l'intento di semplificare tutte le operazioni, risulta **insufficiente** per programmi molto **lunghi** come quelli utilizzati per questo Display **alfanumerico** siglato **LX.1208.**
Infatti questo **EDIT** accetta solo programmi che **non superino** i **30 Kilobyte** quindi, quando li andrete a **salvare**, tutto quello che **eccede** i **30 K** verrà **inesorabilmente cancellato.**
Se perciò vorrete **modificare** e trasferire nella memoria dell'**ST6** un programma per questo Display **alfanumerico** o altri che superino i **30 K,** dovrete **necessariamente** utilizzare l'**Editor** del **DOS** presente nel vostro computer.

Per caricare i due programmi presenti nel dischetto **DF1208** dovrete procedere come segue:

Quando sul monitor appare **C:\>** dovrete inserire il dischetto nel drive **A** e scrivere:

C:\>A: poi premete Enter
A:\> installa poi premete Enter

Il programma vi chiederà su quale **directory** volete installare il contenuto del dischetto e, poiché noi l'abbiamo già definita **LX1208,** dovrete solo premere il tasto Enter.

Si creerà così automaticamente la directory **LX1208** e mentre verranno trasferiti nell'Hard-Disk tutti i programmi presenti nel dischetto floppy verranno anche **scompattati.**

Se usando questo metodo vi apparirà la scritta **error,** vi consigliamo di ricaricare il dischetto nell'Hard-Disk utilizzando questo secondo metodo:

C:\>MD LX1208 poi premete Enter
C:\>COPY A:*.* C:\LX1208 poi premete Enter
C:\>CD LX1208 poi premete Enter
C:\LX1208>installa poi premete Enter

Nota = Per agevolarvi a rispettare le spaziature, abbiamo utilizzato una barra in **colore** che corrisponde ad uno **spazio.**

Ammesso che desideriate modificare il programma **TESTER** presente nel disco **DF.1208,** quando sul monitor appare **C:\>** scrivete:

C:\>CD LX1208 (chiama la directory)
C:\LX1028>Edit TESTER.ASM (chiama Editor del Dos)

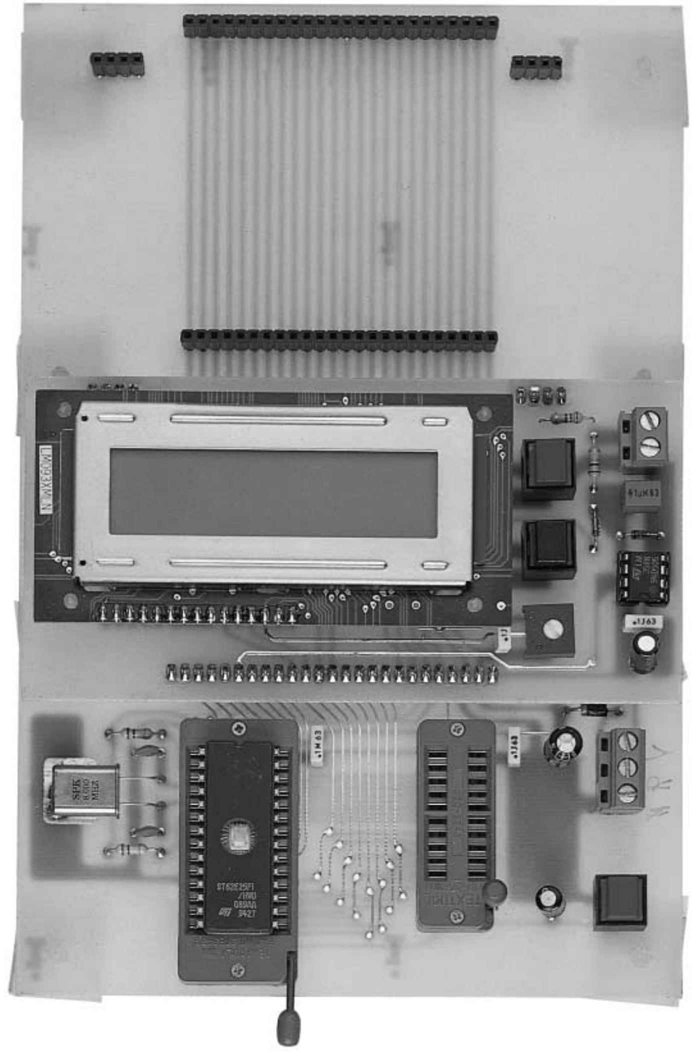


Fig.6 La scheda di questo display andrà inserita nel Bus siglato LX.1202, non dimenticando di innestare nello zoccolo textool un micro ST6 tipo ST6/E25 per trasferire il programma DISP093.HEX oppure il TESTER.HEX.

IMPORTANTE

Con questo programma **LX.1208** non viene più utilizzato l'**Editor** dell'**ST6** ma l'**Editor** del **DOS,** quindi per trasferire i programmi dall'Hard-Disk alla **memoria** dell'**ST62/E25** non dovrete più convertire i programmi da **.ASM** in **.HEX** come vi avevamo insegnato in precedenza per tutti i programmi presenti nel dischetto **LX.1207,** ma dovrete procedere in modo completamente diverso.

Dopo aver eseguito tutte le modifiche sui programmi dovrete premere i tasti **ALT F,** poi portare il cursore sulla riga **SALVA** e premere Enter ed infine sulla riga **ESCI** e premere Enter.

Ammesso di voler compilare il programma **DISP093,** quando sul monitor apparirà **C:\LX1208>** dovrete scrivere**:**

C:\LX1208>A DISP093.ASM

Nota = Dopo la lettera **A** non mettete "**:**" perché questa **A** è un programma **Batch.**

Con questa istruzione convertirete **automaticamente** il programma da **.ASM** a **.HEX**.

Per trasferire i programmi già compilati in **.HEX** nel microprocessore posto sull'interfaccia **LX.1202,** dovrete richiamare la **directory LX1208** e poi scrivere semplicemente:

C:\LX1208>ST6PGM poi premere Enter

A questo punto sul monitor apparirà una maschera che vi chiederà quale programma intendete trasferire e quale **micro** avete scelto e, una volta che avrete risposto a queste domande, potrete memorizzare il vostro micro **ST62/E25**.
Tutte le istruzioni relative a come trasferire un programma dall'Hard-Disk ad un micro **ST6** le abbiamo riportate negli articoli pubblicati sulle riviste **N.172/173-174-175/176-179-180-181,** quindi a chi fosse interessato a questo argomento suggeriamo di procurarsi tali numeri al più presto prima che vengano esauriti.

SCHEMA ELETTRICO

Lo schema elettrico di questo progetto, come potete vedere in fig.4, è quanto di più semplice si possa immaginare.
Abbiamo contrassegnato **otto** dei **15 piedini** del display con le sigle da **DB0** a **DB7** per non confonderli con i segnali da **B0** a **B7** presenti nel connettore che va inserito nella scheda bus **LX.1202.**
Nell'articolo abbiamo spiegato che per gestire questo display bisogna usare un codice di **8 bit,** che andrà applicato sui piedini **DB0 - DB1 - DB2 - DB3 - DB4 - DB5 - DB6 - DB7** (piedini dal numero **7** al numero **14**), mentre osservando lo schema elettrico riportato in fig.4 si potrà notare che i piedini da **DB0** a **DB3** sono collegati a **massa** e che il segnale entra nei soli piedini da **DB4** a **DB7**.
Questa configurazione è stata adottata perché per gestire questo display abbiamo usato un codice **4+4 bit.**
Oltre al display, nello schema elettrico è presente anche un amplificatore operazionale siglato **IC1/A;** a questo proposito vi chiederete se tale amplificatore sia indispensabile per far funzionare questo display e noi vi rispondiamo che **non serve.**
Infatti l'abbiamo **inserito** soltanto per potervi dimostrare come sia possibile, con il **programma Tester**, trasformare questo display in un **voltmetro.**
Dobbiamo subito precisare che nell'ingresso di questo **operazionale** non è possibile inserire delle tensioni superiori ai **5 volt;** per poterlo fare sarà necessario applicare sull'ingresso dei partitori **resistivi** da **1/10 - 1/100**.
Oltre a questo particolare, dobbiamo anche ricordarvi di rispettare la **polarità** della tensione sull'ingresso, perché se **invertirete** il positivo con il negativo sul display appariranno **0 volt**.
Il trimmer **R3** collegato al piedino **3** serve per variare la **luminosità** delle lettere o dei numeri che appariranno nelle diverse caselle.
Tutti i piedini del **display** e quello d'uscita dell'**operazionale** vengono collegati al **Connettore 1/2** che andrà innestato nella scheda bus **LX.1202**.

REALIZZAZIONE PRATICA

Sul circuito stampato siglato **LX.1208** dovrete montare tutti i componenti visibili in fig.7.
Vi consigliamo di iniziare dal **connettore maschio** a **24 terminali** e di procedere inserendo gli altri due **connettori maschi** a **4 terminali** (nello schema pratico si vede solo quello di destra) ed il **connettore femmina** a **15 terminali** che userete come zoccolo per il display.
Completata questa operazione, potrete inserire lo zoccolo per l'integrato **IC1** e tutti gli altri componenti richiesti, cioè pulsanti, trimmer, condensatori e resistenze.
Nel montaggio dovrete solo rispettare la polarità dei due diodi al silicio **DS1-DS2,** posizionando il lato del loro corpo contornato da una **fascia nera** come appare ben visibile nello schema pratico di fig.7.
Completato il montaggio, dovrete inserire nello zoccolo l'integrato **IC1,** rivolgendo la tacca di riferimento a forma di **U** presente sul suo corpo verso il condensatore **C4**.
Per fissare il **display** dovrete inserire nei quattro fori presenti sullo stampato i distanziatori plastici inclusi nel kit, dopodichè potrete innestare i terminali del display nello zoccolo femmina, facendo entrare i perni dei distanziatori nei fori presenti sullo stampato del display.

PROGRAMMI PER LM093

Nel dischetto **DF1208** sono riportati due programmi per utilizzare il display **alfanumerico** in tutti i modi possibili.

Questi due programmi sono denominati:

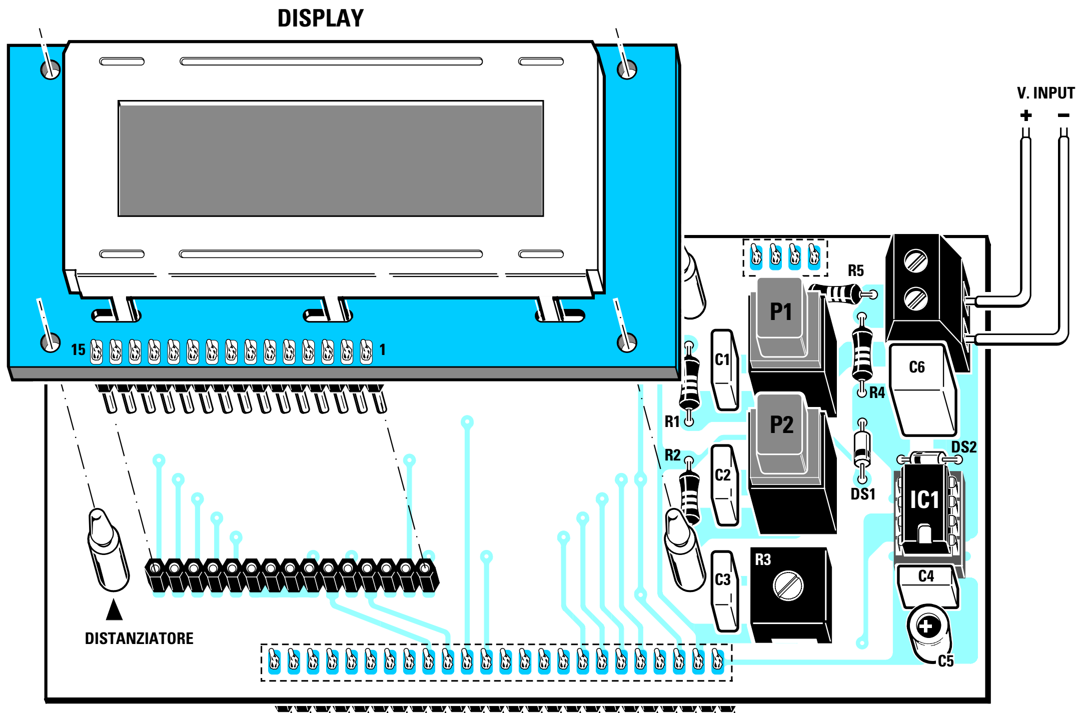
DISP093.ASM
TESTER.ASM

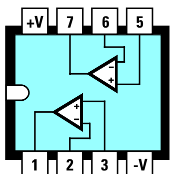
A questi due programmi occorrono altri due **files** chiamati:

TB_CGR01.ASM
TB_CGR02.ASM

Questi due ultimi files **TB_CGR** sono in pratica delle **tabelle** che vi serviranno per utilizzare una **direttiva** denominata **.input**.
Questa direttiva altro non è che una **istruzione** inserita nel file sorgente, che in fase di compilazione "richiama" una serie di **dati** contenuti in un file **diverso** dal sorgente.

Quando **assemblerete** il programma **DISP093.A-**





LM 358

Fig.7 Schema pratico di montaggio della scheda LX.1208. Sul circuito stampato dei display dovrete saldare il connettore maschio a 15 terminali, che dovrete poi innestare nel circuito stampato LX.1208.

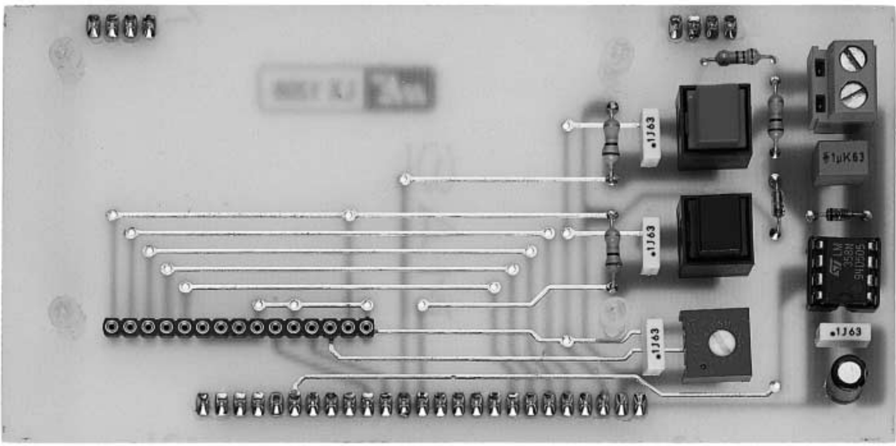


Fig.8 Foto dello stampato LX.1208 con sopra già montati tutti i componenti richiesti. Nota = Nel kit, anzichè trovare un connettore maschio da 24 poli e due da 4 poli potreste trovarne uno solo da 32 terminali, che dovrete tagliare per ottenere i tre pezzi richiesti.

SM, il compilatore andrà a ricercare il file:

TB_CGR01.ASM

e lo ingloberà al suo interno, formando così un **unico file** che si chiamerà: **DISP093.HEX**.

Quando **assemblerete** il programma **TESTER.ASM,** il compilatore andrà a ricercare il file **TB_CGR02.ASM** e lo ingloberà al suo interno, formando così un **unico file** che si chiamerà: **TESTER.HEX**.

Quindi quando **assemblerete** questi due programmi non dovrete assolutamente preoccuparvi di questi files **TB_CGR,** perché le operazioni di ricerca e di inserimento vengono eseguite automaticamente.
La direttiva chiamata **.input** è simile ad una subroutine, con la sola differenza che viene eseguita in fase di compilazione e non durante l'esecuzione del programma stesso.

Per terminare aggiungiamo che una volta inserita la **LX.1208** di questo display **alfanumerico** nel bus **LX.1202,** in quest'ultimo bus non potrete inserire altre schede, tipo **TRIAC** o **RELE'**.

Programma DISP093.HEX

Questo programma, che utilizza i soli pulsanti **P1 - P2** presenti nella scheda **LX.1208,** vi permetterà di visualizzare il vostro **nome** e **cognome** o qualsiasi altra **scritta** sul display, a condizione di utilizzare un **massimo** di **16 caratteri** per riga.
Facciamo presente che è necessario considerare gli **spazi** come **caratteri**.

Il vostro nome e cognome o una qualsiasi altra scritta, dovrà essere scritto all'**interno** del programma **.ASM** nelle **righe** che vi indicheremo.
Dopo aver scritto le parole che dovranno apparire sul display, dovrete **riassemblare** il programma ed infine caricarlo nel micro **ST62/E25,** che andrà inserito nello zoccolo **textool** presente nella scheda bus **LX.1202**.
Se caricherete nel micro **ST62/E25** il programma **DISP093.ASM,** ovviamente dopo averlo assemblato in **DISP093.HEX**, sul display vedrete apparire in ordine alcune scritte:

N.ELETTRONICA
**** DISP093 ****

Queste scritte rimarranno visualizzate per circa **3 secondi**, dopodichè apparirà:

[P1] >
PER PROCEDERE

Se non premerete il pulsante **P1** vedrete alternarsi le due scritte sopra riportate con una cadenza di circa **1 secondo**.

Se invece premerete il pulsante **P1** per più di **3 secondi** circa, apparirà questa nuova scritta:

-NOME-NOME-NOME-
-COGNOME-COGNOME

Come noterete, ogni riga occupa un totale di **16 caratteri**.

Questa scritta rimarrà visualizzata sul display per circa **6 secondi,** dopodichè apparirà la scritta:

LE FUNZIONI
PREVISTE SONO:

Anche questa scritta rimarrà visualizzata per circa **6 secondi**, dopodichè apparirà questa scritta:

1-MAIUSC.> minus
2-ROTAZIONE

Questa rimarrà visualizzata per circa **6 secondi,** dopodichè apparirà:

3-SCOMPOSIZIONE
4-VISUAL.CG RAM

e nuovamente vedrete apparire:

[P1] >
PER PROCEDERE

Come noterete, sulla base delle scritte apparse, potrete ottenere **4 diverse** funzioni che sono numerate **1-2-3-4**.
Solo dopo che saranno apparse **tutte** le scritte che vi abbiamo sopra indicato, potrete utilizzare il **tasto P1** per selezionare **una** della **4 funzioni**.
Se **non premerete** il pulsante **P1** vedrete nuovamente ripetersi all'infinito le stesse scritte.
Quando apparirà **P1 > PER PROCEDERE** dovrete tenere premuto questo tasto per almeno **3** secondi e apparirà la scritta:

SCELTA FUNZIONE
......[?] [P2] >

A questo punto, utilizzando il tasto **P2** potrete scegliere una della quattro funzioni numerate **1-2-3-4**.

Questa scritta rimarrà sui display fino a quando non premerete il pulsante **P2**.
Se terrete premuto per almeno **3 secondi** il pulsante **P2,** apparirà questa scritta:

SCELTA FUNZIONE
......[1] [P2] >

Come potete vedere, nelle due parentesi quadre è sparito il **?** ed è apparso il numero **1.**
Premendo per una **seconda** volta **P2** apparirà il numero **2**, premendo una **terza** volta **P2** apparirà il numero **3** e premendolo per la **quarta** volta apparirà il numero **4.**
Fate attenzione a premerlo per la **quinta** volta, perché se sul display apparirà il numero **5** uscirete dal **menu.**

Ammesso di voler visualizzare la funzione **1-MAIUSC.> minus** quando appare:

SCELTA FUNZIONE
......[1] [P2] >

dovrete premere per circa **3 secondi** il pulsante **P1** e comparirà la scritta:

-NOME-NOME-NOME-
-COGNOME-COGNOME

dopo circa **5 secondi** vi apparirà la stessa scritta ma in **minuscolo**, ovvero:

-nome-nome-nome-
-cognome-cognome

Tale scritta in **minuscolo** resterà visualizzata per circa **5** secondi, dopodichè vi riapparirà nuovamente la scritta:

SCELTA FUNZIONE
......[?] [P2] >

A questo punto, se premerete il tasto **P2** per **due** volte consecutive, sceglierete la funzione **2-ROTAZIONE,** quindi quando apparirà:

SCELTA FUNZIONE
......[2] [P2] >

tenendo premuto per **3 secondi** il tasto **P1,** vedrete apparire un divertente effetto perché il vostro nome partirà dalla seconda riga, scorrendo da sinistra verso destra, per poi riportarsi sulla prima riga scorrendo da destra verso sinistra per **4 volte** consecutive, dopodichè riapparirà la scritta:

SCELTA FUNZIONE
......[?] [P2] >

Se ora premerete il tasto **P2** per **tre** volte consecutive, sceglierete la funzione **3-SCOMPOSIZIONE,** quindi quando apparirà:

SCELTA FUNZIONE
......[3] [P2] >

dovrete tenere premuto il pulsante **P1** per circa **3 secondi** e rilasciandolo vedrete che i caratteri riportati nella sola prima riga cominceranno a **scomporsi**, cioè vedrete i caratteri della **prima riga** allontanarsi **ad uno ad uno** scorrendo verso destra, fino a scomparire totalmente, poi li vedrete ritornare da destra verso sinistra fino a **ricostruire** l'intera parola.
Una volta ricostruiti i **16** caratteri sulla prima riga, nuovamente vedrete apparire la scritta:

SCELTA FUNZIONE
......[?] [P2] >

Se ora premerete il tasto **P2** per **quattro** volte consecutive, sceglierete la funzione **4-VISUAL.CG RAM** quindi quando apparirà:

SCELTA FUNZIONE
......[4] [P2] >

dovrete sempre tenere premuto il tasto **P1** per almeno **3 secondi** e sulla prima riga del display vedrete apparire **8 simboli grafici** generati appositamente a scopo didattico, più un cursore non lampeggiante.

I programmi inseriti nel dischetto **DF.1208** servono principalmente per farvi vedere come si debbano scrivere le varie istruzioni per far funzionare questo display **alfanumerico**.
Solo dopo che avrete preso una certa confidenza con questi programmi, potrete modificarli, o prelevare direttamente dalle nostre **sorgenti** tutte le righe che potrebbero interessarvi.
Una modifica **molto semplice** che potrete apportare è quella di far apparire sui display il vostro **nome** e **cognome** o qualsiasi altra scritta.

Se sulla **prima riga** volete far apparire il vostro **nome** che potrebbe essere **ALESSANDRO - MARCO - VINCENZO,** ecc. dovrete andare alla **riga**

N.684 posta all'interno del programma sorgente **DISP093.ASM** e sostituire la scritta da noi inserita con il vostro nome.

Se in corrispondenza della **seconda riga** volete far apparire il vostro **cognome** che potrebbe essere **BIANCHI - ALBERTAZZI - FANTOZZI,** ecc., dovrete andare alla **riga N.685** e sostituire la scritta da noi inserita con il vostro cognome.

Nota importante = Qualsiasi cosa scriverete nelle righe **684** e **685,** dovrete sempre farlo in caratteri **maiuscoli** e non superare mai i **16 caratteri** per riga compresi gli **spazi**.
Dopo aver eseguito queste modifiche dovrete premere i tasti **ALT F,** poi portare il cursore sulla riga **SALVA** e premere Enter ed infine sulla riga **ESCI** e premere Enter.
Dopodichè dovrete richiamare il programma **LX1208** scrivendo:

C:\>CD LX1208 poi premere Enter

e scrivere:

C:\LX1208>A DISP093.ASM

Nota = Dopo la lettera **A** non mettete **":",** perché questa **A** non è altri che un programma Batch che lancia il compilatore in assembler**.**

Con questa istruzione convertirete **automaticamente** il nostro programma da **.ASM** in **.HEX**.

Per trasferire questo programma già compilato in **.HEX** nel microprocessore posto sull'interfaccia **LX.1202,** dovrete richiamare la **directory LX1208** e poi scrivere semplicemente:

C:\LX1208>ST6PGM poi premere Enter

A questo punto sul monitor apparirà una maschera che vi chiederà quale programma desiderate trasferire e su quale **micro ST6;** fornite al computer le esatte risposte, il vostro programma modificato verrà memorizzato nel micro **ST62/E25**.

TESTER.HEX

Con questo programma dimostrativo desideriamo insegnarvi ad utilizzare l'**A/D** converter presente all'interno del microprocessore **ST6/E25** e per farlo abbiamo realizzato con questo display **alfanumerico** un semplice **voltmetro elettronico** utilizzando entrambe le righe presenti nel display.

Sulla **prima riga** faremo apparire il valore della tensione in **numero,** mentre sulla **seconda riga** faremo apparire una **barra** che si allungherà di **1 riga** ogni **0,1 volt** e di **1 quadretto** ogni **0,5 volt**.
Chi fosse interessato a comprendere come siamo riusciti ad ottenere queste due condizioni, dovrà leggere attentamente il programma **TESTER.ASM** e i **commenti** riportati su ogni riga.
Vogliamo subito far presente che il **massimo valore** di tensione che potremo leggere con questo voltmetro è di soli **5 volt,** quindi non applicate sull'ingresso dell'operazionale **IC1/A** tensioni maggiori.
Per leggere tensioni di **50 volt fondo scala,** dovrete necessariamente utilizzare dei partitori resistivi come illustrato nelle figg.10-11.

La tensione da misurare, applicata sull'ingresso dell'operazionale **IC1/A,** verrà prelevata dal suo piedino d'uscita **7** ed inviata all'**A/D converter** presente all'interno del microprocessore **ST6/E25,** che la convertirà in un numero **decimale** compreso tra **0** e **255**.

L'**A/D converter** per un valore di tensione di **5 volt** ci dà un numero **decimale** di **255;** se dividiamo **255** per **5** otteniamo **51,** quindi è intuitivo che per un valore di tensione di **1 volt** l'**A/D converter** ci darà un numero **decimale** di **51** e per un valore di **2 volt** ci darà un numero **decimale** di **102** e per **3 volt** un numero **decimale** di **153.**
Se misurassimo una tensione di **2,5 volt,** in teoria l'**A/D converter** dovrebbe darci il numero **51 x 2,5 = 127,5**, ma poiché in pratica non ci darà mai un numero con la **virgola,** sulla sua uscita otterremo dei numeri variabili molto prossimi a **127,** ad esempio **127-128-128-129-127-129,** perché l'**A/D converter** dell'ST6 non è molto stabile.

Sommando i **6** numeri del nostro esempio otterremo un totale di **768**, che diviso per **6** ci darà il valore medio:

768 : 6 = 128

Dividendo **128** per **51** otterremo:

128 : 51 = 2,50

Per ottenere una maggiore precisione nel nostro programma leggeremo i **numeri decimali** che l'**A/D converter** ci fornirà per ben **32 volte,** poi una volta **sommati** li divideremo per **32.**
Per far apparire il numero **2,5** metteremo il numero **2** in un **byte** e il numero **5** in un altro **byte.**

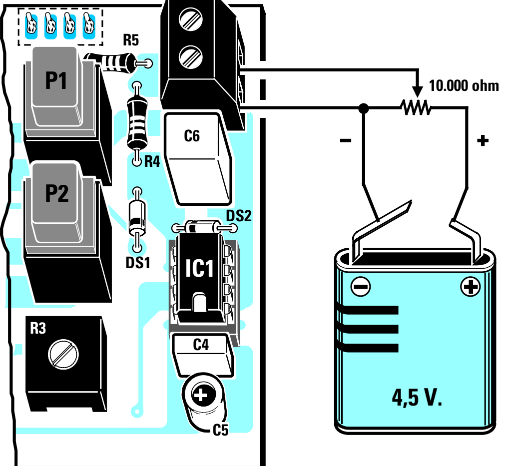


Fig.9 Il programma TESTER.HEX permette di utilizzare questo display alfanumerico in un Voltmetro in grado di misurare un massimo di 5 volt. Per provare questo Voltmetro potrete procurarvi un trimmer da 10.000 ohm ed una pila da 4,5 volt.
Ruotando il suo cursore vedrete apparire sui display il valore della tensione.

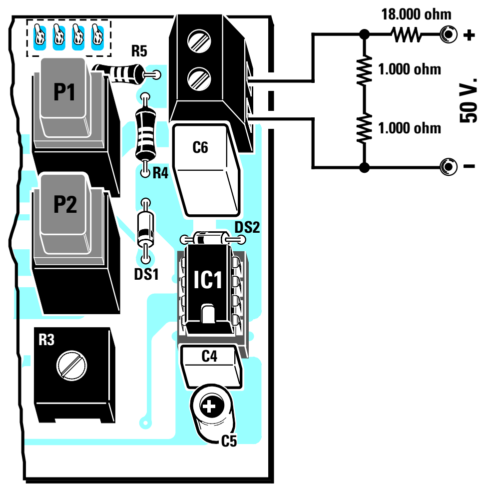


Fig.10 Volendo utilizzare la funzione Voltmetro per leggere tensioni maggiori, dovrete applicare sull'ingresso un partitore resistivo composto da tre sole resistenze.
Con questo partitore potrete leggere fino ad un valore massimo di 50 volt.

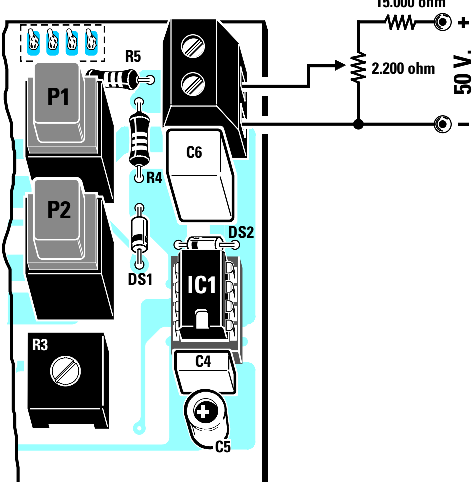


Fig.11 Utilizzando il partitore resistivo di fig.10 la lettura potrebbe non risultare precisa a causa delle "tolleranze" delle resistenze. Per risolvere questo problema, potrete utilizzare una sola resistenza ed un trimmer che andrà tarato in modo da leggere l'esatta tensione applicata sull'ingresso.

Come vi abbiamo già spiegato in precedenza, perché il display **intelligente** faccia apparire un qualsiasi **segno grafico** contenuto all'interno della sua **CGROM,** gli deve giungere un **numero** ben diverso dal **2** e dal **5** inseriti in questi due **byte,** per cui dovremo **sommare** questi due numeri alla **costante 48.**
Otterremo così:

2 + 48 = 50
5 + 48 = 53

Consultando la Tabella N.1 vedremo che il numero **50** corrisponde al **simbolo grafico 2** ed il numero **53** al **simbolo grafico 5**.

Per visualizzare la **barra** che appare sulla **seconda riga** utilizziamo i due numeri **2** e **5,** che abbiamo messo in precedenza nei due **byte,** e con questi due numeri andiamo nel file **TB_CGR02.ASM** per prelevare i **simboli grafici** che ci serviranno per accendere tutte le **caselle** interessate.
Poiché con **5 volt** si accendono **10 caselle** orizzontali, è ovvio che disponendo di una tensione di **2,5 volt** si accenderanno solo **5 caselle**.

Per provare questo voltmetro potrete procurarvi un trimmer da **10.000 ohm,** più una pila da **4,5 volt,** collegandoli come visibile in fig.9.
Ruotando il cursore del trimmer da un estremo all'altro, vedrete variare sulla **prima riga** del display il numero da **0 volt** fino ad un massimo di **4,5 volt** e sulla **seconda riga** la **barra** aumentare progressivamente fino a raggiungere un massimo di **9 quadretti**.

Volendo utilizzare questo tester per misurare tensioni superiori a **5 volt,** dovrete applicare sull'ingresso un partitore resistivo con i valori riportati in fig.10 e, in tal modo, otterrete un **fondo scala** di **50 volt**.
Non è possibile utilizzare dei partitori resistivi che diano dei valori di **fondo scala** di **10-100-200 volt,** perché il programma è impostato per leggere un **massimo** di **5 volt**.
Come noterete, tra le due cifre rimane sempre inserita la **virgola,** quindi se avete utilizzato il **partitore** di fig.10 e sull'ingresso inserite **18 volt,** sul display apparirà il numero **1,8 volt**.
Poiché le resistenze hanno una loro **tolleranza,** il **partitore** di fig.10 potrebbe non fornirvi l'esatto valore di tensione; per risolvere questo problema la soluzione migliore sarebbe quella di utilizzare lo schema riprodotto in fig.11.
Per tarare il **trimmer** potrete prendere una esatta tensione **continua** che non risulti maggiore di **50 volt** e partendo con il cursore tutto ruotato **verso massa,** lo ruoterete lentamente in senso inverso fino a leggere l'esatta tensione applicata sull'ingresso.
Ammesso di aver scelto una tensione di **28 volt,** dovrete ruotare questo trimmer fino a leggere sui display il numero **2,8 volt**.

NOTA IMPORTANTE

Non invertite la polarità della tensione sull'ingresso dell'operazionale **IC1/A,** perché sui display vedrete sempre apparire **0,0 volt**.
Non applicate sull'ingresso tensioni maggiori di **5 volt** (per pochi istanti l'integrato accetta anche **9 volt**), diversamente si potrebbe danneggiare il microprocessore **ST6/E25**.
Al programma **TESTER.ASM, che funziona** solo come voltmetro per leggere una tensione di **5 volt massimi,** non è possibile apportare alcuna modifica.
Questo programma, come già vi abbiamo accennato, è un dimostrativo che vi permetterà di vedere tutte le varie soluzioni che abbiamo adottato per far apparire un numero proporzionale alla tensione e come si utilizzano le tabelle del **TB_CGR02.ASM** e l'**A/D converter**.
Come già saprete, per poter memorizzare il programma **TESTER.ASM** all'interno del micro **ST6,** lo dovrete prima convertire in **.HEX** e per farlo dovrete digitare:

C:\>CD LX1208 poi premete Enter
C:\LX1208>A TESTER.ASM poi premete Enter

Per poter trasferire il programma convertito in **.HEX** sul micro **ST6** dovrete semplicemente scrivere:

C:\LX1208>ST6PGM poi premere Enter

e rispondere, come già sapete, a tutte le domande che appariranno sul monitor del computer.

KIT ESAURITO
vedi LX.1208/N nelle pagine seguenti

COSTO DI REALIZZAZIONE

Tutti i componenti necessari per la realizzazione di questo progetto per Display alfanumerico, che potete vedere riprodotti in fig.7 (Escluso il disco DF.1208) L.78.500

Il programma DF.1208..............................L.12.000

Costo dello stampato LX.1208 ....L.10.000

Ai prezzi riportati, già comprensivi di IVA, andranno aggiunte le sole spese di spedizione a domicilio.

IL KIT LX.1208/N con il nuovo DISPLAY WH.1602A

Poiché il display alfanumerico **LM.093LN** non viene più fabbricato, ci siamo dati da fare per cercare un sostituto che lo rimpiazzasse nel kit **LX.1208**.

Dopo un'accurata ricerca, abbiamo scelto il display **WH.1602A** della **Hitachi**, che è equivalente al display **LM.093N**, eccetto che nella disposizione di alcuni piedini e nella definizione di alcuni caratteri alfanumerici (vedi tabella nelle pagine seguenti).

Proprio perché la piedinatura del display **WH.1602A** non collima perfettamente con quella del display **LM.093N** (se confrontate la fig.5 con la fig.12 vi accorgerete subito che il display della Hitachi ha un piedino in più), abbiamo pensato noi a disegnare e a fare incidere un nuovo circuito stampato al quale abbiamo dato la sigla **LX.1208/N**.

In questo modo non incontrerete alcuna difficoltà nel realizzare la scheda e soprattutto nel montare il nuovo display alfanumerico.
Inoltre, come siamo soliti fare per tutti i nostri kit, abbiamo già montato e collaudato questa scheda, e quindi possiamo assicurarvi che il circuito funziona esattamente come funzionava l'altro.

Vale la pena sottolineare che il display **WH.1602A** utilizzato, è un display **LCD alfanumerico** composto da **due righe** di **16 caratteri**.

Come vi abbiamo anticipato, rispetto al display **LM.093LN**, che aveva solo un piedino per regolare la luminosità, il display **WH.1602A** ha due controlli: con i collegamenti al positivo di alimentazione e a massa dei piedini **15-16** viene retro illuminato, mentre il trimmer **R4** collegato al piedino **3** consente di regolarne il contrasto.

Per la descrizione dello schema elettrico e per il montaggio dei componenti sul circuito stampato, rimandiamo a quanto già descritto nelle pagine precedenti, perché il funzionamento del circuito non è cambiato.
Per quanto riguarda l'**elenco componenti** e i **disegni** degli **schemi** elettrico e pratico, tenete invece presenti quelli riportati in queste pagine.

IL SET dei CARATTERI ALFANUMERICI

Nella pagina seguente abbiamo riportato anche la tabella relativa ai caratteri alfanumerici gestiti dal display **WH.1602A**, che, come vi dicevamo, non coincide perfettamente con il set di caratteri che veniva gestito dal vecchio display. Infatti se la confrontate con la **Tabella N.1** di questo articolo, vedrete che alcuni caratteri sono diversi.

In particolare, il display **WH.1602A** non ha tra i suoi caratteri le due frecce che nella **Tabella N.1** si trovano alle posizioni **126** e **127**.

Per questo motivo alcune delle istruzioni presenti nei programmi **DISP093.ASM** e **TESTER.ASM** vanno modificate, come ora vi spieghiamo.

Dopo la modifica, al posto delle frecce alle posizioni **126** e **127** della **Tabella N.1**, appariranno le frecce alle posizioni **62** e **60** della tabella del set di caratteri del display **WH.1602A**.

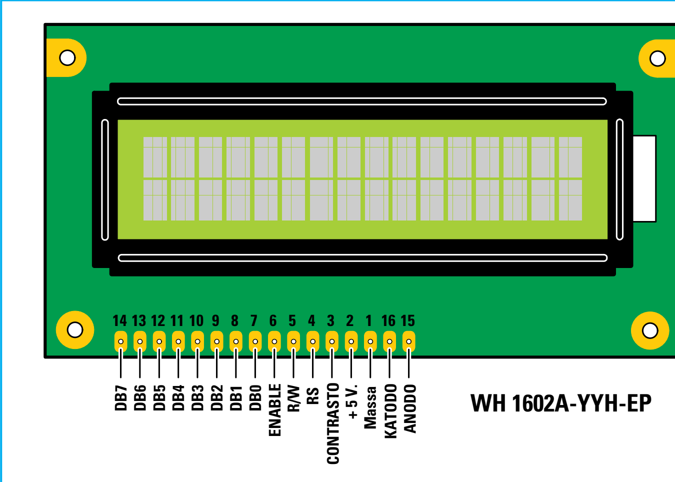


Fig.12 Connessioni del display alfanumerico WH.1602A. Il microprocessore ST6 esterno dialoga con questo display a 4+4 bit attraverso i piedini da DB4 a DB7, che fanno capo ai piedini 11-12-13-14 (vedi fig.13). I piedini da DB0 a DB3, che fanno capo ai piedini dal 7 al 10, vanno collegati a massa perché non vengono utilizzati.

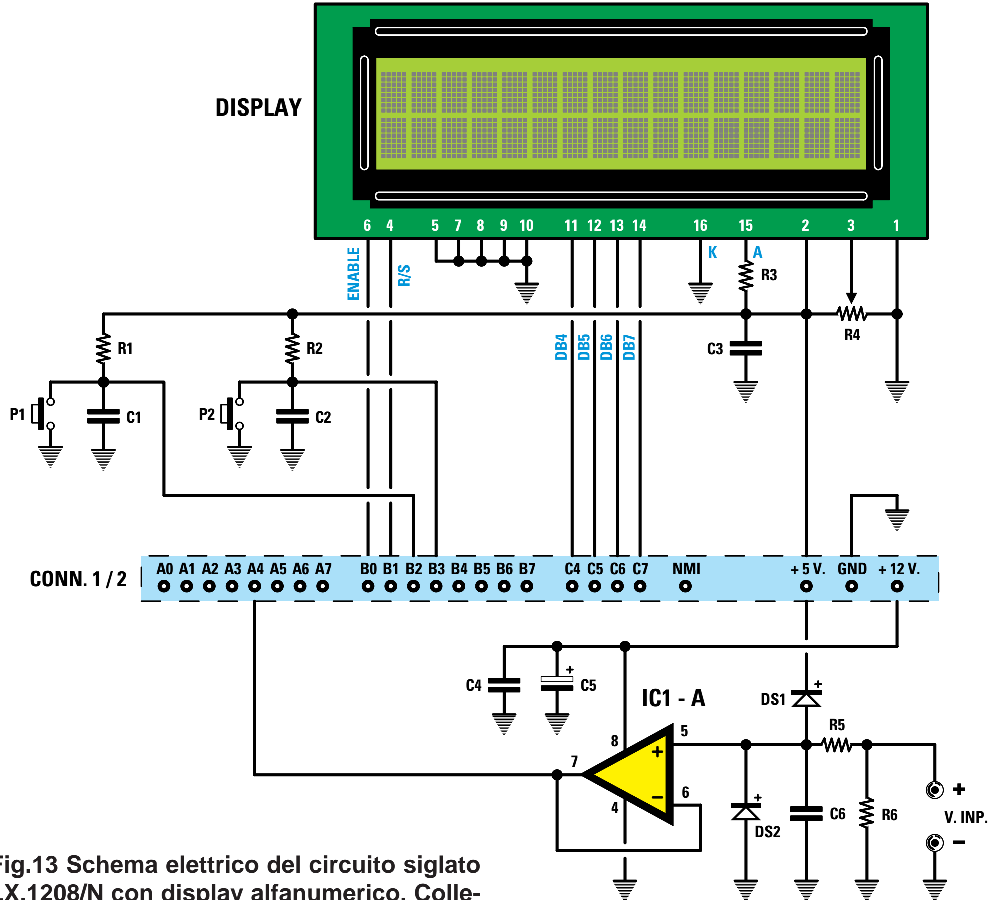
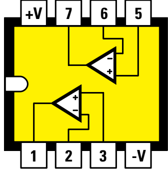


Fig.13 Schema elettrico del circuito siglato LX.1208/N con display alfanumerico. Collegando i piedini 16-15 rispettivamente a massa e al positivo di alimentazione, il display si retro illumina. Per regolare il contrasto, dovete agire sul trimmer R4.



LM 358

Fig.14 Connessioni viste da sopra dell'amplificatore operazionale LM.358, siglato IC1/A nello schema elettrico di fig.13. Come già spiegato nell'articolo, questo amplificatore non è indispensabile al funzionamento del display, ma è stato inserito per dimostrarvi come sia possibile, con opportune istruzioni di programma (vedi programma TESTER.ASM), trasformare il display in un voltmetro.

ELENCO COMPONENTI LX.1208/N

R1 = 10.000 ohm 1/4 watt
R2 = 10.000 ohm 1/4 watt
R3 = 4,7 ohm 1/2 watt
R4 = 10.000 ohm trimmer
R5 = 10.000 ohm 1/4 watt
R6 = 1 Megaohm 1/4 watt
C1 = 100.000 pF poliestere
C2 = 100.000 pF poliestere
C3 = 100.000 pF poliestere
C4 = 100.000 pF poliestere
C5 = 10 microF. elettrolitico
C6 = 1 microF. poliestere
DS1 = diodo tipo 1N.4150
DS2 = diodo tipo 1N.4150
IC1 = integrato tipo LM.358
DISPLAY = LCD tipo WH.1602A
CONN.1/2 = connettore 24 poli
P1 = pulsante
P2 = pulsante

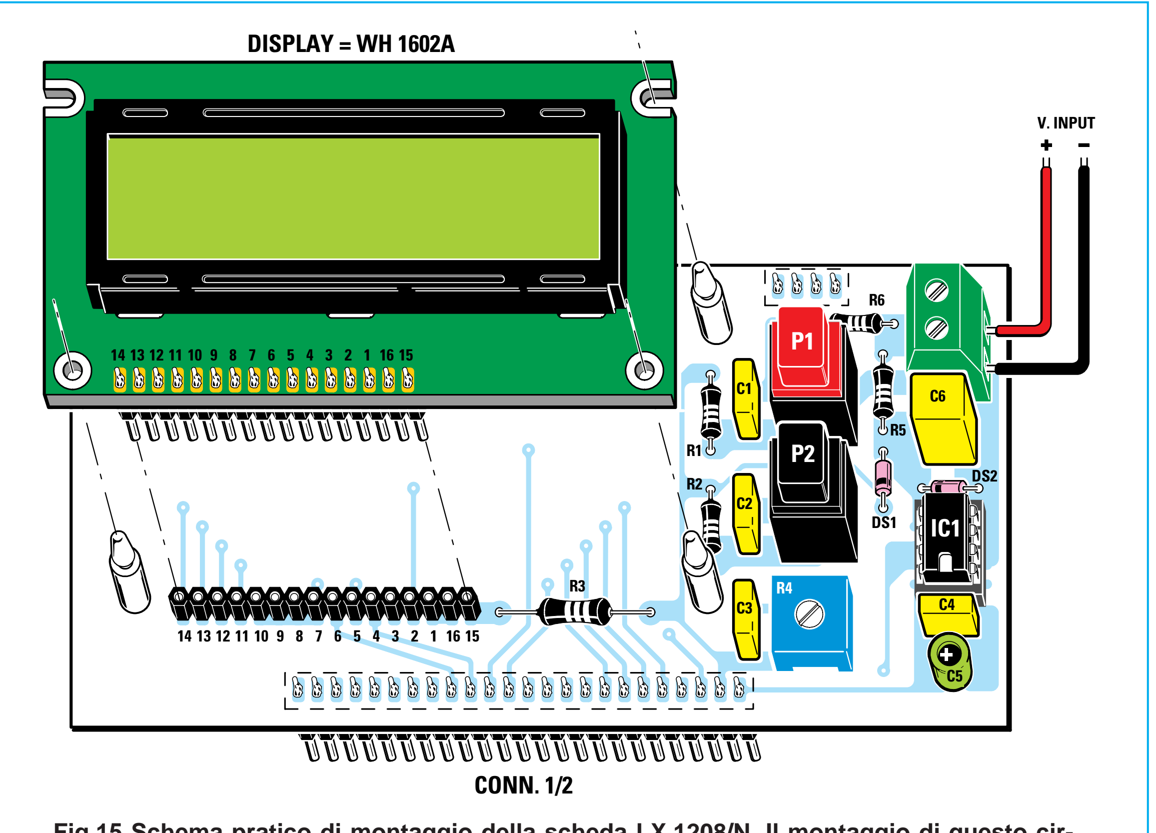


Fig.15 Schema pratico di montaggio della scheda LX.1208/N. Il montaggio di questo circuito non presenta particolari difficoltà e se rispetterete la polarità del condensatore elettrolitico C5 e dei due diodi al silicio DS1-DS2, il circuito funzionerà senza problemi. Sul lato opposto a quello visibile, vanno saldati i connettori a 4 terminali che, insieme al connettore CONN.1/2, vi serviranno per innestare la scheda nel Bus siglato LX.1202.

Nel programma **DISP093.ASM** alle righe **672 - 685 - 692**, dovete sostituire l'istruzione:

.byte 01111110b

con l'istruzione:

.byte 00111110b

Nel programma **TESTER.ASM** dovete invece andare **dopo** la riga **454** e nella tabella riportata, sostituire l'istruzione:

.byte 01111110b

con l'istruzione:

.byte 00111110b

Poi dovete lasciare invariata l'istruzione:

.byte 32,32,32

e infine sostituire l'istruzione:

.byte 01111111b

con l'istruzione:

.byte 00111100b

COSTO di REALIZZAZIONE

Costo dei componenti necessari per la realizzazione del kit con display alfanumerico **WH.1602A**, siglato **LX.1208/N**, visibile in fig.15, **escluso** solo il dischetto **DF.1208**
Euro 26,00

Costo del solo circuito stampato **LX.1208/N**
Euro 5,20

Costo del dischetto **DF.1208** con i programmi per display alfanumerico con micro ST6
Euro 6,20

TABELLA PER DISPLAY LCD tipo WH 1602A

0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111	
32	48	64	80	96	112	160	176	192	208	224	240	xxxx0000
33	49	65	81	97	113	161	177	193	209	225	241	xxxx0001
34	50	66	82	98	114	162	178	194	210	226	242	xxxx0010
35	51	67	83	99	115	163	179	195	211	227	243	xxxx0011
36	52	68	84	100	116	164	180	196	212	228	244	xxxx0100
37	53	69	85	101	117	165	181	197	213	229	245	xxxx0101
38	54	70	86	102	118	166	182	198	214	230	246	xxxx0110
39	55	71	87	103	119	167	183	199	215	231	247	xxxx0111
40	56	72	88	104	120	168	184	200	216	232	248	xxxx1000
41	57	73	89	105	121	169	185	201	217	233	249	xxxx1001
42	58	74	90	106	122	170	186	202	218	234	250	xxxx1010
43	59	75	91	107	123	171	187	203	219	235	251	xxxx1011
44	60	76	92	108	124	172	188	204	220	236	252	xxxx1100
45	61	77	93	109	125	173	189	205	221	237	253	xxxx1101
46	62	78	94	110	126	174	190	206	222	238	254	xxxx1110
47	63	79	95	111	127	175	191	207	223	239	255	xxxx1111

SOFTWARE emulatore per

Quando si scrivono programmi per qualsiasi microprocessore anche i più esperti possono commettere degli **errori** di **sintassi** oppure **logici**.
I primi, cioè quelli di **sintassi**, vengono già rilevati in fase di compilazione, perciò è abbastanza facile scoprirli e correggerli; i secondi, cioè quelli **logici**, possono essere scoperti solo se si dispone di un **emulatore real-time.**
Se non si possiede un **emulatore** il solo sistema per verificare che il programma risulti corretto è quello di trasferirlo in un micro **ST6 riprogrammabile**, cioè provvisto di una **finestra**.
Se, dopo averlo collocato nel circuito che dovrà gestire, si verifica che **non funziona**, bisogna ricontrollare il programma **istruzione** per **istruzione**, correggere gli errori commessi, sempre che si riesca a trovarli, ricompilare il programma con l'assembler, **cancellare** l'**ST6**, ed infine **riprogrammarlo** e "testarlo" nuovamente, perché non è detto che non vi siano altri errori che potrebbero essere sfuggiti ad un primo controllo.

Non è inoltre da escludere che nonostante la buona volontà non si riesca a capire in quale istruzione è presente l'**errore** e per scoprirlo ci potrebbe volere molto tempo e pazienza.

Con un **emulatore** risulta molto più facile ed anche meno costoso programmare qualsiasi **ST6**, perché si può controllare **passo** per **passo** ogni **istruzione** mentre viene eseguita. In questo modo è possibile capire dove e perché si è generato l'**errore**.

Ad esempio, potremmo aver scritto un programma che ad un **tempo** prefissato deve accendere una lampadina, e solo dopo aver programmato l'**ST6** ci accorgiamo che questo tempo risulta **dimezzato** o **raddoppiato** perché non abbiamo tenuto conto della frequenza del **quarzo** oppure abbiamo fatto una **somma** anziché una **moltiplicazione**.

Oppure potremmo aver scritto un programma per svolgere una semplice funzione, ad esempio:

- se suona il campanello
- vai ad aprire la porta

ma se non abbiamo tenuto presente che in casa ci sono diverse porte, quella d'ingresso, quella della cucina e quella della camera da letto e se non indichiamo nel programma quale porta **deve** essere **aperta**, si aprirà una porta qualsiasi e non quella d'ingresso come noi volevamo.

Un **emulatore** ci offre parecchi vantaggi.
Prima di tutto quello di non dover più acquistare un certo numero di **ST6 riprogrammabili** e, poiché il loro prezzo è salito alle stelle, si risparmierà una cifra considerevole.
Inoltre potendo controllare prima il programma, non si perderà del tempo per programmarli, cancellarli e riprogrammarli.

Infatti dopo aver eseguito un **test** completo sul vostro programma, se non rileverete delle anomalie potrete tranquillamente trasferirlo su un **ST6 non riprogrammabile** perché, avendolo già testato, avrete la matematica certezza che funzionerà.

Per questi motivi i softwaristi e gli hobbisti sono alla ricerca di un **emulatore** corredato di **software** che risulti facile da usare, molto economico e che permetta un'emulazione completa ed in **tempo reale** di un micro **ST6**.

Per risolvere questo problema abbiamo acquistato tutti gli **emulatori** per **ST6** e relativo **software** che siamo riusciti a reperire sul mercato, poi ad uno ad uno li abbiamo **testati** inserendo apposta nei nostri programmi degli **errori** con diversi livelli di difficoltà per verificare con quale grado di facilità ci consentivano di individuarli.

Tra tutti quelli provati ne abbiamo trovato **uno** che, a nostro avviso, è molto **valido** ed **evoluto** sia come **hardware** sia come **software**.
Si tratta di quello della **SOFTEC** di **Azzano Decimo**, in provincia di Pordenone.
Il suo **software** è inoltre perfettamente compatibile con il sistema operativo **Windows 3.1** e precedenti ed anche con il più recente **Windows.95**, laddove molti altri software presentano invece dei problemi.
Nota: Il pacchetto **non funziona** sotto **DOS**.

Utilizzando il **software** è possibile risolvere l'**80%** dei problemi relativi alla programmazione.
Noi vi spiegheremo come deve essere usato per controllare **passo** per **passo** ogni istruzione e per scoprire tutti gli **errori logici** che potreste aver commesso nello scrivere un programma.

TESTARE i micro ST6

Con il software "emulatore" della SOFTEC riuscirete a programmare senza difficoltà tutti i micro della serie ST6210/15/20/25 perché se commetterete qualche "errore" potrete "rintracciarlo" e correggerlo. Questo software funziona sotto Windows 3.1 e sotto Windows 95.

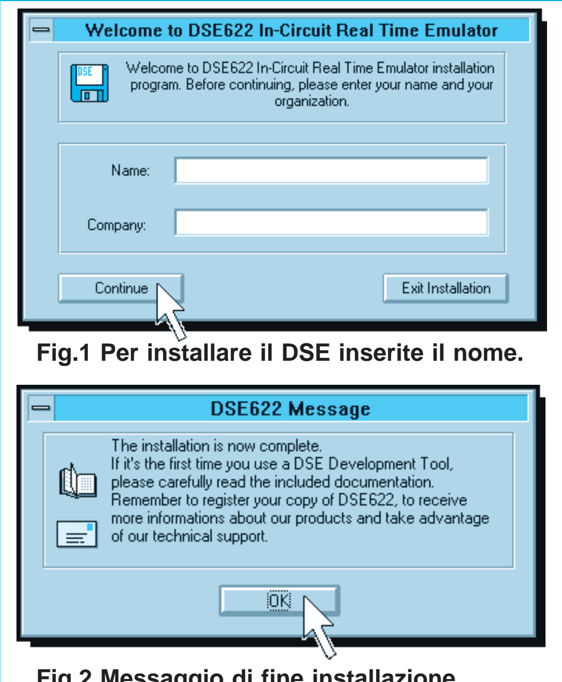


Fig.1 Per installare il DSE inserite il nome.

Fig.2 Messaggio di fine installazione.

Anche se supponiamo che tutti sappiano già come trasferire un programma da un floppy sull'Hard-Disk, riteniamo ugualmente utile ricordare queste poche istruzioni.

INSTALLARE il SOFTWARE sotto WINDOWS 3.1

Se nel vostro computer avete installato **Windows 3.1** o una versione precedente, dopo aver inserito il dischetto del **software DSE622** nel suo Drive entrate in **Program Manager**, poi portate il cursore in alto a sinistra sulla scritta **File** e cliccate, quindi andate sulla scritta **Esegui**, cliccate nuovamente e quando appare la finestra di dialogo digitate:

A:\setup poi cliccate su **OK**

In questo modo il **software DSE622** verrà trasferito dal floppy nell'Hard-Disk.

Quando appare la finestra **Name and Company** (vedi fig.1) inserite il vostro nome poi cliccate sulla scritta **Continue** per completare l'installazione.

Ad installazione avvenuta apparirà la finestra di fig.2: qui cliccate sulla scritta **OK** ed apparirà la finestra di fig.3.

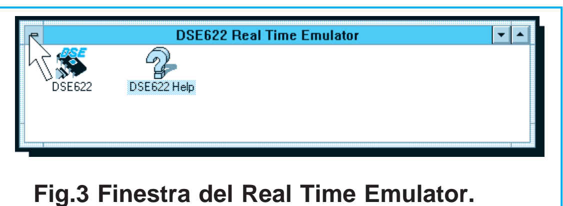


Fig.3 Finestra del Real Time Emulator.

Portate il cursore in alto a sinistra nel quadrettino con il segno **–** e cliccate per far apparire la finestra di fig.4.
Ora portate il cursore sulla scritta **Chiudi** e cliccate in modo che compaia la finestra del:

Program Manager di **Windows 3.1**

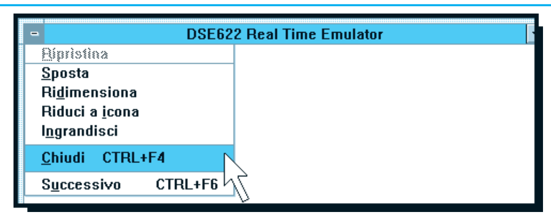


Fig.4 Per chiudere potete usare CTRL+F4.

A questo punto andate con il cursore sul simbolo di **File Manager** (vedi fig.5) e cliccate, quindi andate sul simbolo dell'**unità floppy disk A** (riportato in alto sulla sinistra) e cliccate nuovamente.
Appariranno così sulla destra del video queste tre scritte (vedi fig.6):

atest.asm
btest.asm
setup.exe

Ora dovete trasferire nell'**Hard-Disk** i due soli files **atest.asm** e **btest.asm**, perciò selezionate con il cursore la scritta **atest.asm**, andate sulla scritta **File** posta in alto sulla sinistra e cliccate in modo che appaia la finestra di fig.7.

Fig.5 Icona di File Manager in Windows.

Selezionate il comando **copia** e vedrete apparire la finestra di dialogo di fig.8, in cui dovete specificare dove volete copiare il file **atest.asm**.

Portate il cursore sulla finestra in basso e scrivete:

C:\ST6

poi andate sulla scritta **OK** e cliccate.
Tornerete così alla finestra di fig.6.

Nota: vi abbiamo fatto copiare nella **directory ST6** questo file, perché tutti i precedenti programmi inseriti nel dischetto **DF.1170**, contenenti il **software** di sviluppo dell'**ST6** della **SGS-Thomson**, prevedevano l'installazione in questa directory.

Ripetendo i passaggi appena descritti dovete ora copiare nella directory **ST6** anche il file **btest.asm**.

Al contrario **non dovete** assolutamente copiare il terzo file **setup.exe**, perché questo programma è già stato installato.

Per uscire da **File Manager** cliccate in alto a sinistra su **File** e selezionate la scritta **Esci**.

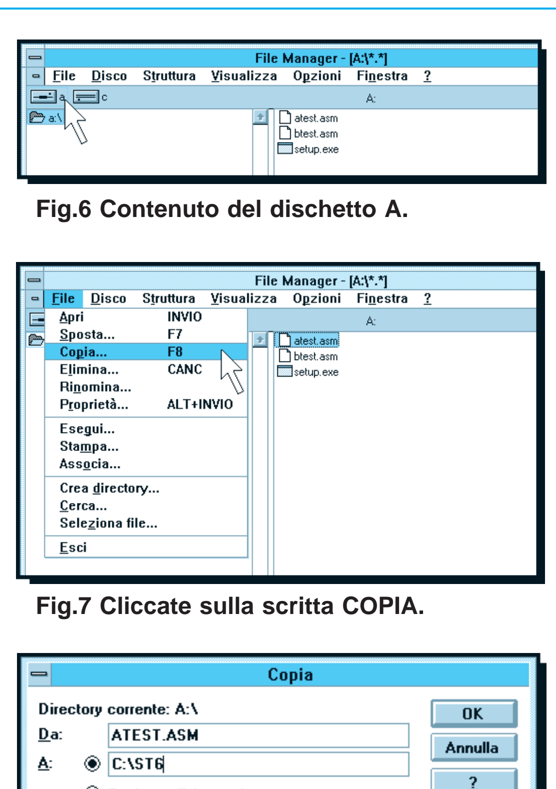


Fig.6 Contenuto del dischetto A.

Fig.7 Cliccate sulla scritta COPIA.

Fig.8 Copiate il file ATEST.ASM in C:\ST6.

Installare il SOFTWARE sotto WINDOWS 95

Se nel vostro computer avete installato **Windows 95**, dopo aver inserito il dischetto con il **software DSE622** nel suo Drive, cliccate sulla scritta **Avvio** posta in basso, quindi andate sulla scritta **Esegui** e cliccate nuovamente.
Quando appare la finestra di dialogo dovete digitare:

A:\setup poi cliccate su **OK**

Il software **DSE622** verrà direttamente installato dal floppy nell'Hard-Disk.

Quando appare la finestra **Name and Company** (vedi fig.9) inserite il vostro nome, quindi andate sulla scritta **Continue** e cliccate.

Ad installazione avvenuta apparirà la finestra di fig.10: ora andate sulla scritta **OK** e cliccate.

Fig.9 Quando appare la finestra dell'installazione del programma, inserite il vostro nome e cliccate su Continue.

Fig.10 Quando l'installazione sarà completata comparirà un messaggio. Per continuare cliccate su OK.

Quando appare la finestra visibile in fig.11, portate il cursore in alto a destra sull'icona con il disegno **X** e cliccate per tornare nella finestra **principale** di **Windows 95** (vedi fig.12).

Fig.11 Per tornare alla finestra principale di Windows 95 cliccate sul simbolo X.

Fig.12 Dalla finestra principale di Windows 95 cliccate su Avvio.

Ora cliccando sulla scritta **Avvio** posta in basso attiverete un sottomenu nel quale dovrete selezionare la scritta **Programmi** e poi **Gestione risorse**. Dopo aver selezionato anche questa scritta apparirà la finestra di dialogo di fig.13.

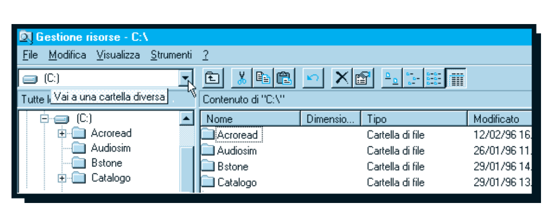


Fig.13 Cliccando sulla scritta Avvio, visibile in fig.12, e seguendo le istruzioni riportate nel testo entrate nella finestra Gestione Risorse di Windows 95.

Cliccate sulla **freccia** che appare nella piccola finestra in alto al cui interno è scritto **C:** per veder

apparire un'altra piccola finestra nella quale dovete selezionare la scritta:

Floppy da 3.5 pollici A:

Dopo aver cliccato (vedi fig.14), apparirà sulla destra il contenuto del dischetto floppy, cioè i tre files:

Atest.asm
Btest.asm
Setup.exe

A questo punto dovete trasferire nell'**Hard-Disk** solo i files **Atest.asm** e **Btest.asm**.
Per prima cosa selezionate il file **Atest.asm**, poi andate sull'icona **Copia** (vedi fig.15) e cliccate.

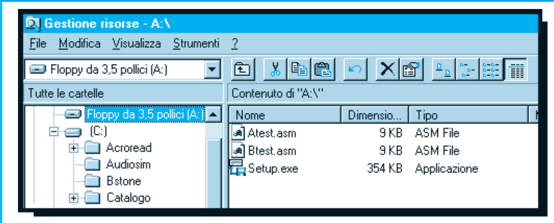


Fig.14 Contenuto del dischetto A.

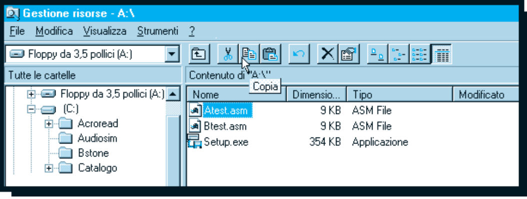


Fig.15 Andate su COPIA e cliccate.

Poiché questo file deve essere copiato nella **directory ST6**, nella finestra a sinistra (vedi fig.16) cercate con il mouse la scritta **ST6**, quindi fermate il cursore su questa riga e cliccate.

Per il momento avete **selezionato** la **directory**, ma il file non è ancora stato trasferito.

Per trasferirlo andate sull'icona **incolla** (vedi fig.17) e cliccate.

Ripetete la procedura visibile in fig.14 per copiare anche il secondo file **Btest.asm**.

Nota: non copiate il file **setup.exe** perché già installato.

Per uscire da questa finestra andate in alto a sinistra su **File**, poi portate il cursore su **Chiudi** oppure pigiate i tasti **ALT+F4**.

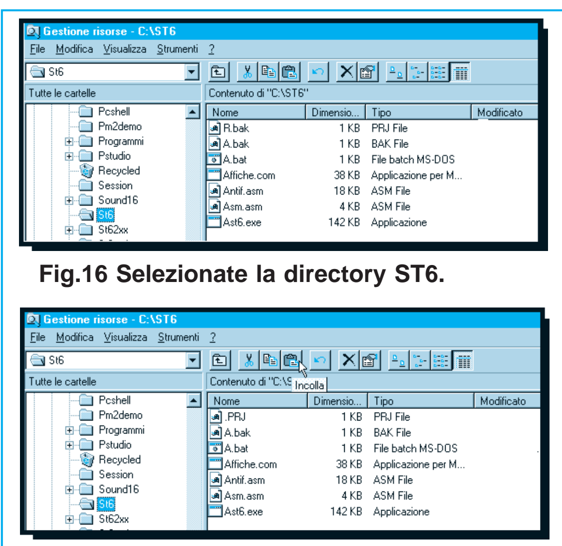


Fig.16 Selezionate la directory ST6.

Fig.17 Copiate il file con l'icona INCOLLA.

I FILES ATEST e BTEST

I programmi che vi abbiamo fatto copiare nella directory **ST6** ci sono serviti per nostri **test** di **simulazione** e ve li proponiamo in modo che possiate imparare ad usare questo **software** di **simulazione**.
Il programma **ATEST** è stato concepito in modo da usare i quattro **piedini PA0 - PA1 - PA2 - PA3** di porta **A** come **ingressi** ed i **piedini PB0 - PB1 - PB2 - PB3** di porta **B** come **uscite**.

Applicando su uno di questi quattro **ingressi** un **livello logico 1**, tramite un interruttore o un microswitch ecc., vorremmo che apparisse sulla **corrispondente uscita** un **livello logico 1** da utilizzare per accendere un **diodo led** oppure per polarizzare la Base di un **transistor** o per eccitare un **relè** o una **sirena**.

Il programma **BTEST** differisce dal precedente solo perché vi abbiamo **inserito** alcuni **errori**, che ci permettono di mostrarvi come il **software** vi aiuti ad individuarli.

COME lavorare con il SOFTWARE DSE622

Come abbiamo già accennato, anche senza la **scheda emulatrice**, che la **SOFTEC** è in grado di fornire ad un prezzo molto competitivo, questo **software** permette di controllare in modo **trasparente** tutte le istruzioni di qualsiasi programma, aiutando così nel loro lavoro tutti i programmatori ed in particolar modo quelli che da poco hanno iniziato a programmare.
Quando si lancia il programma **DSE622**, il software testa se sull'uscita della porta **COM2** è collegata la **scheda emulatrice** della SOFTEC.

Ovviamente se non la trova segnala **"ERRORE"** (vedi fig.18), ma di questo **non dovete** assolutamente preoccuparvi.

Infatti dei tre tasti selezionabili in questa finestra, **Retry - Demo - Parameters**, basterà cliccare sul tasto **Demo** per iniziare a testare il programma.

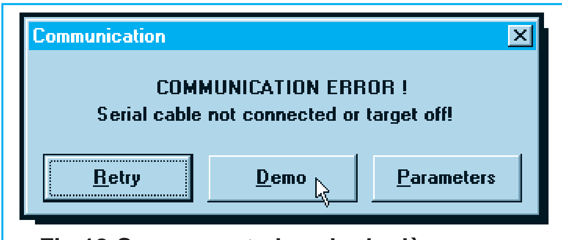


Fig.18 Se non avete la scheda dà errore.

Se cliccate sul tasto **Parameters** apparirà la finestra di fig.19, che, se un domani l'acquisterete, vi permette di indicare al computer su quale **porta seriale** avete collegato la **scheda emulatrice**.

Quando sarete in questa finestra vi consigliamo di cambiare la **velocità** di esecuzione (**Baud Rate**), e, potendo scegliere tra **9.600 - 19.200 - 28.800 - 57.600 - 115.200 Baud**, vi suggeriamo di scegliere la massima velocità, cioè **115.200 Baud**.
Digitate perciò questo numero poi cliccate sulla scritta **OK**.

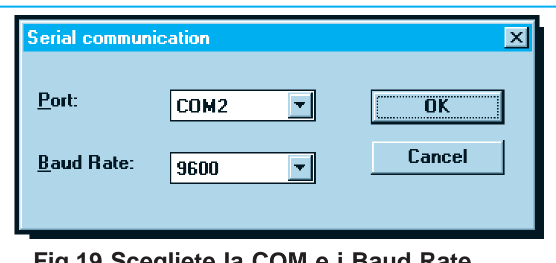


Fig.19 Scegliete la COM e i Baud Rate.

Quando appare la finestra principale del **software** di **simulazione** (vedi fig.20) si possono già iniziare a testare tutti i programmi.

Fig.20 Finestra principale della SOFTEC.

Prima di proseguire riportiamo il significato di alcune parole che sono spesso richiamate nell'articolo.

cliccare - definiamo così l'azione che si effettua premendo il tasto del **Mouse** sulla scritta o icona indicata.

project - chiamiamo con questa parola tutti i files con estensione **.PRJ** che, oltre le caratteristiche del programma, contengono le specifiche definite con il **software simulatore** per testare il programma stesso.

source/file - chiamiamo con questa parola tutti i programmi già assemblati riconoscibili dall'estensione **.HEX**.

simulazione - chiamiamo con questa parola l'esecuzione dei **test dei programmi** con l'ausilio del **software DS622**, senza l'utilizzo dell'**Hardware** dell'**emulatore**.

variabili - chiamiamo con questa parola le definizioni degli indirizzi di memoria **Data Space**.

COME creare la LIBRERIA per l'ST6

Quando sul monitor appare la finestra con i **menu** (vedi fig.20), per creare la **libreria** cliccate sulla scritta **Configure** e vedrete apparire la piccola finestra visibile in fig.21.

- Cliccate sulla riga **Tools** per far apparire la finestra di dialogo visibile in fig.22.

Fig.21 Finestra per creare la libreria.

Fig.22 Finestra di dialogo TOOLS.

- Cliccate sulla scritta **Add** per far apparire la finestra di dialogo di fig.23.

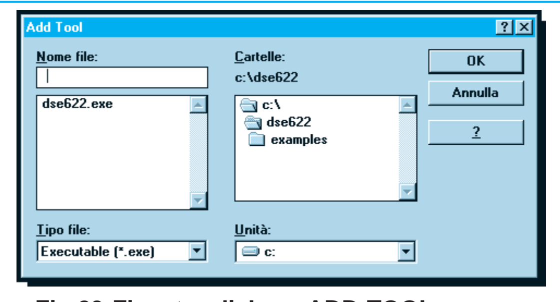


Fig.23 Finestra dialogo ADD TOOL.

- Nel riquadro a destra cliccate sulla riga **C:** e, sempre in questo riquadro, ricercate la **directory ST6**, quindi andate con il cursore su questa riga e cliccate.

- Nel riquadro a sinistra cercate il programma **ST6.EXE** (vedi fig.24) e selezionatelo, poi uscite cliccando su **OK**.
In questo modo nel riquadro **Command Line** (vedi fig.25) apparirà: **C:\ST6\ST6.EXE**

- Nella riga **Menu Title** dovete digitare: **ST6**

- Nella terza riga, **Arguments** (vedi fig.25), dovete digitare: **$2**

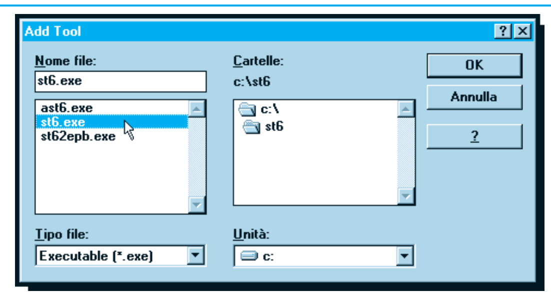


Fig.24 Selezionate il programma ST6.EXE.

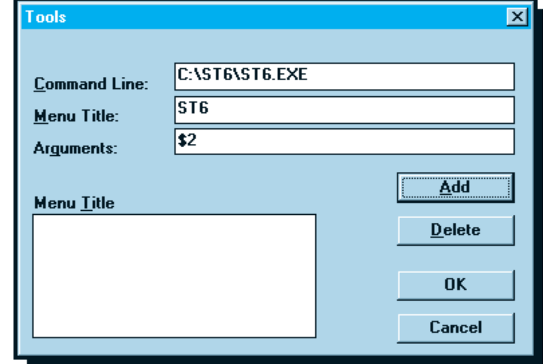


Fig.25 Digitate nelle righe quanto appare.

Definite tutte le specifiche richieste, tornate al menu principale (vedi fig.20) cliccando su **OK**.

In questo modo vi abbiamo fatto inserire nella libreria il programma **ST6\ST6.EXE**, che già da tempo vi abbiamo fornito. Con questo programma potrete assemblare, scrivere, correggere, ricercare un programma, come vi abbiamo spiegato nelle lezioni precedenti (vedi riviste N.172/173 - 174 - 175/176).

COMPILARE in ASSEMBLER il programma ATEST.ASM

Quando sul monitor appare la finestra con i **menu** (vedi fig.20) selezionate la riga **Tools**.
Nella piccola finestra che appare (vedi fig.26) cliccate sulla scritta **ST6** e così apparirà la finestra dei programmi di **sviluppo** dell'**ST6**, visibile in fig.27.

Per aprire la finestra dei files pigiate il tasto **F3** oppure andate sulla scritta **OPEN** e cliccate (vedi fig.28).
Ora cercate il programma:

atest.asm

e quando l'avete trovato cliccate sulla scritta, poi andate su **OPEN** e cliccate nuovamente.
Appariranno così sul monitor tutte le **istruzioni** di questo programma (vedi fig.29).

A questo punto portate il cursore sulla scritta **ST6** (prima riga in alto) e cliccate.
Sotto questa scritta si aprirà una finestra (vedi fig.30) con il cursore già posizionato sulla parola **Assembla** quindi cliccate.

Durante la **compilazione** in **assembler** verranno creati questi files:

ATEST.HEX
ATEST.SYM
ATEST.DSD

Per chi ancora non conoscesse il significato di queste **estensioni**, lo accenniamo qui brevemente:

.HEX - programma eseguibile in formato **Intel-Hex.**

.SYM - file contenente le definizioni delle **etichette** di **salto** ed il relativo indirizzo di memoria **Program Space**

.DSD - file contenente le definizioni delle **variabili**, le loro caratteristiche ed il relativo indirizzo di memoria **Data Space**.

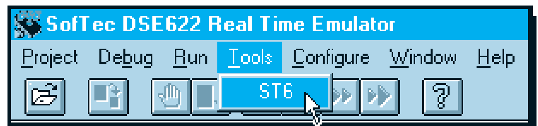


Fig.26 Cliccate sul sottomenu ST6 di Tools.

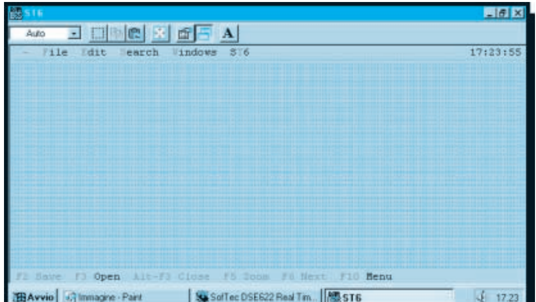


Fig.27 Finestra di sviluppo per micro ST6.

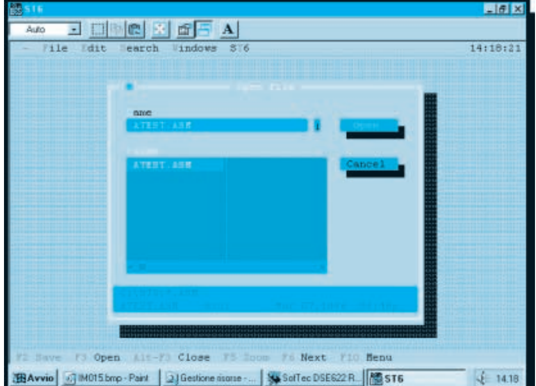


Fig.28 Aprite un programma pigiando F3.

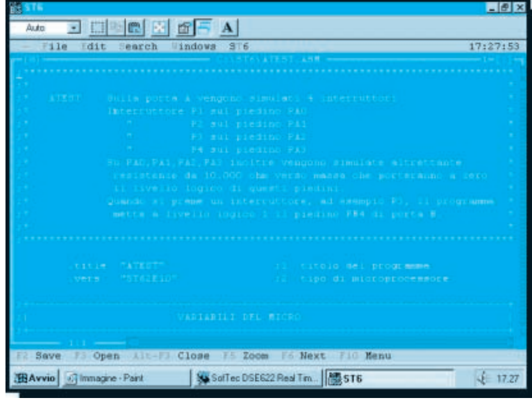


Fig.29 Vedrete sul monitor le istruzioni.

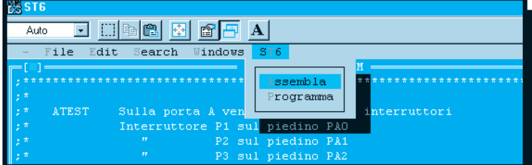


Fig.30 Cliccate sul sottomenu Assembla.

Completata la **compilazione** potete premere un qualsiasi tasto per rientrare nell'**Editor** dell'**ST6**, e quando apparirà la finestra di fig.29 cliccate sulla scritta **ALT-F3** oppure premete i tasti **ALT+F3**.
Apparirà un'altra finestra (vedi fig.27) in cui dovrete digitare **Alt X** per uscire dal programma **ST6** e rientrare automaticamente nel **software DS622**.

Quando appariranno i **menu** del **DS622** selezionate il menu **Project** e, nella finestra che appare visibile in fig.31, andate sulla scritta **New Project** e cliccate.

Questa operazione serve per creare il file con estensione **.PRJ**, che verrà utilizzato dal **simulatore** per testare il programma.

Quando appare la finestra di fig.32, digitate nella riga **Project name** il nome del file **ATEST** (non è necessario riportare dopo il nome l'estensione **.PRJ**).
Tenete presente che potete anche cambiare nome al file **project**, cioè dargli un nome differente dal **source file**.
In altre parole potrete ad esempio cambiare il nome **ATEST** in **BAUBAU** o **PLUTO**, ma se volete evitare che un domani non vi ricordiate più quale nome avevate scelto, vi consigliamo di mantenere lo stesso nome del programma assemblato, cioè nel nostro caso **ATEST**.
Dopo aver digitato il **nome** portate il cursore sulla parola **Create** (vedi fig.33) e cliccate.

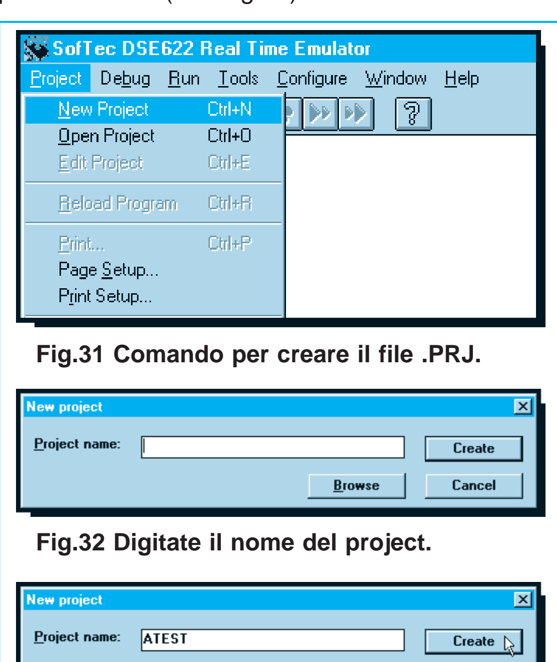


Fig.31 Comando per creare il file .PRJ.

Fig.32 Digitate il nome del project.

Fig.33 Scegliete Create e vedrete la fig.34.

Apparirà la finestra di dialogo **Edit Project** (vedi fig.34) in cui è molto importante inserire le specifiche richieste **senza** commettere **errori**.

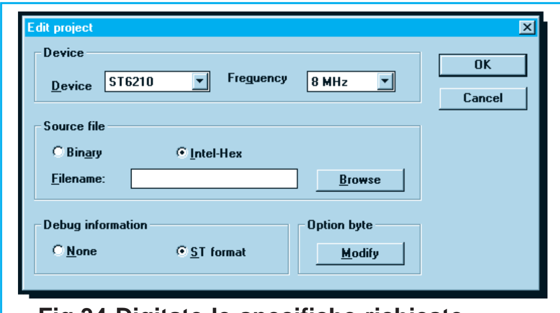


Fig.34 Digitate le specifiche richieste.

Portate il cursore nella finestra **Device** e cliccando la “freccia in giù” cercate la **sigla** del micro **ST6** che volete utilizzare. Ammesso che questo sia un **ST6210** andate sulla riga **ST6210** (vedi fig.35) e cliccate.

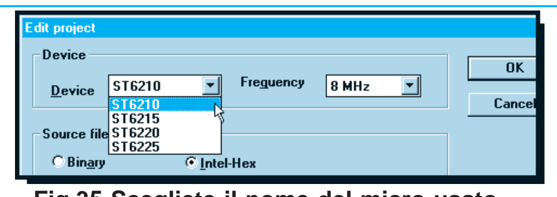


Fig.35 Scegliete il nome del micro usato.

Ora portate il cursore nella finestra **Frequency** (vedi fig.36) dove potete selezionare la **frequenza** del **quarzo** utilizzato per il **clock** del micro scegliendo tra **8 - 4 - 2 - 1 MHz**.
Nota: Se non possedete la **scheda emulatrice** della **Softec** scegliete a caso una di queste quattro frequenze.

Per completare i dati da inserire in questa finestra cliccate nel cerchietto accanto alla scritta **Intel-Hex** (vedi fig.37) cosicché apparirà un **punto**.

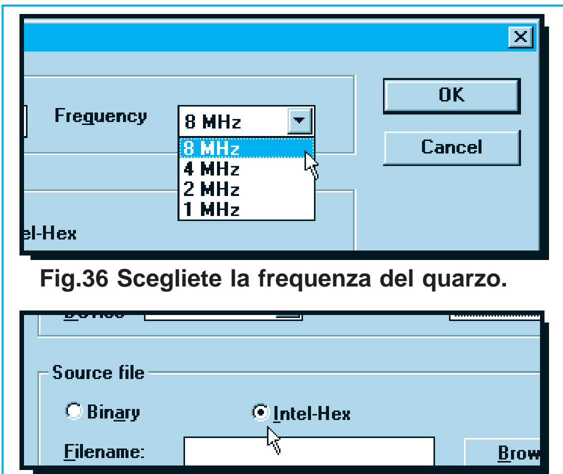


Fig.36 Scegliete la frequenza del quarzo.

Fig.37 Con Intel-Hex vedrete i file .EXE.

In questo modo indicate al programma di visualizzare i soli files con estensione **.HEX**.

A questo punto cliccate sulla scritta **Browse** in modo che appaia la finestra di dialogo di fig.38.

Nel riquadro posto a destra cliccate sulla riga **C:**, poi cercate la scritta **C:\ST6** e quando l’avete trovata selezionatela cliccando.
Nel riquadro a sinistra appariranno tutti i files con estensione **.HEX**.

Cercate tra questi la scritta **ATEST.HEX** (vedi fig.39), cliccate su questa riga, poi uscite cliccando su **OK**. In questo modo nel riquadro **Filename** (vedi fig.40) apparirà la scritta:

C:\ST6\ATEST. HEX

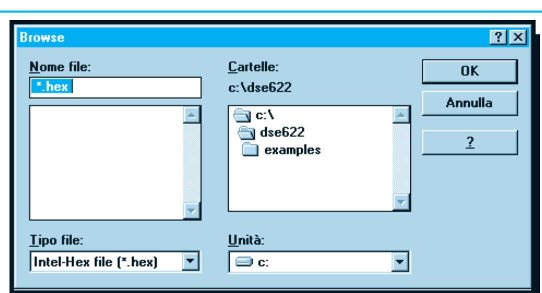


Fig.38 Finestra di dialogo Browse.

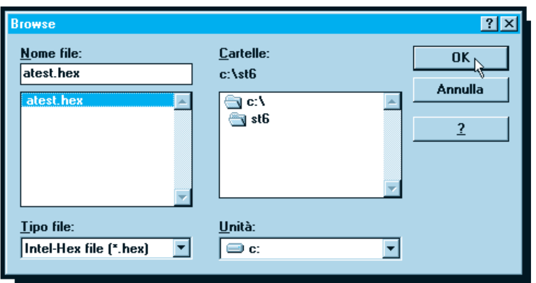


Fig.39 Quando siete nella finestra Browse, selezionate il file ATEST.EXE.

Fig.40 Nel riga Filename apparirà il nome del file selezionato nella finestra Browse.

Ora potete passare al **Debug Information** (vedi fig.34) e se all’interno del **cerchietto** posto a sinistra della scritta **ST format** trovate già un **punto** (vedi fig.42) non dovrete cliccare.

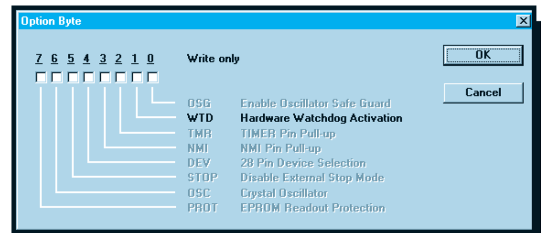


Fig.41 Se siete in simulazione, il WTD, Hardware Watchdog Activazion, non serve.

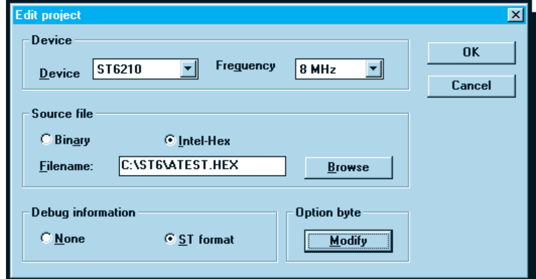


Fig.42 Per creare un file con estensione .PRJ, cliccate sulla scritta OK.

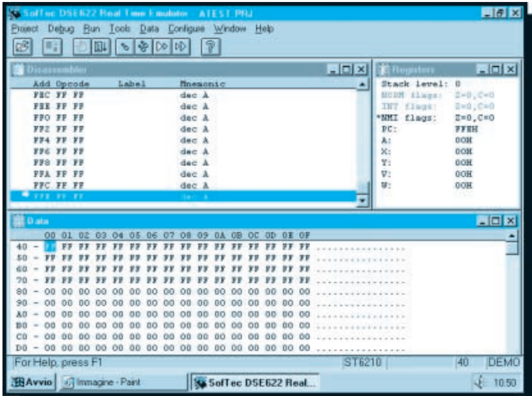


Fig.43 Nella videata principale del DSE622 vengono attivate tre finestre.

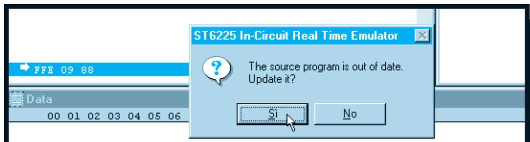


Fig.44 Dalla finestra di fig.27 si passa con ALT+X a questa finestra. Per andare nella finestra di fig.45 cliccate su SI.

Se per errore cliccate all'interno del **cerchietto** posto a sinistra della scritta **None** facendo apparire un **punto**, durante l'esecuzione del programma **non riuscirete** a vedere i **nomi delle variabili** o delle **etichette utilizzate**, quindi vi sarà molto più difficile controllare **passo** per **passo** il programma.

Ora cliccate sulla scritta **Modify** per far apparire la finestra di fig.41 che ha in evidenza questa scritta:

WTD = Hardware Watchdog Activazion

Siccome ci troviamo in **simulazione**, questa selezione non serve perciò portate il cursore su **OK** e cliccate per far apparire la finestra di fig.42.

A questo punto cliccando sulla scritta **OK** verrà creato il **Project ATEST.PRJ** nella directory **DES622** e sarete pronti per testare il programma (vedi fig.45).

Se sul video appare invece una di queste scritte:

source file not found
symbol table file not found
debugger file not found

potreste aver involontariamente **cancellato** dei files di compilazione. Per rigenerarli dovete cliccare su **OK** per far riapparire la finestra di fig.43.

Ora dovete cliccare nuovamente sulla scritta **Tools** e ripetere tutte le operazioni visualizzate dalla fig.20 alla fig.29 e vi ritroverete nella finestra visibile in fig.27.
Per uscire digitate **Alt+X** e sul video vedrete apparire la finestra di fig.44.
Cliccando su **SI** potrete vedere la finestra di fig.45 con tutti i dati e le istruzioni corrette.

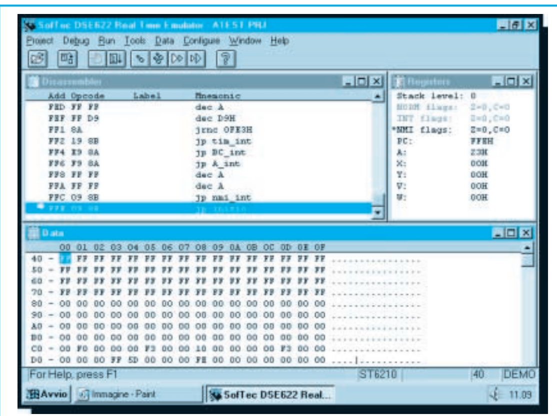


Fig.45 La finestra per testare il programma appare sullo schermo così suddivisa.

Se invece appare la scritta:

C:\ST6\ATEST.HEX emulation buffer overflow

significa che avete scritto il programma per un tipo di micro **ST6** diverso da quello selezionato nella riga del **Device** (vedi fig.35).
Ad esempio potreste aver scritto un programma per il micro **ST62/10** ed aver inserito nel **Device** il micro **ST62/25** o viceversa.
In presenza di questo errore cliccate su **OK** e, quando appare la finestra di fig.43, selezionate il menu **Project** e cliccate su **Edit project** (vedi fig.46).
Vi ritroverete nella finestra di fig.35 dove potrete correggere la scritta corrispondente alla riga **Device** digitando la sigla corretta del micro utilizzato.
Per uscire cliccate su **OK**.

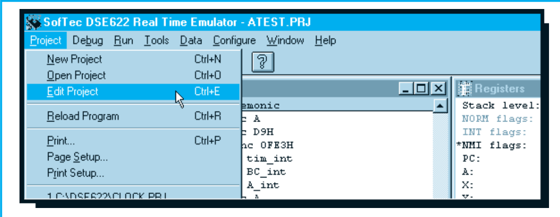


Fig.46 Finestra per correggere la fig.35.

Quando apparirà la finestra di fig.45 sarete pronti per effettuare la **simulazione** del programma **ATEST.HEX** utilizzando il file **Projet ATEST.PRJ**.

Prima di proseguire è necessario che vi spieghiamo cosa contengono le **3 finestre** visibili in fig.45.

Nella finestra **Disassembler**, sotto le due colonne **Label** e **Mnemonic**, avete le istruzioni del programma da **testare** in formato leggibile.
Nelle colonne **Add** e **Opcode** appaiono le medesime istruzioni in formato **Intel.Hex**.

Nella finestra **Register** appaiono tutti i **registri**, lo **stack level**, gli stati dei tre **flags** ed il valore del **Program Counter**.

Nella finestra **Data** appare il contenuto del **Data Memory Space** del micro, cioè il contenuto delle **variabili**, dei **registri** e della **Data Rom Windows** definiti in questo programma.

Avendo sottocchio tutte queste finestre sarete in grado di controllare **passo x passo** in **simulazione** tutti i vostri programmi.
Prima di iniziare il controllo, è a nostro parere necessario configurare ulteriormente il **software DS622** così da vedere sul monitor in **tempo reale** anche altre funzioni che potrebbero risultarvi molto utili.

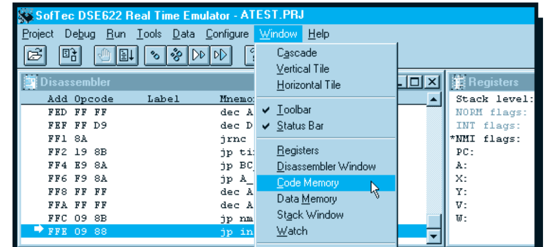


Fig.47 Aprite la finestra Code-Memory.

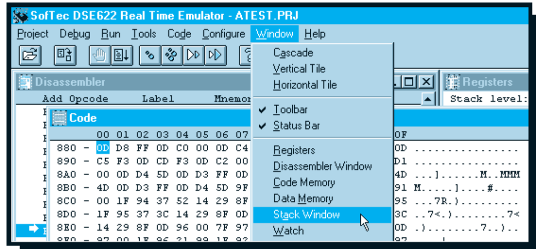


Fig.48 Aprite la finestra Stack-Window.

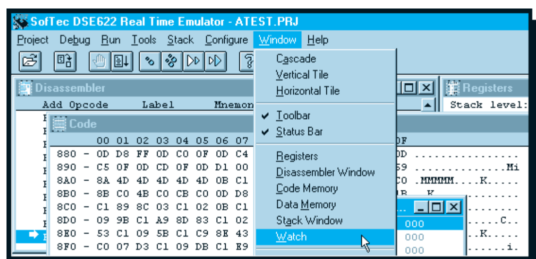


Fig.49 Aprite la finestra Watch.

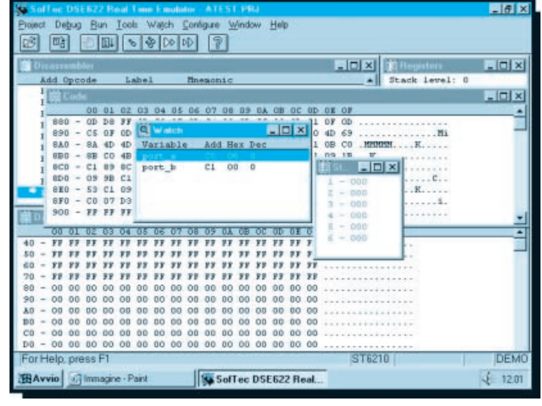


Fig.50 Sul video appaiono altre finestre.

Perciò dopo aver portato il cursore sulla scritta **Window** posta in alto (vedi fig.47) cliccate e nella finestra che appare selezionate la riga **Code Memory**.

Cliccate nuovamente sulla scritta **Window** e selezionate la scritta **Stack Windows** (vedi fig.48).

Ritornate nuovamente a cliccare sulla scritta **Windows** e selezionate la riga **Watch** (vedi fig.49).
Il sottomenu di Window sparirà.

In questo modo alle finestre che già apparivano nella fig.45 si aggiungono **3 supplementari finestre** operative (vedi fig.50), e cioè:

Stack
Code Memory
Watch

Vi consigliamo di rimpicciolire e spostare le finestre (vedi l'esempio riportato in fig.51) in modo da avere sempre sottocchio tutte le specifiche relative alla programmazione di qualsiasi micro **ST6**.

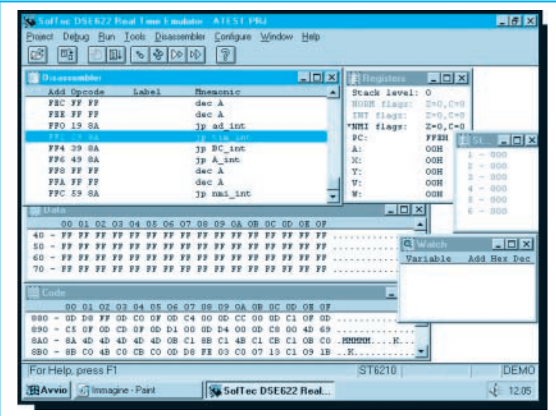


Fig.51 Vi suggeriamo di rimpicciolire e spostare tutte le finestre come visibile in questa figura.

Dopo aver posizionato le finestre nei punti che ritenete più opportuni, tutte le volte che richiamerete il file **ATEST.PRJ** o altri files con la stessa estensione, le finestre riappariranno dove le avevate posizionate.

La finestra **Stack** permette di visualizzare in **tempo reale** il valore dei suoi **6 registri**.

Se nel corso del programma viene eseguita l'istruzione **Call** (vedi rivista **N.174**) o viene attivato un **Interrupt** (vedi rivista **N.175/176**), il contenuto dei primi **5 registri** di **Stack** viene immediatamente **shiftato** di un livello **superiore** , vale a dire che il contenuto del 5° registro viene passato al 6°, il contenuto del 4° registro viene passato al 5° e così via.

A questo punto nel **1° registro** di **Stack** viene memorizzato il contenuto del **Program Counter**, cioè l'indirizzo di ritorno della subroutine richiamata nella **Call** (istruzione **Ret**) o nell'**Interrupt** (istruzione **Reti**), come appare visibile in fig.89.

In questo modo è possibile eseguire più **subroutine**, una all'interno dell'altra, fino ad un massimo di **6**.

La finestra **Code Memory** permette di visualizzare in **esadecimale** il contenuto della **Program Memory Space**, cioè la memoria del micro in cui sono contenute le istruzioni del programma sotto **test**.

La finestra **Watch**, che inizialmente è **vuota**, serve per inserire, come poi vi spiegheremo, le **variabili** delle quali ci interessa controllare il contenuto per vedere come questo si modifica durante l'esecuzione del programma.

Se di queste tre finestre volete che ne appaia **una sola** o **due**, dovete annullare una delle operazioni riportate nelle figg.47-48-49.

ESEMPI di EMULAZIONE

Le prime volte che userete il **software simulatore DSE622** vi consigliamo di **stampare** il listato del programma che volete **testare** (nel nostro esempio **ATEST.ASM**) per poter confrontare le istruzioni stampate con quelle che appariranno sul monitor.
Se avete spento e riacceso il computer, per richiamare il programma **ATEST** già creato come file **project**, che ora si chiamerà quindi **ATEST.PRJ**, dovrete cliccare sull'icona **apri file** (vedi fig.52).

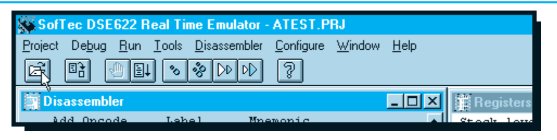


Fig.52 Cliccando sull'icona Apri File potrete aprire il file ATEST.PRJ.

Nella finestra di dialogo che appare, sotto il **Nome file** ci sarà la scritta ***.PRJ** e nella finestra sottostante il solo file **ATEST.PRJ**, perché per ora è stato creato un solo **project**.
Man mano che creerete files con estensione **.PRJ** troverete tutti i loro nomi in questa finestra.

Cliccate sul nome del file desiderato quindi uscite da questa finestra cliccando su **OK**. Apparirà la finestra visibile in fig. 51.

Sull'ultima riga visibile della finestra **Disassembler** (vedi fig.53) potete vedere una riga blu evidenziata da una **freccia rossa**.

Nota: la **freccia rossa** mette in evidenza la prima istruzione che il programma **eseguirà**.

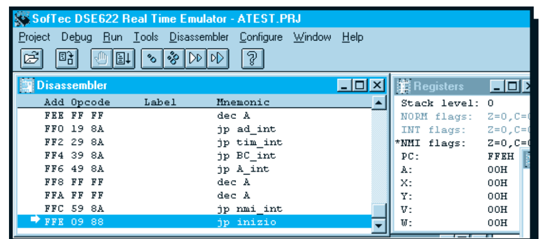


Fig.53 La freccia mostra da dove si parte.

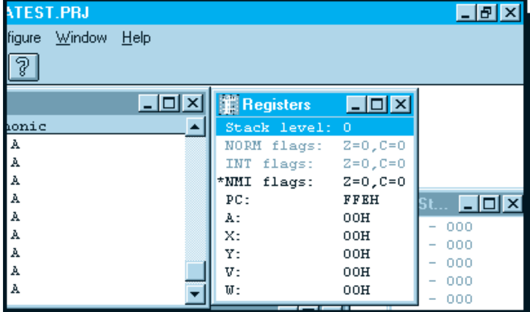


Fig.54 Finestra Registers.

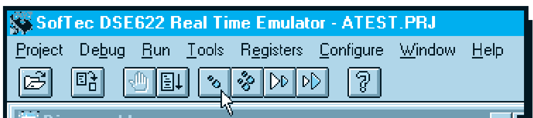


Fig.55 Icona per avanzare passo/passo.

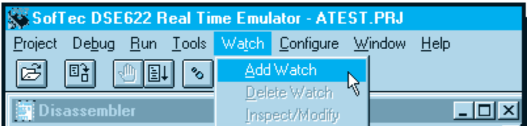


Fig.56 Aprite la fig.57 con Add Watch.

Fig.57 Per le variabili cliccate sulla freccia.

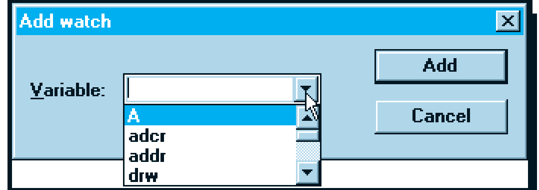


Fig.58 Selezionate la variabile che volete.

Le righe sopra quella evidenziata in blu sono quelle che abbiamo definito nel programma **ATEST** per i **vettori di interrupts**.

Nella finestra **Register** di fig.54 appare lo **Stack level**, i tre **Flags NORM - INT - NMI** tutti a **livello logico 0**, ed il **PC** (Program Counter) posizionato sul valore esadecimale **FFEH** (come già sapete la lettera **H** significa esadecimale), che corrisponde all'indirizzo della prima istruzione eseguibile (vedi la finestra **Disassembler** di fig.53).

Per far partire il programma in modo che ogni istruzione avanzi **passo x passo** dovete portare il cursore sull'icona di fig.55 e cliccare.

Ogni volta che cliccherete su questa icona il programma eseguirà una **sola istruzione** e questo vi consentirà di controllarlo riga per riga.

Come vi spiegheremo tra poco, è possibile eseguire anche più istruzioni per ogni **passo** oppure lanciare un'esecuzione in **automatico**.

La **prima** istruzione che il programma **ATEST** esegue è quella riportata all'indirizzo **FFEH**, come appare in fig.53.

INSERIRE una VARIABILE nella finestra WATCH

Poiché potrebbe risultare utile controllare lo **stato** dei piedini della **porta A** di entrata (input) e quello dei piedini della **porta B** di uscita (output), scrivendo nella finestra **Watch** le definizioni di queste **variabili** potrete averle sempre sottocchio.
In questo modo sarà più facile vedere come cambiano man mano che fate avanzare il programma.

Per visualizzare le due porte nella finestra **Watch** dovete eseguire queste semplici operazioni:

- Andate con il cursore nella finestra **Watch** e cliccate per **evidenziare** questa finestra.

- Come potete notare, nella barra dei menu posta in alto, tra le scritte **Tools** e **Configure**, appare la scritta **Watch** che vi permette di accedere ad un sottomenu dedicato a questa finestra.

Tra le scritte **Tools** e **Configure** appare infatti di volta in volta il menu relativo alla finestra posta in **primo piano**.

- Cliccate sulla scritta **Watch** e selezionate la scritta **Add Watch** (vedi fig.56) in modo da far apparire la finestra di dialogo visibile in fig.57.

- Andate con il cursore sulla **freccia** posta a destra della scritta **Variable** e vedrete apparire in **ordine alfabetico** l'elenco delle **variabili** presenti nel programma **ATEST** (vedi fig.58).

Cliccate su **port_a**, poi andate su **ADD** e qui cliccate. In questo modo nella finestra **Watch** apparirà la **variabile port_a** con l'indirizzo **ADD** di **Data Space**, il valore **esadecimale - Hex** ed il valore **decimale - Dec** (vedi fig.59).

Poiché ci interessa controllare anche i valori della **porta B**, dovrete ripetere tutte le operazione riportate nelle figg.56-57-58, quindi selezionare la variabile **port_b**.
Cliccando poi su **ADD** nella finestra **Watch** appariranno i dati della **port_b** (vedi fig.60).

COME INSERIRE un BREAKPOINT

Un'altra operazione che dovete imparare è come **attivare** un **breakpoint**.
Attivare un **breakpoint** significa mettere un **punto di blocco** ad un'istruzione del programma in modo che durante la simulazione si fermi su quella riga.

Ammesso che vogliate inserire un **breakpoint** sull'ultima riga **in basso** in cui è riportata l'istruzione (vedi fig.61):

898 0D D4 00 ldi tscr,00H

dovrete portare il cursore su questa riga e **cliccare due volte**. Vedrete così apparire sul monitor la finestra di comando visibile in fig.62.
Cliccate sulla scritta **Toggle Breakpoint** e a sinistra della **riga selezionata** comparirà un **punto esclamativo** che vi segnala che su quella riga è stato attivato il **breakpoint** (vedi fig.63).

COME usare il BREAKPOINT

Come già vi abbiamo accennato, il **breakpoint** serve per fermare il programma sulla riga di istruzione che avete **marcato** ogni volta che verrà lanciata una simulazione in modo **automatico** o a **passi multipli**.
Potete mettere quanti **breakpoint** volete, cioè **3 - 5 - 14 - 20 ecc.**
È sottinteso che quando il programma si sarà fermato su un **breakpoint** per proseguire dovrete nuovamente cliccare sull'icona **passo** per **passo** o a **passi multipli** oppure sull'esecuzione **automatica**.
Per togliere il **breakpoint** (ma ora **non toglietelo** perché ve lo faremo usare per fare un po' di pratica) dovrete andare sulla riga interessata e cliccare **due volte**, poi nella finestra che appare dovre-

Fig.59 Esempio della Variabile port_a.

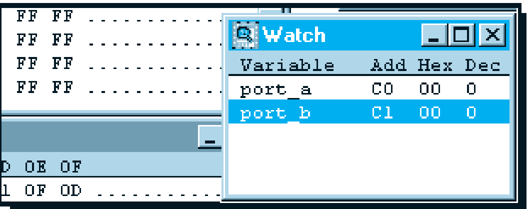


Fig.60 Esempio della Variabile port_b.

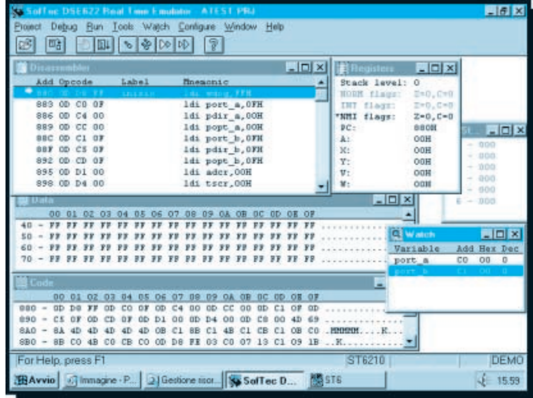


Fig.61 Finestre per testare il programma.

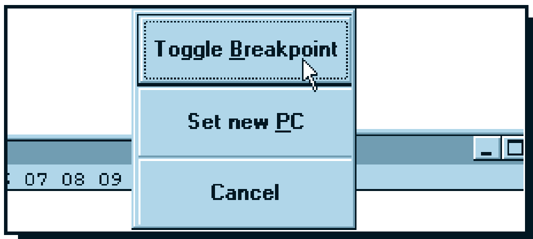


Fig.62 Per attivare un Breakpoint.

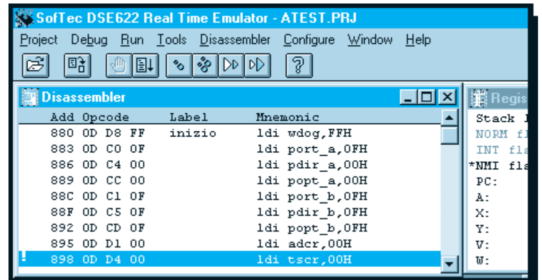


Fig.63 Dove c'è ! è attivato un Breakpoint.

te andare sulla scritta **Toggle Breakpoint** (vedi fig.62) e cliccare.
in questo modo il breakpoint verrà eliminato dalla riga in cui in precedenza era stato inserito.

Se per errore andate sulla scritta **Cancell** e cliccate **non toglierete** il breakpoint, ma farete solo sparire la finestra di fig.62.

ESECUZIONE in AUTOMATICO

Per far avanzare in modo **automatico** le istruzioni senza cliccare tutte le volte il tasto del **passo-passo** riportato in fig.55, dovete portare il cursore sull'icona visibile in fig.64, cioè sull'immagine di una pagina con una freccia rivolta verso il basso.

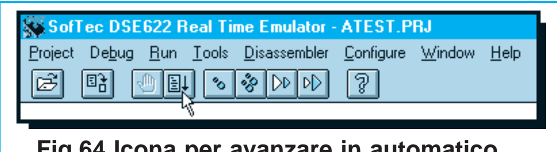


Fig.64 Icona per avanzare in automatico.

Cliccando su questa icona ottenete un'esecuzione sequenziale ed automatica di tutte le istruzioni, che si fermerà solo sull'istruzione in cui avete inserito il **Breakpoint**.

Quando viene eseguita l'istruzione:

883 0D C0 0F   ldi port_a,0FH

se guardate nella finestra **Watch** al contenuto della **port_a** sotto la colonna **Dec.** trovate il numero **15**.

Quando viene eseguita l'istruzione:

88C 0D C1 0F   ldi port_b,0FH

se guardate nella finestra **Watch** al contenuto della **port_b** sotto la colonna **Dec.** trovate il numero **15**.
Infatti sempre nella finestra **Watch** al numero esadecimale **0F** corrisponde il numero **decimale 15**.
Se non sapete ancora convertire un numero **decimale** in un numero **binario** vi consigliamo di andare a **pag.381** del nostro **Handbook** (se ne siete sprovvisti potete richiedercelo) dove troverete:

0 0 0 0 - 1 1 1 1

Per avere una riprova **visiva** di queste condizioni logiche dovete andare nella finestra **Data**.
Questa finestra riporta nella prima colonna gli indirizzi di memoria e nel righello in alto in grigio i valori **esadecimali**:

00-01-02-03-04 —— 09-0A-0B-0C-0D-0E-0F

Poiché l'indirizzo di **port_b** è **C1**, nella prima colonna dovrete cercare l'indirizzo di memoria **C0**, poi, prendendo come riferimento il righello in grigio, scendete dal valore esadecimale **01** fino ad incontrare la riga **C0**, come si farebbe con una **Tavola Pitagorica**, e così troverete il valore **0F**.

Portate il cursore su **0F** poi cliccate **2 volte** e vedrete apparire la finestra di dialogo **Edit data** riportata in fig.65.
Cliccando sulla scritta **Bits** apparirà sullo schermo la finestra di dialogo di **Port B Data Register** di fig.66.

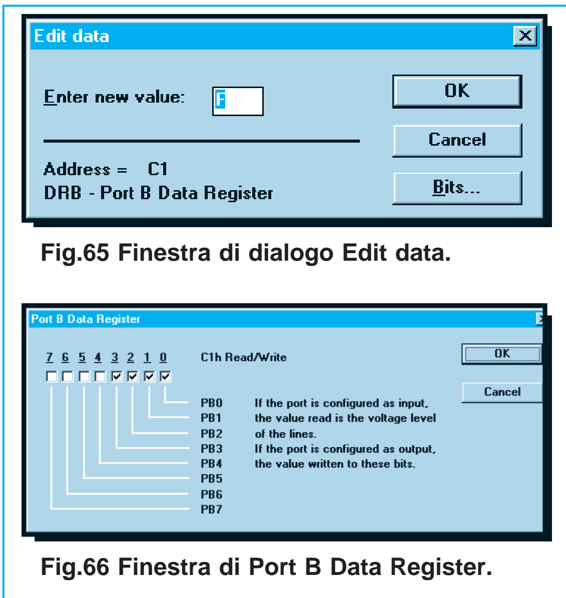


Fig.65 Finestra di dialogo Edit data.

Fig.66 Finestra di Port B Data Register.

In **orizzontale** potete leggere i numeri **7 - 6 - 5 - 4 - 3 - 2 - 1 - 0**, che corrispondono ai piedini della porta B. Sotto questi numeri ci sono delle **caselle** che possono contenere una **V** oppure risultare **vuote**.
Nel nostro caso la **V** è presente solo sulle prime quattro caselle **3 - 2 - 1 - 0**, mentre nelle altre quattro caselle **7 - 6 - 5 - 4** non appare nulla.

Le caselle con la **V** sono quelle che si trovano a **livello logico 1**, cioè sui piedini corrispondenti risulta presente una tensione **positiva**, ed ovviamente quelle **senza** la V sono a **livello logico 0**.
Come già saprete il numero **decimale 15** corrisponde a **0 0 0 0 - 1 1 1 1** in **binario**.

Nella colonna **verticale** troverete gli **otto** piedini della porta **B** siglati **PB0 - PB1 - PB2 - PB3 - PB4 - PB5 - PB6 - PB7**.
Da questa finestra di dialogo potete dunque sapere quale **condizione logica** è presente sugli **otto piedini**.

Nel nostro esempio:

PB7 - PB6 - PB5 - PB4 = livello logico 0
PB3 - PB2 - PB1 - PB0 = livello logico 1

Dopo questa verifica potete cliccare su **OK**, poi cliccate ancora su **OK** nella successiva finestra e tornerete nella finestra **Data** di fig.67.

IMPORTANTE

Durante l'esecuzione del programma tutte le **quattro uscite** si portano a **livello logico 0** fino a quando uno dei **quattro ingressi** non viene collocato a **livello logico 1**.
Potendo vedere nella finestra di fig.66 le **condizioni logiche** presenti sulle **porte**, vi accorgerete subito se nel programma è stato commesso un **errore**.
Ammesso infatti che nel **piedino PB6** di porta B debba risultare presente un **livello logico 1** e non un **livello logico 0** e nel **piedino PB0** i porta B un **livello logico 0** e non un **livello logico 1**, potrete subito vedere la **situazione** sui piedini della porta.
La finestra di fig.66 non solo vi permette di vedere i **livelli logici** sui piedini di **port_b**, ma anche di correggerli. Infatti se, ad esempio, provate a cliccare nella **casella 6** comparirà una **V** che vi indica che questo piedino è passato a **livello logico 1**.

Poiché il programma **ATEST** non contiene **errori** non cambiate i **livelli logici** su **PB0 - PB1 - PB2 - PB4** e se lo fate, rimettete quelli che apparivano in precedenza, diversamente andrete a modificare il corretto proseguimento del **test.**

Allo stesso modo potrete controllare il **livello logico** del registro di controllo dell'**AD/Converter** (nel programma di **ATEST** l'AD/Converter non viene usato).
Andate nella finestra **Data** (vedi fig.67) e cercate negli indirizzi di memoria il valore esadecimale **D0**.

Poiché l'indirizzo di questo registro è **D1**, prendendo come riferimento il righello in grigio, poi scendete dal valore esadecimale **01** fino ad incontrare la riga **D0** e troverete il valore **00**.

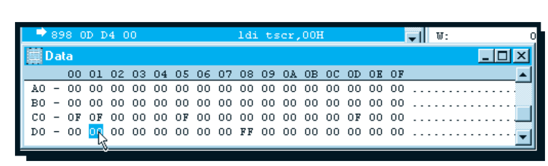


Fig.67 Per attivare la finestra di dialogo Edit Data dovete cliccare due volte su 00.

Ponendo il cursore su **00** e cliccando **due volte** vedrete apparire la finestra dell'**Edit data** di fig.68, ovviamente diversa da quella di fig.65.
Cliccando sulla scritta **Bits** apparirà la finestra di fig.69, leggermente diversa da quella di fig.66.

Fig.68 Finestra di dialogo Edit data.

Fig.69 Dalla finestra di fig.68 cliccate sulla scritta Bits per aprire la finestra dell'A/D Control Register.

Dopo avervi spiegato come sia possibile controllare visivamente i **livelli logici** presenti sui **piedini** del micro, potrete divertirvi a vedere il contenuto delle **variabili** e dei **registri** che non vi abbiamo citato. In questo modo farete un po' di pratica che vi servirà in futuro per scrivere correttamente i vostri programmi.

ESECUZIONE a PASSI MULTIPLI

Per la funzione **passi multipli** sarebbe consigliabile togliere tutti i **breakpoints**, in ogni caso anche se li lascerete potrete ugualmente eseguire questa funzione.
L'esecuzione a **passi multipli** vi dà la possibilità di eseguire in modo **automatico** un numero di istruzioni che voi stessi potrete definire.

Ad esempio, potrebbe risultarvi comodo far eseguire al programma **5 - 8 - 10 istruzioni** di seguito prima i fermarvi per **controllare** i dati**.**
Stabilito il numero di istruzioni da eseguire di seguito, tutte le volte che cliccherete sull'icona di fig.70 verrà eseguito il numero di istruzioni che avete prefissato.

Attualmente il **software** è predefinito per fare **passi** di **2 sole istruzioni**, quindi ammesso che desideriate fare dei passi di **5 istruzioni** dovrete procedere come segue.

- Portate il cursore sulla scritta **Run** visibile in fig.71 e cliccate e nella finestra che appare andate sulla scritta **Multiple Step Value** e cliccate.

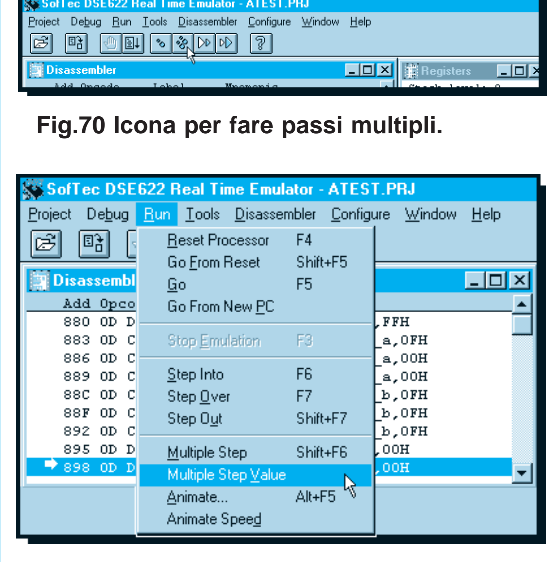


Fig.70 Icona per fare passi multipli.

Fig.71 Finestra per variare i passi/multipli.

- Nella finestra di dialogo che appare (vedi fig.72) andate nella casella **Value** e sostituite il numero **2** con il **5**, poi andate su **OK** e cliccate.

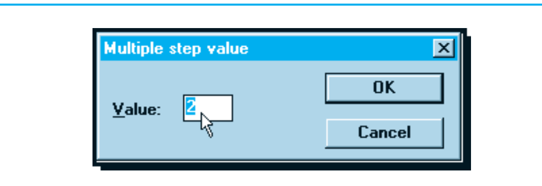


Fig.72 Digitate il numero dei passi multipli.

- Tutte le volte che cliccherete sull'icona in cui sono disegnate due orme (vedi fig.73), il programma avanzerà di **5 istruzioni**.
Cliccando nuovamente sull'icona, il programma avanzerà di altre **5 istruzioni**.

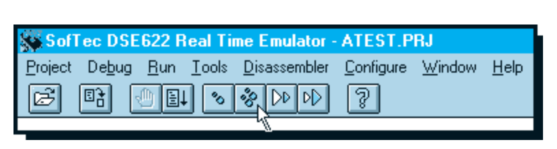


Fig.73 Icona per i passi multipli.

FUNZIONE DEBUG

La funzione **debug** è molto utile per vedere la **mappa** del micro utilizzato e su quali piedini sono posizionate le **porte A - B - C**, inoltre potete vedere quali **livelli logici** sono presenti sui piedini **d'ingresso** o di **uscita** durante l'esecuzione del programma.

Per entrare nella funzione **debug** cliccate sulla scritta **Debug**, visibile nella fascia superiore del menu, e quando appare la maschera di fig.74 cliccate nella riga **Test I/O**.
Apparirà così la finestra di fig.75.
In alto potete leggere la sigla dell'integrato prescelto, nel nostro caso **ST6210 - ST6220**, in basso a sinistra potete vedere le connessioni del micro e su quali **piedini** sono posizionate le **porte A** e **B**, infine sul lato destro è visibile la **mappa** di configurazione logica di queste due porte.

Per la **porta A**, che dispone di soli 4 piedini **PA0 - PA1 - PA2 - PA3**, troverete a destra **4** caselle **grigie** (questo perché i piedini della porta **A** sono solo 4) e **4** caselle indicate **3 - 2 - 1 - 0** che possono essere **vuote** o contrassegnate da una **V**.

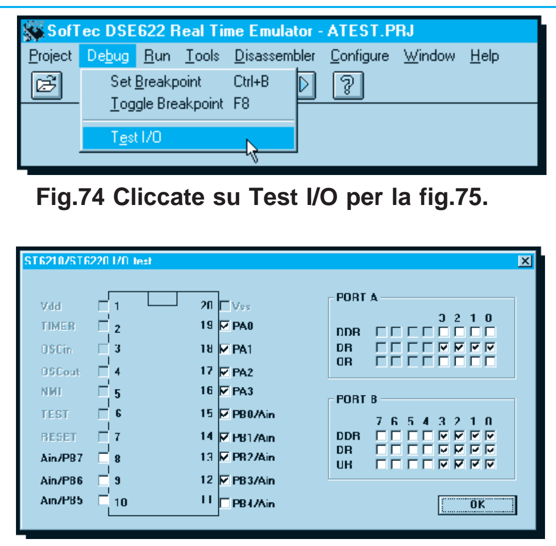


Fig.74 Cliccate su Test I/O per la fig.75.

Fig.75 Mappa di configurazione del micro.

caselle DDR - (Data Direction Register) Se queste caselle risultano **vuote** significa che abbiamo definito i piedini **input** (ingressi), quelle contrassegnate con una **V** indicano che li abbiamo definiti **output** (uscite).

caselle DR - (Data Register) Se queste caselle risultano **vuote** significa che sui piedini è presente un **livello logico 0**, se sono contrassegnate da una **V** significa che è presente un **livello logico 1**.

caselle OR - (Opzion Register) Queste caselle servono per selezionare le varie opzioni delle porte. Se con il **DDR** abbiamo predefinito il piedino come **input**, combinandolo con il **DR** e l'**OR** otterremo queste selezioni:

DDR	DR	OR	opzione come ingressi
0	0	0	pull-up senza interrupt
0	1	0	senza pull-up e senza interrupt
0	0	1	con pull-up e con interrupt
0	1	1	ingresso analogico (vedi nota)

Nota: L'ingresso **analogico** non è consentito per i piedini **PA0 - PA1 - PA2 - PA3**.

Per la **porta B**, che dispone di 8 piedini **PB0 - PB1 - PB2 - PB3 - PB4 - PB5 - PB6 - PB7**, vedrete **8 caselle** su **3 file** indicate **7 - 6 - 5 - 4 - 3 - 2 - 1 - 0** che possono essere **vuote** o contrassegnate da una **V**.

caselle DDR - (Data Direction Register) Se queste caselle risultano **vuote** significa che abbiamo definito i piedini **input** (ingressi), quelle contrassegnate con una **V** indicano che li abbiamo definiti **output** (uscite).

caselle DR - (Data Register) Se queste caselle risultano **vuote** significa che sui piedini è presente un **livello logico 0**, se sono contrassegnate da una **V** significa che è presente un **livello logico 1**.

caselle OR - (Opzion Register) Queste caselle servono per selezionare le varie opzioni delle porte. Se con il **DDR** abbiamo predefinito il piedino come **output**, combinandolo con il **DR** e l'**OR** otterremo queste selezioni:

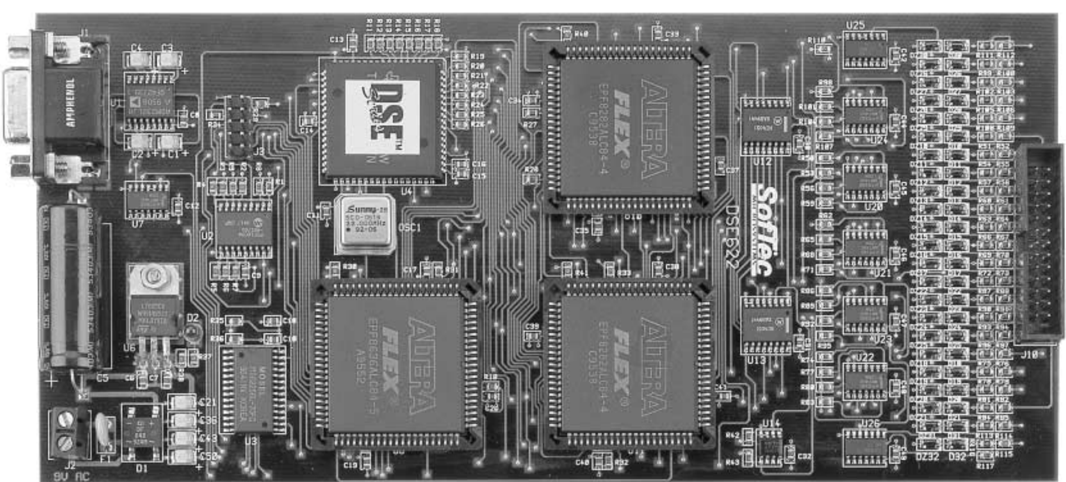
DDR	DR	OR	opzione come uscite
1	0	0	Collettore aperto
1	1	0	Collettore aperto
1	0	1	uscita in Push-Pull

Poiché in molti programmi non si usano i termini **DDR - DR - OR**, ma **PDIR - PORT - POTP**, riportiamo le loro equivalenze:

PDIR corrisponde a **DDR**
PORT corrisponde a **DR**
POTP corrisponde a **OR**

Se rileggerete tutti gli articoli precedenti riguardanti l'**ST6** (vedi Riviste N.172/173 - 174 - 175/176) troverete molti esempi su come procedere per **settare** le porte come **ingressi** e **uscite**.

Potendo **vedere** tramite la finestra visibile in fig.75 tutti i **livelli logici** presenti su questi piedini, potete comprendere quanto risulti semplice accorgersi degli errori, anche perché proseguendo **passo x passo** potete subito verificare come cambiamo i livelli logici sia sugli ingressi sia sulle uscite.



Questo software vi permetterà di vedere nella finestra Disassembler tutte le istruzioni in formato leggibile; nella finestra Register tutti i registri, lo stack level e gli stati dei flags; nella finestra Data il contenuto delle variabili dei registri e della data rom windows ecc. Chi desiderasse acquistare la scheda emulatrice (vedi foto) può rivolgersi a:

SOFTEC MICROSYSTEMS V.le Rimembranze, 19/C 33082 AZZANO DECIMO (PN)
fax 0434-631598 – tel. 0434-640113 – BBS 0434-631904

Nel disegno grafico visibile sulla sinistra di fig.75, potete vedere i piedini che hanno una **casella grigia**, ad esempio:

1 = Vdd (tensione positiva di alimentazione)
2 = Timer
3 = Oscillatore input
4 = Oscillatore uscita
5 = NMI (Interrupt non mascherato)
6 = TEST (piedino di programmazione)
7 = Reset
20 = Vss (tensione negativa di alimentazione)

Le **caselle grigie** non possono essere direttamente testate nella **simulazione** tramite **software**, perché manca la **tensione di alimentazione** ed il **quarzo**, quindi dovrete attivarle con alcuni accorgimenti.

A esempio, non disponendo della **frequenza** di **clock** potrete simulare le funzioni di **timer** solo attivando da programma la **subroutine** legata all'**interrupt del timer**.

La funzione di **reset** può essere invece attivata con un comando presente nella **barra** dei menu.

Chi si procurerà la **scheda emulatrice** sarà in grado di testare in modo automatico anche queste funzioni, perché ha la stessa funzione del **micro**.

In ogni caso risolverete molti problemi già con il solo **software**.
Ad esempio se avete scritto un programma che deve portare a **livello logico 1** il **piedino 6** e simulandolo vi accorgete che è rimasto a **livello logico 0**, vi sarà molto più facile, avanzando **passo** per **passo** e controllando ogni istruzione, trovare quella che, per un banale **errore**, non ha provveduto a modificare il livello logico su tale piedino.

Per uscire dal **debug** è sufficiente portare il cursore sulla scritta **OK** poi cliccare.
Il programma ripartirà **automaticamente** dalla prima **istruzione eseguibile**.
Se lancerete l'esecuzione **automatica** vedrete il programma ruotare all'infinito sulle etichette:

ripeti
main00 - mains1 - main01 - mains2
main02 - mains3 - main03 - mains4

perché non trova premuto nessuno dei quattro interruttori presenti sulla porta **A**.

COME SIMULARE L'INTERRUTTORE

Nel programma **ATEST** tutte le uscite rimangono a **livello logico 0** fino a quando non mettete a **livello logico 1** uno dei quattro **ingressi**.
Poiché siamo in **simulazione** e non avete né un "interruttore" né una tensione "positiva", per portare a **livello logico 1** uno di questi ingressi dovrete forzare l'ingresso desiderato come ora vi spiegheremo.

Cliccate sull'icona di **Stop** visibile in fig.76 rappresentata da una **mano aperta**.

Fig.76 Icona di Stop.

Ammesso di voler portare a **livello logico 1** l'ingresso del piedino **PA2** di **port_a** affinché sull'uscita del piedino **PB2** di **port_b** appaia lo stesso livello logico, per prima cosa dovrete andare nella finestra **Watch**, visibile in fig.77, per controllare l'indirizzo di **port_a**, che risulta essere **C0** (vedi sotto **ADD**).

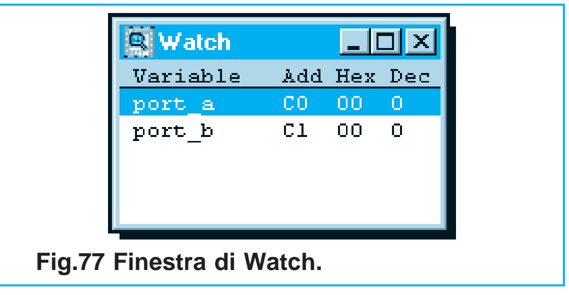


Fig.77 Finestra di Watch.

Per forzare a **livello logico 1** l'ingresso del piedino **PA2** dovete andare nelle finestra **DATA** (vedi fig.78) poi ricercare nella colonna degli indirizzi di memoria il valore esadecimale **C0**.
Trovato **C0** guardate nel righello in alto in cui appaiono i valori esadecimali:

00-01-02-03-04 ——- 08-09-0A-0B-0C-0D-0E-0F

Poiché l'indirizzo di **port_a** è **C0**, scendete dal valore esadecimale **00** fino ad incontrare la riga **C0** e così troverete il valore **00** (vedi fig.78).
Portate il cursore su **00** e cliccate **2 volte**.

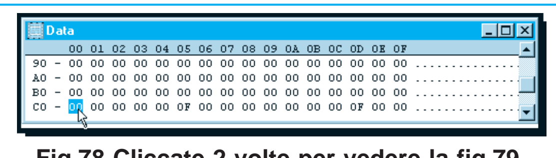


Fig.78 Cliccate 2 volte per vedere la fig.79.

Quando appare la finestra **Edit Data** (fig.79) dovete cliccare sulla scritta **Bits** in modo da far apparire la finestra di fig.80.

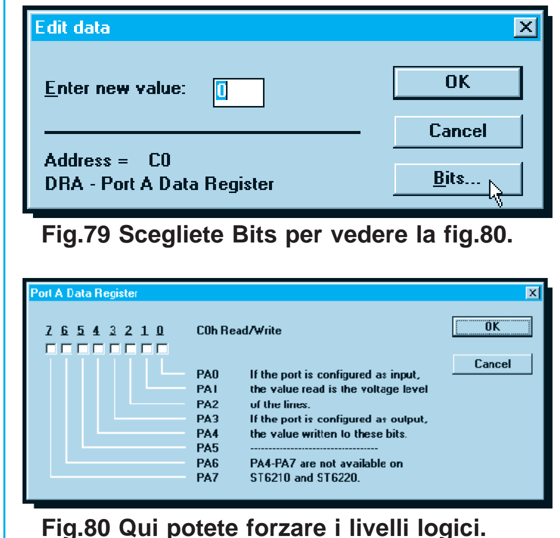


Fig.79 Scegliete Bits per vedere la fig.80.

Fig.80 Qui potete forzare i livelli logici.

In questa finestra potete vedere lo stato logico presente su tutti i piedini della porta **A** e poiché tutte le caselle risultano **vuote**, è ovvio che su tutti i piedini è presente un **livello logico 0**.

Volendo **forzare** a **livello logico 1** il piedino d'ingresso **PA2** dovrete portare il cursore nella casella posta sotto il **numero 2** e cliccare.

Comparirà così **V** a conferma del fatto che sul piedino d'ingresso **PA2** è ora presente un **livello logico 1**.

Nota: se cliccherete una seconda volta tornerà a **livello logico 0**.

A questo punto cliccate su **OK** e nella finestra che appare ritornate a cliccare su **OK**: apparirà così la finestra di fig.81.

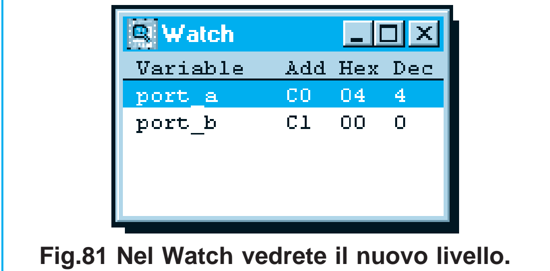


Fig.81 Nel Watch vedrete il nuovo livello.

Se ora guardate all'interno della finestra **Watch** vedrete che qualcosa è cambiato, infatti sotto la colonna **.Hex** troverete **04** e sotto la colonna **Dec.** il numero **4**.

Poiché avevamo "bloccato" il programma pigiando **Stop** (icona con mano) per farlo ripartire dovete ricercare l'etichetta **ripeti** procedendo come segue:

- Attivate la finestra **Disassembler** cliccando all'interno della finestra, quindi cliccate sulla scritta **Disassembler** del menu e poi cliccate sulla scritta **Set New PC** (vedi fig.82).

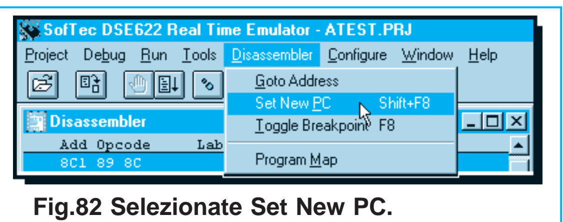


Fig.82 Selezionate Set New PC.

- Apparirà così la finestra di dialogo **New program counter** (vedi fig.83) e cliccando sulla freccia posta a destra della finestra **PC value** dovrete ricercare l'etichetta **ripeti** (vedi fig.84).

Nota: tutte le **etichette** sono in ordine alfabetico.

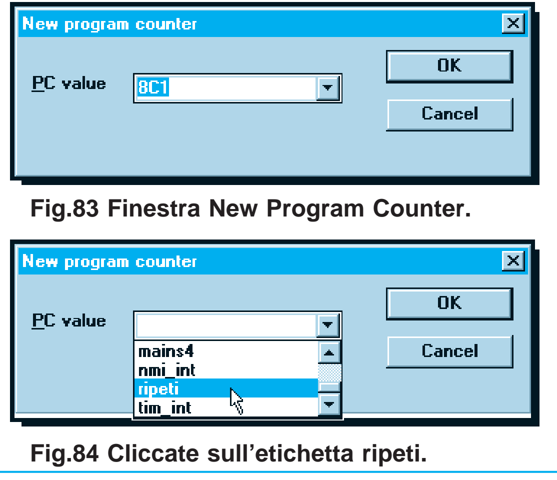


Fig.83 Finestra New Program Counter.

Fig.84 Cliccate sull'etichetta ripeti.

- Ponete il cursore su **ripeti** e cliccate, poi cliccate su **OK** e vedrete apparire la finestra di fig.85. Noterete che la prima riga in alto è ferma sull'etichetta **ripeti**.

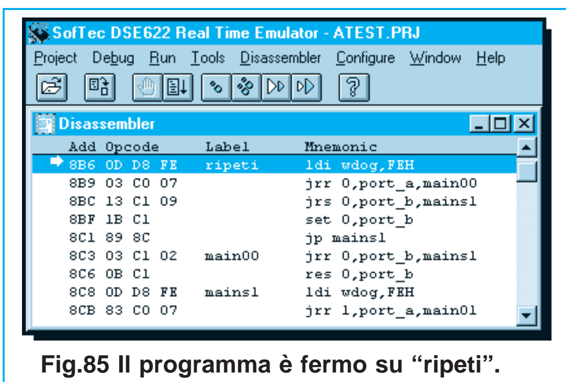


Fig.85 Il programma è fermo su "ripeti".

- A questo punto posizionate il cursore sull'icona di esecuzione **passo** per **passo** e cliccate più volte fino ad arrivare all'istruzione:

8DD 43 C0 07 jrr 2,port_a,main02

Poiché questo bit è a **livello logico 1**, il programma **non salterà** più a **main02** ma proseguirà all'istruzione:

8E0 53 C1 09 jrs 2,port_b,mains3

Poiché **PB2** è a **livello logico 0** non salterà a **mains3**, ma, cliccando sul tasto **passo** per **passo**, proseguirà fino alla successiva istruzione che sarà:

8E3 5B C1 set 2,port_b

Per eseguire questa istruzione cliccate ancora sull'icona **passo passo** ed il piedino **PB2** cambierà il suo livello logico da **0** a **1**.

Per vedere se questa condizione si è verificata potrete guardare nella finestra **Watch** dove leggerete:

port_b C1 04 4

Se volete avere un'ulteriore conferma visiva andate nella finestra **Data** (fig.86), cercate l'indirizzo **C0**, poi andate sotto la colonna **01** e scendendo incontrerete la casella **04**.
Portate il cursore su questa casella e cliccate **due volte**.

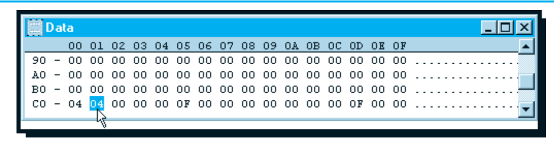


Fig.86 Sulla casella 04 cliccate 2 volte.

Quando appare la maschera **Edit Data** (vedi fig.87) cliccate su **Bits** e comparirà la finestra di dialogo di fig.88 dove potrete vedere che nella casella sotto il numero **2** del piedino **PB2** c'è una **V** ad indicare che questo piedino è a **livello logico 1**.

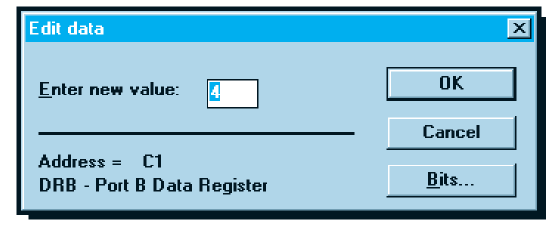


Fig.87 Nell'Edit data cliccate su Bits.

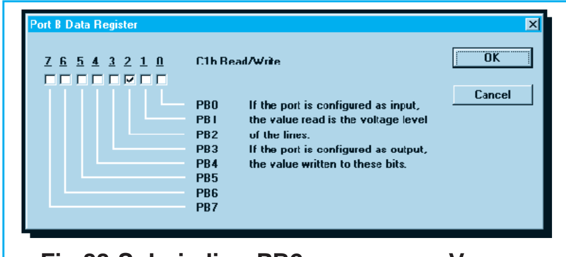


Fig.88 Sul piedino PB2 appare una V.

Per uscire cliccate su **OK** e nella successiva maschera cliccate nuovamente su **OK**: apparirà così la maschera di fig.85.

Ora che avete vi abbiamo spiegato come sia possa modificare un ingresso da **livello logico 0** a **livello logico 1** o viceversa, potete fare un po' di pratica portando a **livello logico 1** anche il piedino di un altro **ingresso** per poi riportarlo a **livello logico 0**, poi verificate se i piedini d'**uscita** sono passati da **livello logico 0** a **livello logico 1**.
Per verificarlo dovrete sempre far ripartire il programma dell'etichetta **ripeti** (vedi fig.85).

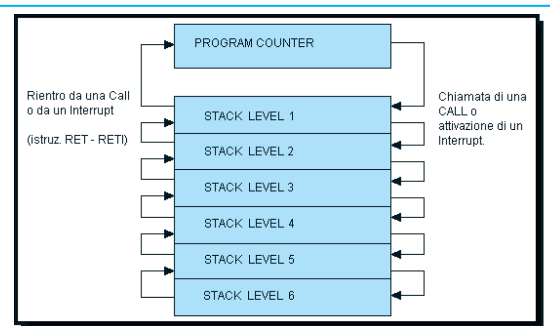


Fig.89 In ogni programma si possono eseguire fino a 6 subroutine nidificate, cioè una all'interno dell'altra.

NON ABBIAMO FINITO

Con questo articolo abbiamo riportato solo una condensata panoramica di quello che riesce a fare questo sofisticato **software di simulazione**.
Per spiegarvi tutto, cioè insegnarvi a capire come scoprire gli **errori**, come **correggerli** ecc., dovremo portarvi tanti altri esempi, e per questo vi rimandiamo al **prossimo** numero.
Comunque quando avrete a disposizione questo **software**, scoprirete voi stessi molte cose ed anche facilmente tramite la funzione di **Help**.

COSTO del dischetto SOFTWARE

Tutti i softwaristi e hobbisti che volessero entrare in possesso di questo dischetto di simulazione per **ST6** siglato **DF622.03** potranno richiederlo alla nostra Direzione. Costo del dischetto .... € 7,75

Prima di insegnarvi le procedure da adottare per cercare gli **errori** nei programmi scritti per i micro **ST6**, vi forniamo per ogni istruzione del linguaggio Assembler una **tabella.**
Queste tabelle vi serviranno come **guida rapida** quando scriverete un programma o quando lo correggerete, perché vi consentono di **decifrare** le codifiche dell'**opcode** e degli **indirizzi** di memoria di ogni istruzione.
Per avere una descrizione particolareggiata e corredata da esempi delle istruzioni in formato **Assembler** vi consigliamo di rileggere quanto già pubblicato sulla rivista **N.174**.

In queste tabelle trovate il **formato**, l'**opcode**, i **bytes**, i **cicli** e i **flags** di ogni istruzione.

C = Carry, registro di stato.

Z = **Zero**, registro di stato.

dst = **byte** che contiene l'indirizzo di una **variabile** o di un **registro** il cui valore può essere modificato dall'istruzione.

e = **5 bits** che esprimono un valore **decimale** composto da un numero da **0** a **63**.

ee = **8 bits** che esprimono un valore **decimale** composto da un numero da **0** a **255**.

MSB = è l'abbreviazione di **M**ost **S**ignificant **B**it, cioè "**bit significativo**". Si tratta del **bit 7** che per la matematica **binaria** quando viene **settato** a **1** vale in **decimale 128** ed in **binario 1000-0000**.

SOFTWARE simulatore per

Formato = composizione di una istruzione.
Opcode = codice operativo in formato **intel.hex**.
Bytes = lunghezza dell'istruzione in **bytes**.
Cicli = **passi** del micro per eseguire un'istruzione.
Flags = particolari bit indicatori che vengono **settati**, cioè posti a **livello logico 1**, oppure **resettati**, cioè posti a **livello logico 0**, a seconda che si verifichino o meno determinate condizioni a seguito dell'esecuzione di una istruzione. Questi **bit** indicatori vengono parcheggiati in speciali registri di **stato** chiamati **Carry** e **Zero**.

Anche se conoscerete già il significato delle parole utilizzate nell'articolo, ne ricordiamo qui alcune:

Variabile = è l'indirizzo di memoria **D**ata **S**pace identificato da un nome, ad esempio **port_a**, contenente un valore che nel corso dell'elaborazione può variare.

Overflow = letteralmente significa "traboccamento". Questo evento si verifica se il risultato di un'operazione matematica ci fornisce un valore più grande della capacità della **variabile** in cui lo dovremmo memorizzare. Ad esempio, se il risultato di una somma è **300** e tentate di memorizzare questo numero in una **variabile** lunga **1 byte** che può contenere un valore compreso tra **0** e **255**, si verificherà un **overflow**.

Le **abbreviazioni** che troverete utilizzate in quasi tutte le istruzioni hanno il seguente significato.

PC = **P**rogram **C**ounter. E' un registro a **12 bit** contenente l'**indirizzo** dell'istruzione in esecuzione.

src = **byte** che contiene l'indirizzo di una **variabile** o di un **registro** il cui valore non è modificabile dall'istruzione.

In ogni Tabella troverete il significato delle abbreviazioni utilizzate che qui non compaiono, e le **operazioni** che effettua il programma quando esegue l'istruzione.
Per capire come usare queste Tabelle di **guida rapida** prendiamo ad esempio quella dell'istruzione **Set**.
Una volta che avrete letto la spiegazione, saprete anche come usare tutte le altre tabelle.

SET Set Bit

Formato	opcode	bytes	cicli	flags	
SET bit,dst				Z	C
SET b,A	b11011 FF	2	4	*	*
SET b,rr	b11011 rr	2	4	*	*

Operazione: **istruzione** che serve per settare uno degli **8 bit** della **Variabile dst**.

A = registro dell'**Accumulatore**.
b = numero binario di tre **bit** di **indirizzo**.
rr = **1 byte** di indirizzo di una **Variabile**.
* = **Z** - **C** non influenzati.

TESTARE i micro ST6

Come vi abbiamo promesso nella rivista precedente, in questo articolo proseguiamo a spiegarvi come usare il "software simulatore DSE622" dandovi alcuni utili suggerimenti sui diversi test che è possibile eseguire sui programmi scritti in linguaggio Assembler. Anche i più esperti infatti possono involontariamente commettere errori nel programmare i microprocessori della famiglia ST6.

Nella colonna **Formato** abbiamo riportato il formato logico dell'istruzione.
L'istruzione **Set** si compone di un comando (**Set**), del bit da settare (**bit**) e della variabile (**dst**) in cui verrà settato (livello logico 1) il bit.

Esempio di – SET b,A

Per settare un bit dell'**accumulatore A** dobbiamo guardare la colonna **opcode**, in cui viene riportata la sua configurazione in formato **intel.hex**.

b11011 FF

b = è la combinazione di **tre bit** utilizzati per definire in **binario** un numero **decimale** da **0** a **7**.

11011 = è la combinazione binaria che il microprocessore riconosce come l'istruzione **Set**, quindi non deve essere mai modificata.

b+11011 = è dunque un numero binario di **8 bit** che il computer utilizza per sapere quale bit dell'accumulatore **A** deve settare. Questo numero binario occupa **1 byte**.

FF = è l'indirizzo di memoria dell'accumulatore **A** in formato esadecimale. Questo indirizzo è di **1 byte**.

Tutta l'istruzione **b11011FF** occupa un totale di **2 byte**, come potete vedere nella **terza** colonna denominata **bytes**.

Nella **quarta** colonna (**ciclo**) sono riportati i numeri di passi necessari al microprocessore per eseguire l'istruzione.

Ammesso di avere un quarzo da **8 MHz**, per conoscere il tempo di esecuzione espresso in **microsecondi** possiamo usare questa formula:

microsecondi = (13 : MHz) x cicli macchina

Questa istruzione verrà perciò eseguita in:

(13 : 8) x 4 = 6,5 microsecondi

Nella **quinta** colonna (**Flags Z - C**) trovate degli asterischi perché l'istruzione **set** non influenza lo stato logico di **Z** e di **C**.

Esempio: Per **settare** il **bit 7** dell'**Accumulatore** dobbiamo scrivere questa istruzione:

set 7,A

Il compilatore Assembler convertirà l'istruzione in questi numeri binari:

11111011 11111111

Nel numero binario **11111011** (esadecimale **FB**) i primi cinque bit partendo da destra, cioè **11011**, corrispondono all'istruzione **Set**.
Gli ultimi **tre bit**, cioè **111**, corrispondono al numero decimale **7**.

Il secondo numero binario, cioè **11111111** (**esadecimale FF**), corrisponde al numero dell'indirizzo dell'**Accumulatore**.

Tutta l'istruzione viene visualizzata sul monitor dal **simulatore** non in numero **binario**, ma in un numero **esadecimale**, cioè:

FB FF

Esempio di – SET b,rr

Per settare un bit della **Variabile rr** dobbiamo guardare la colonna **opcode**, in cui viene riportata la sua configurazione in formato **intel.hex**.

b11011 rr

b = è la combinazione di **tre bit** utilizzati per definire in **binario** un numero **decimale** da **0** a **7**.

11011 = è la combinazione binaria che il microprocessore riconosce come l'istruzione **Set**, quindi non deve essere mai modificata.

b+11011 = è dunque un numero binario di **8 bit** che il computer utilizza per sapere quale bit della variabile **rr** deve settare. Questo numero binario occupa **1 byte**.

rr = è l'indirizzo di memoria della **Variabile** di Data Space. Questo indirizzo è di **1 byte**.

Tutta l'istruzione **b11011rr** occupa un totale di **2 byte**, come potete vedere nella **terza** colonna denominata **bytes**.

Nella **quarta** colonna (**ciclo**) sono riportati i numeri di passi necessari al microprocessore per eseguire l'istruzione.

Nella **quinta** colonna (**Flags Z - C**) trovate degli asterischi perché l'istruzione **set** non influenza lo stato logico di **Z** e di **C**.

Esempio: Per **settare** il **bit 2** di **port_a**, cioè per portare a **livello logico 1** il piedino **PA2** della porta **A**, dobbiamo scrivere questa istruzione:

set 2,port_a

Il compilatore Assembler convertirà l'istruzione in questi numeri binari:

01011011 11000000

Nel numero binario **01011011** (esadecimale **5B**) i primi cinque bit partendo da destra, cioè **11011**, corrispondono all'istruzione **Set**.
Gli ultimi **tre bit**, cioè **010**, corrispondono al numero decimale **2**.

Il secondo numero binario, cioè **11000000** (esadecimale **C0**), corrisponde all'indirizzo di **port_a**.

Tutta l'istruzione viene visualizzata sul monitor dal **simulatore** non in numero **binario**, ma in un numero **esadecimale**, cioè :

5B C0

Di seguito trovate tutte le Tabelle della **guida rapida** in ordine alfabetico.

ADD Addition

Formato	opcode	bytes	cicli	flags	
ADD dst,src				Z	C
ADD A,A	5F FF	2	4	∆	∆
ADD A,X	5F 80	2	4	∆	∆
ADD A,Y	5F 81	2	4	∆	∆
ADD A,V	5F 82	2	4	∆	∆
ADD A,W	5F 83	2	4	∆	∆
ADD A,(X)	47	1	4	∆	∆
ADD A,(Y)	4F	1	4	∆	∆
ADD A,rr	5F rr	2	4	∆	∆

Operazione: il contenuto di una **variabile** viene sommato al contenuto dell'**Accumulatore** ed il risultato dell'operazione è memorizzato nell'**Accumulatore**.

A = registro dell'**Accumulatore**.
X-Y-V-W = **registri** del micro.
rr = **1 byte** di indirizzo di una **Variabile**.
∆ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
∆ = **C** è **resettato** prima dell'operazione e si **setta** automaticamente se l'addizione genera **overflow**.

ADDI Addition Immediate

Formato	opcode	bytes	cicli	flags	
ADDI dst,src				Z	C
ADDI A,nn	57 nn	2	4	∆	∆

Operazione: un **numero** viene sommato al contenuto dell'**Accumulatore** ed il risultato dell'operazione è memorizzato nell'**Accumulatore**.

A = registro dell'**Accumulatore**.
nn = numero di **1 byte** (da **0** a **255**).
∆ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
∆ = **C** è **resettato** prima dell'operazione e si **setta** automaticamente se l'addizione genera **overflow**.

AND Logical AND

Formato	opcode	bytes	cicli	flags	
AND dst,src				Z	C
AND A,A	BF FF	2	4	∆	*
AND A,X	BF 80	2	4	∆	*
AND A,Y	BF 81	2	4	∆	*
AND A,V	BF 82	2	4	∆	*
AND A,W	BF 83	2	4	∆	*
AND A,(X)	A7	1	4	∆	*
AND A,(Y)	AF	1	4	∆	*
AND A,rr	BF rr	2	4	∆	*

Operazione: funzione di **And** tra l'**Accumulatore** ed una **Variabile**. Il risultato della funzione è memorizzato nell'**Accumulatore**.

A = registro dell'**Accumulatore**.
X-Y-V-W = **registri** del micro.
rr = **1 byte** di indirizzo di una **Variabile**.
∆ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
***** = **C** non influenzato.

ANDI LOGICAL AND Immediate

Formato	opcode	bytes	cicli	flags	
ANDI dst,src				Z	C
ANDI A,nn	B7 nn	2	4	∆	*

Operazione: viene eseguita la funzione di **And** di un **numero** con l'**Accumulatore**. Il risultato della funzione è memorizzata nell'**Accumulatore**.

A = registro dell'**Accumulatore**.
nn = numero di **1 byte** (da **0** a **255**).
∆ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
***** = **C** non influenzato.

CALL Call Subroutine

Formato	opcode	bytes	cicli	flags	
CALL dst				Z	C
CALL abc	c0001 ab	2	4	*	*

Operazione: viene utilizzata per chiamare una **subroutine.** Ogni volta che viene eseguita una **Call** il Program Counter viene memorizzato nel livello corrente di **Stack** e quest'ultimo si alza di un livello. Nel micro **ST62** il numero massimo dei livelli di **Stack** è **6**.

abc = etichetta della **subroutine** da eseguire espressa in **3 semibytes** per un totale di **12 bit**.
***** = **Z - C** non influenzati.

CLR Clear

Formato	opcode	bytes	cicli	flags	
CLR dst				Z	C
CLR A	DF FF	2	4	∆	∆
CLR X	0D 80 00	3	4	*	*
CLR Y	0D 81 00	3	4	*	*
CLR V	0D 82 00	3	4	*	*
CLR W	0D 83 00	3	4	*	*
CLR rr	0D rr 00	3	4	*	*

Operazione: serve per resettare l'**Accumulatore**, un **Registro** o una **Variabile**.

A = registro dell'**Accumulatore**.
X-Y-V-W = **registri** del micro.
rr = **1 byte** di indirizzo di una **Variabile**.
∆ = **Z** è **settato**.
∆ = **C** è **resettato**.
***** = **Z - C** non influenzati.

COM Complement

Formato	opcode	bytes	cicli	flags	
COM dst				Z	C
COM A	2D	1	4	∆	∆

Operazione: calcola il complemento al valore contenuto nell'Accumulatore e lo memorizza nell'Accumulatore stesso. A questo scopo utilizza la funzione di **Not** che **inverte** i livelli logici contenuti nell'Accumulatore.

A = registro dell'**Accumulatore**.
∆ = **Z** è settato se il risultato della funzione è **0**, **resettato** se diverso da **0**.
∆ = **C** è **settato** se prima della funzione il **bit 7** è **1**, **resettato** se prima della funzione il **bit 7** è **0**.

Operazione: decrementa di **1** il contenuto dell'**Accumulatore**, del **Registro** o della **Variabile**.

A = registro dell'**Accumulatore**.
X-Y-V-W = **registri** del micro.
rr = **1 byte** di indirizzo di una **Variabile**.
Δ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
* = **C** non viene in alcun modo influenzato, quindi mantiene lo stesso stato, livello logico **0** o livello logico **1**, in cui si trovava prima dell'istruzione.

CP Compare

Formato	opcode	bytes	cicli	flags	
CP dst,src				Z	C
CP A,A	3F FF	2	4	Δ	Δ
CP A,X	3F 80	2	4	Δ	Δ
CP A,Y	3F 81	2	4	Δ	Δ
CP A,V	3F 82	2	4	Δ	Δ
CP A,W	3F 83	2	4	Δ	Δ
CP A,(X)	27	1	4	Δ	Δ
CP A,(Y)	2F	1	4	Δ	Δ
CP A,rr	3F rr	2	4	Δ	Δ

Operazione: compara il contenuto di un **Registro** o di una **Variabile** con il contenuto dell'**Accumulatore**, sottraendo dal contenuto dell'Accumulatore il contenuto della Variabile o del Registro. L'Accumulatore rimane invariato.

A = registro dell'**Accumulatore**.
X-Y-V-W = **registri** del micro.
rr = **1 byte** di indirizzo di una **Variabile**.
Δ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
Δ = **C** è **settato** se l'**Accumulatore** è **minore** del contenuto del **Registro** o della **Variabile**, è **resettato** se l'**Accumulatore** è **uguale** o **maggiore**.

CPI Compare Immediate

Formato	opcode	bytes	cicli	flags	
CPI dst,src				Z	C
CPI A,nn	37 nn	2	4	Δ	Δ

Operazione: compara il contenuto dell'**Accumulatore** con un **numero** contenuto in un **byte**, sottraendo dal contenuto dell'Accumulatore il numero. L'Accumulatore rimane invariato.

A = registro dell'**Accumulatore**.
nn = numero di **1 byte** (da **0** a **255**).
Δ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
Δ = **C** è **settato** se l'**Accumulatore** è **minore** del numero **nn**, è **resettato** se l'**Accumulatore** è **uguale** o **maggiore**.

DEC Decrement

Formato	opcode	bytes	cicli	flags	
DEC dst				Z	C
DEC A	FF FF	2	4	Δ	*
DEC X	1D	1	4	Δ	*
DEC Y	5D	1	4	Δ	*
DEC V	9D	1	4	Δ	*
DEC W	DD	1	4	Δ	*
DEC (X)	E7	1	4	Δ	*
DEC (Y)	EF	1	4	Δ	*
DEC rr	FF rr	2	4	Δ	*

INC Increment

Formato	opcode	bytes	cicli	flags	
INC dst				Z	C
INC A	7F FF	2	4	Δ	*
INC X	15	1	4	Δ	*
INC Y	55	1	4	Δ	*
INC V	95	1	4	Δ	*
INC W	D5	1	4	Δ	*
INC (X)	67	1	4	Δ	*
INC (Y)	6F	1	4	Δ	*
INC rr	7F rr	2	4	Δ	*

Operazione: incrementa di **1** il contenuto dell'**Accumulatore**, del **Registro** o della **Variabile**.

A = registro dell'**Accumulatore**.
X-Y-V-W = **registri** del micro.
rr = **1 byte** di indirizzo di una **Variabile**.
Δ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
* = **C** non viene in alcun modo influenzato, quindi mantiene lo stesso stato, livello logico **0** o livello logico **1**, in cui si trovava prima dell'istruzione.

JP Jump

Formato	opcode	bytes	cicli	flags	
JP dst				Z	C
JP abc	c1001 ab	2	4	*	*

Operazione: viene utilizzata per fare un salto **incondizionato** ad una **etichetta**.

abc = indirizzo di Program Space dell'etichetta. Nel Program Space viene memorizzato questo indirizzo ed il programma "salta" all'etichetta per poi proseguire da questo punto in poi. **abc** è espresso in **3 semibytes** per un totale di **12 bit**.
* = **Z** e **C** non influenzati.

JRC Jump Relative on Carry Flag

Formato	opcode	bytes	cicli	flags	
				Z	C
JRC e	e110	1	2	*	*

Operazione: viene utilizzata per fare un salto **condizionato** dal **Carry Flag** quando questo è **settato**.

e = numero che rappresenta la distanza di **byte** dell'etichetta di salto rispetto al Program Counter. Il numero possibile di **bytes** di salto è **15 prima** e **16 dopo** rispetto al Program Counter.
* = **Z** e **C** non vengono in alcun modo influenzati, quindi mantengono lo stesso stato, livello logico **0** o livello logico **1**, in cui si trovavano prima dell'istruzione.

JRNC Jump Relative on Non Carry Flag

Formato	opcode	bytes	cicli	flags	
				Z	C
JRNC e	e010	1	2	*	*

Operazione: viene utilizzata per fare un salto **condizionato** dal **Carry Flag** quando questo è **resettato**.

e = numero che rappresenta la distanza di **byte** dell'etichetta di salto rispetto al Program Counter. Il numero possibile di **bytes** di salto è **15 prima** e **16 dopo** rispetto al Program Counter.
* = **Z** e **C** non vengono in alcun modo influenzati, quindi mantengono lo stesso stato, livello logico **0** o livello logico **1**, in cui si trovavano prima dell'istruzione.

JRNZ Jump Relative on Non Zero Flag

Formato	opcode	bytes	cicli	flags	
				Z	C
JRNZ e	e000	1	2	*	*

Operazione: viene utilizzata per fare un salto **condizionato** dal **Zero Flag** quando questo è **resettato**.

e = numero che rappresenta la distanza di **byte** dell'etichetta di salto rispetto al Program Counter. Il numero possibile di **bytes** di salto è **15 prima** e **16 dopo** rispetto al Program Counter.
* = **Z** e **C** non influenzati.

JRR Jump Relative if Reset

Formato	opcode	bytes	cicli	flags	
				Z	C
JRR b,rr,ee	b00011 rree	3	5	*	Δ

Operazione: viene utilizzata per fare un salto **condizionato** dal **bit** di una **Variabile** quando questo è **resettato**.

b = numero binario di tre **bit** di **indirizzo**.
rr = **1 byte** di indirizzo di una **Variabile**.
ee = numero che rappresenta la distanza di **byte** dell'etichetta di salto rispetto al Program Counter. Il numero possibile di **bytes** di salto è **126 prima** e **129 dopo** rispetto al Program Counter.
* = **Z** non viene in alcun modo influenzato, quindi mantiene lo stesso stato, livello logico **0** o livello logico **1**, in cui si trovava prima dell'istruzione.
Δ = **C** contiene il valore del **bit testato**.

JRS Jump Relative if Set

Formato	opcode	bytes	cicli	flags	
				Z	C
JRS b,rr,ee	b10011 rree	3	5	*	Δ

Operazione: viene utilizzata per fare un salto **condizionato** dal **bit** di una **Variabile** quando questo è **settato**.

b = numero binario di tre **bit** di **indirizzo**.
rr = **1 byte** di indirizzo di una **Variabile**.
ee = numero che rappresenta la distanza di **byte** dell'etichetta di salto rispetto al Program Counter. Il numero possibile di **bytes** di salto è **126 prima** e **129 dopo** rispetto al Program Counter.
* = **Z** non viene in alcun modo influenzato, quindi mantiene lo stesso stato, livello logico **0** o livello logico **1**, in cui si trovava prima dell'istruzione.
Δ = **C** contiene il valore del **bit testato**.

JRZ Jump Relative on Zero Flag

Formato	opcode	bytes	cicli	flags	
				Z	C
JRZ e	e100	1	2	*	*

Operazione: viene utilizzata per fare un salto **condizionato** dal **Zero Flag** quando questo è **settato**.

e = numero che rappresenta la distanza di **byte** dell'etichetta di salto rispetto al Program Counter. Il numero possibile di **bytes** di salto è **15 prima** e **16 dopo** rispetto al Program Counter.
* = **Z** e **C** non influenzati.

LD Load

Formato	opcode	bytes	cicli	flags	
LD dst,src				Z	C
LD A,X	35	1	4	Δ	*
LD A,Y	75	1	4	Δ	*
LD A,V	B5	1	4	Δ	*
LD A,W	F5	1	4	Δ	*
LD X,A	3D	1	4	Δ	*
LD Y,A	7D	1	4	Δ	*
LD V,A	BD	1	4	Δ	*
LD W,A	FD	1	4	Δ	*
LD A,(X)	07	1	4	Δ	*
LD (X),A	87	1	4	Δ	*
LD A,(Y)	0F	1	4	Δ	*
LD (Y),A	8F	1	4	Δ	*
LD A,rr	1F rr	2	4	Δ	*
LD rr,A	9F rr	2	4	Δ	*

Operazione: serve per **caricare** il valore contenuto in una **Variabile**, nell'**Accumulatore** o in un **Registro**. Può caricare il valore anche tra **Registro** ed **Accumulatore** Per questa istruzione bisogna sempre utilizzare l'**Accumulatore**.

A = registro dell'**Accumulatore**.
X-Y-V-W = **registri** del micro.
rr = **1 byte** di indirizzo di una **Variabile**.
Δ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
* = **C** non viene in alcun modo influenzato, quindi mantiene lo stesso stato, livello logico **0** o livello logico **1**, che aveva prima dell'istruzione.

LDI Load Immediate

Formato	opcode	bytes	cicli	flags	
LDI dst,src				Z	C
LDI A,nn	17 nn	2	4	Δ	*
LDI X,nn	0D 80 nn	3	4	*	*
LDI Y,nn	0D 81 nn	3	4	*	*
LDI V,nn	0D 82 nn	3	4	*	*
LDI W,nn	0D 83 nn	3	4	*	*
LDI rr,nn	0D rr nn	3	4	*	*

Operazione: serve per **caricare** un numero da **0** a **255** in una **Variabile**, nell'**Accumulatore** o in un **Registro**.

A = registro dell'**Accumulatore**.
X-Y-V-W = **registri** del micro.
nn = numero di **1 byte** (da **0** a **255**).
rr = **1 byte** di indirizzo di una **Variabile**.
Δ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
* = **Z** - **C** non influenzati.

NOP No Operation

Formato	opcode	bytes	cicli	flags	
				Z	C
NOP	04	1	2	*	*

Operazione: viene normalmente utilizzata per creare dei piccoli **ritardi**. Ogni **NOP** crea un ritardo di **2 cicli**.

* = **Z** - **C** non influenzati.

RES Reset Bit

Formato	opcode	bytes	cicli	flags	
RES bit, dst				Z	C
RES b,A	b01011 FF	2	4	*	*
RES b,rr	b01011 rr	2	4	*	*

Operazione: serve per **resettare** uno degli **8 bit** della **Variabile** o dell'**Accumulatore** di destinazione.

A = registro dell'**Accumulatore**
b = numero binario di tre **bit** di **indirizzo**.
rr = **1 byte** di indirizzo di una **Variabile**.
* = **Z** - **C** non influenzati.

RET Return from Subroutine

Formato	opcode	bytes	cicli	flags	
				Z	C
RET	CD	1	2	*	*

Operazione: viene utilizzata per ritornare da una **subroutine** al punto della chiamata **Call**. Quando viene eseguita una **RET** si abbassa di un livello lo **Stack** ed il Program Counter assume il valore relativo al livello corrente di Stack.

* = **Z** e **C** non influenzati.

RETI Return from Interrupt

Formato	opcode	bytes	cicli	flags	
				Z	C
RETI	4D	1	2	Δ	Δ

Operazione: viene utilizzata per ritornare da una **routine** di **interrupt** al punto precedente all'evento di **interrupt**. Quando viene eseguita una **RETI** si abbassa di un livello lo **Stack** ed il Program Counter assume il valore relativo al livello corrente di Stack.

Δ = **Z** e **C** vengono riportati alla condizione logica in cui si trovavano prima dell'interrupt.

RLC Rotate Left Through Carry

Formato	opcode	bytes	cicli	flags	
				Z	C
RLC A	AD	1	4	∆	∆

Operazione: serve per spostare di un posto verso **sinistra** gli **8 bit** dell'**Accumulatore.** Il **bit 7** passa nel **Carry** spostando il valore che si trovava sul **Carry** sul **bit 0** dell'**Accumulatore**.

A = registro dell'**Accumulatore**.
∆ = **Z** è settato se il risultato è **0**, **resettato** se diverso da **0**.
∆ = **C** riporta il valore del **bit 7**.

SET Set Bit

Formato	opcode	bytes	cicli	flags	
SET bit,dst				Z	C
SET b,A	b11011 FF	2	4	*	*
SET b,rr	b11011 rr	2	4	*	*

Operazione: serve per **settare** uno degli **8 bit** della **Variabile** o dell'**Accumulatore** di destinazione.

A = registro dell'**Accumulatore**
b = numero binario di tre **bit** di **indirizzo**.
rr = **1 byte** di indirizzo di una **Variabile**.
* = **Z** - **C** non influenzati.

SLA Shift Left Accumulator

Formato	opcode	bytes	cicli	flags	
				Z	C
SLA A	5F FF	2	4	∆	∆

Operazione: serve per spostare di un posto verso **sinistra** gli **8 bit** dell'**Accumulatore**. Il **bit 7** passa nel **Carry** cancellando il valore che risultava presente (equivale ad una moltiplicazione per **2**).

A = registro dell'**Accumulatore**.
∆ = **Z** è settato se il risultato è **0**, **resettato** se diverso da **0**.
∆ = **C** riporta il valore del **bit 7**.

STOP Stop Operation

Formato	opcode	bytes	cicli	flags	
				Z	C
STOP	6D	1	2	*	*

Operazione: serve per bloccare l'oscillatore del **clock** mettendo in **stand-by** tutto il micro **ST62**.

* = **Z** - **C** non influenzati.

SUB Subtraction

Formato	opcode	bytes	cicli	flags	
SUB dst,src				Z	C
SUB A,A	DF FF	2	4	∆	∆
SUB A,X	DF 80	2	4	∆	∆
SUB A,Y	DF 81	2	4	∆	∆
SUB A,V	DF 82	2	4	∆	∆
SUB A,W	DF 83	2	4	∆	∆
SUB A,(X)	C7	1	4	∆	∆
SUB A,(Y)	CF	1	4	∆	∆
SUB A,rr	DF rr	2	4	∆	∆

Operazione: il contenuto di una **variabile** viene sottratto all'**Accumulatore** ed il risultato dell'operazione viene memorizzato nell'**Accumulatore**.

A = registro dell'**Accumulatore**.
X-Y-V-W = **registri** del micro.
rr = **1 byte** di indirizzo di una **Variabile**.
∆ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
∆ = **C** è **settato** se il contenuto dell'**Accumulatore** è minore della **Variabile** o del **Registro**, **resettato** se maggiore o uguale.

SUBI Subtraction Immediate

Formato	opcode	bytes	cicli	flags	
SUBI dst,src				Z	C
SUBI A,nn	D7 nn	2	4	∆	∆

Operazione: un **numero** contenuto in un **byte** viene sottratto all'**Accumulatore** ed il risultato dell'operazione viene memorizzato nell'**Accumulatore**.

A = registro dell'**Accumulatore**.
nn = numero di **1 byte** (.da **0** a **255**).
∆ = **Z** è **settato** se il risultato è **0**, **resettato** se diverso da **0**.
∆ = **C** è **settato** se il contenuto dell'Accumulatore è minore del **numero**, **resettato** se maggiore o uguale.

WAIT Wait Processor

Formato	opcode	bytes	cicli	flags	
				Z	C
WAIT	ED	1	2	*	*

Operazione: serve per mettere in **stand-by** il micro **ST62**, ma l'oscillatore del **clock** rimane attivo.

* = **Z** - **C** non influenzati.

Chi, subito dopo aver letto l'articolo apparso sulla rivista **N.184**, si è affrettato ad acquistare il dischetto con il software **DSE.622**, che serve a "testare" tutti i programmi per i micro **ST6**, si è subito accorto con quanta facilità sia possibile simulare i programmi in Assembler.
Questo software consente di individuare dove e perché il programma non funziona e di correggere gli **errore logici**, facendo risparmiare così non solo tempo ma anche denaro, perché non è più necessario acquistare gli **ST6 riprogrammabili** per provare i programmi.

Abbiamo ricevuto molte lettere di elogio soprattutto dagli **uffici tecnici** delle piccole e medie Industrie che usano gli **ST6** per le loro macchine, e molti **Professori** che insegnano negli **Istituti Tecnici** ci hanno fatto sapere che lo considerano un valido supporto **didattico** alle loro lezioni teoriche.
In realtà questi complimenti non sono molto meritati perché noi ci siamo soltanto limitati a cercare tra i tanti **software** disponibili in commercio quello che ci sembrava il più **valido** come hardware e software e, quando l'abbiamo trovato, abbiamo spiegato sulla rivista in modo molto semplice e con tanti esempi il suo funzionamento e l'utilità delle sue funzioni più importanti.

Prima di spiegarvi come cercare e correggere gli errori che si possono commettere quando si scrive un programma, vogliamo aprire una parentesi per insegnarvi a generare il file **.SYM.**

SE NON APPARE IL FILE .SYM

Nella rivista N.184 vi abbiamo detto che il file ***.PRJ** utilizzato dal simulatore per testare il programma è formato:

– dal file **.HEX**, che contiene il programma eseguibile in formato **INTEL.HEX**.

– dal file **.SYM**, che contiene le definizioni delle **etichette** ed il relativo indirizzo di memoria **Program Space**.

– dal file **.DSD**, che contiene le definizioni, le caratteristiche ed il relativo indirizzo di memoria **Data Space** delle **variabili**.

– dalle **specifiche** proprie che vengono scelte da chi crea il file **.PRJ** dal file **.HEX**.

Molti lettori ci hanno segnalato che quando compilano in **assembler** non riescono a vedere il contenuto del file **.SYM**, quindi sul video non compare la parte del programma relativa alle **etichette** in formato simbolico, ma solo il loro indirizzo di memoria (vedi fig.90).

Per questo motivo quando avete assemblato il file **ATEST.ASM** per generare il file **ATEST.PRJ**, sono stati creati solo i files:

ATEST.HEX
ATEST.DSD

e non il file: **ATEST.SYM**

Anche se questo file non viene creato, il simulatore svolge ugualmente **tutte** le sue funzioni, ma invece di mostrarvi nel DSE le etichette in formato simbolico, fornisce solo la loro codifica in **esadecimale**.

Dal momento che invece è molto più semplice in fase di **simulazione** lavorare con il formato **.SYM**, vi spieghiamo come generarlo.

Come prima operazione caricate il programma **DSE622** e quando compare la finestra di fig.91, selezionate la scritta **Demo** per entrare nella finestra principale.
Cliccate sulla scritta **Tools** sulla barra dei **menu** e selezionate **ST6** (vedi fig.92).

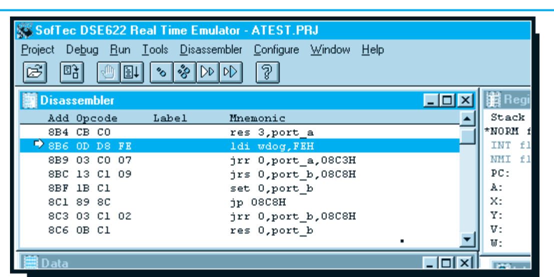


Fig.90 Se in fase di compilazione non è stato creato il file *.SYM, nel file .PRJ le etichette sono codificate in esadecimale.

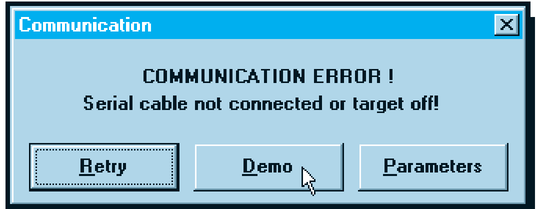


Fig.91 Scegliete l'opzione Demo per entrare nella finestra principale del DSE622.

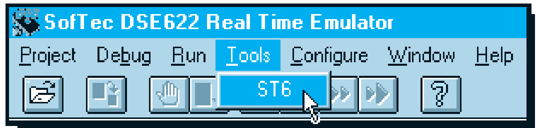


Fig.92 Per entrare nell'editor dell'ST6 dal DSE, scegliete ST6 dal menu Tools.

In questo modo entrerete nell'**editor** dell'**ST6**.
Per aprire il file usate il tasto **F3** e, nella riga **Name**, digitate ***.BAT** come visibile nella fig.93.
Cliccate su **Open** e vedrete apparire la finestra riportata in fig.94, dove risulta già selezionato il file **A.BAT**.
Cliccate su **Open** e sul monitor apparirà il contenuto di questo file, cioè **ast6 %1**.

Per generare il file **.SYM**, è necessario inserire in questa riga l'opzione **-S**.

Per aggiungere questa opzione dovete portare il cursore dopo la scritta **ast6**, digitare uno spazio e scrivere **-s**, quindi separare con uno spazio la scritta **1%**.
In altre parole deve apparire:

ast6 -s %1

come visibile nella finestra di fig.95.

A questo punto **salvate** il file premendo il tasto funzione **F2**, poi uscite premendo i due tasti **Alt+F3**.
Per completare la modifica dovete nuovamente compilare i files:

ATEST.ASM
BTEST.ASM

Per compilare il file **ATEST.ASM** pigiate il tasto funzione **F3**, poi selezionate il programma **ATEST.ASM**, quindi portate il cursore su **Open** e cliccate.

Nella finestra che appare (vedi fig.96) cliccate sulla scritta **ST6** poi su **Assembla**.
Quando il programma sarà compilato premete un tasto **qualsiasi**, poi premete **Alt+F3** per chiudere il file.

La stessa operazione deve essere effettuata per il file **BTEST.ASM** e per **tutti** quei files che avete compilato prima di aggiungere l'opzione **-S** al file **A.BAT**.

Terminata questa operazione potete uscire dall'**editor** premendo i tasti **Alt+X**.

Rientrerete così nel software di simulazione del **DSE622** dove tutti i files con estensione **.PRJ** che avete generato prima di questa modifica sono stati automaticamente aggiornati e contengono quindi anche le informazioni in formato simbolico relative al file **.SYM**.

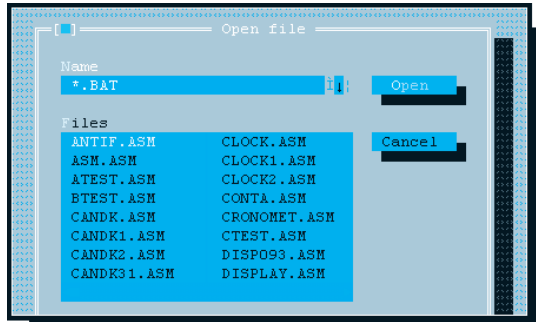


Fig. 93 Utilizzate il tasto funzione F3 per aprire il file con estensione .BAT.

Fig.94 Il file che dovete modificare per generare il file .SYM si chiama A.BAT.

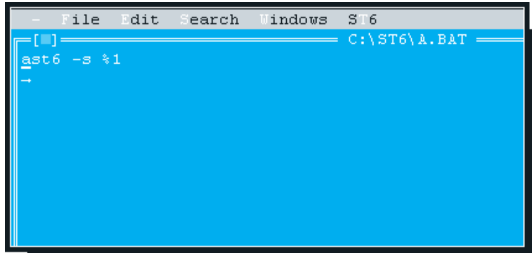


Fig.95 L'opzione -s, che serve a generare il file .SYM in fase di compilazione, deve essere inserita tra le scritte ast6 e 1% separandola con degli spazi.

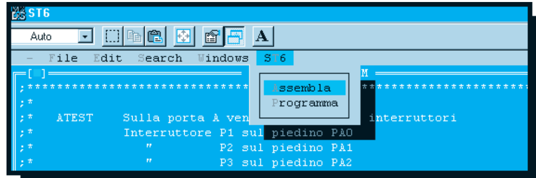


Fig.96 Tutti i files che avete compilato prima di aggiungere l'opzione -s devono essere nuovamente assemblati.

LA CORREZIONE DEGLI ERRORI

Dopo questa parentesi riprendiamo la descrizione del funzionamento del **DSE** fornendovi alcuni suggerimenti per controllare **passo-passo** le istruzioni e di conseguenza **correggere** gli **errori** che si possono commettere quando si scrive un programma.

Per correggere un **errore** possiamo optare tra due soluzioni:

– correggere in maniera **temporanea** il file **.PRJ**
– correggere in maniera **definitiva** il file **.ASM**

Le correzioni **temporanee** riguardano il solo file **.PRJ**, quindi spegnendo il computer o comunque uscendo dal programma **DSE622** vengono tutte perdute.

Le correzioni **definitive** possono essere portate solo sul file **.ASM**, ma spegnendo il computer o uscendo dal programma **rimangono** in memoria.

Leggendo quanto sopra sembrerebbe più logico fare le correzioni direttamente nel **sorgente**, cioè nel file con estensione **.ASM**, ma non sempre conviene andare in questa direzione per i seguenti semplici motivi:

– potrebbero esserci altri **errori** oltre a quello che avete corretto,
– potreste inserirne **uno** proprio durante la correzione.

Inoltre per ricontrollare il programma dovreste nuovamente compilare il file **.ASM**, creare il file **.PRJ** e **settare** daccapo i **piedini** per effettuare una corretta simulazione.

E' quindi consigliabile apportare, dove possibile, le correzioni in modo **temporaneo** sul file **.PRJ**, poi simulare l'esecuzione del programma e, una volta verificato che tutto funziona regolarmente, potrete correggere **definitivamente** il sorgente.

Tuttavia non tutte le correzioni si possono apportare nel file **.PRJ**.
Ad esempio non si può sostituire un'**istruzione** lunga **2 bytes** con una lunga **3 bytes**, perché il **byte** in eccesso andrebbe a ricoprire il **primo byte** dell'istruzione **successiva** sconvolgendo completamente le funzioni del programma.
Quindi prima di sostituire un'istruzione con un'altra bisogna sempre controllare che la nuova istruzione non risulti più lunga di quella da sostituire.
Ad esempio l'istruzione **JRZ** (che è lunga **1 byte**) non può essere sostituita dall'istruzione **JRR** (che è lunga **3 bytes**), ma si può invece fare il contrario utilizzando l'istruzione **NOP** per coprire i **bytes** non utilizzati.

Esempio: Ammesso di voler modificare l'istruzione:

jrr 0,potr_b,mains1 ;(istruzione lunga 3 byte)

con l'istruzione:

jrz mains1 ;(istruzione lunga 1 byte)

poiché mancano 2 bytes dovremo aggiungere due istruzioni **Nop** per avere di nuovo **3 bytes**:

jrz mains1 nop nop

Ad ogni modo ricordatevi di non eccedere con i **NOP** perché occupereste solo della memoria per eseguire istruzioni a vuoto.
Quindi se questo vi accadesse vi converrà correggere e rivedere tutto il programma direttamente nel sorgente.
Per conoscere la lunghezza in **byte** di tutte le istruzioni potete consultare le **tabelle** della **guida pratica** che trovate all'inizio dell'articolo.

GLI errori nel programma BTEST

Come vi abbiamo spiegato nella rivista **N.184**, nel programma **BTEST.ASM** abbiamo inserito degli **errori**, per la precisione **tre**, al fine di mostrarvi come procedere per poterli **individuare** e di conseguenza **correggere**.
La **tipologia** degli errori che vi proponiamo con questo esempio pratico ci permette di spiegarvi quali **test** vanno eseguiti e come vanno eseguiti per trovare gli errori.
Inoltre vi spiegheremo come apportare le modifiche temporanee e definitive in qualsiasi programma in linguaggio Assembler per micro ST6.

Per simulare il programma **BTEST** è prima necessario che lo compiliate in **Assembler**, in modo da creare il file **BTEST.HEX**, e che generiate il rispettivo **project**, cioè il file **BTEST.PRJ**, che viene utilizzato dal **simulatore** per **testare** il programma.

Per eseguire tutte queste operazioni rimandiamo a quanto già ampiamente descritto nel paragrafo "Compilare in assembler il programma atest.asm" riportate a **pag.112** e seguenti della rivista **N.184**.

Quando appare la finestra di fig.97, aprite il file selezionando dal menu **Project**, in alto a sinistra sulla barra del **menu**, il comando **Open Project**.

Si aprirà così la finestra di dialogo **File Open** visibile in fig.98.
Nella finestra a sinistra selezionate la scritta **BTEST.PRJ** quindi cliccate su **OK.**
A video compariranno tutte le finestre visibili in fig.99, che vi consentono di **controllare** istruzione per istruzione il programma.

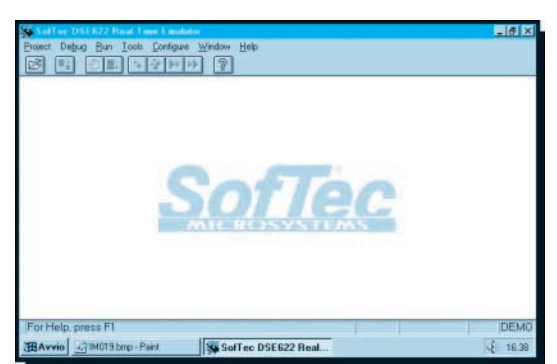


Fig.97 Dopo il nome del DSE, nella prima riga in alto trovate la barra dei menu a tendina, nella riga immediatamente sotto trovate la barra delle icone o degli strumenti che vi consente di accedere rapidamente ai comandi di frequente utilizzo.

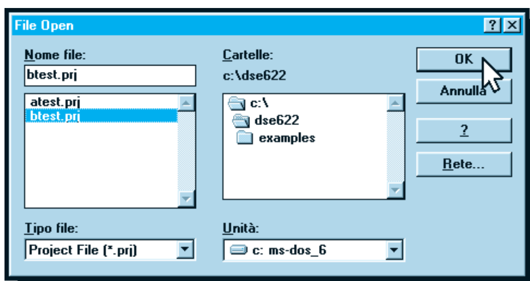


Fig.98 In questo articolo esaminiamo attentamente gli errori del file BTEST.PRJ.

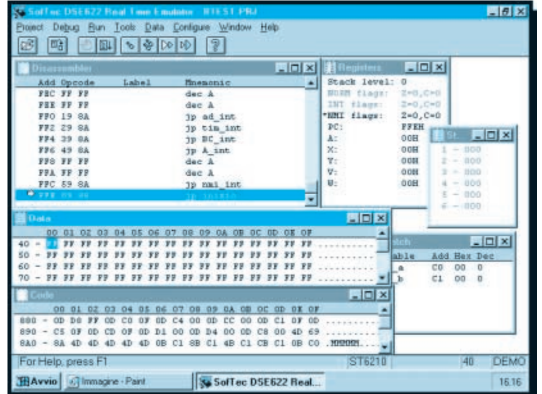


Fig.99 Le finestre del DSE per simulare l'esecuzione dei programmi.

Prima di **testare** il programma dovete inserire le variabili della **porta A** e della **porta B** nella finestra **Watch**.
Anche per questa operazione vi consigliamo di rileggere quanto spiegato nella rivista precedente sotto il paragrafo "Inserire una variabile nella finestra Watch".
In questo modo potrete verificare per ogni **istruzione** lo stato logico dei piedini d'**ingresso** (porta **A**) e d'**uscita** (porta **B**).

Eseguite tutte queste operazione potrete **simulare** le funzioni del programma.

PRIMO TEST

Se avete seguito quanto fin qui detto, nella finestra **Disassembler** sarà evidenziata l'istruzione:

FFE 09 88 jp inizio

visibile anche in fig.99.

Cliccate sull'icona **passo-passo** (la **5°** posta sul righello in alto vedi fig.100) fino ad arrivare all'etichetta **ripeti** visibile in fig.101 cioè:

8B6 0D D8 FE ripeti ldi wdog,FEH

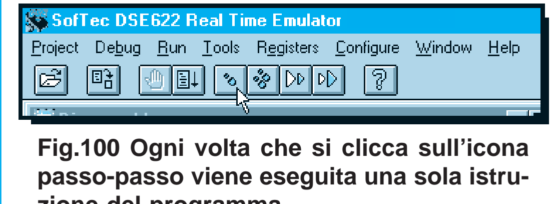


Fig.100 Ogni volta che si clicca sull'icona passo-passo viene eseguita una sola istruzione del programma.

Prima di proseguire riteniamo necessario ricordarvi che il programma **BTEST** utilizza:

– i quattro piedini **PA0 - PA1 - PA2 - PA3** della porta **A** come **ingressi**
– i quattro piedini **PB0 - PB1 - PB2 - PB3** della porta **B** come **uscite**

Una delle funzioni del programma serve a portare a **livello logico 1** un piedino d'**uscita** quando sul corrispondente piedino d'**ingresso** viene applicato un **livello logico 1**.

Portiamo un **esempio**: questo programma potrebbe essere usato per **accendere** un **diodo led** o per polarizzare la Base di un **transistor**, in modo che **ecciti** un **relè**, collegato su un piedino d'uscita quando si preme un **pulsante** che porta a **livello logico 1** il corrispondente piedino d'**ingresso**.

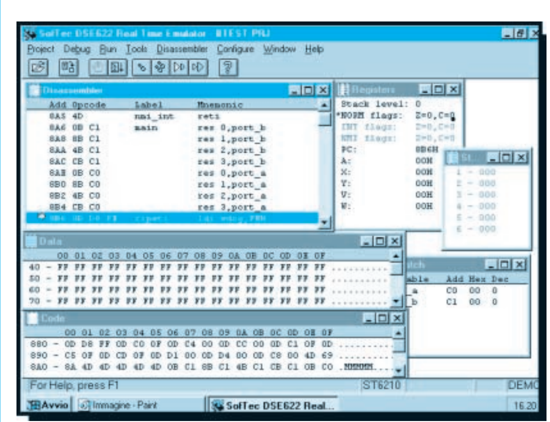


Fig.101 Iniziamo a simulare il programma B-TEST.PRJ dall'etichetta ripeti memorizzata all'indirizzo 8B6 (vedi colonna Add).

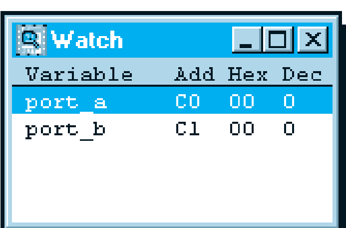


Fig.102 La finestra Watch ci permette di controllare in qualunque momento il contenuto delle variabili port_a e port_b.

Quindi se pigiamo il pulsante collegato sull'ingresso **PA1**, si deve accendere il **diodo led** collegato sull'uscita **PB1**, e lasciandolo si deve spegnere.
Se pigiamo il pulsante collegato sull'ingresso **PA4** si deve accendere il **diodo led** collegato sull'uscita **PB4**, e lasciandolo si deve spegnere.

Ora possiamo verificare se le istruzioni rispondono a questa funzione.

Dall'etichetta **ripeti** (vedi fig.101), cliccando sull'icona **passo-passo** per avanzare di un'istruzione, viene evidenziata:

8B9 03 C0 07 jrr 0,port_a,main00

L'istruzione **JRR** significa letteralmente fai un **salto** se il bit di una **variabile** è resettato, cioè se si trova a **livello logico 0**.

Nota: per la descrizione di tutte le istruzioni del linguaggio **Assembler** rimandiamo alla rivista **N.174** che vi consigliamo di leggere attentamente.

Nel nostro caso questa istruzione segnala al programma di saltare all'etichetta **main00** se il bit **0** di **port_a**, cioè se il piedino **PA0**, è **resettato**.

Controlliamo il contenuto della porta **A** nella finestra **Watch** e, come potete vedere in fig.102, il contenuto della variabile **port_a** all'indirizzo **C0** è **00**, quindi i piedini sono resettati.

Se quindi avanziamo di un'altra **istruzione** il programma salterà all'istruzione con etichetta **main00**, ed in effetti premendo passo-passo viene evidenziata questa istruzione:

8C3 03 C1 02 main00 jrr 0,port_b,mains1

Poiché **PA0** è resettato (livello logico 0) anche **PB0** deve essere resettato (livello logico 0), ma il programma prima di portarlo a **livello logico 0** controlla che questa uscita non si trovi già in questa condizione.
Questa istruzione ha proprio il compito di verificare se il piedino **PB0** di **port_b** è settato (livello logico 1), cioè se l'ipotetico **diodo led** collegato a questo piedino è **acceso**, e solo in questo caso lo **spegne**, cioè porta l'uscita a livello logico 0.

Se controllate la finestra **Watch** noterete che il contenuto della **port_b** all'indirizzo **C1** è **00**. In altre parole il piedino è già resettato quindi non è necessario resettarlo.
Avanzando di un'istruzione il programma salta perciò all'istruzione con etichetta **mains1**:

8C8 0D D8 FE mains1 ldi wdog,0feh

Questa istruzione ripristina il **watchdog**.

Nota: abbiamo descritto la funzione **watchdog** sulla rivista **N.175/176** e poiché sappiamo che non vi manca nessun numero di **Nuova Elettronica**, non avrete difficoltà a rinfrescarvi la memoria.

Cliccando sull'icona **passo-passo** viene evidenziata l'istruzione:

8CB 83 C0 07 jrr 1,port_a,main01

Come avrete già intuito, avanzando **passo-passo** il programma controlla gli altri piedini della porta **A**, cioè **PA1 - PA2 - PA3**, ed i rispettivi piedini della porta **B**, cioè **PB1 - PB2 - PB3**, come ha appena fatto per il piedino d'ingresso **PA0** e quello d'uscita **PB0**.
Essendo il contenuto di **port_a** e **port_b** uguale a **0** (vedi finestra **Watch**), cliccando sempre su **passo-passo** alla fine del **controllo** il programma tornerà all'etichetta **ripeti** visibile in fig.101.

SECONDO TEST

Se ci fermassimo a questo **primo** superficiale controllo potremmo affermare che il programma funziona **correttamente**.
Noi però sappiamo che il **BTEST** contiene **tre errori**, perché li abbiamo messi di proposito, quindi ora vi spieghiamo quali altri controlli vanno effettuati per testare ulteriormente il programma.

Un'altra prova che va fatta è quella di simulare un **interruttore**, cioè portare a **livello logico 1** tutti i quattro ingressi della **porta A** per verificare se anche le rispettive uscite si portano a **livello logico 1**.

Per portare a **livello logico 1** gli ingressi della **porta A** attivate la finestra **Data** cliccando sulla scritta corrispondente (vedi fig.103).

Nel paragrafo "Esecuzione in automatico" (vedi rivista N.184) vi abbiamo spiegato come trovare il valore **esadecimale** di un indirizzo di memoria, nel nostro caso il valore di **port_a**.
Poiché esiste anche una strada diversa per conoscere questo valore, riteniamo opportuno che la conosciate, e quindi ora ve la spieghiamo.

Nella barra degli strumenti visibile sulla parte alta del monitor cliccate sulla scritta **Data** e nella piccola finestra che appare selezionate la scritta **Goto Address** (vedi fig.104).
Apparirà la finestra di dialogo di fig.105.

Cliccate sulla **freccia giù** fino a trovare la variabile **port_a** e quando l'avete trovata selezionatela, quindi cliccate su **OK** (vedi fig.106).

Nella finestra **Data** verrà evidenziato il valore esadecimale **00** (vedi fig.107), che corrisponde al contenuto della variabile **port_a**.

Cliccate nuovamente sul menu **Data** e selezionate questa volta la scritta **Edit Data**.
Apparirà la finestra di dialogo di fig.108.

Cliccate sulla scritta **Bits** per far apparire la finestra di dialogo di fig.109.

Ora portate a **livello logico 1** i quattro piedini d'ingresso **PA0 - PA1 - PA2 - PA3** di **port_a**, cliccando con il cursore nelle caselle **0 - 1 - 2 - 3**.
In queste caselle apparirà una **V** (vedi fig.110).

Cliccate su **OK** per tornare alla finestra di dialogo **Edit Data**, in cui viene ora segnalato il nuovo valore esadecimale della variabile **port_a**, cioè **F**,

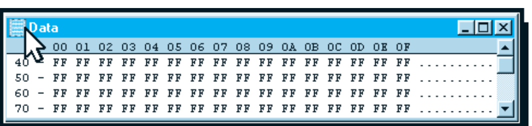


Fig.103 Per accedere al sottomenu relativo ad una finestra, nel nostro caso Data, occorre rendere attiva la finestra cliccando sul nome corrispondente, cioè Data.

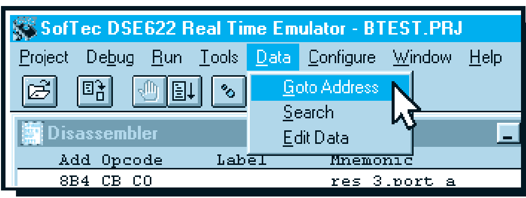


Fig.104 Scegliendo il sottomenu Goto Address potrete posizionarvi direttamente sul valore esadecimale dell'indirizzo di memoria in cui è memorizzata la variabile.

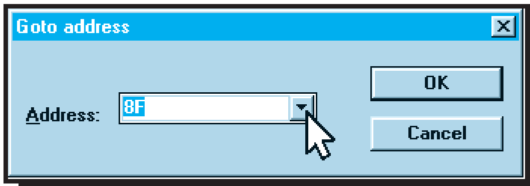


Fig.105 Nella riga Address cercate il nome della variabile cliccando sulla freccia giù.

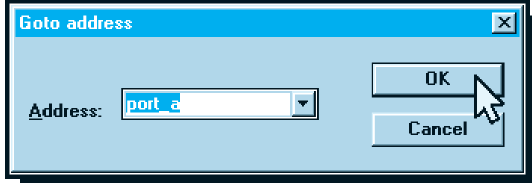


Fig.106 Quando avete selezionato la variabile cliccate su OK per tornare alla finestra Data, visibile in fig.107.

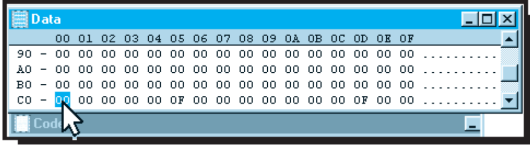


Fig.107 In questa finestra viene evidenziato il valore esadecimale dell'indirizzo della variabile selezionata nella fig.106.

che, come sapete, corrisponde al valore decimale **15** ed al valore binario **00001111**.
Cliccate nuovamente su **OK**.

Nota: Nel nostro volume **Handbook** a **pag.372** troverete un articolo dedicato al linguaggio **esadecimale - binario - decimale** e a **pag.381** una Tabella di **conversione** che potrà risultarvi molto utile.

A riprova di quanto detto nelle finestre **Watch** e **Data** vedrete il nuovo valore assunto da **port_a**.

A questo punto possiamo far ripartire il programma per verificare se, portando a **livello logico 1** un piedino d'ingresso di **port_a**, ritroviamo un **livello logico 1** anche sul corrispondente piedino d'uscita di **port_b**.

Se il programma non è posizionato sull'etichetta **ripeti** (vedi fig.101), attivate la finestra **Disassembler** cliccando sulla scritta corrispondente, quindi portate il cursore sulla scritta **Disassembler** sulla barra dei menu e cliccate così che appaia la finestra di fig.111 e selezionate l'opzione **Set New PC**.

Nella finestra di dialogo **New program counter** cliccate sulla **freccia giù** posta sulla destra fino a quando non trovate l'etichetta **ripeti**, quindi selezionatela e cliccate su **OK** (vedi fig.112).

Prima di far ripartire il programma sarà utile inserire un **breakpoint**.

Nella finestra **Disassembler** cliccate **due** volte su **ripeti** e quando compare la finestra di dialogo di fig.113 cliccate sulla scritta **Toggle Breakpoint**.
In questo modo a sinistra di questa istruzione apparirà un punto esclamativo (**!**) come visibile in fig.114.

A questo punto potete lanciare l'esecuzione **automatica** del programma cliccando sull'icona che rappresenta una **pagina** con una **freccia giù** (vedi fig.115).
Vi accorgerete che il programma, arrivato all'etichetta **mains3**, esegue in maniera ciclica e all'**infinito** (in gergo si dice che c'è un **loop**) un certo numero di istruzioni, e precisamente quelle relative alla gestione dei piedini **PA3** e **PB3**.
Questo evento ci mette sull'avviso che tra queste istruzioni c'è un errore.
Per bloccare l'esecuzione del ciclo, così da scoprire l'errore, cliccate sull'icona **stop** (vedi fig.116) all'istruzione:

8EF C3 C0 07 jrr 3, port_a,main03

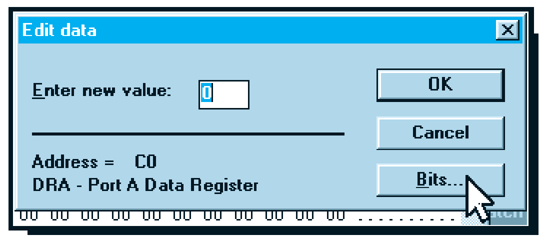


Fig.108 Questa finestra di dialogo appare quando si seleziona il sottomenu Edit Data dal menu Data. Per cambiare lo stato logico dei piedini cliccate su Bits.

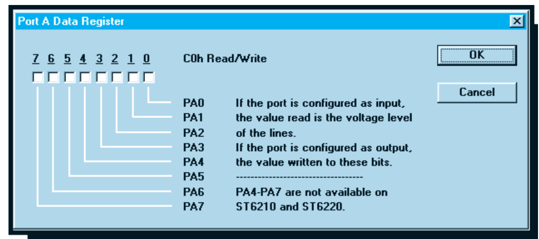


Fig.109 Le caselle che interessano i piedini PA0 - PA1 - PA2 - PA3 sono vuote perché questi piedini sono resettati.

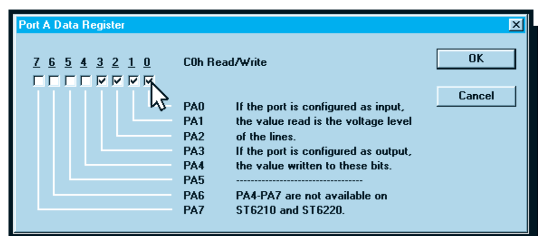


Fig.110 Per portare i piedini PA0 - PA1 - PA2 - PA3 a livello logico 1 cliccate sulle caselle corrispondenti.

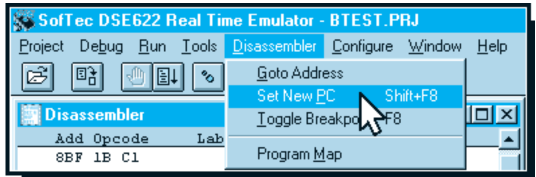


Fig.111 Il sottomenu Set New PC vi consente di aprire la finestra di dialogo New program counter (vedi fig.112).

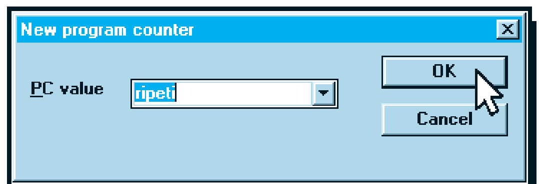


Fig.112 Grazie a questa finestra di dialogo potete far partire l'esecuzione del programma dall'etichetta ripeti.

Se cliccando sull'icona **stop** non riuscite a fermarvi su questa istruzione, utilizzate il comando **Set New Pc** del menu **Disassembler** (vedi fig.111). Nella finestra di dialogo che appare digitate l'indirizzo **8EF** (vedi fig.117) quindi cliccate su **OK.**

Ora conviene utilizzare il comando **passo-passo** per vedere dove abbiamo commesso l'**errore**.

L'istruzione memorizzata all'indirizzo **8EF** dice che se il piedino **PA3** della **porta A** è a **livello logico 0**, il programma deve saltare all'istruzione con etichetta **main03**.
Siccome però noi abbiamo posto questo piedino a **livello logico 1**, cliccando su **passo-passo** il programma dovrebbe proseguire all'istruzione successiva e **non saltare** all'etichetta **main03.**
Infatti cliccando su **passo-passo** il programma evidenzia l'istruzione successiva:

8F2 D3 C1 09 jrs 3,port_b,mains4

Questa istruzione controlla se il piedino **PB3** della **porta B** è a **livello logico 1**, e se si trova in questa condizione salta direttamente all'istruzione con etichetta **mains4**.
Cliccando su **passo-passo** il programma non salta a **mains4**, ma prosegue all'istruzione successiva perché deve prima portare a livello logico 1 il piedino **PB3**:

8F5 DB C0 set 3, port_a

Ecco dov'è l'**errore**. Infatti questa istruzione dovrebbe servire a settare il **bit 3** di **port_b**, cioè **PB3**, mentre noi abbiamo scritto erroneamente di settare il **bit 3** di **port_a**, cioè **PA3.**

In questo caso possiamo correggere l'istruzione apportando una modifica **temporanea**, così da poter continuare poi i nostri **test**, sostituendo all'indirizzo di **port_a** l'indirizzo di **port_b**.

Questa modifica **temporanea** ci consente di proseguire il controllo del programma dal punto in cui ci troviamo senza dover ripristinare tutti i parametri compreso il settaggio dei piedini delle porte.

Per capire come correggere questo **errore**, rivediamo l'istruzione sbagliata:

8F5 DB C0 set 3, port_a

L'istruzione **set 3,port_a** è tradotta dal compilatore in formato **intel.hex** nel valore esadecimale **DB C0**, che potete vedere nella finestra **Disassembler** sotto la colonna **Opcode** (vedi fig.118).

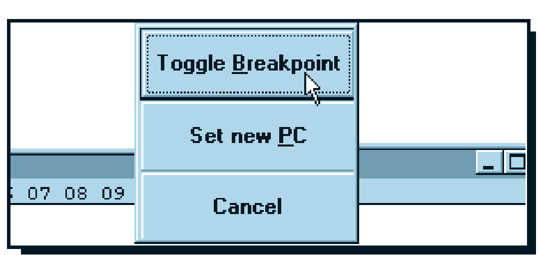


Fig.113 Se volete inserire un breakpoint in una determinata istruzione, cliccate due volte sull'istruzione corrispondente per attivare questo menu di scelta rapida.

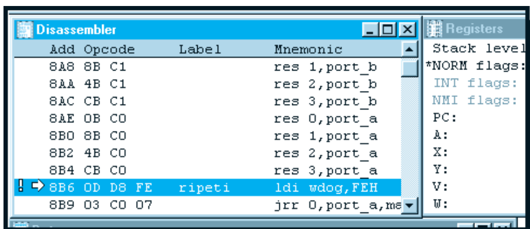


Fig.114 Cliccando su Toggle Breakpoint di fig.113, a sinistra dell'istruzione apparirà un punto esclamativo (!).

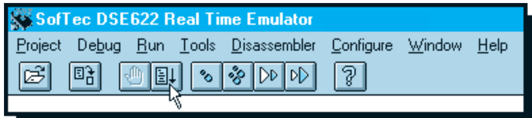


Fig.115 L'icona segnalata dal cursore permette di lanciare l'esecuzione automatica del programma.

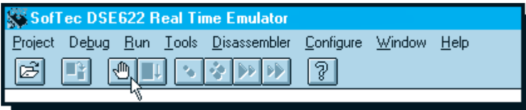


Fig.116 L'icona segnalata dal cursore permette di fermare l'esecuzione automatica del programma.

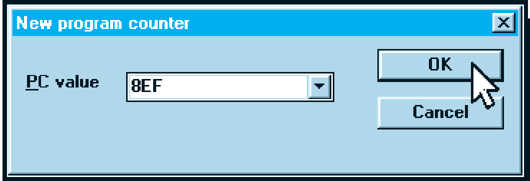


Fig.117 Come abbiamo già detto (figg.111-112), potete far ripartire l'esecuzione dal programma da qualsiasi punto. In questo caso dall'indirizzo 8EF.

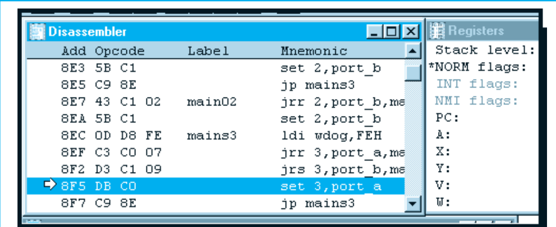


Fig.118 L'istruzione set 3 è tradotta dal compilatore nel valore DB e memorizzata all'indirizzo 8F5, port_a è tradotta nel valore C0 e memorizzata all'indirizzo 8F6 (vedi colonne Opcode e Add).

La stessa istruzione viene memorizzata all'interno del micro all'indirizzo **8F5** - **8F6** di **Program Space** (infatti se guardate la finestra del **Disassembler** sotto la colonna **Add**, l'istruzione seguente è memorizzata a partire dall'indirizzo **8F7**).

In questo caso il valore **DB** corrisponde all'istruzione **set 3** e si trova all'indirizzo **8F5**, mentre il valore **C0** corrisponde all'indirizzo dell'operando **port_a** e si trova all'indirizzo seguente, cioè **8F6**, pertanto è a questo indirizzo che dobbiamo operare la nostra modifica.

Sostituendo nell'indirizzo di memoria **8F6** il valore corrispondente a **port_a** (cioè **C0**) con il valore corrispondente a **port_b** (cioè **C1**) elimineremo questo **errore** senza modificare il sorgente **BTEST.ASM**.
Per conoscere gli indirizzi di **port_a** e **port_b** dovete guardare nella finestra **Watch** sotto la colonna **Add**.

Per andare all'indirizzo **8F6** attivate la finestra **Code** cliccando sul nome corrispondente, quindi dal menu **Code** della barra degli strumenti selezionate il comando **Goto Address** (vedi fig.119).
Nella finestra di dialogo digitate **8F6** (vedi fig.120), quindi cliccate su **OK**.
Ritornerete così nella finestra **Code** dove vedrete evidenziato il valore **C0** che dovete correggere (vedi fig.121).
Cliccate su questo valore **2 volte** e nella finestra di dialogo che appare (vedi fig.122) digitate **C1**, che come abbiamo già detto, è l'indirizzo di **port_b**.
Cliccate su **OK** e nella finestra **Disassembler** vedrete che l'istruzione si è modificata in:

8F5 DB C1 **set 3, port_b**

Ora che abbiamo corretto questo **errore** possiamo continuare la simulazione del programma premendo l'icona esecuzione **automatica**, cioè il disegno con la pagina e la freccia giù (vedi fig.115).
In questo modo il programma prosegue in modo automatico fino all'etichetta **ripeti**, dove, come ricorderete, abbiamo messo un **breakpoint**.

Ora dobbiamo verificare se effettivamente quando i **quattro** piedini d'ingresso (cioè **PA0 - PA1 - PA2 - PA3**) sono a **livello logico 1** anche i **quattro** piedini di uscita (cioè **PB0 - PB1 - PB2 - PB3**) si trovano a **livello logico 1**.
Abbiamo una riprova **visiva** di ciò guardando la finestra **Watch**, dove sia **port_a** sia **port_b** hanno lo stesso valore **esadecimale 0F** che corrisponde al valore **decimale 15** ed al valore **binario 00001111**, come potete anche vedere nella Tabella riportata a **pag.381** del nostro volume **HANDBOOK** che pensiamo sarà sempre a portata di mano.

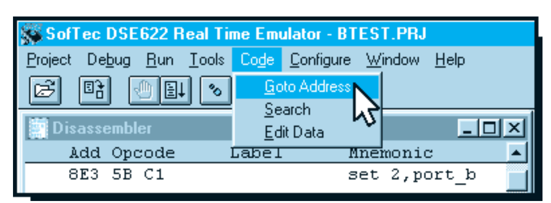


Fig.119 Selezionando il sottomenu Goto Address del menu Code aprite la finestra di dialogo visibile in fig.120.

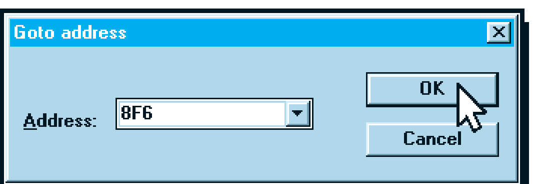


Fig.120 Poiché dovete correggere la variabile memorizzata all'indirizzo 8F6, digitate questo numero quindi date l'OK.

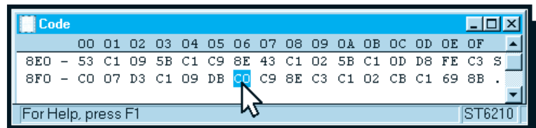


Fig.121 Nella finestra Code viene evidenziato il valore C0, che corrisponde alla variabile port_a. Cliccate due volte su C0.

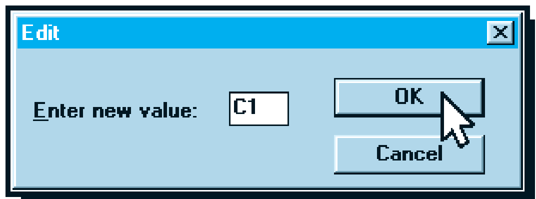


Fig.122 Nella finestra Edit digitate C1, cioè l'indirizzo della variabile che dovete sostituire che corrisponde a port_b.

Un altro modo per verificare lo stato dei piedini di **port_b** è ripetere la procedura eseguita all'inizio di questo paragrafo per portare a **livello logico 1** i piedini di **port_a**, sostituendo l'indirizzo di **port_a** con l'indirizzo di **port_b**, cioè sostituendo **C0** con **C1**.

Attivate la finestra **Data** cliccando sulla scritta corrispondente, quindi cliccate sul menu **Data** della barra dei menu e selezionate **Goto Address**.
Nella finestra di dialogo che appare cercate la variabile **port_b**, selezionatela e quindi cliccate su **OK**.
Nella finestra **Data** verrà evidenziato il valore esadecimale **0F** che corrisponde al contenuto della variabile **port_b**.
Cliccate nuovamente sul menu **Data** e questa volta selezionate **Edit Data**.
Nella finestra di dialogo che appare cliccate sulla scritta **Bits** per aprire la finestra di dialogo visibile in fig.123.

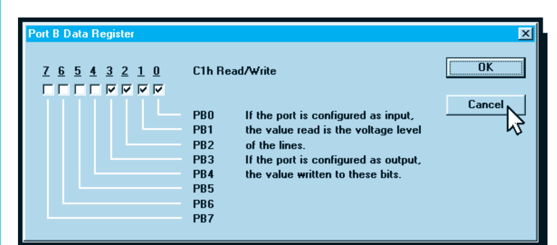


Fig.123 Come potete vedere tutti i piedini della porta B sono settati, cioè sono a livello logico 1. Uscite cliccando su Cancel.

Come potete verificare, ai 4 piedini settati della porta **A** corrispondono 4 piedini settati della porta **B** (infatti le caselle 0 - 1 - 2 - 3 hanno una V).

Uscite da questa finestra senza fare nessuna modifica cliccando sulla scritta **Cancel**.

Vi consigliamo di eseguire in automatico una o due volte il programma per essere certi di non aver modificato senza volere i valori di **port_a** e **port_b**.

Premete sull'icona esecuzione automatica e se tutto procede in modo regolare il programma si fermerà al **breakpoint**.

TERZO TEST

La seconda funzione del programma **BTEST** è quella di **resettare** i piedini di **port_b** quando vengono **resettati** i rispettivi piedini di **port_a**, in altre parole di portare a **livello logico 0** i piedini d'uscita quando vengono posti a **livello logico 0** i piedini d'ingresso.

Il terzo test si propone di verificare questa funzione.

Innanzitutto se nella finestra **Disassembler** non viene evidenziata l'etichetta **ripeti**, cliccate sulla scritta **Disassembler** nella barra dei menu e selezionate **Set New PC** (vedi fig.111).
Nella finestra di dialogo che appare cercate l'etichetta **ripeti** cliccando sulla **freccia giù** e quando l'avrete trovata selezionatela quindi cliccate su **OK** (vedi fig.112).

Per resettare i piedini della **porta A** attivate la finestra **Data** cliccando sul nome corrispondente.
Cliccate sulla scritta **Data** della barra dei menu in modo che appaia la piccola finestra visibile in fig.124 e selezionate **Goto Address**.
Se nella finestra di dialogo non compare **C0**, digitate manualmente questo indirizzo (vedi fig.125) e cliccate su **OK**.
Ora cliccate **due volte** sul valore **0F** evidenziato nella finestra **Data** e nella finestra di dialogo che appare cliccate su **Bits**.
Cliccando nelle caselle in cui appare la **V**, riportate tutti i piedini d'ingresso di **port_a** a **livello logico 0** (le caselle 0 - 1 - 2 - 3 devono essere vuote) come visibile in fig.126.
A questo punto cliccate su **OK** per tornare alla finestra principale del **DSE**.

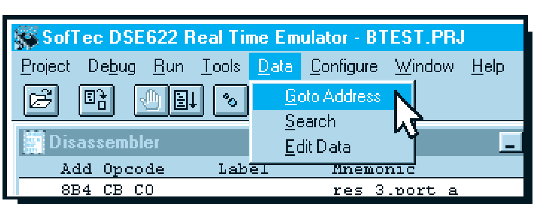


Fig.124 Per portare a livello logico 0 i piedini della porta A, dal menu Data scegliete Goto Address.

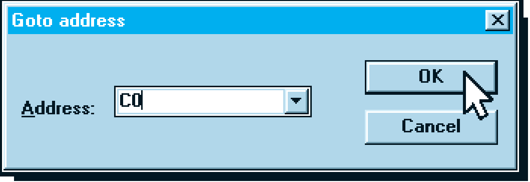


Fig.125 Poiché l'indirizzo C0 corrisponde a port_a, digitate questo valore quindi cliccate su OK.

Ora potete eseguire il programma in **automatico**, cliccando cioè sull'icona con una **pagina** con la freccia in giù (vedi fig.115), e vedrete che si fermerà al **breakpoint**.
Si potrebbe supporre che il programma rispetti anche questa funzione, ma se guardate nella finestra **Watch** (vedi fig.127) noterete che pur essendo tutti i piedini di **port_a** a **livello logico 0** (valore esadecimale 00) i piedini di **port_b** non sono tutti a livello logico 0 (valore esadecimale **04**).

Nel programma c'è quindi un errore.

Se non sapete a quale valore **binario** corrisponde il valore **esadecimale 04**, cioè se non sapete quale bit è a livello logico 1, attivate la finestra **Data** e dal menu **Data** selezionate **Goto Address**.
Nella finestra di dialogo che appare digitate l'indirizzo di **port_b**, cioè **C1** (vedi fig.128), poi cliccate su **OK**.
Quindi cliccate **due volte** sul valore **04** evidenziato nella finestra **Data** e nella finestra di dialogo che appare selezionate la scritta **Bits**.
Come visibile in fig.129, il piedino **PB2** risulta ancora a **livello logico 1** sebbene l'ingresso corrispondente, cioè **PA2**, risulti a **livello logico 0**.

E' dunque ovvio che nel programma c'è una istruzione **sbagliata**, che non porta a **livello logico 0** il piedino **PB2** quando il corrispondente piedino d'ingresso **PA2** è a **livello logico 0**.

Senza modificare nulla chiudete le finestre di dialogo cliccando su **Cancel**.

La gestione dei piedini PA2 e PB2 è associata all'etichetta **mains2**, quindi ricontrolliamo il programma partendo da questa etichetta.

Attivate la finestra **Disassembler** cliccando sulla scritta corrispondente, quindi dal menu Disassembler della barra dei menu selezionate la scritta **Set New PC** (vedi fig.111).
Cliccando sulla **freccia giù** cercate la scritta **mains2** e selezionatela (vedi fig.130), poi cliccate su **OK**.

Il programma si posizionerà sull'istruzione:

8DA 0D D8 FE mains2 ldi wdog,0FEH

che serve a caricare il **watchdog**.

A questo punto cliccate sull'icona **passo-passo** e verrà evidenziata l'istruzione successiva:

8DD 43 C0 C7 jrr 2,port_a,main02

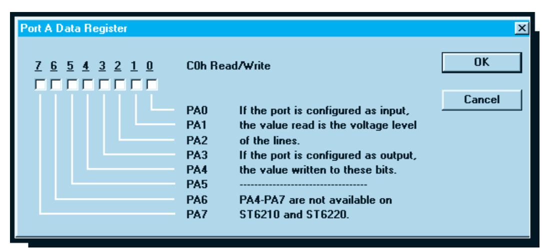


Fig.126 Per resettare i piedini di porta A cliccate sulle caselle 0 - 1 - 2 - 3.

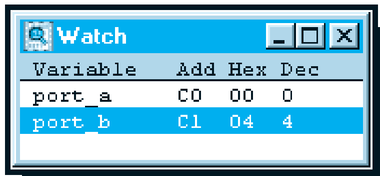


Fig.127 Controllando la finestra Watch vi accorgerete che sebbene il contenuto di port_a sia 00, il contenuto di port_b è 04.

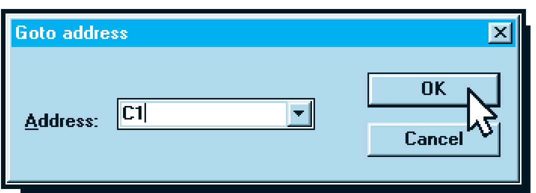


Fig.128 Per sapere quale bit della porta B è a livello logico 1, in Goto Address digitate C1, che è l'indirizzo di port_b.

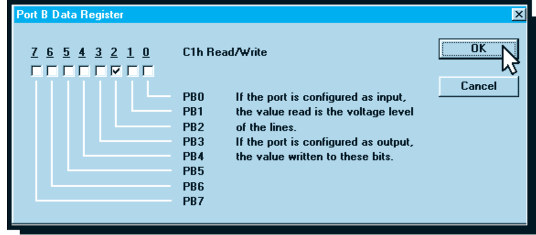


Fig.129 Il valore esadecimale 04 che avevate visto nella finestra Watch (fig.127) corrisponde al piedino PB2 settato.

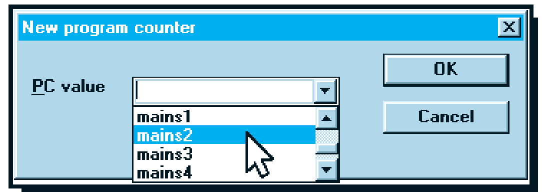


Fig.130 Ricontrollate il programma partendo dall'etichetta mains2, che gestisce i piedini PA2 e PB2.

Questa istruzione significa: salta a **main02** se il piedino **2** di **port_a** è **resettato**, cioè se **PA2** si trova a **livello logico 0**.
Siccome avevamo posto a **livello logico 0** tutti gli ingressi della **porta A**, cliccando su **passo-passo** il programma salterà all'istruzione corrispondente all'etichetta **main02**:

8E7 43 C1 02 main02 jrr 2,port_b,mains3

Questa istruzione significa: salta a **mains3** se il piedino **2** di **port_b** è **resettato**, cioè se **PB2** è a **livello logico 0**.
In altre parole verifica lo **stato logico** del piedino **2** di **port_b**, perché se questo risulta già a **livello logico 0** non lo resetta nuovamente.
Tuttavia noi sappiamo già che questo piedino è rimasto a **livello logico 1**, perché l'abbiamo controllato tramite la finestra **Watch** (vedi figg.127 e 129).
Di conseguenza cliccando su **passo-passo** il programma non salta a **mains3**, ma prosegue all'istruzione successiva che **resetta** il piedino **PB2**:

8EA 5B C1 set 2,port_b

Ecco dov'è l'**errore**: infatti il piedino non deve essere **settato**, ma **resettato** quindi l'istruzione **errata** è **set** e quella giusta è: **res 2,port_b**

Su questa istruzione possiamo apportare una correzione **temporanea** anche se la correzione risulta un poco più **complessa**, perché, come avrete già intuito, non dobbiamo modificare la parte dell'**opcode** che si riferisce all'**operando**, ma quella che contiene l'**istruzione** vera e propria.

L'**opcode** dell'istruzione **SET** è (vedi tabella a pag.103):

b11011 rr (Setta un piedino)

L'opcode dell'istruzione **RES** (vedi tabella a pag.102) è invece:

b01011 rr (Resetta un piedino)

Il **secondo byte** di queste istruzioni è dato da **rr**, che corrisponde all'indirizzo dell'**operando**, nel nostro caso **port_b**, cioè **C1**.
Il **primo byte** dell'istruzione **RES** è dato da **b+01011**, dove **b** equivale a **3 bit** che definiscono quale **bit** dell'operando da **0** a **7** deve essere **resettato** mentre **01011** equivale all'istruzione **res** in **binario**.

In altre parole con un numero **binario** di **8 bit** riusciamo a definire l'**istruzione**, nel nostro caso **RES** che occupa i primi **5 bit** da **0** a **4**, ed il **piedino** a cui l'istruzione si riferisce, nel nostro caso il **bit 2** che occupa gli ultimi **3 bit** da **5** a **7.**

posizione bit	7	6	5	4	3	2	1	0
codice opcode	—	b	—	0	1	0	1	1

Poiché dovete **resettare** il **bit 2**, per trasformare questo numero **decimale** in un numero **binario** potete utilizzare la Tabella sotto riportata.

Tabella N.1

DECIMALE	BINARIO		
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Come potete vedere il numero **decimale 2** corrisponde al numero **binario 0 1 0.**
Pertanto per trasformare l'istruzione **Set 2**, che è **errata**, cioè:

010 11011 (i **3 bit** di sinistra sono **b** dell'opcode)

nella corretta istruzione **Res 2**, dobbiamo considerare questo numero binario:

010 01011

Per convertire questo numero **binario** in un numero **esadecimale** potete utilizzare le tabelle di conversione a **pag.381** del nostro volume **Handbook**.
Se non disponete di questo **Handbook** vi conviene procurarvelo perché troverete spiegato come si fa a **convertire** un numero **binario** in un numero **decimale** o **esadecimale**.

Nelle Tabelle dell'**Handbook** potete vedere che l'istruzione **Set 2**:

0101-1011 equivale al numero **esadecimale 5B**

e che l'istruzione di **Res 2**:

0100-1011 equivale al numero **esadecimale 4B**

Ora che sapete come correggere l'istruzione, attivate la finestra **Code** quindi dal menu **Code** selezionate il comando **Goto Address** e nella finestra

di dialogo che appare digitate **8EA** (vedi fig.131), che è l'indirizzo di memoria **Program Space** corrispondente all'istruzione **set 2, port_b**.
Potete vedere questo indirizzo nella finestra **Disassembler** sotto la colonna **Add**.
Se cliccate su **OK**, nella finestra **Code** verrà evidenziato il numero **5B** (vedi fig.132).
Cliccate **due volte** su questo numero quindi nella finestra di dialogo che appare scrivete il nuovo valore, cioè **4B** (vedi fig.133).

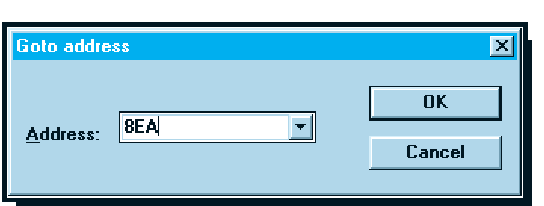


Fig.131 Poiché dovete correggere l'istruzione set 2, port_b, nella finestra Goto Address digitate 8EA, che è l'indirizzo di memoria Program Space di questa istruzione. Quindi cliccate su OK.

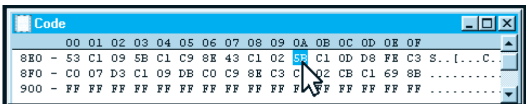


Fig.132 Nella finestra Code viene evidenziato 5B, che è il valore esadecimale corrispondente all'istruzione set 2. Cliccate due volte su questo valore, per aprire la finestra di dialogo visibile in fig.133.

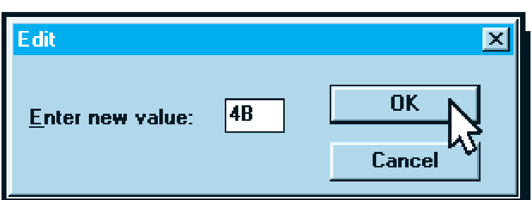


Fig.133 Ora potete correggere il valore esadecimale 5B, che equivale a set 2, con il valore esadecimale 4B, che equivale all'istruzione res 2.

Cliccate su **OK** e nella finestra **Disassembler** vedrete che questa istruzione sarà stata modificata come sotto riportato:

8EA 4B C1 **res 2, port_b**

Cliccate sull'icona **passo-passo** per far eseguire l'istruzione e nella finestra **Watch** vedrete che il valore esadecimale di **port_b** è diventato **00**.

Cliccate ora sull'icona esecuzione **automatica** (vedi fig.115) ed il programma, eseguita qualche istruzione, si fermerà di nuovo al **breakpoint**.
Per essere sicuri di non aver variato altri valori mentre correggevate l'istruzione, è meglio far eseguire il programma un paio di volte cliccando sempre su esecuzione **automatica**.
Se non esistono altri **errori** il programma eseguirà un ciclo completo e si fermerà sempre al **breakpoint**.
Durante questa fase potrete osservare nella finestra **Watch** che le variabili **port_a** e **port_b** non cambiano di valore.

Anche se il programma ci conferma che non esistono altri **errori** e tutto procede regolarmente, sappiamo che esiste un altro **errore**, che noi abbiamo volutamente inserito nel file **BTEST.ASM.**
Per poter scoprire quest'ultimo **errore** occorre necessariamente fare un **quarto test**.

QUARTO TEST

Il **terzo errore** da noi inserito riguarda un passaggio insidioso e molto **subdolo**, perché non cambia la **logica** dell'esecuzione quindi potrebbe **non essere mai trovato** da chi non ha molta esperienza.
Infatti malgrado ci sia questo **errore** il programma funziona correttamente.

Posizionatevi sull'etichetta **ripeti** e lasciate il **breakpoint**.
Guardate nella finestra **Watch** dove le variabili **port_a** e **port_b** dovrebbero essere a **0**.

Per trovare questo **errore** dovete riportare a **livello logico 1** il piedino **PA3** di **port_a**
Riteniamo che la procedura per settare i piedini di una porta vi sia già familiare (vedi figg.104-110), comunque, nel caso ancora non aveste preso confidenza con il programma, dovete attivare la finestra **Data** ed utilizzare il comando **Goto Address** del menu **Data**. Digitate il valore di **port_a,** cioè **C0**, quindi, dopo aver dato l'**OK**, cliccate **due volte** su **00** poi nella finestra di dialogo che appare cliccate su **Bits**.

Dopo aver portato a **livello logico 1** il piedino **PA3** potrete uscire cliccando su **OK**.

In questo modo tornate nella finestra principale e vedrete nella finestra **Watch** che la variabile **port_a** ha valore **8** (vedi fig.134).

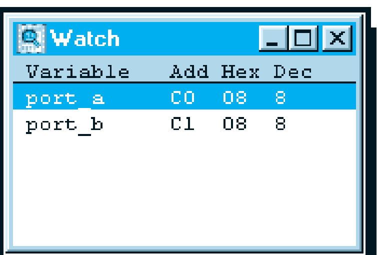


Fig.134 Avendo portato a livello logico 1 il piedino PA3, nella finestra Watch viene evidenziato il nuovo contenuto di port_a.

A questo punto rieseguite il programma **passo-passo** fino all'istruzione:

8EC 0D D8 FE mains3 ldi wdog, FEH

che serve a caricare il **watchdog**.
Premete ancora sull'icona **passo-passo** per passare alla istruzione successiva:

8EF C3 C0 07 jrr 3,port_a, main03

Questa istruzione significa: se il piedino **3** di **port_a** è **resettato**, cioè a **livello logico 0**, il programma deve saltare all'istruzione con etichetta **main03**.
Poiché abbiamo appena posto questo piedino a **livello logico 1**, il programma non salta a **main03** ma prosegue all'istruzione successiva ed infatti cliccando sull'icona **passo-passo** viene evidenziata:

8F2 D3 C1 09 jrs 3, port_b, mains4

Questa istruzione dice che se il piedino **3** di **port_b** è **settato**, cioè è a **livello logico 1**, il programma deve saltare all'istruzione con etichetta **mains4**.
Poiché questo piedino è a **livello logico 0** (come possiamo vedere nella finestra **Watch**), cliccando **passo-passo** il programma prosegue all'istruzione successiva:

8F5 DB C1 set 3, port_b

Questa istruzione **setta** il piedino d'uscita **PB3** e cliccando su **passo-passo** viene evidenziata:

8F7 C9 8E jp mains3

Il programma ha svolto regolarmente la sua funzione: controlla se **PA3** è **settato**, cioè se si trova a **livello logico 1**, poi controlla ed eventualmente modifica la porta d'uscita **PB3.**
Per averne una verifica immediata controllate il valore della variabile **port_b** nella finestra **Watch** e vedrete che è **8** (vedi fig.134).

A questo punto il programma dovrebbe proseguire andando a controllare gli ultimi piedini, cioè **PA4** e **PB4**, quindi dovrebbe saltare all'istruzione con etichetta **mains4**, ma il realtà l'ultima istruzione esegue un salto incondizionato all'etichetta **mains3** come potrete constatare cliccando nuovamente su **passo-passo**:

8EC 0D D8 FE mains3 ldi wdog,FEH

Cliccando sull'icona **passo-passo** viene evidenziata l'istruzione:

8EF C3 C0 07 jrr 3,port_a, main03

Poiché il piedino **PA3** è sempre **settato** cliccando **passo-passo** il programma prosegue all'istruzione successiva:

8F2 D3 C1 09 jrs 3, port_b, mains4

A questo punto però **PB3** è già stato **settato**, quindi premendo **passo-passo** il programma prosegue all'istruzione con etichetta **mains4** per controllare lo **stato logico** degli ultimi piedini.

In pratica il programma esegue **due volte** una serie d'istruzioni per controllare i piedini **PA3** e **PB3.**
Questo **doppio controllo** non crea nessun problema sulla funzionalità, però se un domani apporterete delle modifiche al programma **inserendo** altre istruzioni proprio tra **queste ultime** righe che abbiamo analizzato, questo potrebbe crearvi dei grossi problemi e potrebbe diventare difficile individuare l'errore.

Ad esempio se inserite un **contatore** che si **incrementa** di **una unità** ogni volta che il programma controlla i **quattro piedini**, constaterete che mentre per gli altri piedini la **somma** si incrementa di **una unità**, per la routine del piedino **PA3** si incrementa di **due unità** perché esegue per **due volte** consecutive questa routine.

Per rintracciare questo **errore** non è necessario che chi scrive il programma sappia a memoria tutte le istruzioni, ma è importante che abbia ben chiaro lo **schema logico**.

E’ per questo motivo che questi tipi di **errori** sono difficili da individuare.

Per concludere l’**errore subdolo** è l’istruzione:

8F7 C9 8E **jp mains3**

che fa ripetere per **due volte** consecutive questa routine.
L’istruzione va corretta con **jp mains4**.

Conoscendo i **3 errori** da noi inseriti nel programma **BTEST.PRJ**, possiamo andare direttamente nel file **BTEST.ASM** per correggerli tutti definitivamente.
Nel caso voleste conservare gli errori presenti in questo programma per eventuali **test**, prima di apportare le modifiche copiate il file **BTEST.ASM** con un altro nome, ad esempio **CTEST.ASM**, utilizzando le funzioni di **copia** di **Windows**.

Per fare le **correzioni** andate direttamente nell’editor di **ST6** selezionando dal menu **Tools** la scritta **ST6** (vedi fig.135).
Apparirà così la finestra dell’editor di **ST6**.
Premete **F3** per aprire il file e nella finestra di dialogo che appare selezionate **BTEST.ASM** (vedi fig.136) quindi cliccate su **Open**.
Appariranno sul vostro monitor le istruzioni del programma **BTEST.ASM**.
Utilizzate la freccia giù fino all’istruzione da noi numerata come **87** (accanto a questo numero vedrete anche un asterisco) che corrisponde al numero **147:1** dell’editor.
Modificate **set 2,port_b** in **res 2,port_b**.

Ora proseguite fino al secondo asterisco ***91** (che corrisponde a **153:1**) e modificate **set 3,port_a** in **set 3,port_b**.

Infine andate al terzo asterisco ***92** (che corrisponde a **154:1**) e modificate **jp mains3** in **jp mains4**.

Nella fig.137 potete vedere la parte del programma con le istruzioni già corrette.

Tutte le modifiche devono essere **salvate** pigiando semplicemente il tasto funzione **F2**.

Poiché avete apportato delle correzioni al **sorgente** del programma, dovete ricompilarlo.
Cliccate quindi sul menu **ST6** e su **Assembla** (vedi fig.138).
Se avete apportato tutte queste modifiche in maniera corretta la **compilazione** si concluderà regolarmente.

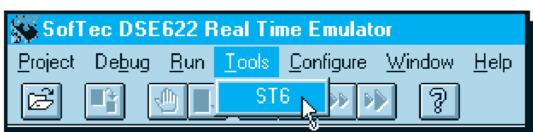


Fig.135 Per correggere in maniera definitiva il programma BTEST è necessario apportare le correzioni nel sorgente, cioè nel file BTEST.ASM. Dal DSE è possibile accedere direttamente all’editor dell’ST6, scegliendo ST6 dal menu Tools.

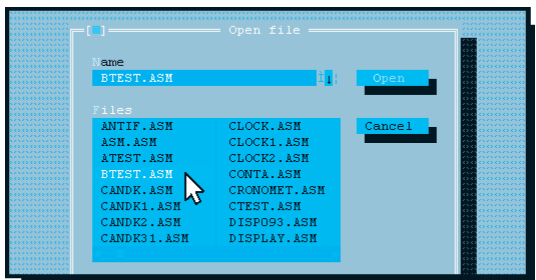


Fig.136 Per aprire un file nell’editor dell’ST6 potete usare il tasto funzione F3, che attiva questa finestra di dialogo. Selezionate con il cursore file BTEST.ASM, quindi cliccate su Open.

```
main02  jrr     2,port_b,mains3         ;86 salta a mains3 se
        res     2,port_b                ;*87 mette a 0 PB2

mains3
        ldi     wdog,0feh               ;88 ricarica il watchd
        jrr     3,port_a,main03         ;89 salta a main41 se
        jrs     3,port_b,mains4         ;90 salta a main41 se
        set     3,port_b                ;*91 mette a 1 PB3
        jp      mains4                  ;*92 salta a mains4
main03  jrr     3,port_b,mains4         ;93 salta a mains4 se
```

Fig.137 In questa figura potete vedere la parte del programma BTEST.ASM con le istruzioni già corrette.

Fig.138 Prima di chiudere il file dovete ricompilare il programma BTEST.ASM, quindi dal menu ST6 scegliete Assembla.

Per tornare all'**editor** premete un tasto qualsiasi poi premete **ALT+F3** per chiudere il programma **BTEST.ASM** ed **Alt+X** per uscire dall'**editor** di **ST6**.
Rientrerete così nella finestra del **DSE622** aperta su **BTEST.PRJ** e sul video comparirà un messaggio che vi informa del fatto che il **project** ha una data precedente al **sorgente** del programma (vedi fig.139).

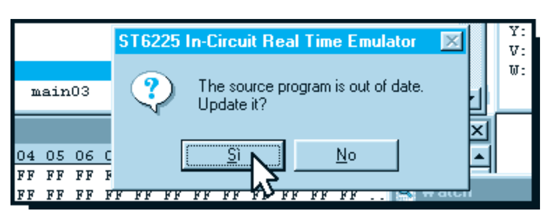


Fig.139 Per aggiornare le modifiche anche sul file BTEST.PRJ dovete cliccare su Sì.

Infatti avendo appena ricompilato il programma, il **project** che è attualmente attivo su **DSE622** verrà aggiornato solo cliccando su **Sì**.
In questo modo le modifiche apportate diventeranno definitive anche in **BTEST.PRJ**.
Comparirà un altro messaggio che vi avvisa che l'aggiornamento del **project** annullerà i **breakpoint** (vedi fig.140) e a questo punto potrete cliccare su **OK**.

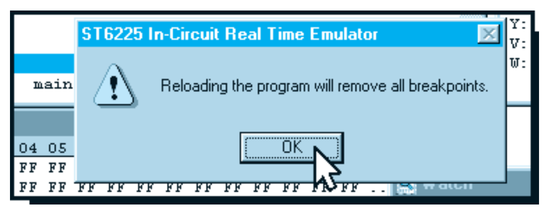


Fig.140 Il DSE622 vi informa che l'aggiornamento del file .PRJ annullerà i breakpoint. Cliccate pure su OK.

Rileggendo quanto fin qui scritto ci siamo accorti che la spiegazione per cercare e **correggere** gli **errori** con il simulatore **DSE** è stata abbastanza lunga, ma in questo modo siamo certi che questo articolo risulterà per voi molto utile perché ora sapete in quale finestra dovete controllare i diversi **livelli logici**, come si fa per trasformare un numero **esadecimale** in un **decimale** o in un **binario**, e se inizialmente tutto questo vi sembrerà **difficile** e **complesso** con un po' di **pratica** capirete quanto invece risulti facile e semplice.
Anche la primissime volte che avete iniziato ad andare in **bicicletta** dover rimanere in equilibrio, pedalare e fermarsi, potevano sembrare manovre difficilissime, poi con un poco di perseveranza e di pratica riuscite ora a pedalare anche controllando il manubrio con una sola mano.

Alcuni lettori ci hanno inviato valide soluzioni per far girare sotto Windows 95 il programma ST6PGM della SGS-Thomson e per richiamare velocemente il sistema operativo MS-DOS. Noi ve le proponiamo per poter risolvere i problemi che ora riscontrate.

Windows 95 e ST6

Sulla rivista N.183 vi avevamo proposto una veloce soluzione per riuscire a caricare ed utilizzare, pur avendo installato WINDOWS 95, i programmi che utilizzano il sistema operativo MS-DOS 6.2.

Infatti a causa dei problemi incontrati nel caricare i “vecchi” programmi qualcuno aveva addirittura abbandonato **WINDOWS 95** ed era ritornato a **Windows 3.1**.
Tra i nostri lettori però ci sono anche dei softwaristi molto esperti che hanno cercato e trovato soluzioni alternative alla nostra e subito hanno provveduto a segnalarcele affinché potessimo renderle di dominio pubblico tramite la rivista.

Tra le tante lettere che ci sono pervenute ve ne proponiamo oggi due che ci sembrano particolarmente utili ed interessanti, ma non escludiamo di pubblicare anche le altre nei prossimi numeri.
Fin da oggi desideriamo **ringraziare** tutti questi lettori per la loro collaborazione.

Sig. Luca Montefiore - Teramo

La prima proposta ci viene dal Sig. **Luca Montefiore** che è riuscito a lanciare il programma **ST6PGM.BAT**, scritto per i microprocessori **ST6**, sotto **Windows 95** aggiungendo semplicemente una riga di istruzione al file **CONFIG.SYS**.

Se anche a voi interessa aggiungere questa riga dovete procedere come segue:

– Quando siete in **Windows 95** portate il cursore sulla scritta **Avvio** (vedi fig.1) e cliccate. Nella finestra che appare scegliete **Programmi** e nel menu a destra portate il cursore su **Gestione Risorse** quindi cliccate (vedi fig.2).

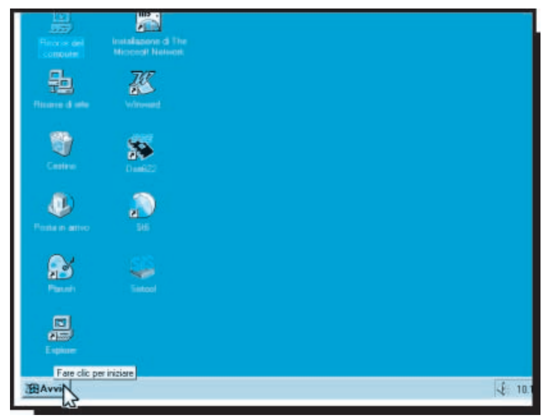


Fig. 1

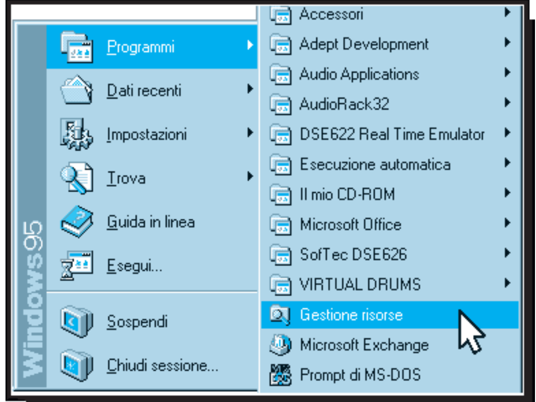


Fig. 2

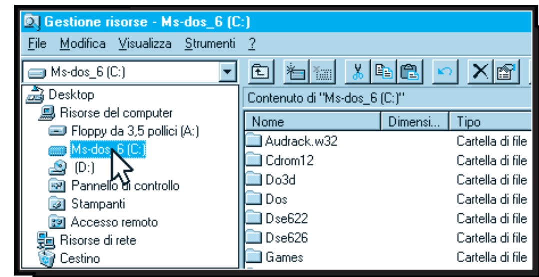


Fig. 3

– A sinistra della finestra che appare selezionate l'unità **Ms-dos_6 (C)** (vedi fig.3), quindi attivate il menu a tendina di **Visualizza** e cliccate su **Opzioni** (vedi fig.4).

– Nella finestra di dialogo che appare scegliete **Tutti i file** cliccando con il mouse sul cerchietto visibile in fig.5, poi portate il cursore sulla scritta **OK** e cliccate. Nella finestra a destra vedrete apparire tutti i file, compresi quelli **nascosti**.

– Utilizzate il tasto freccia giù per cercare il file **CONFIG.SYS** e quando l'avete trovato selezionatelo cliccando una **sola volta** con il tasto **destro** del mouse.

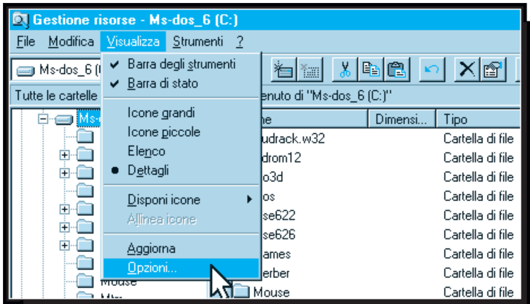


Fig. 4

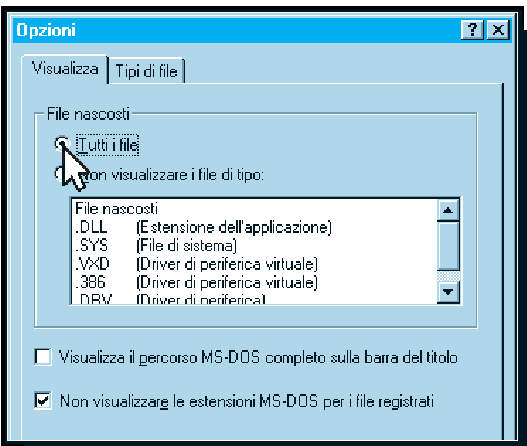


Fig. 5

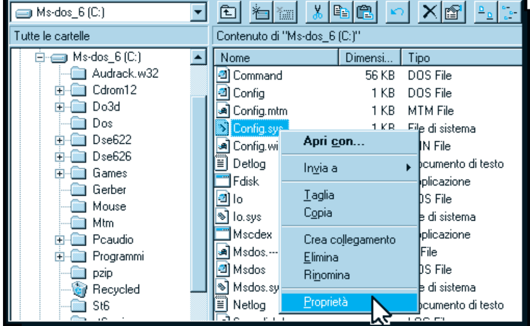


Fig. 6

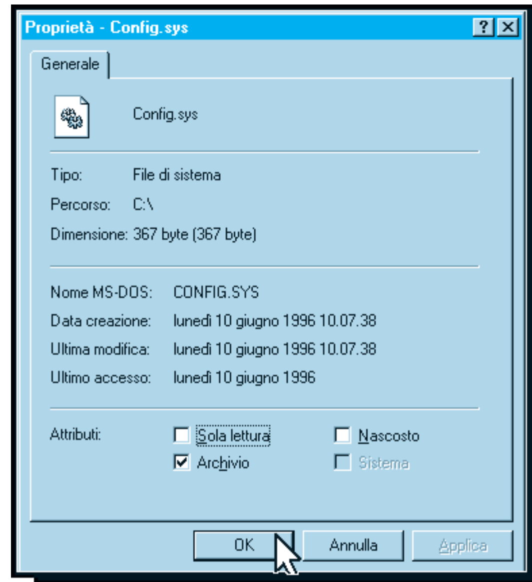


Fig. 7

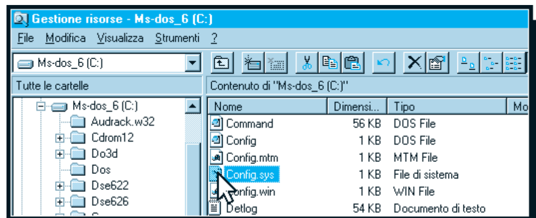


Fig. 8

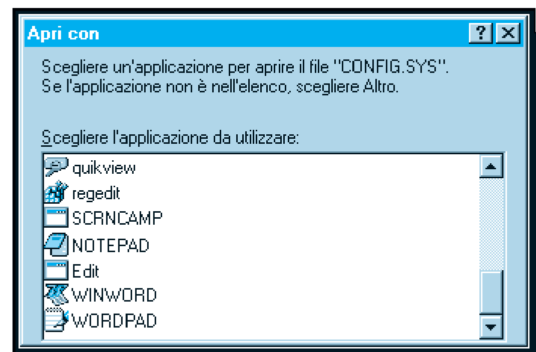


Fig. 9

– Nel menu che appare cliccate sulla scritta **Proprietà** (vedi fig.6) e quando appare la finestra di dialogo visibile in fig.7 controllate che **non** siano selezionate le opzioni **solo lettura** e **nascosto**, nel qual caso cliccate nelle rispettive caselle per togliere la selezione.

– Dopo questa verifica cliccate su **OK** (vedi fig.7) per tornare al file **CONFIG.SYS** (vedi fig.8) che sarà ancora selezionato e cliccate **due volte** ma con il tasto **sinistro** del Mouse.

– Apparirà la finestra di dialogo **Apri con** (vedi fig.9) in cui dovrete selezionare uno di questi programmi

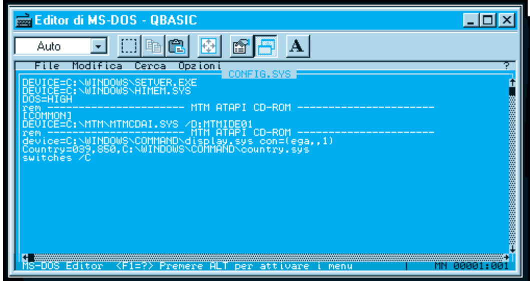


Fig. 10

di gestione testi, **EDIT**, **NOTEPAD**, **WINWORD**. Poiché questi programmi sono equivalenti potrete indifferentemente scegliere l'uno o l'altro cliccando **due volte** sul nome corrispondente.
Noi abbiamo usato **Edit**.

– Nell'ultima riga del programma dovete inserire l'istruzione **SWITCHES /C** come visibile in fig.10.

– Per salvare il file cliccate sul menu **File**, poi cliccate sulla scritta **SALVA** (vedi fig.11) ed uscite.

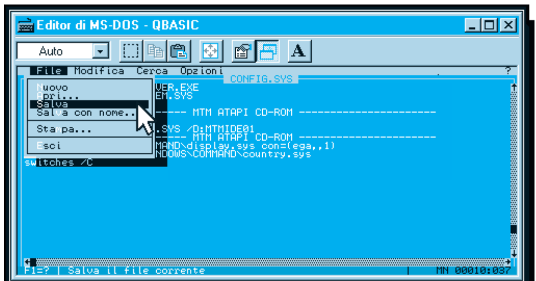


Fig. 11

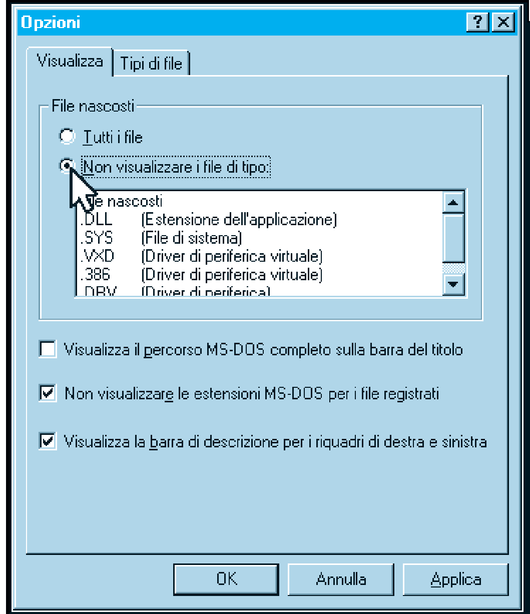


Fig. 12

– A questo punto dovete nuovamente **ripristinare** l'opzione di **file nascosti**, perché se questo file di sistema rimane visualizzato potreste per errore cancellarlo o modificarlo.

– Sulla barra dei menu posta in alto cliccate su **Visualizza** e dal menu che appare scegliete la scritta **Opzioni** e cliccate nuovamente. Nella finestra che appare cliccate sul **cerchietto** posto a sinistra della scritta **Non visualizzare i file tipo** (vedi fig.12) in modo che appaia un **punto**. Per chiudere questa finestra cliccate su **OK**.

– Uscite dal programma **Gestione Risorse**, chiudete **Windows 95** quindi ricaricatelo.
A questo punto la modifica proposta dal Sig. Montefiore sarà operativa.

Sig. Fabio Chiribiri - Marola (La Spezia)

La seconda proposta che sottoponiamo alla vostra attenzione ci viene dal Sig. **Fabio Chiribiri** che è riuscito a richiamare il sistema operativo **MS-DOS 6.2** senza utilizzare il tasto funzione **F8**.

Sulla rivista **N.183** vi avevamo spiegato che premendo il tasto funzione **F8** si attivava un menu col quale era possibile scegliere tra varie modalità di caricamento sia di **WINDOWS** sia di **MS-DOS**.
Questo tasto doveva essere premuto al momento giusto altrimenti il computer si poteva bloccare.

Il Sig. **Chiribiri** ci ha spiegato che modificando il file **MSDOS.SYS** si può fare a meno di premere il tasto funzione **F8**.
Per modificare il programma **MSDOS.SYS** dovete procedere come segue:

– Quando siete in **Windows 95** portate il cursore sulla scritta **Avvio** (vedi fig.1) e cliccate. Nella finestra che appare scegliete **Programmi** e nel menu a destra portate il cursore su **Gestione Risorse** quindi cliccate (vedi fig.13).

– A sinistra della finestra che appare selezionate l'unità **Ms-dos_6 (C)** (vedi fig.14). Ora attivate il menu a tendina di **Visualizza** e cliccate su **Opzioni** (vedi fig.15).

– Nella finestra di dialogo che appare scegliete **Tutti i file** cliccando con il mouse sul cerchietto visibile in fig.16, poi portate il cursore sulla scritta **OK** e cliccate. Nella finestra a destra vedrete apparire tutti i file, compresi quelli **nascosti**.

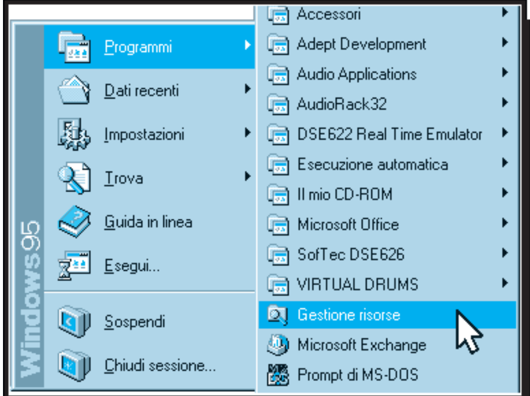


Fig. 13

Fig. 14

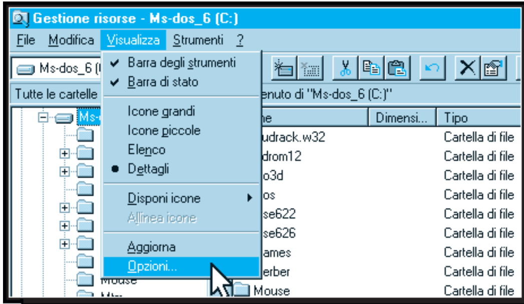


Fig. 15

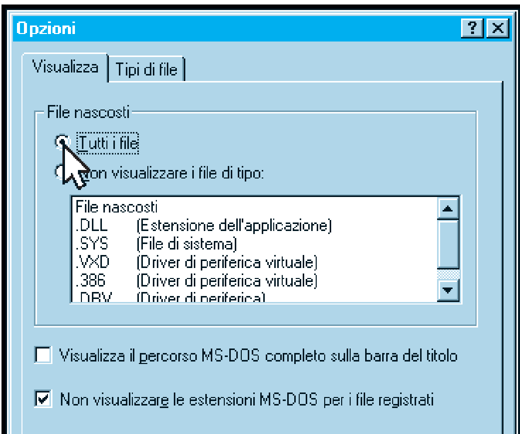


Fig. 16

– Utilizzate il tasto freccia giù per cercare il file **MSDOS.SYS** e quando l'avrete trovato selezionatelo cliccando una **sola volta** con il tasto **destro** del mouse.

– Nel menu che appare cliccate sulla scritta **Proprietà** (vedi fig.17) e quando appare la finestra di dialogo visibile in fig.18 controllate che **non** siano selezionate le opzioni **solo lettura** e **nascosto**, nel qual caso cliccate nelle rispettive caselle per togliere la selezione.

– Dopo questa verifica cliccate su **OK** (vedi fig.18) per tornare al file **MSDOS.SYS** (vedi fig.19) che sarà ancora selezionato e cliccate **due volte** ma con il tasto **sinistro** del Mouse.

Fig. 17

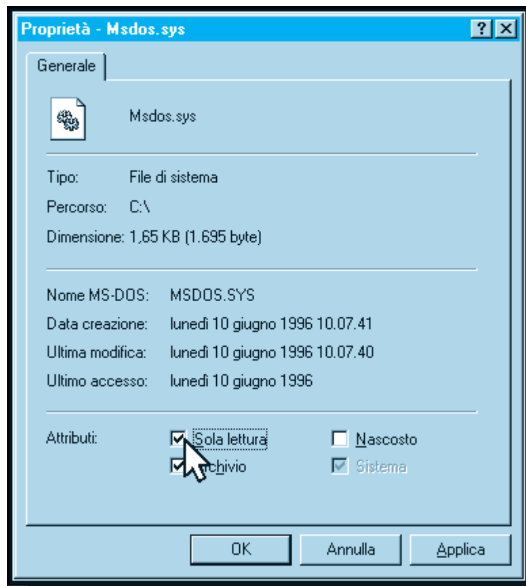


Fig. 18

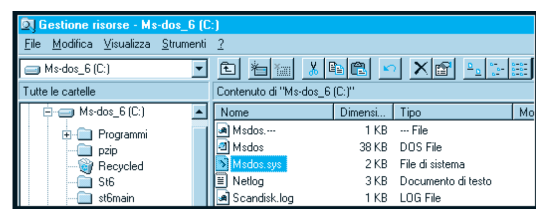


Fig. 19

– Apparirà la finestra di dialogo **Apri con** (vedi fig.20) in cui dovrete selezionare uno di questi programmi di gestione testi, **EDIT**, **NOTEPAD**, **WINWORD**. Poiché questi programmi sono equivalenti potrete indifferentemente scegliere l'uno o l'altro cliccando **due volte** sul nome corrispondente. Noi abbiamo usato **Edit**.

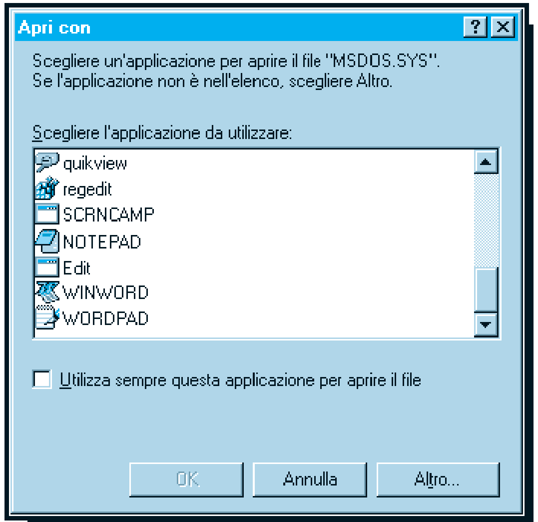


Fig. 20

– Sotto la scritta **[Options]** dovrebbero apparire queste due scritte:

BootGui=1
BootMulti=1

Se non compare **BootMulti=1** dovete necessariamente inserirla. Le altre scritte che dovete inserire come visibile in fig.21 sono:

BootMenu=1
BootMenuDelay=10

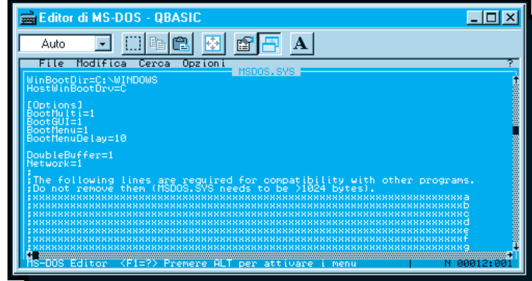


Fig. 21

L'opzione **BootMenu=1** ci mostra automaticamente ad ogni avvio il menu delle modalità di caricamento di **MS-DOS** e di **Windows 95** senza premere **F8**.
L'opzione **BootMenuDelay** stabilisce per quanto tempo, espresso in **secondi,** questo menu deve rimanere a video.

Noi abbiamo scelto un tempo di **10 secondi**, ma potete dare a questa variabile un altro valore.
Scaduto questo tempo, se non avete scelto nessuna modalità, viene automaticamente avviato **Windows 95** in modalità normale.

Nota: se non desiderate far apparire il logo di **Windows 95** inserite in coda alle altre la scritta **Logo=0** (vedi fig.22).

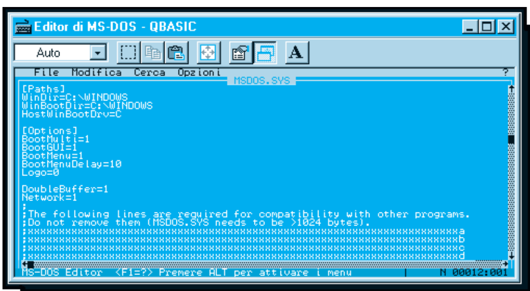


Fig. 22

Salvate il file utilizzando il comando **Salva** dal menu **File** ed uscite.

– Siccome **MSDOS.SYS** oltre ad essere un file nascosto è un file di sola lettura, dovete cliccare una volta con il tasto **destro** del mouse sulla scritta **MSDOS.SYS** poi selezionare **Proprietà** (vedi fig.17).

– Nella finestra di dialogo che appare cliccate accanto alla scritta **sola lettura** per ripristinare questa opzione, poi cliccate su **OK** (vedi fig.18).

– Ora cliccate sulla scritta **Visualizza** del menu di **Gestione Risorse** e selezionate **Opzioni** (vedi fig.15).

– Nella finestra di dialogo che appare ripristinate la condizione di file nascosti cliccando sul cerchietto, quindi uscite cliccando su **OK** (vedi fig.23).

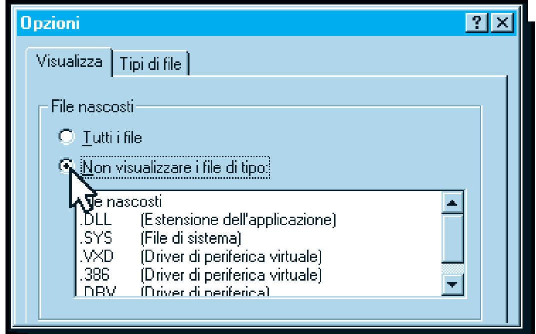


Fig. 23

– Uscite dal programma **Gestione Risorse**, chiudete **Windows 95** quindi ricaricatelo.
A questo punto la modifica proposta dal Sig. Chiribiri sarà operativa.

Anche se negli ultimi numeri della rivista non sono apparsi degli articoli relativi al microprocessore **ST6**, non pensate volessimo abbandonarlo.
Purtroppo dobbiamo accontentare anche quei lettori che non vogliono sentir parlare di computer, di software e di microprocessori, ma solamente di progetti **Hi-Fi**, oppure di ricevitori, microspie, strumenti di misura, ecc.

Ad ogni modo durante questa **pausa** forzata ci sono state richieste da parte di Istituti Tecnici e piccole Industrie una infinità di spiegazioni supplementari e ciò significa che abbiamo spiegato poco o in modo non sufficientemente comprensibile.

Prendendo spunto da tutte le domande ricevute oggi vogliamo "tentare" di darvi delle spiegazioni più chiare, avvertendovi al tempo stesso di non fare troppo affidamento in quanto riportato nei diversi manuali per **ST6**.

PER PROGRAMMARE

In quest'ultimi infatti vi sono molti errori e nessuna **errata** corrige, quindi in presenza di un insuccesso si è indotti ad autoaccusarsi di incapacità, mentre la colpa è di chi pubblica questi manuali senza aver mai visto o utilizzato in pratica un solo **ST6**.

I CICLI MACCHINA

Nella rivista N.185 vi abbiamo spiegato che per **cicli macchina** si intende il numero di **passi** necessari al micro per eseguire un'istruzione.
Poiché questa nostra spiegazione non è stata per tutti sufficientemente chiara, cercheremo di illustrarvela meglio con un semplice esempio.

Prendiamo in considerazione due istruzioni molto utilizzate in un programma, cioè **call** e **ret**.

L'istruzione **call** esegue **4 cicli** macchina, vale a dire che il microprocessore quando esegue questa istruzione compie **4 passi**:

1° passo o **ciclo –** il microprocessore riconosce il codice operativo **opcode** della istruzione **call**.

2° passo o **ciclo –** il microprocessore sposta al livello superiore il contenuto dei registri di **Stack**, innalzando di **1 livello** anche lo **Stack level** (vedi **Riv.184**).

3° passo o **ciclo –** il microprocessore memorizza nel registro di **Stack 1** l'indirizzo di **Program Space** nell'istruzione che si trova immediatamente dopo l'istruzione **call**.

4° passo o **ciclo –** il microprocessore muove nel **PC** (Program Counter) l'indirizzo della subroutine della **call**.

L'istruzione **ret** esegue **2 cicli** macchina, vale a dire che il microprocessore quando esegue questa istruzione compie **2 passi**:

1°passo o **ciclo –** il microprocessore riconosce il codice operativo **opcode** della istruzione **ret**.

2° ciclo – il microprocessore sposta il contenuto del registro **Stack 1** nel **PC** (Program Counter) trasferendo ad un livello più basso i valori contenuti nei rimanenti Stack (vedi **Riv.184**).
Molti ci hanno chiesto quanto tempo dura un **ciclo macchina** e, poiché questo dipende dalla frequenza del **quarzo**, riportiamo la semplice operazione necessaria per ricavarlo:

microsecondi = (13 : MHz) x cicli

Se usiamo un quarzo da **8 MHz**, per una istruzione **call** occorre un tempo di:

(13 : 8) x 4 = 6,5 microsecondi

e per una istruzione **ret** un tempo di:

(13 : 8) x 2 = 3,25 microsecondi

Usando un quarzo da **4 MHz** i tempi raddoppiano.

Nota = Vogliamo far presente a tutti coloro che utilizzano il **Simulatore DSE622** della Softec **senza** l'emulatore, che la gestione dei registri di **Stack**, anche se viene eseguita correttamente, sul **video** appare in senso **inverso** a quanto sopra riportato, e l'indirizzo di rientro viene memorizzato nel livello più **alto** disponibile ed evidenziato con il simbolo **>**, mentre il contenuto degli altri livelli non viene trasferito.

correttamente i micro ST6

Nella rivista N.185 abbiamo pubblicato delle tabelle che vi permettono di decifrare le decodifiche dell'Opcode e degli Indirizzi di Memoria. Oggi cercheremo di spiegarvi i Cicli macchina, il Watchdog, la funzione Reset e tante altre cose.

RESET

Sono tanti i lettori che ci hanno chiesto perché inizialmente il microprocessore si posiziona nella locazione **FFEH** di Program Space.

Il micro si posiziona in questa locazione di memoria quando si verificano queste tre condizioni:

1° - si alimenta il microprocessore.
2° - viene messo un **livello logico 0** sul piedino di **Reset** del microprocessore **ST6**.
3° - il contatore **Watchdog** arriva a **0**.

In questa locazione di memoria **FFEH** il microprocessore troverà la prima istruzione che dovrà eseguire, ad esempio **Jp inizio.**

```
Add Opcode   Label    Mnemonic
--- -- --             --- --
FFA FF FF             dec A
FFC 59 8A             jp nmi_int
FFE 09 88             jp inizio
```

In pratica la locazione **FFEH** è una cella di memoria nella quale, durante la stesura del programma, è stato scritto cosa deve fare il microprocessore quando si attiva l'**Interrupt** generato dalla funzione **reset**.

Se volete sapere qualcosa di più sugli **interrupts** vi consigliamo di rileggere la rivista **N.175/176**.

WATCHDOG

Il **Watchdog**, come già vi abbiamo spiegato nella rivista **N.174/175**, è un **contatore** che si decrementa con una frequenza legata al **Clock** del **microprocessore** e che genera un **reset** ogni volta che arriva a **0**.

Per gestire il **Watchdog** il microprocessore utilizza un registro chiamato **D**igital **W**atchdog **R**egister che è formato da **8 bit** come visibile in fig.1.

Mettendo a **1** il bit **C** il **Watchdog** si attiva, mentre mettendolo a **0** si disattiva.

Nei micro **ST6** con estensione **/SWD** l'attivazione e la disattivazione sono gestibili tramite **software**, mentre nei micro **ST6** con estensione **/HWD** sono gestite direttamente dall'**hardware** e quindi **non** si possono modificare tramite **software**.

Nei micro con estensione **/SWD** se mettiamo a **0**, tramite **software**, il bit denominato **C**, il **Watchdog non** risulta attivato e i bits **T1-T2-T3-T4-T5-T6-SR** possono essere utilizzati come **timer** a **7 bits.**

Quando il **Watchdog** viene attivato, cioè il bit **C** è a **1**, se tramite **software** mettiamo a **0** il bit **SR**, il microprocessore si **resetta.**

Quando il **Watchdog** risulta **attivato** utilizza come **contatore** solo **6 bit**, cioè quelli siglati **T1-T2-T3-T4-T5-T6** (vedi fig.1); per questo solo particolare registro bisogna tener presente che il bit **più significativo** è il **T6** e il **meno significativo** è il **T1.**

Il **peso** di questi **bits** è perciò il seguente:

bit T1 = peso 1
bit T2 = peso 2
bit T3 = peso 4
bit T4 = peso 8
bit T5 = peso 16
bit T6 = peso 32

Sommando tutti questi **pesi** otteniamo un peso totale di **63** e se a questo sommiamo il ciclo **0,** che per il **Watchdog** è significativo, otteniamo **64 cicli.**

Per sapere dopo quanti **microsecondi** il **Watchdog** si decrementa di una **unità** dobbiamo usare questa formula:

microsecondi = (1 : MHz) x 3.072

In questa formula la frequenza **MHz** è quella del **quarzo** utilizzato per il Clock del microprocessore.

Se nel nostro microprocessore è inserito un **quarzo** da **8 MHz**, il **Watchdog** sarà decrementato di **una** unità ogni:

(1 : 8) x 3.072 = 384 microsecondi

Poiché il numero massimo di **cicli** è **64**, potremo raggiungere un massimo di:

384 x 64 = 24.576 microsecondi

che corrispondono a **24 millisecondi** circa.

Conoscere il tempo di **decremento** del **Watchdog** è molto importante, perché quando inseriamo una **routine** possiamo calcolare con buona approssimazione la **cifra** da caricare nel **Watchdog Register**, in modo che la **somma** dei tempi di **ciclo macchina** delle istruzioni risulti sempre **minore** del tempo **totale** di **decremento.**

Tutto questo si fa per poter ricaricare al termine di una **routine** il **Watchdog Register** prima che arrivi a **0**, generando in questo modo un **reset** indesiderato nel microprocessore.

Usando questo artificio, se il microprocessore dovesse andare in **loop** a causa di un impulso spurio, il **Watchdog Register** arriverebbe a **0** in un tempo brevissimo attivando il **reset.**

Qui sotto riportiamo un esempio di calcolo effettuato su una routine, utilizzando un ST6 tipo **/HWD** con un quarzo da **8 MHz.**

Esempio:

```
rout01    ldi    wdog,130      (4)
          call   dsend         (4)
          ldi    wdog,130      (4)
---------------------------------------
dsend     ld     save1a,a      (4)
          ld     a,ddata       (4)
          andi   a,11110000b   (4)
          ld     port_c,a      (4)
          res    0,port_b      (4)
          set    0,port_b      (4)
          call   delay         (4)
          ld     a,ddata       (4)
          sla    a             (4)
          sla    a             (4)
          sla    a             (4)
          sla    a             (4)
          ld     port_c,a      (4)
          res    0,port_b      (4)
          set    0,port_b      (4)
          call   delay         (4)
          ld     a,save1a      (4)
          ret                  (2)
---------------------------------------
delay     ldi    del1,30       (4)
delay_A   dec    del1          (4 x 30)
delay_B   jrnz   delay_A       (2 X 30)
delay_C          ret           (2)
```

Accanto ad ogni istruzione abbiamo indicato i **cicli macchina.**
Come potete notare, nella routine **dsend** troviamo due **call delay** e di conseguenza questa **sub-routine** viene richiamata **due volte.**

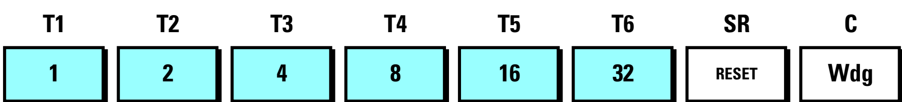


Fig.1 Il Watchdog utilizza come contatore i soli 6 bits indicati T1-T2-T3-T4-T5-T6. Come visibile nel disegno, il bit T1 vale 1 e il bit T6 vale 32, quindi sommando tutti questi "pesi" otteniamo 63 e sommando a questi il ciclo 0 otteniamo un totale di 64.

Quindi nel calcolo del tempo **totale** dovremo sommare **due volte** il tempo di esecuzione della **subroutine delay**.
Iniziamo ora il conteggio dei tempi.

- In **rout01** vi sono **3** istruzioni di **4 cicli** macchina per un totale di **3 x 4 = 12 cicli**, quindi otterremo un tempo totale di:

12 x 1,625 microsec. = 19,50 microsec.

- rout01 richiama anche la **call dsend**, quindi nel calcolo dovremo sommare anche i tempi di **dsend**.

- Nel **dsend** vi sono **18** istruzioni: di queste ve ne sono **17** di **4 cicli** macchina (**17 x 4 = 68 cicli**) ed una istruzione di **2** cicli macchina (**1 x 2 = 2 cicli**), quindi otterremo un tempo totale di:

(68 + 2) x 1,625 microsec. = 113,75 microsec.

Anche se nella subroutine **delay** vi sono **2** istruzioni di **4 cicli** macchina, occorre far presente che le due istruzioni **delay_A** e **delay_B** vengono eseguite **30 volte**. Svolgendo i nostri calcoli otterremo:

delay = 4 cicli macchina
delay_A = 4 x 30 =120 cicli macchina
delay_B = 2 x 30 = 60 cicli macchina
delay_C = 2 cicli macchina

Facendo la somma otterremo un totale di **186 cicli** macchina, quindi un tempo di:

186 x 1,625 microsec. = 302,25 microsec.

Poiché questa subroutine viene eseguita **due volte** questo tempo **raddoppierà** in **604,50 microsec.**

Sommando tutti i tempi **parziali** otterremo un tempo **totale** di esecuzione di:

tempo rout01	**19,50**
tempo dsend	**113,75**
tempo delay	**604,50**
tempo totale	**737,75 microsecondi**

Poichè sappiamo che un ciclo di **Watchdog** dura **384 microsec.**, per evitare che il Watchdog vada a **0** prima di **737,75 microsecondi** dovremo necessariamente fargli fare **2 cicli** in modo da ottenere un tempo totale di **384 x 2 = 768 microsecondi.**

A questo punto sembrerebbe logico che per fare eseguire **2 cicli** al Watchdog occorra caricare sul suo **registro** il numero **2**, invece dovremo caricare un numero inferiore di **una** unità, cioè **2-1 = 1** essendo necessario conteggiare anche il ciclo **0.**
Al valore così ottenuto dobbiamo poi **sempre** sommare **2**, perchè il bit **SR** del **Watchdog Register** deve essere sempre a **1** altrimenti il microprocessore si resetta.
Quindi nella prima istruzione per far eseguire **2 cicli** dovremo scrivere:

```
rout01    ldi    wdog,130
```

Se volessimo eseguire **4 cicli** dovremmo scrivere:

```
rout01    ldi    wdog,194
```

Per eseguire **11 cicli** dovremmo scrivere:

```
rout01    ldi    wdog,82
```

A questo punto vi chiederete perché per **2 cicli** abbiamo scritto **wdog,130**, per **4 cicli** abbiamo scritto **wdog,194**, mentre per **11 cicli** abbiamo scritto **wdog,82.**

Per farvelo capire utilizziamo la tabella qui sotto riportata:

Tabella di corrispondenza Pesi/Cicli

128	64	32	16	8	4	2	1	peso Binario
1	2	4	8	16	32	SR	C	Wdog

Poiché per ottenere **2 cicli** dobbiamo fare **2-1= 1**, basta guardare nella riga **sotto** per scoprire che il numero **1** corrisponde ad peso **binario 128.**
Al valore ottenuto ora dobbiamo sommare **2** perchè il bit **SR** sia sempre settato a **1**, perciò avremo **128+2 = 130.**

Per ottenere **4 cicli** dobbiamo fare **4-1= 3**, ma guardando la riga **sotto** non troveremo questo numero, quindi per ottenerlo dovremo necessariamente sommare **1+2 = 3** e poiché il numero **1** corrisponde ad un peso **binario** di **128** e il numero **2** ad un peso **binario** di **64**, dovremo fare la **somma** di questi due **pesi** (**128+64** = **192**).
A questo valore andiamo ora ad aggiungere sempre **2** e a questo punto avrete certamente compreso perché abbiamo scritto **wdog,194.**

Per ottenere **11 cicli** dobbiamo fare **11-1= 10** che, non essendo presente nella riga **sotto**, potremo ottenere soltanto sommando **2+8 = 10.**
Poiché **2** corrisponde ad un peso **binario** di **64** e **8** corrisponde ad un peso **binario** di **16** dovremo calcolare la somma dei due pesi **64+16 = 80**, aggiungere sempre il valore **2** e di conseguenza scrivere **wdog,82.**

Per ottenere **51 cicli** dovremo fare **51-1 = 50**, poi vedere nella colonna del **wdog** quali numeri dovremo sommare per ottenere **50** e qui avremo una sola possibilità:

32 + 16 + 2 = 50

Se sommeremo i **pesi binari** corrispondenti ai numeri sopra riportati otterremo **4+8+64 = 76**, valore al quale andremo ad aggiungere **2** e quindi nel registro di Watchdog scriveremo **wdog,78.**

Se scriveremo **wdog,254** eseguiremo il **massimo** dei cicli, cioè **64**, ma, come già abbiamo accennato, se il microprocessore va in **loop** dovremo attendere molto tempo prima che si **resetti.**

Nota = Quando il **Watchdog** è **attivo**, se in un programma abbiamo inserito l'istruzione **stop**, il microprocessore esegue al suo posto l'istruzione **wait** e **non** blocca il **Clock** (rivista **N.174**).

GESTIONE OTTIMALE delle PORTE

Tutti i micro con **20 piedini** (vedi fig.2) hanno due porte **I**nput-**O**utput contraddistinte da **A-B.**
Tutti i micro con **28 piedini** (vedi fig.3) hanno tre porte **I**nput-**O**utput contraddistinte da **A-B-C.**

Come noterete, i terminali della **porta A** sono siglati **A0-A1-A2**, ecc., quelli della **porta B** sono siglati **B0-B1-B2**, ecc., e quelli della porta **C** sono siglati **C4-C5-C6-C7.**

Come già vi abbiamo spiegato nella rivista **N.174/175** (sarebbe opportuno rileggerla), la configurazione e l'utilizzo di queste porte e di conseguenza dei singoli loro piedini, avviene tramite una serie di **tre registri** chiamati:

pdir_a	**popt_a**	**port_a**	(per la Porta **A**)
pdir_b	**popt_b**	**port_b**	(per la Porta **B**)
pdir_c	**popt_c**	**port_c**	(per la Porta **C**)

Nella **Tabella N.3** riportiamo cosa bisogna scrivere nei tre registri **pdir - popt - port** per predisporre questi piedini come **Ingressi** o come **Uscite.**

Quando si scrive un programma, si dovrebbe cercare di utilizzare i piedini di ogni singola **porta** tutti come **Ingressi** oppure tutti come **Uscite.**

Poiché spesso ci si trova nella necessità di utilizzare i piedini della stessa **porta** alcuni come **Ingressi** e altri come **Uscite**, non è consigliabile usare le istruzioni **SET(Set Bits)** e **RES(Reset Bits)** direttamente sulla porta che stiamo gestendo, perché il microprocessore potrebbe generare dei falsi impulsi sui piedini compromettendo la corretta esecuzione del programma.

Un piccolo stratagemma per ovviare a questo inconveniente è quello di utilizzare una **variabile** definita in **Data Space** come area di **parcheggio**, caricare al suo interno il contenuto del registro della porta, settare o resettare il bit relativo e infine copiare nuovamente il contenuto nel registro della porta come qui sotto riportato:

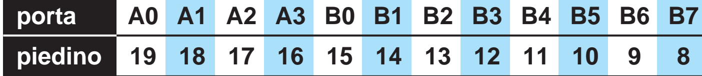
```
save_pa         def     xxx

        ld      a,port_a
        ld      save_pa,a
        set     1,save_pa
oppure  (res 1,save_pa)
        ld      a,save_pa
        ld      port_a,a
```

Nota = Il passaggio dati da una variabile ad un'altra, come ben sapete, deve avvenire tramite l'utilizzo intermedio dell'accumulatore **a.**

Vi sono tanti altri casi in cui si possono generare degli errori e falsi impulsi sui piedini della stessa porta; ad esempio quando nel corso di un programma si passa più volte da **Ingressi** a **Uscita** e viceversa, come spesso avviene quando, **dialogando** con un integrato esterno, **inviamo** un treno di dati e lo stesso integrato ce li rimanda per poterli **leggere.**
Se non si eseguono dei **passaggi** con un ordine ben definito si verificheranno sempre degli **errori.**

Questi passaggi sono visibili in fig.5.

TABELLA N.1 per micro ST62/E10 - ST62/E20 e per micro ST6/T10 - ST6/T20



porta	A0	A1	A2	A3	B0	B1	B2	B3	B4	B5	B6	B7
piedino	19	18	17	16	15	14	13	12	11	10	9	8

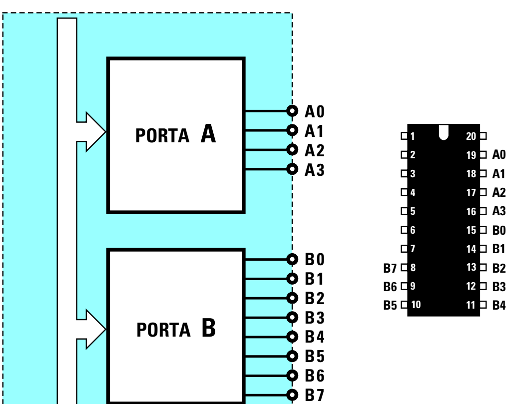


Fig.2 All'interno dei micro della serie T10-T20 non riprogrammabili e della serie E10-E20 che sono riprogrammabili, troviamo due sole porte indicate A-B.
Nella Tabella soprariportata abbiamo indicato a quale numero di piedino corrispondono le due porte A-B.

TABELLA N.2 per micro ST62/E15 - ST62/E25 e per micro ST62/T15 - ST62/T25

porta	A0	A1	A2	A3	A4	A5	A6	A7	B0	B1	B2	B3	B4	B5	B6	B7	C4	C5	C6	C7
piedino	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	9	8	7	6

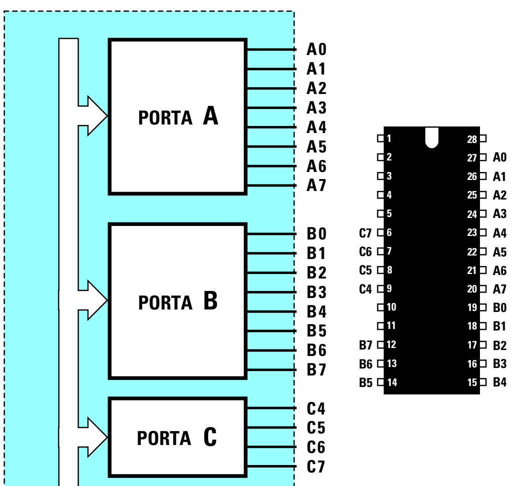


Fig.3 All'interno dei micro della serie T15-T25 non riprogrammabili e della serie E15-E25 che sono riprogrammabili, troviamo tre porte indicate A-B-C.
Nella Tabella soprariportata abbiamo indicato a quale numero di piedino corrispondono le tre porte A-B-C.

TABELLA N.3 per predisporre gli ingressi e le uscite

Registri	INGRESSI				USCITE			
	con pull-up	senza pull-up	con interrupt	segnali analogici	open collector		uscita push-pull	
pdir	0	0	0	0	1	1	1	1
popt	0	0	1	1	0	0	1	1
port	0	1	0	1	0	1	0	1

Fig.4 In questa Tabella indichiamo il numero che occorre scrivere nei tre registri "pdir-popt-port" per far funzionare una porta come Ingresso o come Uscita. Per settare un piedino come INGRESSO con PULL-UP dovremo scrivere nei tre registri 0-0-0.

Per spiegarci meglio, se dalla configurazione:
- Ingresso **Pull-Up 000**
volessimo passare alla configurazione:
- Uscita **Push-Pull 111**
dovremmo effettuare questi passaggi di configurazione:
000-100-110-111 oppure **000-001-101-111**

Se passeremo direttamente da **000** a **111** o se faremo **000-100-111**, ci ritroveremo con un programma che potrebbe generare delle anomalie.
Se da un'**Uscita Push-Pull 111** volessimo passare direttamente ad un **Ingresso Pull-up** con **Interrupt 010** dovremmo effettuare queste configurazioni:
111-101-001-000-010
oppure seguire l'altra configurazione, cioè:
111-110-100-000-010
Osservando le **frecce** presenti nella fig.5 in ciascuna di queste configurazioni, si può facilmente comprendere qual è la strada da seguire per passare da una configurazione all'altra.

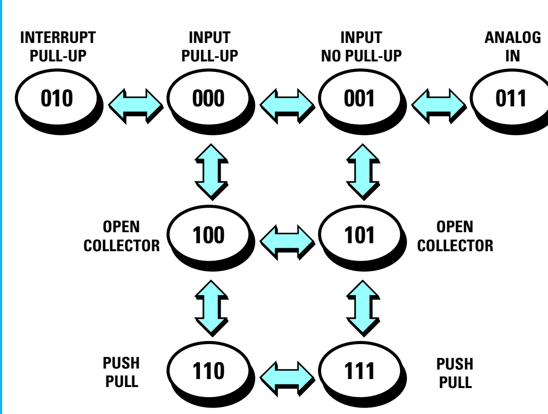


Fig.5 In questo disegno indichiamo i passaggi ottimali per portarsi da una configurazione ad un'altra (vedi Tabella N.4). Se non seguirete questi passaggi obbligati otterrete delle anomalie.

ESPRESSIONI

Una **Espressione** è costituita da **numeri**, da **variabili** e da **operatori**.
Qui sotto riportiamo la **priorità** di questi **operatori**, che ci sarà utile per stabilire l'**ordine** di esecuzione di calcolo nel caso fossero presenti **espressioni** con più **operatori**.
Ad esempio, se nella stessa espressione troviamo una **somma**, una **And** e una **inversione** di **bit**, il compilatore eseguirà prima l'**inversione** di **bit**, poi la **somma** e per ultima la funzione **And**.

Operatori	Funzione	Priorità
–	valore negativo	1
~	inversione di bit	1
*	moltiplicazione	2
/	divisione	2
%	modulo	2
>>	sposta a destra	2
<<	sposta a sinistra	2
+	somma	3
–	sottrazione	3
&	funzione And	4
|	funzione Or	6

Dobbiamo far presente che se mettiamo delle **parentesi** in una istruzione, ad esempio:

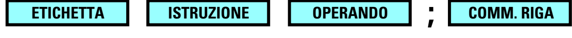
ldi var1,(seg1 - offset) /4

nonostante la **divisione /4** abbia una priorità **2**, viene eseguita prima l'operazione **seg1 – offset** anche se ha una priorità **3**.

Le **espressioni** vengono eseguite quando si effettua la Compilazione in **Assembler** e **non** durante l'esecuzione del programma.

Nota = In molti degli **esempi** che riporteremo troverete delle **Direttive Assembler** che in seguito vi spiegheremo meglio, anche se nella rivista **N.174** abbiamo già accennato qualcosa in merito.

Come noto, una istruzione **Assembler** è composta da:



L'**Operando** può essere costituito da una **Variabile,** un **Registro**, una **Etichetta**, un **valore assoluto**, cioè un **numero** che può essere espresso in Binario, in Esadecimale o in Decimale.

Anche una **espressione** può essere utilizzata come **operando**, ottenendo in questo modo il vantaggio di poter **spostare** dei **blocchi** di **variabili** da un punto ad un altro di una **memoria** con una sola operazione, riducendo così eventuali errori.

Sempre utilizzando una **espressione** come **operando**, daremo al **compilatore** la possibilità di selezionare i **blocchi** di istruzioni di una **macro** e di includerli in un nuovo programma.

Per farvi capire come usare una **espressione** come **operando** vi proponiamo alcuni esempi che potrete inserire in un qualsiasi vostro programma di

prova, verificandone il risultato con il **software simulatore** che vi abbiamo presentato nelle riviste **N.184** e **N.185**.

1° Esempio

In questo esempio vi facciamo vedere come si può utilizzare una **espressione** con la funzione **valore negativo**.

Etichetta	Istruzione	Operando
seg1	.def	088h
offset	.set	-20h
inizio	ldi	seg1, offset
	ldi	x,-40h
	ldi	a,-offset

Come si potrà notare, nella prima istruzione la **Direttiva .def** associa la variabile **seg1** all'indirizzo di **Data Space 088h**.
Nella seconda istruzione la **Direttiva .set** associa alla costante definita **offset** il valore risultante dall' **Espressione** –**20h** (**–32 decimale**) e il compilatore lo converte in **E0h** (**224 decimale**): questo perché la **funzione valore negativo** genera il **complemento** a **256** di **32** che corrisponde a **224**.

Nella terza istruzione, all'etichetta **inizio**, **ldi** carica nella variabile **seg1** il valore associato a **offset** e cioè **E0h**.

Nella quarta, l'istruzione **ldi** carica nel registro **x** il valore risultante dalla **Espressione** –**40h** (–**64** in **decimale**) e il compilatore lo converte in **C0h** (**192 decimale**), perché anche in questo caso genera il **complemento** a **256** di **64** che corrisponde a **192**.

Nella quinta, l'istruzione **ldi** carica nell'accumulatore **"a"** il valore risultante dalla **Espressione** –**offset** (–**224** in **decimale**) e il compilatore lo converte in **20h** (**32 decimale**), perché anche in questo caso si ottiene il **complemento** a **256** di **224** che corrisponde a **32**.

2° Esempio

In questo esempio vi facciamo vedere come si può utilizzare una **espressione** con la funzione **somma**.

Etichetta	Istruzione	Operando
seg1	.def	088h
seg2	.def	seg1+1
seg3	.def	seg2+1

Nella prima istruzione la **Direttiva .def** associa la variabile **seg1** all'indirizzo di **Data Space 088h**.

Nella seconda istruzione la **Direttiva .def** associa il valore risultante della **Espressione** "**seg1+1"** alla variabile **seg2**.

Essendo **seg1** definito all'indirizzo di memoria **088h**, l'espressione **seg1+1** viene semplificata durante la **compilazione** in **Assembler** nel numero esadecimale **088h + 1 = 089h**.
Questa **variabile seg2** viene perciò associata all'indirizzo di **Data Space 089h**.

Nella terza istruzione la **Direttiva .def** associa il valore risultante dalla **Espressione** "**seg2+1"** alla variabile **seg3**. Essendo **seg2** definito all'indirizzo di memoria **089h**, l'espressione "**seg2+1**" viene semplificata in **089h + 1 = 08Ah**.

Dichiarando le **Variabili** come qui sopra riportato, ridurremo l'**errore** di definire due o più variabili nella stessa cella di memoria.
Nell'esempio qui sotto riportato si può notare che per **errore** la **seg4 = 08ah** risulta collocata nella stessa cella di memoria di **seg3**.

```
seg1    .def    088h
seg2    .def    089h
seg3    .def    08ah
seg4    .def    08ah
```

e questo **errore** non viene segnalato dal **Compilatore**.

Un altro vantaggio che deriva dall'utilizzo della soluzione sopra consigliata si presenta nel caso volessimo spostare la **variabile** di memoria **seg1** in un altra cella mantenendo sempre la successione di **seg2** e **seg3**.
Infatti non saremo più costretti a **modificare** tutte tre le istruzioni, perché basterà cambiare soltanto la prima **.def**, mentre le successive si rilocano automaticamente.
I vantaggi più evidenti però li otterremo con le Direttive **.macro** e **.input** per l'utilizzo di **moduli** e di **macro-routines**.

3° Esempio

In questo esempio vi facciamo vedere come si può utilizzare una **espressione** con le funzioni **divisione** e **sottrazione**.

Etichetta	Istruzione	Operando
seg1	.def	088h
offset	.set	10h
inizio	ldi	x,(seg1 - offset)/ 2

Nella prima istruzione la **Direttiva .def** associa la variabile **seg1** all'indirizzo di **Data Space 088h**.

Nella seconda istruzione la **Direttiva .set** associa alla costante **offset** il valore **10h**.

Nella terza istruzione contrassegnata dall'etichetta **inizio**, l'istruzione **ldi** carica nel registro **x** il risultato dell'espressione **(seg1 - offset) / 2**.
L'istruzione **seg1 - offset** è stata racchiusa tra parentesi perché **deve** essere eseguita prima della **divisione** per **2**.
Infatti, come vi abbiamo già spiegato nelle **priorità**, la **divisione** verrebbe altrimenti eseguita prima della **sottrazione**.

divisione priorità **2**
sottrazione priorità **3**

Se eseguite queste istruzioni con un **software simulatore** potrete vedere all'etichetta **inizio** che l'espressione **(seg1 - offset) / 2** è stata sostituita dal valore **3Ch**.

E ora vi spiegheremo il perché.

Poiché viene data la priorità a **(seg1 - offset)**, il compilatore Assembler eseguirà la sottrazione:

088h - 10h = 78h

quindi l'espressione **(seg1 - offset) / 2** verrà semplificata in **78h / 2**.

Successivamente il compilatore **Assembler** eseguirà la divisione **78h / 2** e come risultato finale otterremo **3Ch**.

Se volessimo ragionare in **decimale**, già sappiamo che i numeri **esadecimali** equivalgono a:

088h = 136 decimale
10h = 16 decimale
78h = 120 decimale
3Ch = 60 decimale

Eseguendo queste stesse operazioni in **decimale** otterremo:

136 - 16 = 120 che corrisponde a **78h**
120 : 2 = 60 che corrisponde a **3Ch**

4° Esempio

In questo esempio vi facciamo vedere come si può utilizzare una **espressione** con le funzioni **And**, **Inversione** di **bit** e **sposta** a **destra**.

Etichetta	Istruz.	Operando
seg1	**.def**	**088h**
offset	**.set**	**0Fh**
valtst	**.set**	**04h**
inizio	**ldi**	**seg1,offset&(~(valtst>>1))**

Nella prima istruzione la **Direttiva .def** associa la variabile **seg1** all'indirizzo di **Data Space 088h**.

Nella seconda istruzione la **Direttiva .set** associa alla costante **offset** il valore **0Fh**.

Nella terza istruzione la **Direttiva .set** associa alla costante **valtst** il valore **04h**.

Nella quarta, all'etichetta **inizio**, l'istruzione **ldi** carica nella variabile **seg1** il risultato della **Espressione** "**offset&(~(valtst>>1))**".

Analizziamo ora questa **Espressione** per capire ogni singola funzione.

Subito viene data la priorità a **(valtst>>1)** essendo racchiusa tra parentesi **interne** e viene semplificata in **04h>>1 = 02h**.
Infatti **valtst** vale **04h** e **l'operatore ">>"** che indica di **spostare** a **destra**, esegue uno **shift binario a destra** di tante posizioni quante risultano indicate nell'espressione.
Nel nostro esempio con **1** si ottiene il numero **esadecimale 02h**.

Convertendo in **binario** il valore **esadecimale 04h** questo calcolo diventerà più comprensibile.

04h = binario **00000100b**

Se ora spostiamo a **destra** il valore di ogni singolo bit otterremo:

00000010b

Se riconvertiremo questo numero da **binario** in **esadecimale** otterremo **02h**.

Perciò il compilatore Assembler semplificherà la espressione:

offset&(~(valtst>>1)) in **offset&(~02h)**

Nota = Se avete dei problemi a **convertire** un numero **binario** in **esadecimale** o **decimale** vi consigliamo di consultare il nostro volume **HANDBOOK** a **pag.372**.

Attenzione = Poiché **non** esiste nessun controllo sul numero di bits che vengono spostati a destra, il compilatore non segnalerà mai nessun **errore**.

Quindi se spostate verso **destra** più di **7 bit** otterrete come risultato un valore uguale a **0.**

Se, ad esempio, nella espressione **(valtst>>1)** scriveremo per errore **(valtst>>10)**, come risultato otterremo **0.**

Poiché la nostra l'espressione è stata semplificata in **offset&(~02h)** la priorità passa a **(~02h).**

Il **compilatore** Assembler eseguendo l'**inversione** della configurazione **02h** (si noti il segno **~** della funzione **inverti** i bit) ci darà come risultato il valore esadecimale **FDh.**

Convertendo in **binario** il valore **esadecimale 02h** questo calcolo diventerà più comprensibile:

02h = binario **00000010b**

Se ora **invertiremo** questi bits otterremo:

11111101b

Se riconvertiremo questo nuovo numero da **binario** in **esadecimale** otterremo **FDh.**

Quindi l'espressione **offset&(~02h)** verrà semplificata in **offset&FDh.**

Il compilatore Assembler eseguendo la funzione **AND** (si noti il segno **&**) tra **offset** (che vale **0Fh**)e **FDh** otterrà come risultato **0Dh.**

Convertendo in **binario** questi due numeri **esadecimali** ed eseguendo poi la funzione **And** otterremo:

```
      0Fh  =  00001111b
      FDh  =  11111101b
              ---------
risultato  =  00001101b
```

Come noterete, nella funzione **And** solo se nella colonna sopra e in quella sotto è presente un valore logico **1** otteniamo come **risultato 1.**
Ogni altra combinazione ci darà **risultato 0.**

Se riconvertiamo il numero **binario 00001101b** in **esadecimale** otterremo **0Dh.**

Pertanto la quarta istruzione :

ldi   seg1,offset&(~(valtst>>1))

il compilatore **Assembler** l'ha semplificata in:

ldi   seg1,0Dh.

5° Esempio

In questo esempio vi facciamo vedere come si può utilizzare una **espressione** con le funzioni **Or** e **sposta** i bits a **sinistra.**

<table>
<tr><th>Etichetta</th><th>Istruz.</th><th>Operando</th></tr>
<tr><td>seg1</td><td>.def</td><td>088h</td></tr>
<tr><td>offset</td><td>.set</td><td>80h</td></tr>
<tr><td>valtst</td><td>.set</td><td>05h</td></tr>
<tr><td>inizio</td><td>ldi</td><td>seg1,offset | (1<<valtst)</td></tr>
</table>

Nella prima istruzione la **Direttiva .def** associa la variabile **seg1** all'indirizzo di **Data Space 088h.**

Nella seconda istruzione la **Direttiva .set** associa alla costante **offset** il valore **80h.**

Nella terza istruzione la **Direttiva .set** associa alla costante **valtst** il valore **05h.**

Nella quarta, all'etichetta **inizio**, l'istruzione **ldi** carica nella Variabile **seg1** il risultato della espressione **offset|(1<<valtst).**

Analizziamo ora questa **Espressione** per capire ogni singola funzione.

Viene data la priorità a **(1<<valtst)** essendo racchiusa fra parentesi e viene semplificata in **20h.**

Attenzione = Con questa istruzione molti cadono in inganno perché pensano che si **sposti** di **1 bit** verso **sinistra** il valore di **valtst.**
Invece è il valore **valtst** che dice di quanti **bits** occorre spostare verso sinistra il numero **decimale 1.**

Convertendo in **binario** il numero **decimale 1** otterremo:

00000001b

Poichè il valore di **valtst** è **05h** che in **decimale** equivale a **5**, il compilatore sposterà di **5 posizioni**, verso **sinistra**, il numero **1**:

00100000b

Se convertiamo questo numero **binario** in **esadecimale** otterremo **20h.**

Pertanto la quarta istruzione:

ldi seg1,offset | (1<<valtst)

verrà semplificata dal compilatore in:

ldi seg1,offset | 20h

Attenzione = Il compilatore **Assembler** segnala **errore** di **Overflow** nel caso si tenti di spostare a sinistra bits significativi, oltre la capacità di **1 Byte.**

Ad esempio, se nel nostro numero sono presenti sulla sinistra quattro zeri = **00001000** potremo effettuare uno spostamento solo di quattro, ottenendo così **10000000.**
Se invece sulla sinistra del numero sono presenti solo due zeri = **0011100** lo spostamento potrà essere solo di due, ottenendo così **11100000.**

Tornando al nostro esempio, quando il compilatore incontra l'espressione **offset | 20h**, eseguendo la funzione **OR** tra **offset** (che vale **80h**) e **20h** otterremo come risultato **A0h.**

Infatti, se convertendo in **binario** i numeri **esadecimali 80h** e **20h** proviamo ad eseguire la funzione **OR**, otterremo:

80h = 10000000b
20h = 00100000b
risultato = **10100000b**

che convertito in **esadecimale** ci darà **A0h.**

Come noterete nel **risultato** vengono riportati tutti gli **1** presenti nelle **due** colonne.

Pertanto la quarta istruzione:

ldi seg1, offset | (1<<valtst)

il compilatore la semplificherà in:

ldi seg1,A0h

6° Esempio

In questo esempio vi facciamo vedere come si può utilizzare una **espressione** con la funzione **modulo** indicata con il simbolo **%** che esegue una **divisione** e ci dà come risultato il **resto** se questo è presente, altrimenti ci dà come risultato **0**.

Etichetta	Istruzione	Operando
	.block	64-$%64
disp01	.ascii	"PROVA"

Il simbolo **"$"** in **Assembler** significa **Valore del Program Counter Relativo**, perciò in fase di **Compilazione** (e non di esecuzione) al posto di **"$"** viene inserito l'indirizzo della cella di memoria **Program Space Relativa.**

Nella prima istruzione la **direttiva .block** definisce un'area di **Program Space** la cui estensione è il risultato della **Espressione 64–$%64.**

Questa espressione va letta come segue:

- Sottrai da **64** l'eventuale **resto** risultante dalla divisione fra il valore del **Program Counter Relativo** e il numero **64.**

Supponiamo che la direttiva:

.block 64–$%64

si trovi all'indirizzo di memoria:

Program Space 894h

Il **compilatore** sostituirà questo indirizzo di memoria al simbolo **$** e semplificherà questa espressione come segue:

.block 64–894h%64

che espressa in decimale diventerà:

.block 64–2196%64

La **funzione modulo** ha priorità **2**, mentre la **funzione sottrazione** ha priorità **3**, perciò verrà svolta prima l'operazione **2196%64.**
Eseguendo questa operazione con la calcolatrice si otterrà:

2196 : 64 = 34,3125

Per ricavare il **resto** di questa divisione sarà sufficiente moltiplicare **0,3125** per **64**:

0,3125 x 64 = 20

L'**Espressione** verrà ulteriormente semplificata come segue:

.block 64–20 = 44

Come risultato finale questa direttiva definirà un'area di **44 byte** in **Program Space** a partire dall'indirizzo **894h.**
Vale a dire che vengono lasciati **liberi 44 bytes** in

modo da posizionare il **1° byte** della successiva direttiva:

disp01 .ascii "PROVA"

nel **1° byte** del blocco successivo di **64 bytes** del **Program Space.**

La direttiva **.ascii** definisce una **stringa** di caratte- re **ASCII** in **Program Space** la cui lunghezza in bytes è definita dal numero di caratteri inseriti tra le **virgolette**, e vi associa una **etichetta**.

Nel nostro esempio la stringa **"PROVA"** è lunga **5 bytes**, e **disp01** è l'etichetta associata.

Tutte queste operazioni sono indispensabili perché il microprocessore **non** permette di utilizzare direttamente le stringhe di dati definite in **Program Space.**

Quindi se vogliamo utilizzarle dobbiamo trasferirle tramite un apposito **registro,** in un'area di memoria di **64 bytes** definita **Data Rom Window.**

Poiché quest'**area** ha una **capacità** di soli **64 bytes**, caricando una seconda stringa questa cancellerà quella precedente e la sostituirà con i nuovi **dati.**

A questo punto cercheremo di spiegarvi con un semplice **esempio** perché occorre far rientrare le stringhe di dati all'inizio di ogni **blocco** da **64 bytes.**

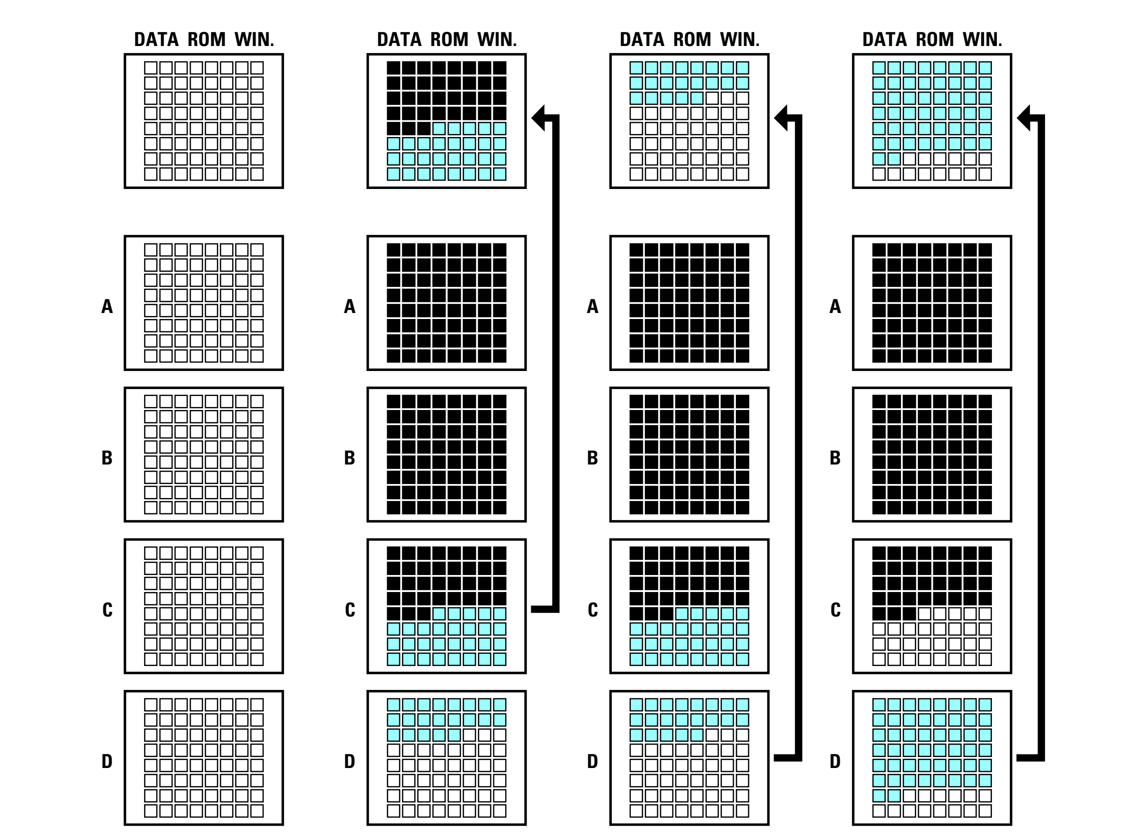


Fig.6 Con questo esempio dei vassoi contenenti 64 pedine vogliamo spiegarvi perchè è necessario collocare la stringa .ASCII all'inizio del blocco successivo in cui terminano le "istruzioni" del programma. Poichè il micro per poter utilizzare la stringa di "dati" nel Program Space la deve prelevare da uno dei vassoi e caricare nel primo vassoio in alto della Data Rom Window, occorre che il primo byte di questa stringa (quadrettini azzurri) inizi da un blocco da 64 bytes. L'istruzione ".block" serve proprio per posizionare questa stringa nel "vassoio" successivo da 64 byte. Come vedesi nella seconda e terza colonna verticale dove non si è usato il ".block", se trasferiamo il vassoio C o D nella Data Rom Window la stringa dei dati risulterà incompleta, mentre usando il ".block" (vedi ultima colonna di destra) riusciremo a portare nel vassoio della Data Rom Window la stringa completa dei "dati" presenti nel vassoio D.

Ammettiamo di avere un certo numero di vassoi in grado di contenere ciascuno un massimo di **64 pedine**, che nel nostro esempio sarebbero i **bytes**.

Anche l'area di memoria definita **Data Rom Window** riservata per poter utilizzare i **dati** contenuti nelle **tabelle** è in grado di contenere un massimo di **64 pedine** (vedi fig.6).

Se abbiamo un programma composto da **163 pedine** riusciremo a riempire completamente i primi due vassoi **A-B** e per **metà** il terzo vassoio **C** (vedi fig.6) fino ad arrivare alla casella **35**.
Se di seguito inseriamo una stringa di dati composta da **50 pedine** (senza inserire l'espressione **.block**), queste verranno inserite partendo dalla casella **36** fino ad arrivare alla casella **64** e nel successivo vassoio **D** l'Assemblatore inserirà le altre **21** pedine.

Se ora volessimo trasferire la nostra stringa di dati nella **Data Rom Window**, poiché possiamo prelevare solo dei blocchi di **64 byte** potremmo trasferire il solo blocco **C** o il solo il blocco **D** ed in questo caso avremmo sempre una stringa di dati spezzata ed incompleta.

Si può ovviare a questo inconveniente facendo calcolare all'Assemblatore lo spazio necessario per portare tutte le **pedine** della stringa nel blocco **D**, lasciando inutilizzate le caselle dalla **36** fino alla **64** del blocco **C**, inserendo questa solo **espressione**:

.block 64–$%64

Avendo in questo modo collocato tutta la stringa nel blocco **D**, quando la trasferiremo nella **Data Rom Window** (vedi fig.6) il **1° byte** della stringa coinciderà con il **1° byte** della **Data Rom Window**.

Importante = Come vi abbiamo spiegato, il **.block** permette di spostare tutta la stringa dei **dati** nel vassoio **D** (vedi fig.6). Uno dei vantaggi che offre il **.block** è quello di non doverci più preoccupare se nel vassoio **C** aggiungiamo o togliamo delle **righe** di istruzione, perchè il **compilatore** provvederà automaticamente a calcolare l'area necessaria per fare saltare tutto il blocco dei dati nel vassoio **D**.

A questo punto dobbiamo **forzatamente** interrompere questo articolo. Per il momento non ponetevi domande sul concreto utilizzo degli esempi che vi abbiamo presentato, ma limitatevi ad osservarne il risultato.
Nel prossimo numero ci addentreremo nelle **direttive assembler** e tratteremo **in pratica** e con **esempi** le **Espressioni**.

NUOVO software SIMULATORE

Chi desidera un software simulatore per testare i micro ST6 più completo del DSE.622 presentato nelle riviste N.184 e N.185, potrà installare sotto Windows 3.1 o 95 il nuovo software che qui vi presentiamo.

Non possiamo iniziare questo articolo senza prima ringraziare pubblicamente il Sig. **Cesarin Ivano** di **Porpetto** (Udine) che ha realizzato questo **software simulatore** per i micro **ST62** tipo **10-15-20-25** che, oltre alle funzioni presenti normalmente nei software in circolazione, offre la possibilità di **inserire - variare** e soprattutto **memorizzare** tutti i segnali sia sui piedini d'**ingresso** che su quelli di **uscita** compresi i piedini del **Timer** e dell'**NMI**.

Con questo simulatore è infatti possibile memorizzare sui piedini d'**ingresso** di ogni singola porta, compresi quelli del **Timer** e dell'**NMI**, degli stati digitali e analogici con dei precisi **tempi** che possiamo noi stessi prefissare da **0** fino ad un massimo di **500.000 microsecondi**, condizione questa che consentirà di lanciare delle simulazioni **Batch**.

Inoltre permette di visualizzare questi segnali memorizzati su una valida **finestra grafica** e di vedere così sul monitor tutti gli eventi che si sono verificati sui piedini del **micro** nell'arco di tempo di **500.000 microsecondi**.

I segnali d'**ingresso** possono poi essere memorizzati sui files con estensione **.CMD**, mentre i segnali d'**uscita** sui files con estensione **.DAT**.

Facciamo presente che le restanti funzioni risultano pressochè **identiche** a quanto già descritto nelle riviste **N.184** e N.**185**, quindi non dovete fare altro che rileggerle attentamente insieme a tutti gli esempi riportati.

Chi non dispone di queste riviste potrà richiederle quando ordinerà il dischetto software.

Per spiegarvi come usare questo software iniziamo dalla videata **principale** di un **test** di esempio inserito dall'Autore, che apparirà chiamando il file **PEDALI.PROG.**

Tralasciamo di spiegare come si deve procedere per far apparire la finestra di dialogo dell'**Edit Data** (vedi fig.1), comunque se non lo ricordate vi consigliamo di prendere la rivista **N.184** e di rileggere quanto scritto.

Come noterete, sull'**Edit Data** di fig.1 appare il registro **dati** di **Port_B**. Cliccando sui **Bits** apparirà la nuova finestra di fig.2 con la **mappa binaria** completa di questa porta.

Nella fig.3 vi facciamo vedere la finestra che apparirà facendo una **Attivazione/Disattivazione** di un **BreakPoint**.

Potete trovare una esauriente spiegazione della funzione **BreakPoint** nella rivista **N.148**.

L'unica differenza che noterete è che nella fig.62 tutte le scritte sono in inglese, mentre nella fig.3 sono riportate in italiano.
Facciamo presente che la scritta **Breakpoint** equivale a **Interruzione ON/OFF**.

Nelle figg.4-5-6-7 abbiamo riportato i **menu** disponibili affinchè possiate rendervi conto delle possibilità che offre questo simulatore.

per micro ST6

I comandi supplementari che troverete in questo **nuovo** software sono:

Dati in ingresso
Scrivi File Dati in ingresso
Scrivi File Dati in Uscita

Cronologia Porte
Scrivi Dati in Uscita
Test I/O

Se si attiva la funzione **Dati in ingresso** il programma accetta la selezione di un file solo con estensione **.CMD** come visibile in fig.8.

Consigliamo di usare sempre lo stesso **nome** del programma che si desidera simulare, quindi se avete denominato il programma **Tester.Hex** o **Led.Hex** lo dovete chiamare **Tester.CMD**. o **Led.CMD**.

Importante! Il file deve risultare presente già prima di effettuare la **prima** simulazione, quindi lo dovrete **creare** con un **Editor** qualsiasi sulla **directory** di lavoro.

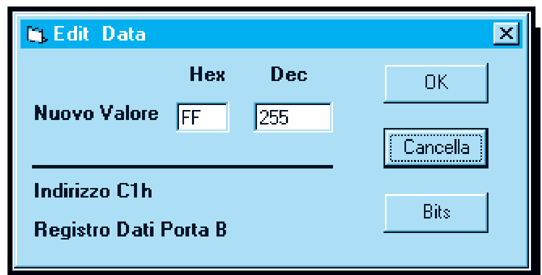


Fig.1 Quando appare la finestra di dialogo dell'Edit Data dovete premere Bits.

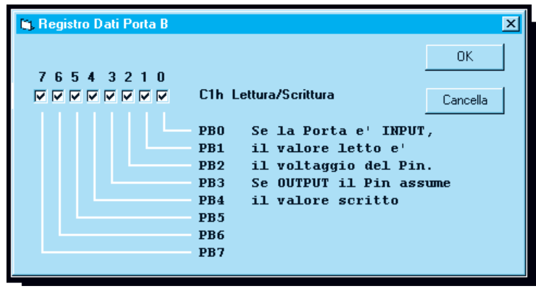


Fig.2 Automaticamente vi apparirà la finestra della mappa binaria della porta B.

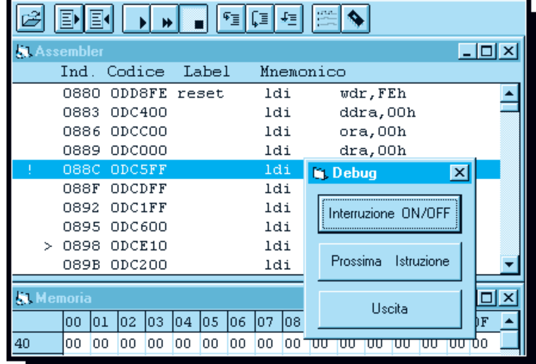


Fig.3 Ecco la finestra che apparirà attivando o disattivando un BreakPoint.

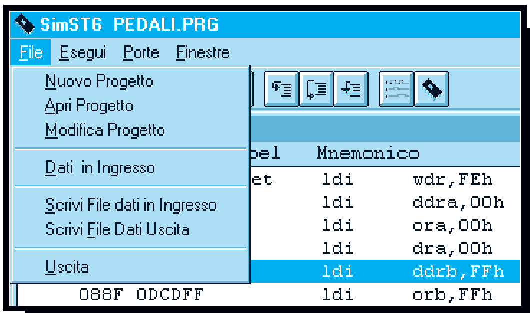


Fig.4 Cliccando File apparirà la finestra del menu con l'elenco dei comandi.

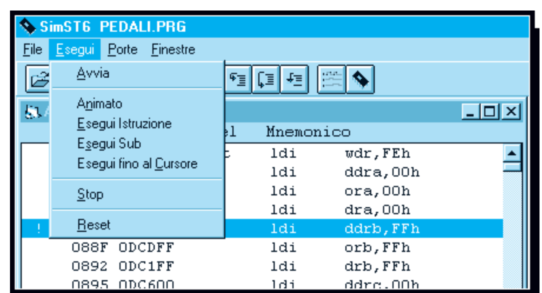


Fig.5 Cliccando Esegui apparirà questa finestra e tutti i relativi comandi.

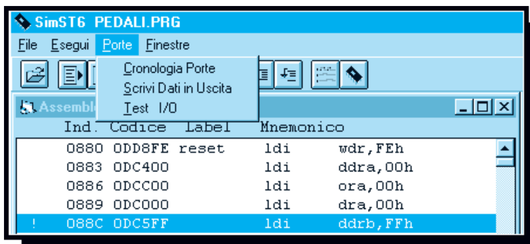


Fig.6 Cliccando Porte appariranno le tre funzioni che potrete eseguire.

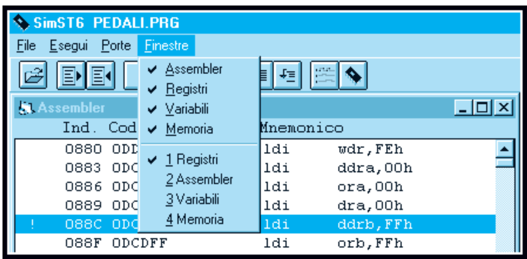


Fig.7 Cliccando Finestre potrete scegliere quali funzioni visualizzare.

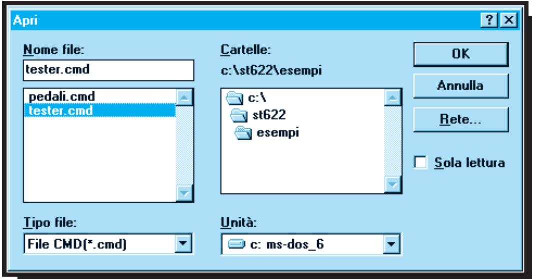


Fig.8 Cliccando nella fig.4 Dati in Ingresso apparirà questa finestra.

Cliccando ora sulla funzione **Cronologia Porte** (vedi fig.6) è sottointeso che la prima volta la finestra di fig.9 apparirà completamente **vuota**.
In questa finestra potete inserire o variare su ogni porta i **livelli logici 1-0** oppure i **livelli analogici** come visibile in fig.10 (vedi in alto le scritte **PA-PB-PC-Varie**).

Se selezionerete **Varie**, potrete agire sui piedini del **TIMER** e dell'**NMI**.
Oltre agli stati logici è possibile **selezionare** sia la **base dei tempi** (vedi riga indicata **Ampiezza**) che l'**offset** (durata dell'impulso) fino ad un tempo massimo di **500.000 microsecondi**.

Nota = Questo tempo di **500.000 microsecondi**, corrispondente a **0,5 secondi**, che a voi può sembrare **irrisorio**, per il microprocessore è un tempo **esagerato** perchè corrisponde a diverse **migliaia** di cicli macchina.

In fig.9 abbiamo riportato un **esempio grafico** della funzione **Cronologia Porte**.

Se, a questo punto, volete inserire nel programma di prova i dati visibili in fig.9, dovrete innanzitutto cliccare sulla barra di scorrimento orizzontale posta in basso (vedi scritta **Ampiezza**) e selezionare una base dei tempi di **12,5 msec**.
Cliccando poi sul cerchietto **PB** selezionerete la porta B e, posizionandovi con il mouse su **PB0** e **PB1**, dovrete inserire i seguenti valori:

piedino **PB0**
a **livello logico 1** da **0** a **25 msec**.
a **livello logico 0** da **26** a **50 msec.**
a **livello logico 1** da **51** a **75 msec.**
a **livello logico 0** da **76** a **84 msec.**
a **livello logico 1** da **85 msec.** in poi

piedino **PB1**
a **livello logico 0** da **0** a **37 msec**.
a **livello logico 1** da **38** a **81 msec.**
a **livello logico 0** da **82 msec.** in poi

Nota = Cliccando con il mouse al di sotto della linea tratteggiata orizzontale relativa a **PB0** o **PB1** si inserisce un **livello logico 0**, mentre cliccando al di sopra si inserisce un **livello logico 1**.
Per conoscere gli **esatti tempi** di salita e di discesa degli stati logici dovrete tenere premuto il tasto **destro** del **mouse** e, in questo modo, sul video apparirà una **riga verticale** di colore **viola** che potrete spostare in orizzontale tenendo ovviamente sempre premuto il tasto del mouse.

Come potete vedere infatti in fig.12, il piedino **PB1** si porta a **livello logico 1** dopo **37 msec**.

Potrete leggere questo numero nel piccolo riquadro posto in alto a destra.

Spostando la riga **viola** verso destra (vedi fig.13) potrete constatare che il piedino **PB1** si porta a **livello logico 0** dopo un tempo di **81 msec**.

In fig.14 sono riportati gli stessi stati logici di fig.9 ma con una **base tempi** di **50 msec**.

Questo simulatore permette anche l'inserimento di segnali analogici e non soltanto di stati logici.

Ad esempio, ammesso che nel nostro programma il piedino **4** della **porta A** sia stato predisposto come ingresso per un **segnale analogico**, poichè l'**AD/converter** dell'**ST6** accetta un massimo di **5 volt** e utilizza un registro di **8 bit** per la conversione che corrisponde ad un numero **decimale 255**, è ovvio che **1 volt** corrisponde al numero:

255 : 5 = 51

Ammesso di voler simulare una tensione di **3 volt** dopo **125 microsecondi**, dovrete cliccare sulla funzione **Cronologia Porte** visibile in fig.6 e, in questo modo, apparirà la finestra di fig.9; a questo punto dovrete cliccare nel cerchietto **PA**.

La prima operazione da effettuare è quella di andare con il cursore sulla barra di scorrimento con la scritta **Offset** fino a quando sul video non apparirà un **tempo** di **125 msec**.

Se porterete il cursore sulla riga **orizzontale** in corrispondenza della scritta **PA4** e poi vi sposterete fino ad incontrare la riga **verticale** in corrispondenza dei **125 msec** e qui cliccherete il mouse, apparirà la finestra centrale visibile in fig.10.

All'interno del riquadro posto sulla destra scriverete **153** che corrisponde ad un valore di **3 volt**, infatti **51 x 3 = 153**.

Se a questo punto cliccherete su **OK** apparirà la finestra di fig.15 e sulla riga orizzontale della porta **PA4** apparirà il numero **153**.

Per **memorizzare** tutti i segnali riportati nelle figg.9-15 nel file con l'estensione **.CMD** dovrete premere il tasto **OK** e, in tal modo, comparirà la finestra riprodotta in fig.11.

Portando il cursore sull'icona **File** e cliccando Enter apparirà la finestra di fig.4 e a questo punto sarà sufficiente cliccare sulla scritta:
Scrivi File Dati in ingresso.

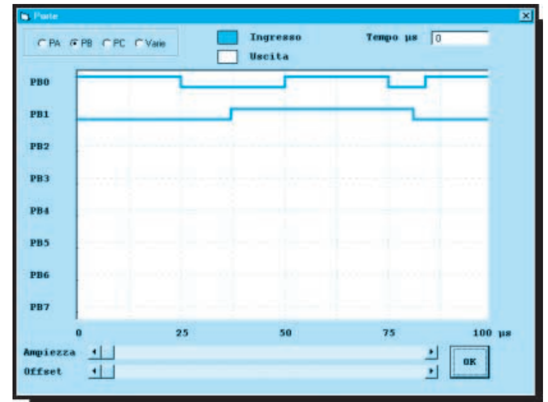


Fig.9 Esempio grafico della funzione Cronologia Porte che permette di vedere i livelli logici presenti sui piedini del micro.

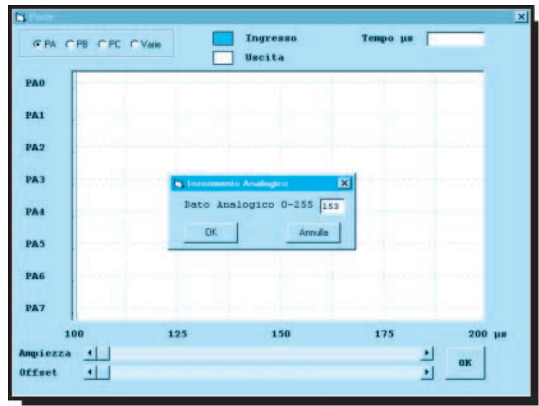


Fig.10 Se un piedino della porta è configurato Analog In, apparirà una finestra con un valore che potrete modificare.

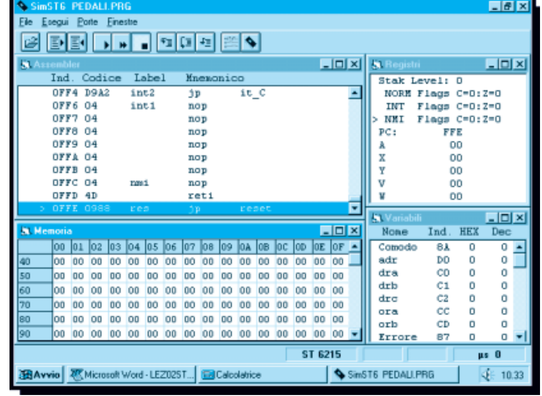


Fig.11 Videata principale delle quattro finestre Assembler - Registri - Variabili - Memoria attivabili tramite la fig.7.

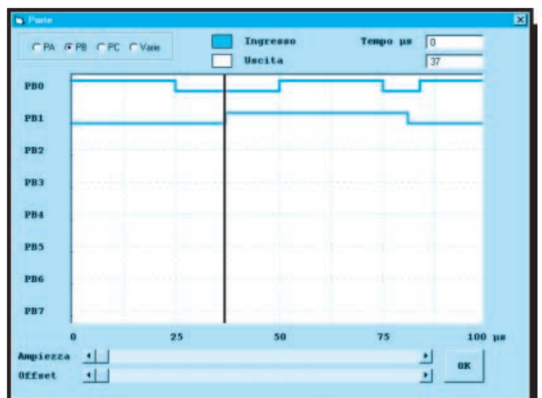


Fig.12 Portando nella Cronologia Porte di fig.9 la riga verticale su un segnale potrete leggere il suo tempo esatto.

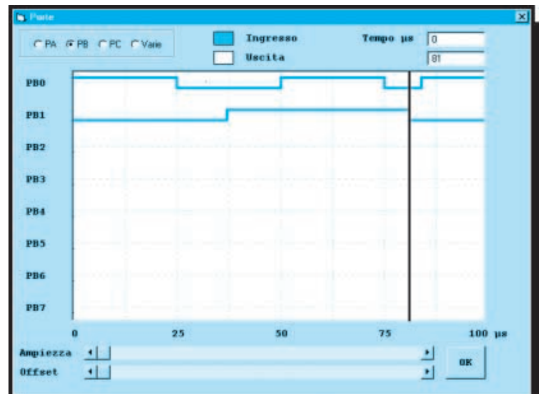


Fig.13 Per sapere quando la porta PB1 cambia da 1 a 0 dovrete spostare la riga verticale nella posizione visibile in figura.

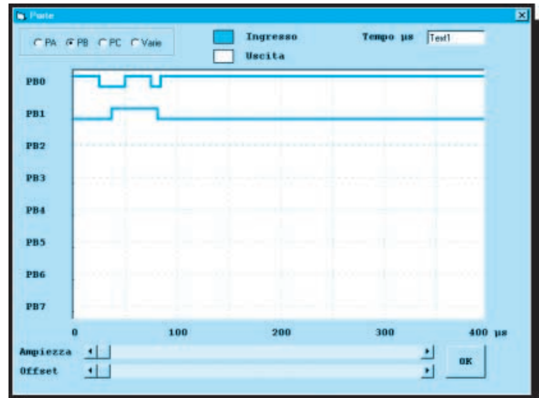


Fig.14 Modificando la base dei tempi da 12,5 msec. (vedi fig.9) a 50 msec potrete aumentare il campo di visualizzazione.

Una volta memorizzati i **dati**, se si rilancia la **simulazione** del programma selezionando la funzione **Dati in ingresso**, il simulatore, leggendo il file **.CMD** preleverà i segnali modificati e li inserirà in **automatico** nei piedini da voi assegnati senza interrompere la simulazione.

In tutti gli altri normali programmi di simulazione si è costretti a **fermare** l'esecuzione, **inserire** le modifiche e poi ripartire.

Concluse le spiegazioni relative alla funzione **Dati in Ingresso**, possiamo passare alla funzione **Scrivi Dati in Uscita**.

Portando il cursore sul menu **Porte** e cliccando Enter apparirà la finestra di fig.6 e ovviamente qui dovrete selezionare la riga con la scritta:

Scrivi Dati in Uscita e cliccare

Da questo preciso istante il simulatore **memorizzerà** tutti gli eventi presenti sui piedini del microprocessore configurati come **uscita** sia come valore **digitale** che **analogico**, compresa la loro **durata.**
Il tempo massimo di memorizzazione, come già accennato, non può superare i **500.000 msec** che corrispondono esattamente a **0,5 secondi**.

Per **vedere graficamente** sul monitor quanto avete memorizzato in modo da verificare istante per istante come cambiano i livelli logici o analogici sui piedini delle **porte** e controllare così se le routine eseguono le funzioni da voi richieste, dovrete andare alla maschera di fig.6 e selezionare la riga **Cronologia porte**.

Cliccando su questa riga comparirà la finestra visibile in fig.16 dove, in **rosso**, appaiono tutti i livelli logici delle porte d'uscita.

La barra verticale azzurra visibile sulla destra dell'esempio segnala il punto esatto di fine registrazione eventi.
Nel nostro caso è di **181 msec** (come visibile anche in alto a destra).

Nel nostro esempio i segnali sono riferiti alla porta **B**, quindi se avessimo predisposto come uscita la porta **A** avremmo dovuto selezionare il **cerchietto** posto in alto con la scritta **PA** anzichè **PB**.

Per **salvare** definitivamente questi dati memorizzati dovrete premere **OK** e, in tal modo, riapparirà la finestra di fig.11 e su questa dovrete cliccare sulla scritta **File** in modo da far apparire la finestra riprodotta in fig.4.

Portate il cursore sulla riga **Scrivi File Dati Uscita**, poi premete Enter e, così facendo, apparirà la finestra visibile in fig.17 e nel riquadro sottostante, in corrispondenza della scritta **Nome file**, inserite il **nome** del file con estensione **.DAT**.
Premete infine **OK.**

Nell'esempio di fig.17 abbiamo denominato questo file **TESTER.DAT.**
Tutti i file memorizzati con estensione **.DAT** e anche quelli con estensione **.CMD** possono essere richiamati, visualizzati e stampati con un qualsiasi programma di **Editor**.
Troverete la spiegazione della struttura dei **dati** contenuti in questi due files nelle **NOTE** che l'Autore ha inserito nel software.
Per stampare queste **note** dovrete andare su **File Manager** se avete il **Windows 3.1** oppure su **Risorse del Computer** se avete **Windows 95**, poi vi dovrete posizionare nella **directory** in cui avete installato questo software.
In **directory** cercate **manuale.wri**, poi cliccate e sul monitor potrete leggere queste **note**.

L'ultima funzione **Test I/O** serve per **visualizzare** ed eventualmente **modificare** la configurazione delle **Porte** e del **Timer** del microprocessore e anche i **livelli logici** dei **dati d'ingresso**.

Nella fig.18 potete vedere la maschera che compare sul monitor quando si attiva questa funzione.
Osservando in basso a destra, sotto la scritta **Modo** potrete notare che è stata selezionata la funzione **Linee** che permette di visualizzare i **segnali** che sono presenti sui piedini del microprocessore nel preciso momento in cui è stata attivata la funzione **Test I/O**.
Sul lato **sinistro** è riportata la zoccolatura del microprocessore (nel nostro esempio è riportata la zoccolatura dell'**ST62/15 - ST62/25**) e sul lato **destro** la piedinatura di ogni **porta**.
Il segno **V** presente nei riquadri sta ad indicare che nel corrispondente piedino è presente un **livello logico 1**.
Ammesso di voler modificare lo stato logico del piedino **PA2** della porta **A**, sarà sufficiente portare il cursore nel riquadro corrispondente e **cliccare** con il mouse e, così facendo, apparirà una **V**; se cliccherete una seconda volta la **V** sparirà.
Se sotto la scritta **Modo** selezionerete la modalità **Configurazione** (vedi fig.19), vedrete come risultano configurate tutte le **porte** del micro.
Come potete notare nel riquadro che appare sulla destra, tutti i piedini della **porta B** sono configurati **Out Push Pull**.
Il piedino **4** della porta **A** è configurato **Analog In**, mentre i restanti piedini di questa porta sono configurati **Input Pull Up**.

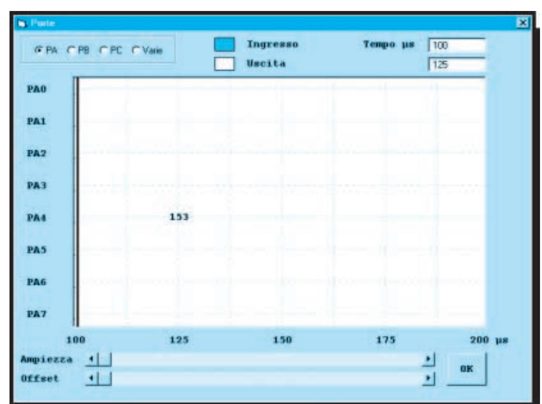


Fig.15 Poichè nel nostro esempio abbiamo assegnato alla porta PA4 un valore di 3 volt vedrete apparire il numero 153.

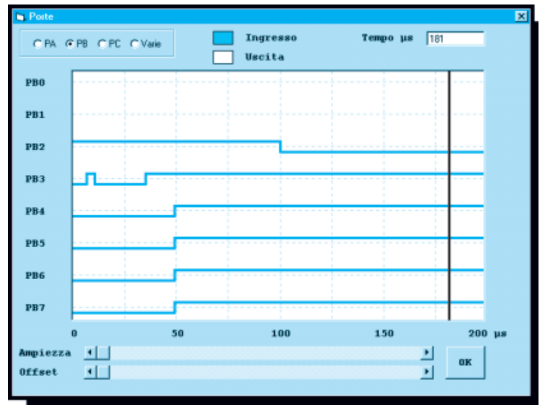


Fig.16 In questo grafico potete vedere tutti i livelli logici 1-0 che risulteranno presenti sulle porte configurate Out.

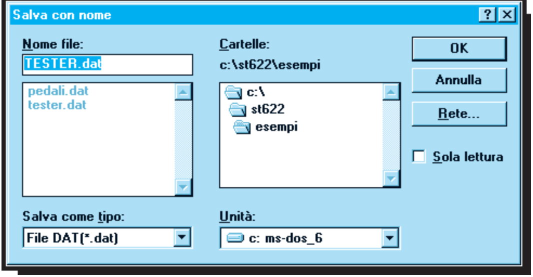


Fig.17 Per salvare in un file i segnali presenti sui piedini d'uscita (vedi fig.16) dovrete utilizzare questa finestra.

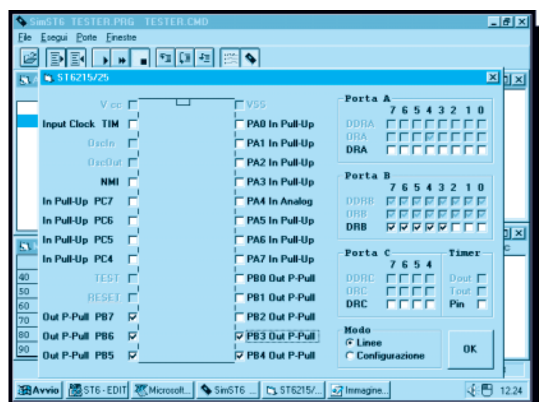


Fig.18 Questa finestra appare attivando la funzione Test I/0 in Modo Linee.

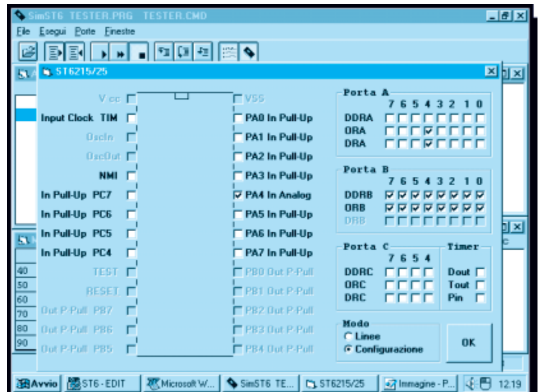


Fig.19 Questa finestra appare attivando Test I/0 in Modo Configurazione.

Fig.20 In fase d'installazione, quando appare questo messaggio inserite il disco 2.

Per modificare la configurazione di una **porta** dovrete rispettare i valori **logici** che abbiamo indicato nella **Tabella N.1**.
Premendo **OK** uscirete da questa maschera e poichè avete apportato delle modifiche vi converrà **salvarle** sul file con estensione **.CMD** tramite la funzione **Scrivi File Data in Ingresso** dopo aver fatto apparire la maschera di fig.4.
A questo punto rilanciate la **simulazione** del vostro programma, selezionando **Dati in ingresso** e verificando quale risultato si ottiene con le modifiche apportate.

INSTALLAZIONE del SOFTWARE sotto WINDOWS 3.1

Inserite nell'unità floppy disk il dischetto **ST622-1** e nel menu del **Program Manager** portate il cursore in alto a sinistra sulla scritta **File** e cliccate.
Andate sulla scritta **Esegui**, cliccate nuovamente e quando apparirà la finestra di dialogo digitate:

A:\setup poi cliccate su OK

In questo modo il computer inizierà a leggere il contenuto del **primo** dischetto e quando questo risulterà trasferito, apparirà la maschera di fig.20.
Togliete il dischetto **ST622-1**, inserite il dischetto **ST622-2** e a questo punto cliccate su **OK**.
Completata la lettura anche di questo dischetto sul video apparirà la maschera di fig.21.
Cliccate su **OK** per proseguire.
Dopo qualche istante apparirà la maschera riprodotta in fig.22.
Se a questo punto volete modificare la directory di installazione, cliccate su **cambia directory** e seguite le indicazioni su video, altrimenti cliccate sull'icona contenente l'immagine di un computer per installare il software sotto la directory **c:\ST622**.
Completata l'installazione, apparirà la maschera di fig.23 per confermarvi che nell'hard-disk risulta inserito questo software sotto la directory **ST622**.
Come noterete, nel **Program Manager** verrà generata la relativa **icona**.

Registri	INGRESSI				USCITE			
	con pull-up	senza pull-up	con interrupt	segnali analogici	open collector		uscita push-pull	
DDR	0	0	0	0	1	1	1	1
OR	0	0	1	1	0	0	1	1
DR	0	1	0	1	0	1	0	1

TABELLA N.1 = Per modificare la configurazione di una porta d'ingresso o di uscita dovrete rispettare questi livelli logici. Quindi per un ingresso SENZA PULL UP dovrete assegnare il registro DDR a 0, il registro OR a 0 e il registro DR a 1.

INSTALLAZIONE del SOFTWARE sotto WINDOWS 95

Inserite nell'unità floppy disk il dischetto **ST622-1** e cliccate sulla scritta **Avvio** posta in basso a sinistra, poi andate sulla scritta **Esegui** e cliccate nuovamente e quando apparirà la finestra di dialogo digitate:

A:\setup poi cliccate su OK

In questo modo il computer inizierà a leggere il contenuto del **primo** dischetto e quando questo risulterà trasferito, apparirà la maschera di fig.20.
Togliete il dischetto **ST622-1**, inserite il dischetto **ST622-2** e a questo punto cliccate su **OK.**
Completata la lettura anche di questo dischetto, sul video comparirà la maschera di fig.21.
Cliccate su OK per proseguire.
Dopo qualche istante, apparirà la maschera riprodotta in fig.22.
Se, a questo punto, volete modificare la directory di installazione, cliccate su **cambia directory** e seguite le indicazioni che appariranno sul video, altrimenti cliccate sull'"icona" contenente l'immagine di un computer per installare il software sotto la directory **c:\ST622**.
Completata l'installazione apparirà la maschera di fig.23 per confermarvi che nel vostro hard-disk risulta inserito il software sotto la directory **ST622** e automaticamente verrà generata la relativa **icona**.

Nota = Nel dischetto **ST622-2** è presente una directory **Esempi** in cui l'Autore ha inserito un programma chiamato **PEDALI** con varie **estensioni** (**.Dat** - **.Cmd** - **.Asm**, ecc.), che potranno servirvi per impratichirvi nell'uso del simulatore.
In fase di installazione questa directory **non** viene copiata nell'Hard-Disk, quindi per trasferirla dovrete usare la funzione **Copy File**.

CONCLUSIONE

Questo **software simulatore** sarà molto utile a tutti i softwaristi che programmano dei microprocessori tipo **ST6** perchè, a differenza di altri, permette di **vedere - modificare - variare** con estrema facilità le condizioni logiche su tutte le porte d'ingresso e d'uscita.
Più prenderete confidenza con l'uso di questo software più vi renderete conto dei tanti problemi che esso è in grado di risolvere.

COSTO del SOFTWARE

Costo dei due dischetti floppy **ST622-1** e **ST622-2**
Iva inclusa .... € 10,32

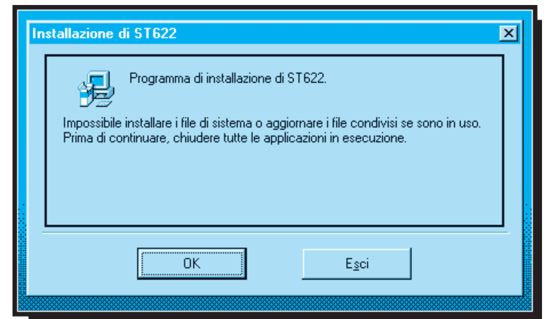


Fig.21 Se in fase d'installazione appare questo messaggio dovrete uscire e poi chiudere tutte le altre applicazioni attive.

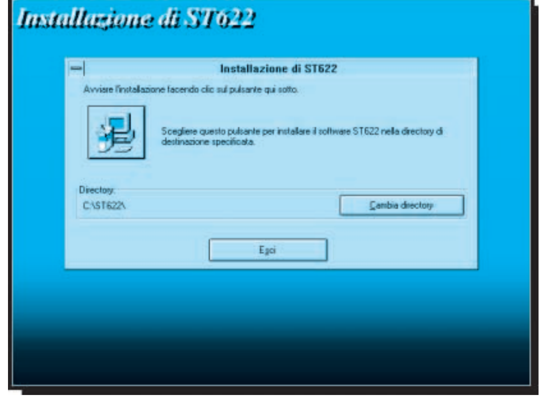


Fig.22 Letti tutti e due i dischetti, il programma d'installazione chiederà se volete cambiare oppure no la directory.

Fig.23 Completata l'installazione sull'Hard Disk, apparirà l'icona che dovrete utilizzare in seguito per lanciare il software.

SOFTWARE emulatore per

Se imparerete a conoscere e ad utilizzare le Direttive dell'Assembler riuscirete ad ottimizzare i vostri programmi, a gestire le librerie di moduli e di macro, a sezionare i programmi in pagine logiche, a generare moduli in formato .obj ecc., risparmiando così del tempo nella stesura.

Nell'Assembler per **ST62** esistono delle **istruzioni** chiamate **direttive**, che in pratica esauriscono la loro funzione in fase di **compilazione** e non generano, come le altre istruzioni, una **OPCODE** eseguibile.
In parole povere queste istruzioni **non fanno** eseguire **nulla** al microprocessore ma agiscono solo sul **Compilatore**.

A questo punto qualcuno si chiederà quale utilità possano avere queste **direttive** che non fanno **nulla** di concreto, quindi se volete scoprirlo dovrete leggere questo articolo e alla fine capirete che molte di esse sono **indispensabili**.

Per fare un esempio iniziamo dalla **direttiva .display** che possiamo chiamare digitando:

.display "stringa"

La parola stringa che abbiamo inserito tra le virgolette, può essere sostituita con una parola diversa oppure con una frase di messaggio, ad esempio:

.display "Compilato routine PIPPO.ASM"

Quando il **Compilatore** legge questa **Direttiva**, provvede a visualizzare sul **Monitor** del Computer la frase:

Compilato routine PIPPO.ASM

In questo caso è il **Compilatore Assembler** che esegue il comando **.display** e non l'**ST6**.

In programmazione, questa tecnica di **segnalazione** sul **monitor** viene molto utilizzata in combinazione con le direttive **.input** oppure **.macro** e **.ifc** perchè consente di vedere direttamente sul monitor, in tempo reale, quali routine sono state caricate nel programma in fase di **Compilazione**.

Chiusa questa breve parentesi, proseguiamo riportando l'elenco di tutte queste **Direttive**.

A quanti hanno già acquistato e utilizzato i programmi **didattici** contenuti nei nostri dischetti molte di queste istruzioni risulteranno familiari.

Vi sono ben **36** di queste **direttive** e qui sotto ve le elenchiamo raggruppate per funzioni:

Direttive usate per la definizione di **Dati** nell'area del **Programma**:
.block **.byte** **.word** **.ascii** **.ascii**

Direttiva usata per la definizione di **Variabili** nell'area dei **Dati**:
.def

Direttive usate per la definizione delle **Costanti Simboliche**:
.equ **.set**

Direttive usate per il **Correlatore di Moduli** definito anche con il nome di **Linker**:
.glob **.window** **.windowend** **.transmit** **.notransmit**

TESTARE i micro ST6

Direttive relative al solo **Hardware**:

.pp_on
.dp_on
.w_on
.page_d
.section

Direttive di carattere **Generale**:

.display
.end
.input
.org
.error
.warning

Direttive da utilizzare per impaginare il **listato**:

.eject
.list
.pl
.linesize
.title
.comment

Direttive da usare per la **Compilazione Condizionata** in **Assembler**:

.ifc
.else
.endc

Direttive da utilizzare per la gestione delle **Macro**:

.macro
.endm
.mexit

Completato l'elenco delle **direttive**, poichè non esiste un ordine ben preciso per iniziare da una direttiva anzichè da un'altra, prendiamo in considerazione quella denominata **.w_on**, strettamente legata alla **Data Rom Window**, alla quale abbiamo già fatto cenno nella rivista **N.189**.

La direttiva **.w_on** ha un'unica funzione che consiste nell'abilitare la **Data Rom Window** all'interno del programma.

Se all'inizio del programma non inseriamo la diret-

tiva **.w_on** non potremo usufruire di quest'area ed il **Compilatore** ci segnalerà:

Error .W_ON Directive Required

Una volta abilitata, per accedere e utilizzare la **Data Rom Window** dovremo inserire in coda a determinate istruzioni le due sigle **.w** e **.d**.

A questo punto apriamo una parentesi per parlare di **Data Rom Window** e del suo corretto utilizzo tramite **.w** e **.d**.

Già nella precedente rivista **N.189** abbiamo accennato che per poter utilizzare una stringa di dati, sia alfanumerici che numerici, definiti nel **Program Space**, occorre prima caricarli nella **Data Rom Window** (che è un'area di **Data Space** lunga **64 bytes** che inizia dalla locazione **40h**) tramite il **Data Window Register**, che è un registro a **8 bits** definito alla locazione di **Data Space C9h**.

Supponiamo quindi di avere un programma in cui abbiamo definito una serie di stringhe **dati** nella locazione **A40h** di **Program Space** come qui sottoriportato:

```
A40h test01     .ascii "TESTO DI PROVA"
                .ascii "PER VISUALIZZARE"
                .ascii "CARATTERI  ALFAN"
                .ascii "UMERICI  - FINE -  "
```

Come potete notare, abbiamo definito **4** direttive **.ascii** lunghe **16** Bytes cadauna ed alla prima abbiamo associato l'etichetta **test01**.

Per caricare questi **dati** in **Data Rom Window** dobbiamo scrivere:

ldi drw,test01.w

Nota: dopo **ldi** appare la variabile **drw** che abbiamo utilizzato per definire il **Data Window Register**. È sottointeso che possiamo sostituire **drw** con qualsiasi altra sigla, ad esempio **pippo**, oppure **reg01** ed anche **kkkk**, importante è non superare **8 caratteri**.

Come noterete l'etichetta **test01**, che corrisponde al bytes di inizio della stringa, termina con **.w**.

Se per errore scriviamo l'etichetta **test01** senza inserire il **.w** come qui sottoriportato:

ldi drw,test01

il compilatore ci segnalerà questo errore:

- **8 bit value overflow**

nel caso le stringhe di **test01** risultino inserite in coda alle istruzioni del programma.

TABELLA N.1 di conversione binario/decimale

bit	11	10	9	8	7	6	5	4	3	2	1	0
peso	2048	1024	512	256	128	64	32	16	8	4	2	1
binario	1	0	1	0	0	1	0	0	0	0	0	0

Il codice binario 101001.000000 posto sotto al relativo peso.

TABELLA N.2 di conversione binario/decimale

bit	11	10	9	8	7	6	5	4	3	2	1	0
peso	2048	1024	512	256	128	64	32	16	8	4	2	1
binario	0	0	0	0	0	0	1	0	1	0	0	1

I primi 6 numeri del codice binario 101001 sono spostati tutti verso destra.

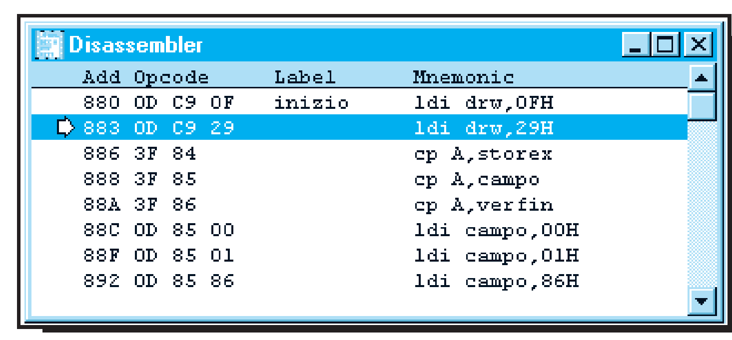


Fig.1 Con il software simulatore potrete notare nella finestra del Disassembler, che l'istruzione:
ldi drw,test01.w
si sarà tramutata in:
ldi drw,29H

Se invece le stringhe di **test01** risultano inserite prima delle istruzioni del programma, apparirà questa diversa scritta:

- **operand may not reference program space simbol test01**

Questi **errori** vengono segnalati dal **Compilatore** tutte le volte che incontra una istruzione che carica un indirizzo di **Program Space** (sempre espresso con **12 bits**) in una variabile o registro in grado di contenere solo **8 bits**.

Inserendo invece in coda all'istruzione la sigla **.w** come qui riportato:

ldi drw,test01.w

il **Compilatore** utilizza solo i **6** bits più significativi dell'indirizzo (quelli di sinistra) di **Program space** di **test01**, che verranno poi caricati, in fase di esecuzione, nel registro **drw** senza generare errore.

A questo punto dobbiamo spiegarvi come utilizzando solo **6** bits non si generi **nessun errore**.

Se convertiamo in **binario** il numero **esadecimale A40h** che corrisponde all'indirizzo di **test01** otterremo:

101001-000000

Se collochiamo questo numero **binario**, composto di **12 bits**, nella **Tabella N.1** (vedi tabella a sinistra) e sommiamo i loro i **pesi** otterremo:

2048 + 512 + 64 = 2624

Se prendiamo i **6** bits di **sinistra** della **Tabella N.1**, cioè **101001**, e li spostiamo tutti verso **destra** come possiamo vedere nella **Tabella N.2** e sommiamo i suoi **pesi** otterremo:

32 + 8 + 1 = 41

Nota = I **pesi** da **sommare** sono solo quelli che si trovano a **livello logico 1**.

Se ora proviamo a **dividere** la somma dei **pesi** della **Tabella N.1** per la somma dei **pesi** della **Tabella N.2** otterremo:

2624 : 41 = 64

In pratica quando il Compilatore assembla una istruzione con **.w**, divide per **64** l'indirizzo di **Program Space** dell'etichetta:

2624 : 64 = 41 (esadecimale **29H**)

perdendo l'eventuale resto e memorizza il risultato della divisione (nel nostro esempio **29H**) al posto dell'operando **test01.w**.

Disponendo di un **software** simulatore (vedi fig.1) sarà possibile vedere, tramite la finestra Disassembler, che l'istruzione:

ldi drw,test01.w

sarà diventata:

ldi drw,29H

Una volta assemblato il programma, quando il microprocessore incontrerà questa istruzione il risultato verrà memorizzato nel **Data Window Register** che, come già saprete, deve essere tassativamente definito all'indirizzo di memoria **Data Space C9h** (vedi fig.2).

In pratica l'indirizzo di **test01** all'interno del registro **drw** viene espresso in **64esimi**.

Il microprocessore inoltre riconosce che il registro **drw**, definito alla locazione di memoria **C9h**, è il **Data Window Register**, quindi carica nella **Data**

```
psc      .def    0d2h                        ;20 registro del timer
tcr      .def    0d3h                        ;21 contatore del timer
tscr     .def    0d4h                        ;22 registro prescaler del time
wdog     .def    0d8h                        ;23 registro del watchdog
drw      .def    0c9h                        ;24 registro data rom window
ram      .equ    084h
endram   .equ    08fh+1
;+------------------------------------------------------------------
```

Fig.2 Il risultato della istruzione ldi drw,29H verrà memorizzato nella locazione C9h.

Rom Window tutti i **64** bytes, cioè:

"TESTO DI PROVA "
"PER VISUALIZZARE"
"CARATTERI ALFAN"
"UMERICI - FINE - "

Infatti, se prendiamo il numero **41** contenuto nel **Data Window Register** e lo moltiplichiamo per **64** otterremo un numero **decimale**:

41 x 64 = 2624

che corrisponde al numero esadecimale **A40h** che è esattamente l'indirizzo di:

```
A40h test01    .ascii "TESTO DI PROVA "
               .ascii "PER VISUALIZZARE"
               .ascii "CARATTERI ALFAN"
               .ascii "UMERICI - FINE - "
```

Nota = Nel nostro esempio abbiamo volutamente dichiarato **test01** in un indirizzo di memoria **Program Space** esattamente divisibile per **64**, quindi, seguendo quanto detto sopra, **test01** inizia esattamente nel primo byte del **41** blocco; pertanto il primo byte della stringa di dati che inizia con **test01**, verrà posizionato nel **primo** byte della **Data Rom Window**.

Giunti a questo punto abbiamo **memorizzato** i **64** bytes della stringa dei dati nella **Data Rom Window**, ma per poterli utilizzare dovremo eseguire un'altra operazione, cioè caricare l'indirizzo del primo byte di questa stringa (nel nostro esempio sarebbe la **T** affinchè la nostra stringa inizi con **TE**-STO) in un registro (**x** , **y**, ecc.).

Se useremo il registro **x**, ogni volta che vorremo visualizzare questo testo dovremo come prima istruzione scrivere:

ldi x,test01.d

Come noterete l'etichetta **test01** è seguita da **.d**.

Se per errore scriveremo:

ldi x,test01

senza inserire **.d** in coda a **test01**, il compilatore ci segnalerà **errore** dal momento che tentiamo di caricare nel registro **x** un indirizzo di memoria **Program Space**.

Scrivendo correttamente:

ldi x,test01.d

quando verrà eseguita questa istruzione, nel registro **x** verrà caricato l'indirizzo di **Data Rom Window** di **test01** e cioè **40h**.

Se notate abbiamo detto **Data Rom Window** e non **Program Space** come dovrebbe essere dal momento che **test01** (e la relativa stringa di dati) è stato definito inizialmente all'indirizzo **A40h di Program Space**.

Quindi quando il **Compilatore** assembla questa istruzione:

ldi x,test01.d

divide l'indirizzo di **Program Space** di **test01** per **64**, preleva il **resto** di questa divisione, **somma** a questo **resto** il valore dicimale **64** e lo memorizza nell'istruzione stessa al posto dell'operando **test01.d**.

Nel nostro caso, poichè **test01** è stato definito all'indirizzo di **Program Space A40h** che corrisponde al valore decimale **2624**, avremo:

2624 : 64 = 41 con un resto **= 0**
resto 0 + 64 = 64

che in esadecimale corrisponde a **40h**.

Utilizzando il **software** simulatore possiamo anda-

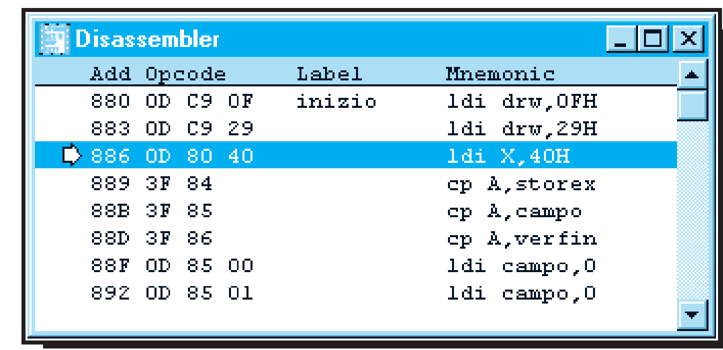


**Fig.3 Con il software simulatore potrete notare nella finestra del Disassembler, che l'istruzione:
ldi X,test01.d
si sarà tramutata in:
ldi X,40H**

re nella finestra Disassembler (vedi fig.3) e qui vedere che l'istruzione:

ldi x,test01.d

sarà diventata:

ldi x,40h

Quando verrà lanciato il programma assemblato, nel registro **x** verrà caricato il valore **40h** che corrisponde esattamente all'indirizzo di inizio della **Data Rom Window** (vedi fig.4) che, nel nostro caso, corrisponde al **primo byte** della stringa:

"TESTO DI PROVA "
"PER VISUALIZZARE"
"CARATTERI ALFAN"
"UMERICI - FINE - "

A questo punto con una routine ciclica che voi stessi potrete creare, sarà possibile "muovere" la stringa sul display e visualizzarla.

Molti si domanderanno che utilità pratica offre la sigla **.d** cioè:

ldi x,test01.d

quando per caricare nel registro **x** il valore **40h** lo stesso risultato lo potremmo ottenere scrivendo semplicemente:

ldi x,40h

Infatti in entrambi i casi nel registro **x** verrebbe sempre caricato l'indirizzo iniziale della **Data Rom Window**.

La soluzione di scrivere **ldi x,40h** che sembrerebbe anche la più semplice, è da **scartare** e con gli esempi che ora riporteremo ne capirete il motivo.

Ammettiamo di avere un programma che utilizza un display **alfanumerico** composto di **2 righe** di **16** caratteri cadauna e che all'inizio occorra visualizzare:

INSERIMENTO DATI
-PROVA DISPLAY-

poi successivamente:

PREMI PULSANTE
-SPEGNI DISPLAY-

Il testo da visualizzare lo abbiamo definito così all'interno del programma:

```
A40h   test01   .ascii "INSERIMENTO DATI"
A50h            .ascii "-PROVA  DISPLAY- "
A60h   test02   .ascii "PREMI PULSANTE
A70h            .ascii "-SPEGNI DISPLAY-"
```

In questo caso abbiamo **4** direttive **.ascii** lunghe ciascuna **16** bytes.

Fig.4 Nel registro X verrà caricato il valore di 40H che contiene il nostro testo.

Abbiamo aggiunto a sinistra le locazioni di memoria **Program Space** di queste direttive, cioè:

A40h - A50h - A60h - A70h

Come noterete vi sono anche **2** etichette **test01** e **test02** associate rispettivamente alle locazioni **Program Space A40h** e **A60h.**

Se ora scriviamo:

ldi drw,test01.w

a partire dall'etichetta **test01** verranno caricati in **Data Rom Window** tutti i **64 bytes.**

Se si dispone di un **software** simulatore che permette di visualizzare la **Data Rom Window** si potrà infatti vedere memorizzato (vedi fig.5):

INSERIMENTO DATI
-PROVA DISPLAY-
PREMI PULSANTE
-SPEGNI DISPLAY-

Se noi scriviamo:

ldi x,test01.d

nel registro **x** verrà caricato l'indirizzo **40h.**

A questo punto vi sarà una routine che provvederà a portare i **32** caratteri del primo testo da visualizzare, pertanto sul display comparirà la scritta:

INSERIMENTO DATI
-PROVA DISPLAY-

Siccome questo programma non prevede l'utilizzo di altri dati da caricare in **Data Rom Window**, i **64** bytes inizialmente **caricati** sono ancora memorizzati in questa area, pertanto per far apparire sui display il secondo testo:

PREMI PULSANTE
-SPEGNI DISPLAY-

sarà sufficiente scrivere:

ldi x,test02.d

e, così facendo, nel registro **x** verrà caricato l'indirizzo di memoria della **Data Rom Window** corrispondente al **primo** byte del secondo testo (etichetta **test02**) e cioè **60h.**

Quando il **Compilatore** assembla l'istruzione:

ldi x,test02.d

divide l'indirizzo di **Program Space** di **test02** per **64**, poi al **resto** di questa divisione **somma 64** e **memorizza** il risultato così ottenuto nell'istruzione stessa al posto dell'operando **test02.d.**

Nel nostro caso, siccome **test02** è stato definito all'indirizzo di **Program Space A60h** che corrisponde al valore decimale **2656**, avremo:

2656 : 64 = 41,5 (rimane **0,5**)
64 x 0,5 = 32 sarebbe il **resto**
resto 32 + 64 = 96

che in esadecimale corrisponde a **60h.**

Se disponete di un **software** simulatore potrete verificare nella finestra **Disassembler**, che la nostra istruzione:

ldi x,test02.d

dopo la compilazione (vedi fig.6) sarà diventata:

ldi x,60h

Quando verrà lanciato il programma assemblato, eseguendo questa istruzione nel registro **x** verrà caricato il valore **60h** che corrisponde esattamente all'indirizzo di inizio del secondo testo (etichetta **test02**) in **Data Rom Window** e cioè:

PREMI PULSANTE
-SPEGNI DISPLAY-

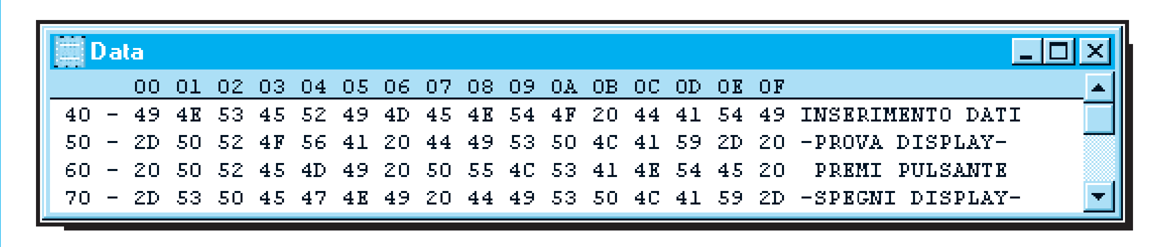


Fig.5 Anche gli spazi sono considerati caratteri e vengono memorizzati.

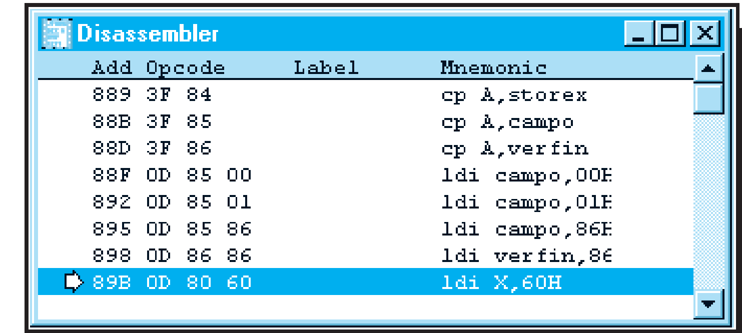


Fig.6 Se controllate con il software la finestra del Disassembler, potrete notare che l'istruzione:
ldi X,test02.d
si sarà tramutata in:
ldi X,60H

e una successiva routine provvederà a visualizzare sui display i **32** caratteri di questo testo.

Questo vi fa capire l'importanza dell'utilizzo della sigla **.d**, che permette di caricare nel registro voluto l'indirizzo di inizio di una stringa di dati memorizzati in **Data Rom Window**, quando al suo interno vi sono più stringhe indirizzabili.

Nel prossimo esempio vi spieghiamo cosa succede se i dati definiti in **Program Space** non iniziano esattamente con un **indirizzo** che non è esattamente un **multiplo** di **64**.

In questo esempio infatti abbiamo volutamente inserito un **errore** nella definizione delle stringhe.

Prendiamo sempre come esempio un programma che deve gestire un display alfanumerico composto di **2** righe di **16** caratteri, che faccia apparire sulla **prima** riga del display la scritta:

INIZIO

Supponiamo che in un secondo tempo questa scritta scompaia e in sua sostituzione appaia:

INIZIALIZZAZIONE

che poi anche questa scritta scompaia per essere sostituita dalla scritta:

ATTIVAZIONE BOX

che, dopo pochi secondi, anche la scritta sopra riportata scompaia per essere sostituita da:

BOX OK

Infine, che scompaia anche questa scritta per essere sostituita da una scritta che faccia apparire sulle due **2 righe** del display:

-ATTENDERE FASE-
-DI SPEGNIMENTO-

Nell'ultima istruzione del programma che termina all'indirizzo di **Program Space C03h** abbiamo definito le stringhe da visualizzare direttamente al byte successivo senza utilizzare la direttiva **.block 64-$%64** (vedi rivista 189):

```
C04h   test01   .ascii "INIZIO"
C0Ah   test02   .ascii "INIZIALIZZAZIONE"
C1Ah   test03   .ascii "ATTIVAZIONE "
C26h   test04   .ascii "BOX  OK"
C2Ch   test05   .ascii "-ATTENDERE FASE-"
C3Ch            .ascii "-DI SPEGNIMENTO-"
```

Tralasciamo tutte le istruzioni del programma che **non** sono strettamente legate all'argomento che stiamo trattando ed arriviamo subito alla **fase** in cui il programma deve visualizzare sul display queste scritte.

Innanzitutto il programma le carica in **Data Rom Window** con l'istruzione che già conoscete:

ldi drw,test01.w

Poichè, contrariamente agli esempi precedenti, **test01** è stato definito volutamente ad un indirizzo **C04h** di **Program Space** che in **decimale** corrisponde a **3.076**, valore non divisibile per **64**, avremo dei decimali:

3.076 : 64 = 48,0625

Quando il Compilatore assembla questa istruzione con **.w**, in pratica divide l'indirizzo di **Program Space** dell'etichetta per **64**, perdendo il **resto** e **memorizza** il risultato della divisione nell'istruzione stessa al posto dell'operando **test01.w**.

Quindi al posto dell'operando **test01.w** il Compilatore sostituirà il valore **48** che in esadecimale corrisponde a **30h**.

Se disponete di un **software** simulatore troverete che nella finestra Disassembler l'istruzione:

ldi drw,test01.w

sarà diventata:

ldi drw,30h

Una volta assemblato e lanciato il programma viene eseguita questa istruzione e il risultato viene memorizzato nel **Data Window Register**.

Il valore contenuto nel **Data Window Register** indicherà l'indirizzo (espresso in blocchi di **64** bytes) della stringa di dati che verrà così caricata in **Data Rom Window**.

Il microprocessore riconosce che il registro **drw** definito alla locazione di memoria **C9h** è il **Data Window Register**, quindi carica in **Data Rom Window** **64** bytes del nostro testo a partire dal **48°** blocco (di **64 bytes** l'uno) di **Program Space**.

Se prendiamo il valore **48** contenuto nel **Data Window Register** e lo moltiplichiamo per **64** otterremo:

48 x 64 = 3072

Espresso in esadecimale **3072** vale **C00h**.

Perciò come risultato finale l'istruzione:

ldi drw,test01.w

caricherà in **Data Rom Window**, **64** bytes di dati definiti in **Program Space** a partire dall'indirizzo **C00h**, cioè **4 bytes** prima e non da **C04h** come richiesto.

Eseguendo una simulazione di questo programma nella finestra di **Data Rom Window** (vedi fig.7), vedrete nella parte di **sinistra** le locazioni di memoria e il contenuto espresso in esadecimale, mentre nella parte **destra** la relativa decodifica in caratteri **ASCII**.

Negli indirizzi da **40h** a **43h** saranno entrati dei valori che nulla hanno a che vedere con le nostre stringhe: di conseguenza perderemo dei dati ed infatti noteremo che l'ultima stringa riporterà solo:

ENDERE FASE —DI

quindi verrà perso **SPEGNIMENTO-.**

La stringa **INIZIO** anche se parte da **44h** anzichè da **40h** riusciremo sempre a visualizzarla.

Quindi quando digiteremo l'istruzione con **.d**:

ldi x,test01.d

il **Compilatore** assemblerà questa istruzione dividendo l'indirizzo di **Program Space** di **test01** per **64**, poi preleverà il **resto** di questa divisione, **sommerà** a questo **resto** il valore decimale **64** e lo memorizzerà nell'istruzione stessa al posto dell'operando **test01.d**.

Nel nostro caso, poichè **test01** è stato definito all'indirizzo di **Program Space C04h** che corrisponde al valore decimale **3076**, avremo:

3076 : 64 = 48,0625 (rimane **0,0625**)
64 x 0,0625 = 4 (che sarebbe il **resto**)
resto 4 + 64 = 68

che in esadecimale corrisponde a **44h**.

Nota = L'esempio dei **resti** in tutte le operazioni che abbiamo riportato sono quelli che ci ritroveremo usando una normale calcolatrice tascabile.

Se disponete di **software** simulatore potrete notare che nella finestra **Disassembler** (vedi fig.8) la nostra istruzione:

ldi x,test01.d

si sarà convertita in:

ldi x,44h

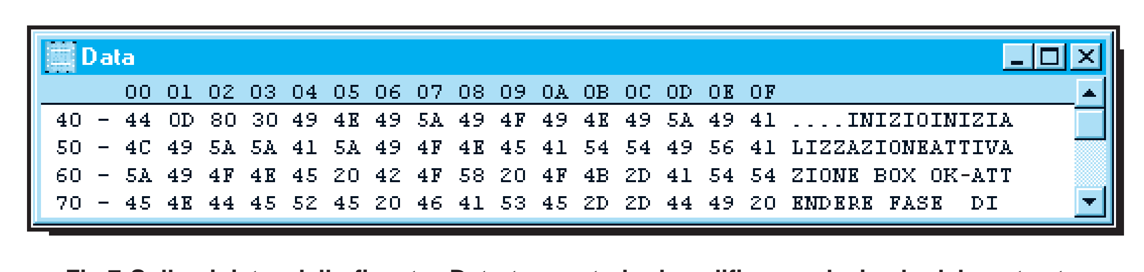


Fig.7 Sulla sinistra della finestra Data troverete la decodifica esadecimale del nostro testo e sulla destra la decodifica ASCII (49 = I, 4E = N, 5A = Z, 4F = O).

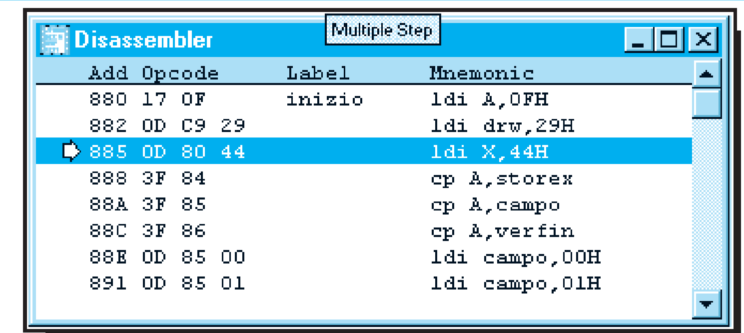


**Fig.8 Se controllate con il software la finestra del Disassembler, potrete notare che l'istruzione:
ldi X,test01.d
si sarà tramutata in:
ldi X,44H**

Quando verrà lanciato il programma assemblato, nel registro **x** verrà caricato il valore **44h.**

Se guardate la fig.7 potrete constatare che in **Data Rom Window** l'indirizzo **44h** corrisponde in effetti al primo byte della stringa "**INIZIO**".
A questo punto occorre soltanto inserire una routine ciclica che provvederà a portare i **6** caratteri del testo **INIZIO** sul display.

Successivamente il programma deve visualizzare la scritta:

INIZIALIZZAZIONE

e se guardate in fig.7 vedrete che anche la stringa "**INIZIALIZZAZIONE**" è completa e quindi per visualizzarla si dovrà digitare:

ldi x,test02.d

Quando il **Compilatore** assembla questa istruzione, divide l'indirizzo di **Program Space** di **test02** per **64**, preleva il resto di questa divisione, **somma** a questo **resto** il valore decimale **64** e lo memorizza nell'istruzione al posto dell'operando **test02.d.**

Nel nostro caso, poichè **test02** è stato definito all'indirizzo di **Program Space C0Ah** che corrisponde al valore decimale **3082**, avremo:

3082 : 64 = 48,15625 (rimane **0,15625**)
64 x 0,15625 = 10 (che sarebbe il **resto**)
resto 10 + 64 = 74

che corrisponde al numero esadecimale **4Ah.**

Se controllate la finestra Disassembler, noterete che l'istruzione:

ldi x,test02.d

sarà diventata:

ldi x,4Ah

Quando verrà lanciato il programma assemblato, l'istruzione soprariportata caricherà nel registro **x** il valore **4Ah.**

Se guardate in fig.7 potrete constatare che in **Data Rom Window** l'indirizzo **4Ah** corrisponde in effetti al primo byte della stringa "**INIZIALIZZAZIONE**".

A questo punto dovremo creare una routine ciclica che provveda a portare i **16** caratteri del testo **INIZIALIZZAZIONE** sul display.

Proseguendo, dopo un certo tempo il programma deve visualizzare la scritta:

ATTIVAZIONE BOX

Anche questa stringa **ATTIVAZIONE BOX** è completa in **Data Rom Window**, perciò alla istruzione:

ldi x,test03.d

quando il **Compilatore** assembla questa istruzione divide l'indirizzo di **Program Space** di **test03** per **64**, preleva il resto di questa divisione, **somma** a questo **resto** il valore decimale **64** e lo memorizza nell'istruzione al posto dell'operando **test03.d.**

Nel nostro caso, poichè **test03** è stato definito all'indirizzo di **Program Space C1Ah** che corrisponde al valore decimale **3098**, avremo:

3098 : 64 = 48,40625 (rimane **0,40625**)
64 x 0,40625 = 26 (che sarebbe il **resto**)
resto 26 + 64 = 90

che in esadecimale corrisponde al numero **5Ah.**

Osservando la finestra Disassembler noterete che l'istruzione:

ldi x,test03.d

sarà diventata:

ldi x,5Ah

Quando verrà lanciato il programma assemblato l'istruzione soprariportata caricherà nel registro **x** il valore **5Ah**.

Se guardate in fig.7 potrete constatare che in **Data Rom Window** l'indirizzo **5Ah** corrisponde in effetti al primo byte della stringa "**ATTIVAZIONE BOX**".
A questo punto andrà solo inserita una routine ciclica che provvederà a portare i **16** caratteri del testo **ATTIVAZIONE BOX** sul display.

Lo stesso dicasi per la scritta:

BOX OK

per visualizzare la quale occorre solo digitare:

ldi x,test04.d

A questo punto il programma per terminare deve visualizzare sulle **2 righe** del display:

–ATTENDERE FASE–
–DI SPEGNIMENTO–

ma, se ricordate (vedi fig.7), in **Data Rom Window** risulta caricata soltanto:

–ATTENDERE FASE—DI

perchè le stringhe dichiarate sono più lunghe di **64** bytes, infatti in totale abbiamo **72** bytes.

Poichè la prima stringa l'abbiamo definita per **errore** all'indirizzo **C04h** anzichè **C00h**, osservando la fig.9 potrete capire perchè si perde l'ultima parola:

SPEGNIMENTO

A questo punto per visualizzare la scritta completa verrebbe logico pensare che risulti sufficiente ricaricare la frase completa:

"-ATTENDERE FASE-"
"-DI SPEGNIMENTO-"

in **Data Rom Window** con l'istruzione:

ldi drw,test05.w

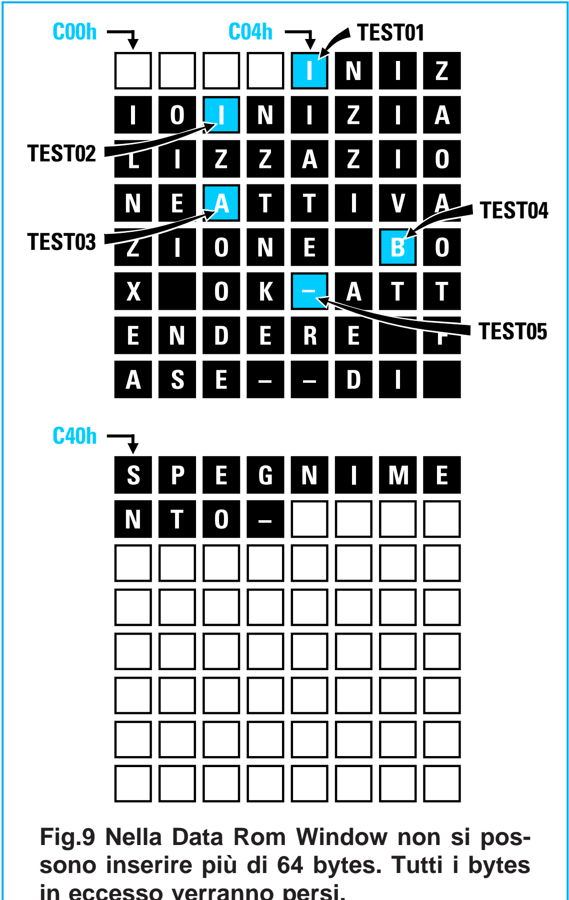


Fig.9 Nella Data Rom Window non si possono inserire più di 64 bytes. Tutti i bytes in eccesso verranno persi.

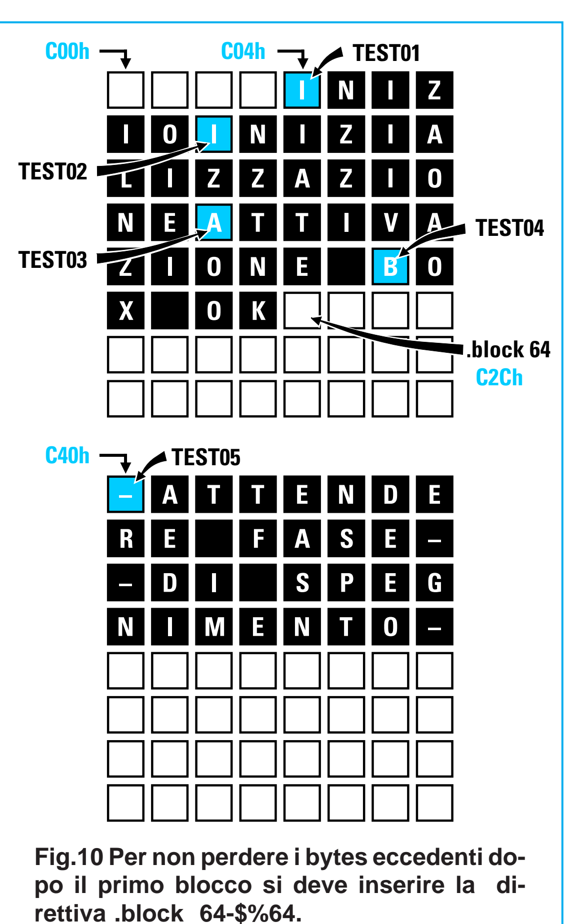


Fig.10 Per non perdere i bytes eccedenti dopo il primo blocco si deve inserire la direttiva .block 64-$%64.

e poi indirizzarla nel registro **x** scrivendo:

ldi x,test05.d

e successivamente visualizzarla con una routine sul nostro display.

Se eseguirete queste due operazioni commettereste il più grossolano degli **errori**.

Infatti, come già saprete, con l'istruzione:

ldi drw,test05.w

l'indirizzo di **Program Space** dell'etichetta **test05** viene diviso per **64.** Il risultato di questa divisione viene poi memorizzato nel **Data Window Register (drw)** perdendo l'eventuale **resto**.

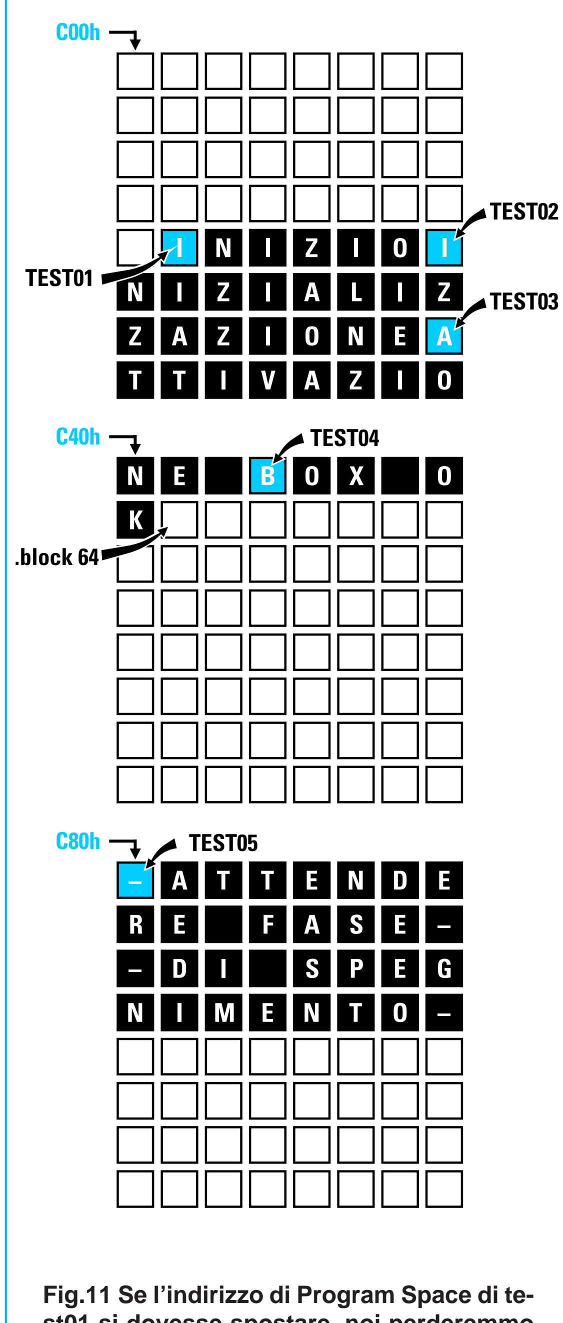


Fig.11 Se l'indirizzo di Program Space di test01 si dovesse spostare, noi perderemmo tutte le parole che fuoriescono dall'area del primo blocco da 64.

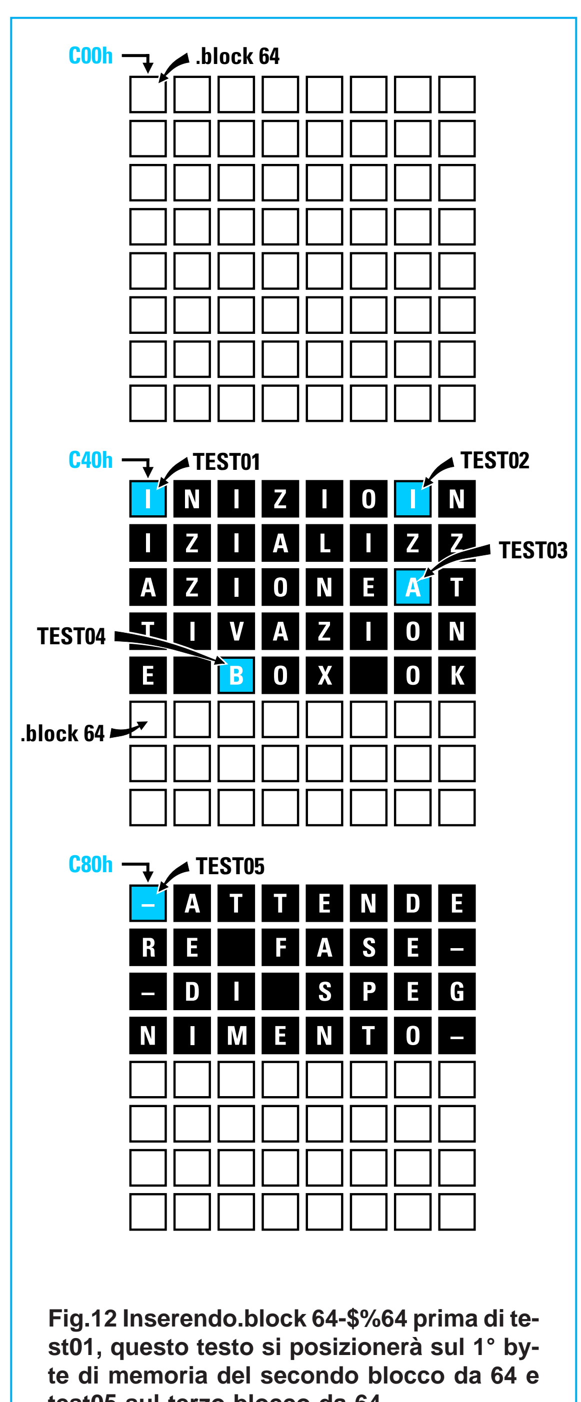


Fig.12 Inserendo.block 64-$%64 prima di test01, questo testo si posizionerà sul 1° byte di memoria del secondo blocco da 64 e test05 sul terzo blocco da 64.

Nel nostro caso **test05** è stato definito all'indirizzo di **Program Space C2Ch** che, convertito in decimale, corrisponde al numero **3116.**
Se divideremo questo numero per 64 otterremo:

3116 : 64 = 48,6875 (rimane **0,6875**)
64 x 0,6875 = 44 (che sarebbe il **resto**)

Poichè in questo caso **perdiamo** il **resto 44**, ci rimane il solo numero **48** che verrà caricato nel **Data Window Register**.

Se moltiplichiamo **48** per **64** bytes otterremo sempre il numero:

48 x 64 = 3072

che corrisponde al valore esadecimale **C00h**, quindi come risultato finale l'istruzione:

```
ldi drw,test05.w
```

caricherà in **Data Rom Window, 64** bytes di dati definiti in **Program Space**, a partire dall'indirizzo **C00h**; pertanto ci ritroveremo sempre nella condizione di fig.7, cioè con la frase incompleta.

Per ovviare a questo inconveniente dovremo inserire la **direttiva**:

.block 64 - $%64

come qui sotto riportato:

```
        .block    64 - $%64
test05  .ascii “–ATTENDERE FASE–”
        .ascii “–DI SPEGNIMENTO–”
```

quindi avremo:

```
test01  .ascii “INIZIO”
test02  .ascii “INIZIALIZZAZIONE”
test03  .ascii “ATTIVAZIONE “
test04  .ascii “BOX OK”
        .block 64 - $%64
test05  .ascii “–ATTENDERE FASE–”
        .ascii “–DI SPEGNIMENTO–”
```

Nella rivista **N.189** abbiamo spiegato la funzione completa di **.block 64 - $%64**.

Dal punto in cui viene definito **.block 64 - $%64** il Compilatore calcola quanti bytes deve lasciare **liberi** per posizionare l'inizio della stringa **test05** sul **primo** byte del blocco successivo di **64** bytes come è possibile vedere in fig.10.

In pratica, quando il programma sarà **Compilato** in Assembler, la direttiva:

```
.block 64 - $%64
```

definirà un'area di **20** bytes **vuota** a partire dalla locazione di **Program Space C2Ch** così che la successiva definizione:

```
test05  .ascii “–ATTENDERE FASE–”
        .ascii “–DI SPEGNIMENTO–”
```

inizi dalla locazione di memoria **Program Space C40h** corrispondente al valore decimale **3136.**

A questo punto l'istruzione:

```
ldi drw,test05.w
```

caricherà in **Data Rom Window** il testo completo:

–ATTENDERE FASE–
–DI SPEGNIMENTO–

Per poterla caricare in un registro dovremo semplicemente scrivere:

```
ldi x,test05.d
```

Per poter trasferire questa frase sul display dovremo utilizzare una routine ciclica.

Con la soluzione soprariportata non pensate di aver risolto il problema, purtroppo è ancora presente un **errore**.

Come potete vedere in fig.10 la stringa **test01** non inizia dalla locazione **C00h** e per questo motivo la stringa **test04** termina alla locazione **C2Ch**.
Avendo inserito in questo punto **.block 64 - $%64** obbligheremo **test05** ad iniziare dalla locazione **C40H**.

Se eseguendo dei test ci trovassimo costretti a **inserire** o **togliere** nel programma delle istruzioni, è ovvio che l'indirizzo di **Program Space** delle nostre stringhe verrebbe automaticamente **variato** e potremmo così correre il rischio di **non** caricare in **Data Rom Window** tutto il nostro testo.

Ammettiamo che in fase di controllo abbiamo dovuto inserire delle nuove istruzioni all'interno del programma e che, in questo modo, **test01** passi da **C04h** all'indirizzo **C21h** (vedi fig.11).

Se ora caricassimo in **Data Rom Window** la stringa **test01** perderemmo le ultime lettere:

NE -BOX -OK

perchè fuoriescono dall'area **64** bytes di **Data Rom Window**.
Non avremo invece nessun problema per **test05** perchè ce lo ritroveremo nella successiva area di memoria **C80h** (vedi fig.11).

Per evitare questo **errore** è sufficiente inserire **.block 64 - $%64** prima della stringa **test01** come qui sottoriportato:

```
        .block  64 - $%64
test01  .ascii "INIZIO"
test02  .ascii "INIZIALIZZAZIONE"
test03  .ascii "ATTIVAZIONE "
test04  .ascii "BOX  OK"
        .block 64 - $%64
test05  .ascii "-ATTENDERE FASE-"
        .ascii "-DI SPEGNIMENTO-"
```

Non dovremo perciò più preoccuparci se durante il **test** del programma **aggiungiamo** o **togliamo** istruzioni dal programma, perchè automaticamente **.block 64-$%64** provvederà a calcolare l'area necessaria per allineare le definizioni di dati al blocco ottimale (divisibile per **64**) di **Program Space** come visibile in fig.12.

Come visibile in fig.12 la stringa di **test01** partirà sempre dal primo byte di memoria del secondo blocco da **64**, che nel nostro esempio è **C40h**.

Il successivo **blocco** di **test05** inizierà dalla locazione di memoria **C80h**.

Dopo questa spiegazione molti, per evitare di incorrere in uno degli **errori** sopracitati, abuseranno di questo **.block 64 - $%64**, ma in questo modo potrebbero sprecare inutilmente molti blocchi di **64** bytes di **Program Space**.

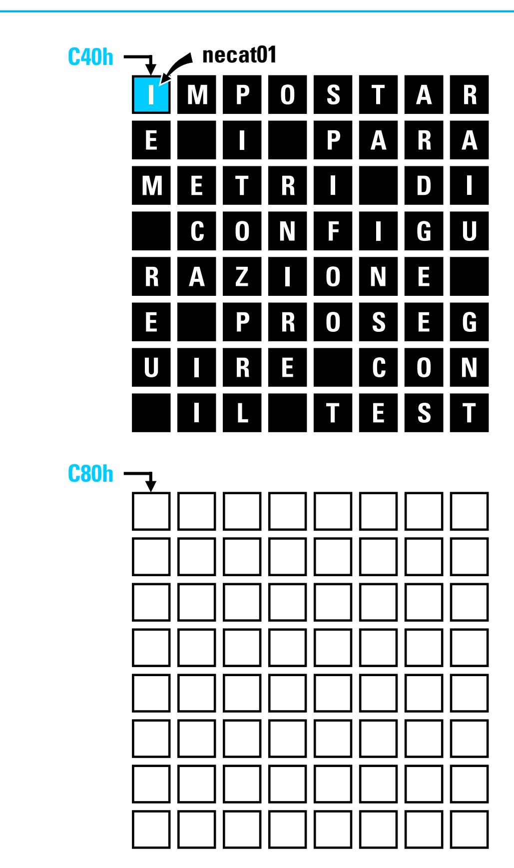


Fig.13 Se la stringa di dati inizia esattamente da un indirizzo di memoria Program Space divisibile per 64 senza resto, non è necessario inserire .block 64-$%64.

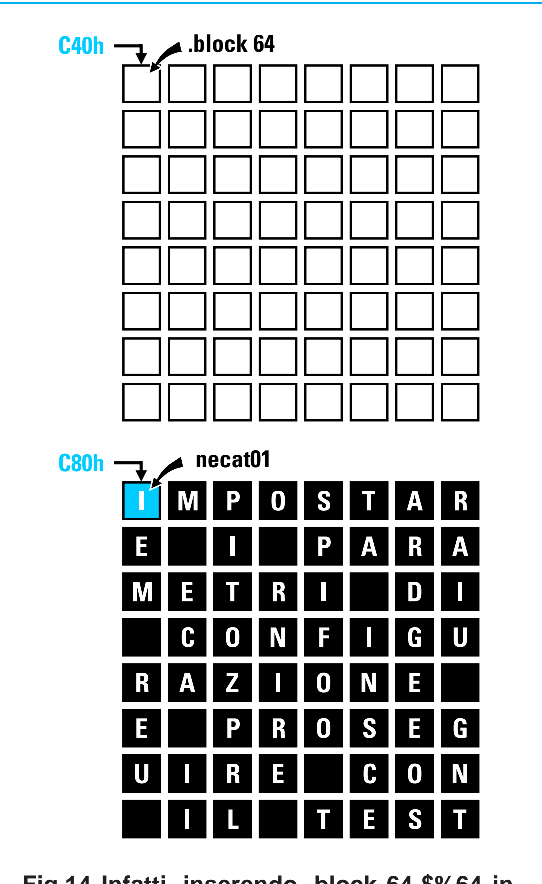


Fig.14 Infatti, inserendo .block 64-$%64 in una locazione di memoria divisibile per 64 senza resto, il testo viene posto nel blocco seguente sprecando l'area del primo.

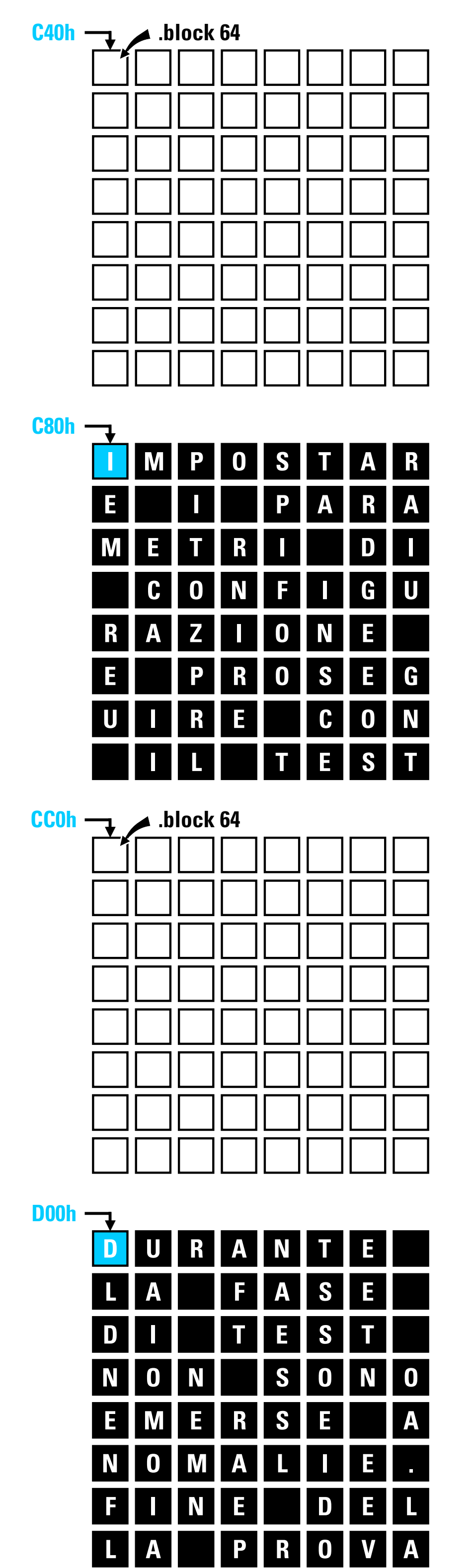


Fig.15 In questo esempio non è stata usare la direttiva .ifc ne ($%64) e per questo motivo il 1° e il 3° blocco sono vuoti.

Questo spreco blocchi si verifica ogniqualvolta la direttiva **.block 64 - $%64** viene inserita in una locazione di memoria di **Program Space** perfettamente divisibile per **64** senza **resto**.

Ammettiamo di avere una stringa di dati lunga esattamente **64 bytes** (vedi fig.13) che inizia da **C40h**, cioè:

```
necat01   .ascii "IMPOSTARE PARA"
          .ascii "METRI DI CONFIGU"
          .ascii "RAZIONE E PROSEG"
          .ascii "UIRE CON IL TEST"
```

Durante la stesura del programma, se non eseguiremo dei calcoli, non sapremo mai se questa stringa inizia esattamente da **C40h**; pertanto se, per evitare errori, prima di questa stringa inseriamo **.block64-$%64**, il Compilatore provvederà a collocare questa stringa all'indirizzo **C80h** sprecando i precedenti **64** bytes come illustrato in fig.14.

Poichè di questi casi se ne potrebbero presentare diversi, sprecheremo inutilmente molti blocchi di memoria (vedi fig.15).

Per ovviare a questo inconveniente c'è una semplice soluzione che non tutti conoscono, che utilizza la **direttiva**:

.ifc che significa **Compila solo se...**

Questa **direttiva** ci permette di compilare parti di programma o di inserire moduli a scelta **solo se** sono **Vere** (o **False**) le condizioni specificate.

Pertanto se nel programma, **prima** delle definizioni di **dati** in **Program Space** inseriremo:

```
 .ifc ne ($%64)
       .block 64 - $%64
 .endc

necat01   .ascii "IMPOSTARE I PARA"
          .ascii "METRI DI CONFIGU"
          .ascii "RAZIONE E PROSEG"
          .ascii "UIRE CON IL TEST"
```

otterremo che **.block 64 - $%64** sarà **Compilata** in assembler solo se **non** si trova in una locazione di **Program Space** divisibile esattamente per **64**.

Quindi con le **tre** istruzioni riportate prima della stringa **necat01**, il Compilatore compila l'istruzione **.block64-$%64** solo se il risultato della **Espressione ($%64)** non è uguale a **zero** (ne).

Facciamo presente che la direttiva **.endc** deve essere **sempre** inserita come istruzione **finale** quando si usa **.ifc**.

Nella rivista **N.189** abbiamo spiegato cosa sono le Espressioni e come procedere al loro svolgimento.

LE DIRETTIVE dell'assembler ST6

Poichè in nessun manuale è spiegato in modo comprensibile come usare correttamente le Direttive dell'Assembler dell'ST6, cercheremo di risolvere questo problema spiegandovi anche tutti quei piccoli segreti di cui pochi sono al corrente. In questo articolo, tutto sulle direttive .ASCII .ASCIZ .DEF.

Già saprete che la **memoria** dell'**ST6** è suddivisa in:

Memoria Ram = definita anche **Data Space**
Memoria Rom = definita anche **Program Space**

La **Memoria Ram** è **riscrivibile**, quindi si utilizza nei programmi come **memoria dinamica** per memorizzare risultati di calcoli o di dati variabili in apposite **celle** e, poichè è di tipo **volatile**, quando viene tolta tensione al microprocessore questi dati vengono automaticamente **cancellati**.

La **Memoria Rom** (**R**ead **o**nly **m**emory) si utilizza per inserire le **istruzioni** del programma, quindi una volta che queste risultano memorizzate nel microprocessore non si possono più modificare nè cancellare.

LA DIRETTIVA chiamata .ASCII

La direttiva **.ascii** serve per definire dei **dati** nella **Program Space** che è l'area **Rom** riservata alle istruzioni del programma.

In pratica questa direttiva viene utilizzata per definire nella **Program Space** delle stringhe di caratteri **alfanumerici** e per associare eventuali **etichette** in quei programmi che generano messaggi o scritte di vario genere su **video** o su **stampa**.

Ogni tentativo di utilizzarla per definire **dati** nella **Data Space** darà un **errore** di compilazione.

La **lunghezza** in **bytes** è definita dal numero di caratteri **ascii** inseriti fra le **virgolette**.

Chiaramente queste stringhe così definite non sono modificabili durante il corso del programma perchè definite nella memoria **ROM**.

Per poterle utilizzare dovremo **caricarle** in **Data Rom Window** con le stesse modalità e gli stessi accorgimenti già spiegati nella **rivista N.190** nel paragrafo riguardante la direttiva **.w_on**.

Anche per la loro definizione, in fase di stesura del programma, bisognerà attenersi a quanto riportato

nella parte riguardante la direttiva **.block** sempre descritta nella **rivista N.190**.

Il suo utilizzo permette di usufruire di una notevole quantità di **messaggi** in **Program Space** senza **riempire** inutilmente l'area di **Data Space** che, disponendo di soli **60 bytes**, potremo sfruttare per delle **Variabili** tramite la direttiva **.def**.

Il formato logico della direttiva **.ascii** è il seguente:

[etichetta] .ascii "stringa"

[etichetta] = Nome dell'etichetta che si vuole associare al **primo byte** della stringa. Questo nome è opzionale quindi può essere anche omesso.

"stringa" = In tale stringa si inseriscono i caratteri **alfanumerici** che si vogliono definire in **Program Space**, racchiudendoli sempre fra **virgolette**.

Per rendere più chiaro quanto detto finora vi proponiamo questo semplice esempio:

scritta1 .ascii "/*-INIZIO-*\"
.ascii "/*-Premi-P1 > per uscire"

La prima stringa di caratteri /***-INIZIO-*** è composta da **12 bytes**, mentre la seconda stringa di caratteri, cioè /***-Premi-P1 > per uscire**, è composta da **24 bytes**. Facciamo presente che gli **spazi** sono anche questi dei **caratteri** alfanumerici, quindi vanno conteggiati.

In fase di **Compilazione** l'**Assembler** definisce nella **Program Space** la seguente stringa:

/***-INIZIO-*\/*-Premi P1 > per uscire**

per un **totale** di **12 + 24 = 36 bytes**

ed associa all'indirizzo di **Program Space** del **primo** byte della stringa l'etichetta **scritta1**.

LA DIRETTIVA chiamata .ASCIZ

Anche questa direttiva viene utilizzata per definire in **Program Space** delle stringhe di caratteri **alfanumerici** ed associarvi eventuali **etichette**.

Con la direttiva **.ASCIZ**, il **compilatore** inserisce in coda ad ogni singola stringa **1 byte** contenente il valore **00h** che è un carattere non editabile (**null**).

La lunghezza in bytes di questa stringa è definita dal numero di **caratteri** alfanumerici inseriti fra le virgolette, **addizionando** a questi **1 byte**.

Perciò se scriviamo:

scritta1 .asciz "/*-INIZIO-*\"
.asciz "/*-Premi-P1 > per uscire"

la prima stringa risulterà composta da:
12 bytes + 1 = 13
mentre la seconda stringa sarà composta da:
24 bytes +1 = 25

In fase di compilazione verrà definita nella **Program Space** la seguente stringa:

/***-INIZIO-*\(null)**/***-Premi P1 > per uscire (null)**

dove **(null)** rappresenta **1 byte** contenente **00h** che non è rappresentabile in formato **ASCII**, pertanto questa stringa risulterà lunga:

12 + 1 + 24 + 1 = 38 bytes

Quindi per definire delle **stringhe** di **dati** in **Program Space** ogni programmatore potrà scegliere indifferentemente sia **.ascii** che **.asciz**.

Scegliendo **.ascii** il compilatore **Assembler** definisce nella **Program Space** la seguente stringa:

/***-INIZIO-*\/*-Premi T1 > per uscire**

Se, per esempio, si desidera far apparire sul monitor la parola della **prima** stringa /***-INIZIO-*** composta di **12 caratteri**, dovremo realizzare una **routine** che conti esattamente i 12 caratteri da inviare sul video, altrimenti si correrà il rischio di veder apparire anche caratteri della **seconda** stringa.

Se si utilizza **.asciz**, il compilatore **Assembler** definisce nella **Program Space** la seguente stringa:

/***-INIZIO-*\(null)**/***-Premi P1 > per uscire (null)**

e in questo caso non è necessario realizzare una **routine** che **conti** i caratteri, ma una diversa **routine** che provveda ad inviare sul video tutti i caratteri della stringa precisando che si deve **fermare** quando incontra **00h** = **null**.
In pratica questo **00h** equivale ad un comando di **stop** lettura.

LA DIRETTIVA chiamata .DEF

La direttiva **.def** viene utilizzata per definire delle **etichette** associandole ad una cella di memoria di **Data Space** il cui indirizzo, come già saprete, è contenuto nell'operando **indiriz**.

Il formato logico della direttiva **.def** è il seguente:

[etich] **.def** indiriz,[R-mask],[W-ask],[value],[M]

Nota = Gli operandi posti fra parentesi quadra sono **opzionali** e possono essere omessi, togliendo anche le parentesi quadre. Inserite tutte le **virgole** come visibile nell'esempio.

[etich] = nome della **variabile** che si vuole associare all'indirizzo di memoria.

indir, = è l'indirizzo della **cella** di memoria **Data Space**. Questo valore può essere in **Binario**, **Decimale**, **Esadecimale** o una **Espressione**.

[R-mask] = utilizzando questo operando potremo definire quale degli **8 bits** della variabile può risultare leggibile (**R** sta per READ = leggi).

Se ad esempio scriviamo:

pippo .def 08Dh,00100000b,

sapremo già che la variabile **pippo** risulta collocata nella locazione di Program Space **08Dh** e che l'operando che segue, cioè **00100000b**, è **R-mask**.

Di questa variabile risulta leggibile il solo **5° bit** perchè settato a **1** (vi ricordiamo che i bit si leggono da destra verso sinistra 7-6-**5**-4-3-2-1-0), mentre gli altri **non** risultano leggibili perchè settati a **0**.

Perchè **tutti** gli **8 bit** risultino leggibili occorre omettere **R-mask** e ciò si ottiene scrivendo semplicemente:

pippo .def 08Dh

[W-mask] = utilizzando questo operando potremo definire quali degli **8 bits** della variabile possono risultare scrivibili (**W** sta per WRITE = scrivi).

Se ad esempio scriviamo:

pippo .def 08Dh,00100000b,10000000b

sapremo già che la variabile **pippo** risulta collocata nella locazione di Program Space **08Dh** e che l'operando **00100000b** è **R-mask** e quello che segue, cioè **10000000b** è **W-mask**.

Di questa variabile risulta scrivibile solo il **7° bit** perchè settato a **1** (bit di sinistra), mentre gli altri **non** risultano scrivibili perchè settati a **0**.

Se nell'istruzione vogliamo omettere **R-mask** ed utilizzare solo **W-mask** dovremo riportare due **virgolette** in sostituzione di **R-mask** come indicato in questo esempio:

pippo .def 08Dh,,10000000b

[value] = questo operando **non** risulta utilizzabile, quindi l'istruzione che abbiamo riportato in precedenza, cioè:

[etich] **.def** indiriz,[R-mask],[W-ask],[value],[M]

la potremo semplificare omettendo **value**, scrivendo quindi soltanto:

[etich] **.def** indiriz,[R-mask],[W-ask],[M]

[M] = questo operando (potremo anche scriverlo in **minuscolo**), se inserito, mette un **marker** nella **variabile** [etich] nel file **.DSD** (vedi fig.1).
Questo marker ci sarà utile in fase di **Debug**, perchè potremo automaticamente vedere su video, utilizzando ovviamente un **Simulatore**, tutte le variabili a cui è stato associato appunto un **marker**: ciò ci consentirà di controllarne il valore in **tempo reale** (vedi fig.2).

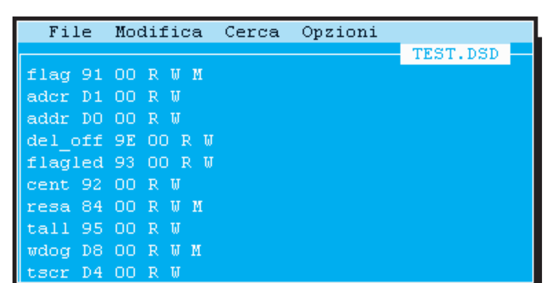


Fig.1 Poichè nel programma TEST.ASM, preso come esempio, in coda alle variabili flag - resa - wdog risulta inserita una M, in fase di compilazione nel file TEST.DSD apparirà la lettera M ad indicare che in queste tre variabili è presente un marker.

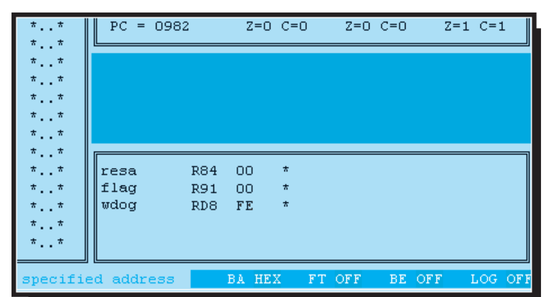
Dopo avervi spiegato il formato della direttiva **.def** ed il significato dei suoi operandi dovremo chiarire che cosa s'intende per **bit leggibili** e **bit scrivibili** ed indicare tutti i vantaggi di **R-mask** e **W-mask**.

Ad esempio se scriviamo questa istruzione:

prova jrs 3,store,finepr

il programma salterà (**jrs**) all'etichetta **finepr** solo se il **bit 3** della **variabile store** è settato.

In pratica l'istruzione **jrs** (jump relative set) deve quindi **leggere** lo stato del **bit 3** e se, per ipotesi,



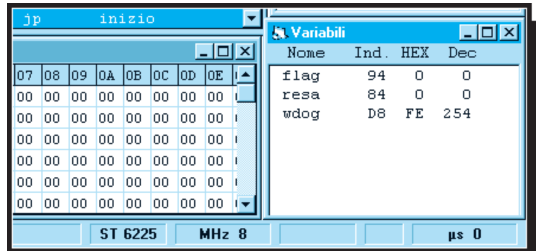


Fig.2 In fase di simulazione potrete vedere sul vostro monitor tutte le variabili contrassegnate con il marker M, complete del loro contenuto.
Ogni tipo di simulatore farà apparire sul monitor una sua particolare videata.

avessimo definito la variabile **store** come qui sotto riportato:

store .def 08Dh,11110111b

avremmo definito **leggibili** tutti i **bit** tranne il **bit 3**, perchè avremmo messo uno **0** anzichè **1**, quindi il compilatore segnalerà **errore**.

Perchè il compilatore non segnali **nessun** errore dovremo rendere leggibile il **bit 3** scrivendo:

store .def 08Dh,00001000b

Utilizzando **R-mask** e dichiarando "leggibili" i soli bits che ci interessano di una **variabile**, sarà lo stesso **compilatore** a trovare l'**errore** che potremmo aver commesso involontariamente, facendo apparire sul video l'istruzione che ha tentato di **leggere** il bit che **non** doveva leggere.

Lo stesso dicasi per **W-mask** quando si utilizzano le istruzioni che "scrivono" (**set** , **res** ,**ldi** ecc.) nei diversi bits.
In questi casi il **compilatore** non assemblerà il programma, quindi durante la **simulazione** eviteremo di trovare delle condizioni logiche non desiderate sui piedini del microprocessore.

Se poi il programma che stiamo scrivendo è molto complesso, oppure richiama molti **moduli** o **macro**, questo tipo di **errore** sul **test** o sul **settaggio** dei bits delle **variabili** sarà più frequente di quanto si possa supporre.

Ammettiamo per esempio di voler testare in una **variabile** i bits **1 - 4 - 6** con le istruzioni:

jrs 1,status,flag1
jrs 4,status,flag4
jrs 6,status,flag6

Se per errore scrivessimo come visibile in fig.4:

jrs 1,status,flag1
jrs 3,status,flag4 (errore)
jrs 6,status,flag6

quando il programma passerà sulla seconda riga dove è presente l'errore, **non** salterà mai sul **flag4**, quindi **non** usando **R-mask** perderemmo tempo prima di individuarlo.

Usando **R-mask** subito apparirà sul video il **numero** della riga dov'è presente l'errore (vedi **ASM 58**) con indicato il tipo di errore = **113** (vedi fig.5).

Vi sono comunque delle precise regole che dovremo osservare nel dichiarare **R-mak** e **W-mask** e per farvelo meglio comprendere vi proponiamo alcuni esempi:

1) Esempio

storex	**.def**	**084h,10001000b,00001111b,M**
prova	**jrs**	**0,storex,finepr**
	set	**1,storex**
finepr	**res**	**7,storex**

Fig.3 In questa figura vi facciamo vedere come appariranno sul monitor le cinque righe delle istruzioni del 1° esempio . Come potete notare, a fianco di ogni istruzione abbiamo riportato dopo il ; il numero della relativa riga.

```
seg1    .def    084h                    ; prova
seg2    .def    seg1+1
status  .def    086h,01010010B,        _; R_mask bit 1,4,6
;+--------------------------------------------------------
        .org    0800h

inizio  call    read_uno
        jrs     1,status,flag1
        jrs     4,status,flag4
        jrs     6,status,flag6
        jp      inizio
go_due  call    read_due
        jrs     1,status,flag1
        jrs     4,status,flag4
        jrs     6,status,flag6
        jp      go_due
go_tre  call    read_due
        jrs     1,status,flag1
        jrs     3,status,flag4
        jrs     6,status,flag6
        jp      go_tre
fine_t  call    termina
```

Fig.4 Inserendo R-MASK nella variabile STATUS, se commetterete un errore involontario (ad esempio jrs 3,status,flag4) il compilatore lo segnalerà immediatamente (vedi fig.5).

```
C:\ST6>ast6 -x -s -m C:\ST6\UTEST.ASM
ST6 MACRO-ASSEMBLER version 4.00  - August 1992
Error C:\ST6\UTEST.ASM 58: (113) no read-access rights to operand 2
Execution time: 1 second(s)
One error detected
No object created

C:\ST6>

Press a key to return..._
```

Fig.5 Poichè abbiamo tentato di compilare la riga 58 che è errata, il compilatore subito ce lo segnalerà indicandoci anche il tipo di errore = 113.

```
C:\ST6>ast6 -x -l -s C:\ST6\TEST.ASM
ST6 MACRO-ASSEMBLER version 4.00  - August 1992
Error C:\ST6\TEST.ASM 71: (113) no read-access rights to operand 2
Error C:\ST6\TEST.ASM 73: (114) no write-access rights to operand 2
Execution time: 0 second(s)
2 errors detected
No object created

C:\ST6>

Press a key to return...
```

Fig.6 Nel 1° esempio solo la istruzione 72 è corretta, mentre le due istruzioni 71-73 che sono errate vengono subito segnalate dal compilatore.

Con la prima istruzione:

storex .def 084h,10001000b,00001111b,M

il **compilatore** associa la variabile **storex** alla cella di memoria **084h** di **Data Space** e in più definisce nell' **R-mask** leggibili i soli bits **3-7**.
Nuovamente facciamo presente che i bits si leggono da destra verso sinistra **7-6-5-4-3-2-1-0**.
Poi definisce scrivibili nel **W-mask** i bits **0-1-2-3**.

Inoltre nel file **.DSD** (generato in compilazione) alla variabile **storex** viene associato un **marker**.

La successiva istruzione:

prova jrs 0,storex,finepr

significa **salta** all'etichetta **finepr** se il bit **0** di **storex** è settato.
Poichè nel **R-mask** il bit **0** non è leggibile, sul video apparirà un messaggio di **errore**.

La successiva istruzione:

set 1,storex

che significa metti a **1** il bit **1** di **storex**, viene correttamente compilata perchè il bit **1** definito in **W-mask** è "scrivibile".

L'ultima istruzione:

finepr res 7,storex

significa metti a **0** il bit **7** di **storex**.

In questo caso il **compilatore** farà apparire un messaggio di **errore** (vedi fig.5) perchè il bit **7** di **storex** definito nella **W-mask** non è "scrivibile" essendo presente **0**.

In presenza di questi **errori** dovremo ricontrollare tutto il programma per scoprire se l'istruzione utilizza un **bit sbagliato** oppure se sono sbagliati i bits inseriti in **R-mask** o **W-mask**.

2) Esempio (vedi fig.7)

```
a       .def    0FFh
storex  .def    084h,00000000b,00000111b
prova   ldi     storex,6
        ldi     storex,32
        ldi     a,32
        ld      storex,a
        cp      a,storex
```

Spiegazione:

Con la prima definizione:
a .def 0FFh
abbiamo definito l'accumulatore "**a**".

```
;+-------------------------------------------
;|      VARIABILI DEL PROGRAMMA
;+-------------------------------------------
storex  .def    084h,00000000B,00000111B
;+-------------------------------------------
        .org    0800h           ;73

prova   ldi     storex,6        ;74
        ldi     storex,32       ;75
        ldi     a,32            ;76
        ld      storex,a        ;77
        cp      a,storex        ;78
```

Fig.7 In questa figura vi facciamo vedere come si presentano sul monitor le sette righe delle istruzioni del 2° esempio. Di fianco ad ogni istruzione abbiamo riportato dopo il ; il numero di riga.

Con la seconda istruzione:

storex .def 084h,00000000b,00000111b

il **compilatore** associa la variabile **storex** alla cella di memoria **084h** di **Data Space** e in più definisce **non** leggibili gli **8** bits della variabile **R-mask** e scrivibili nella variabile **W-mask** i soli bits **0-1-2**.

La successiva istruzione:

prova ldi storex,6

significa carica nella variabile **storex** il valore decimale **6**.
Poichè si sa che **6** equivale a **00000110b** (vedi a **pag.381** del nostro volume **Handbook**), quando questo numero viene caricato nella variabile **storex** si riesce a modificare lo stato dei tre bits **0-1-2** perchè in **W-mask** li abbiamo configurati **scrivibili**, pertanto questa istruzione viene correttamente compilata.

La quarta istruzione:

ldi storex,32

significa carica nella variabile **storex** il valore decimale **32**.
Poichè si sa che **32** equivale a **00100000b** (vedi sempre **pag.381** del volume **Handbook**), quando viene caricato nella variabile **storex** non riuscirà a modificare lo stato del **quinto bit**, perchè questo in **W-mask** non è stato configurato scrivibile.
Infatti su questo bit è presente uno **0** e non un **1**.
In questo caso verrà subito segnalato un **errore** di compilazione (vedi ASM 75 in fig.8).

La quinta istruzione:

ldi a,32

significa carica nell'accumulatore "**a**" il valore **32**.

Questa istruzione viene correttamente compilata perchè nell'accumulatore "**a**" non abbiamo definito nè **R-mask** nè **W-mask**, comunque consigliamo di **non utilizzarle** mai nell'accumulatore perchè potrebbero bloccare qualche altra funzione.

La sesta istruzione:

ld storex,a

significa carica nella variabile **storex** il valore contenuto nell'accumulatore "**a**" che, nel nostro esempio, corrisponde al numero **32**.
Questa istruzione verrà compilata anche se sappiamo, per averlo spiegato nella **quarta** istruzione, che **non** è possibile caricare il valore **32** in **storex** perchè in **W-mask** sono stati configurati **scrivibili** i soli **bits 0-1-2**.

Il **compilatore** non può segnalare questo **errore** perchè, quando compila, non è in grado di controllare il contenuto nell'accumulatore "**a**".

Infatti per il compilatore è sufficiente che risulti scrivibile anche **uno solo** degli **8 bits** di **W-mask** di **storex** per ritenere questa istruzione corretta, quin-

di quando si usa una istruzione con **due** variabili, nel nostro esempio "**storex**" e "**a**", occorre sempre confrontare il valore contenuto nell'accumulatore "**a"** con i bits della **W-mask** della variabile **storex** durante la stesura del programma.

La settima istruzione:

cp a,storex

significa **confronta** il valore di "**a**" con il valore di **storex**.

Questa istruzione ci segnalerà **errore** (vedi ASM 78 in fig.8), perchè per eseguire un confronto fra i due valori l'istruzione **cp** deve **leggerli**, ma poichè in **R-mask** di **storex** è riportato **00000000b**, nessuno dei suoi bits è leggibile.

Come già vi abbiamo accennato, il numero **binario** di **32** è **00100000b**, quindi giustamente potreste pensare che si possa evitare l'errore rendendo leggibile il **quinto bit** della **R-mask** di **storex**. Purtroppo se ci sbagliamo e rendiamo leggibile il **quarto** bit o un **qualsiasi** altro bit, il compilatore non segnalerà piu' **nessun errore** perchè, non essendo in grado di controllare il contenuto nell'accumulatore "**a**", è sufficiente che un qualsiasi bit di **R-mask** risulti leggibile perchè esso ritenga valida l'istruzione.

Se ne volete una conferma inserite le istruzioni di questo nostro esempio in un qualsiasi vostro programma di prova, poi andate a modificare la **R.mask** di **storex** da:

storex .def 084,**00000000b**,00000111b

in una delle due istruzioni qui sotto riportate:

storex .def 084,**11000000b**,00000111b
storex .def 084,**00000011b**,00000111b

In questo caso il compilatore **dovrebbe** segnalare un **errore** perchè nel primo esempio abbiamo definito **leggibili** i bits **7-6** e nel secondo esempio abbiamo definito **leggibili** i bits **1-0**, mentre nel numero **32** dovrebbe risultare leggibile il solo bit **5**.

Invece il compilatore **non** segnalerà **nessun errore** in **cp a,storex** perchè, non riuscendo a controllare il contenuto nell'accumulatore "**a**", è sufficiente che un qualsiasi bits di **storex** riportato nel **R-mask** risulti **leggibile** per considerare l'istruzione valida.

3) Esempio

Se vogliamo rendere leggibili tutti i bits di **R-mask** anzichè scrivere **11111111b** potremo mettere due sole **virgole** come qui sotto riportato:

storex .def 084h, ,00001111b

oppure inserire **0ffh** tra le due virgole:

storex .def 084h,0ffh,00001111b

Se vogliamo rendere **leggibili** e **scrivibili** tutti i bits di una variabile **mask** dovremo semplicemente scrivere:

storex .def 084h

4) Esempio

```
storex  .def   084h,0ffh,00001111b
campo   .def   storex+1, 0ffh,00010000b
verfin  .def   campo+1,m
valfix  .set   storex+2
inizio  cp     a,storex
        cp     a,campo
        cp     a,verfin
        clr    campo
        ldi    campo,valfix
        ldi    verfin,valfix
```

La prima istruzione:

storex .def 084h,0ffh,00001111b

significa, associa la variabile **storex** alla cella di memoria **084h** di **Data Space** e poichè abbiamo reso leggibili tutti i bits di **R-mask** con **0ffh**, il com-

```
C:\ST6>ast6 -x -l -s C:\ST6\TEST.ASM
ST6 MACRO-ASSEMBLER version 4.00  - August 1992
Error C:\ST6\TEST.ASM 75 : (112) no write-access rights to operand 1
Error C:\ST6\TEST.ASM 78 : (113) no read-access rights to operand 2
Execution time: 1 second(s)
3 errors detected
No object created

C:\ST6>

Press a key to return..._
```

Fig.8 Nel 2° esempio solo le istruzioni 73-74-76-77 sono corrette, mentre le 75-78 risultando errate verrano subito segnalate dal compilatore.

pilatore non effettuerà nessun controllo di lettura, mentre effettuerà un controllo nella **W-mask** perchè abbiamo definito scrivibili i soli bits **0-1-2 3.**

La seconda istruzione:

campo .def storex+1, 0ffh,00010000b

associa la variabile **campo** alla cella di memoria **085h** (ormai dovreste essere esperti nel decodificare l'espressione "**storex + 1**") di **Data Space**.
Tutti gli **8** bits della variabile **R-mask** sono leggibili mentre nella **W-mask** è scrivibile solo il bit **4**.

```
;+------------------------------------------
;|      VARIABILI DEL PROGRAMMA
;+------------------------------------------
storex  .def    084h,0ffh,00001111b
campo   .def    storex+1,0ffh,00010000b
verfin  .def    campo+1,m
valfix  .set    storex+2
;+------------------------------------------
        .org    0800h           ;80

inizio  cp      a,storex        ;81
        cp      a,campo         ;82
        cp      a,verfin        ;83
        clr     campo           ;84
        ldi     campo,valfix    ;85
        ldi     verfin,valfix   ;86
```

Fig.9 In questa figura vi facciamo vedere come si presentano sul monitor le dieci righe delle istruzioni del 4° esempio. Anche in questo caso accanto ad ogni istruzione abbiamo riportato dopo il ; il numero di riga per ritrovarle più facilmente.

La terza istruzione:

verfin .def campo+1,m

associa la variabile **verfin** alla cella di memoria **086h** ("**campo + 1**") e omette sia **R-mask** che **W-mask**, ma inserisce un **marker** alla variabile **verfin**.
Per omettere **R-mask** e **W-mask** non è necessario scrivere come molti potrebbero supporre:

verfin .def campo+1,0ffh,0ffh,m

Per la quarta istruzione:

valfix .set storex+2

come già vi abbiamo spiegato nel capitolo riguardante le **Espressioni**, la direttiva **.set** associa una etichetta ad un valore e **non** ad un indirizzo di **Memoria Data Space** come avviene con **.def**.

Nel nostro esempio all'etichetta **valfix** viene associato il valore **086h** (**storex+2**).

Le tre successive istruzioni:

inizio cp a,storex
cp a,campo
cp a,verfin

confrontano il valore contenuto nell'accumulatore **a** con i valori contenuti rispettivamente in **storex**, **campo** e **verfin**.
In fase si compilazione non viene segnalato nessun **errore**, perchè tutte e tre le variabili sono state dichiarate leggibili in **R-mask**.

La successiva istruzione:

clr campo

che significa **azzera** il valore della variabile **campo**, viene normalmente compilata senza problemi e senza generare degli errori.

La penultima istruzione:

ldi campo,valfix

significa **carica** nella variabile **campo** il valore associato a **valfix** e **non**, come molti ritengono, il valore contenuto in **valfix.**
Poichè il compilatore controlla il valore da caricare nella variabile **campo**, in fase di compilazione segnalerà un **errore** perchè nel nostro esempio **valfix** vale **086h** (il suo numero binario è **10000110b**) e nella **W-mask** di **campo** abbiamo definito scrivibile il solo bit **4** e non i bits **1-2-7**.

L'ultima istruzione:

ldi verfin,valfix

significa **carica** nella variabile **verfin** il valore associato a **valfix** e viene regolarmente compilata senza segnalare **errori** perchè in **verfin** non è stata definita la **W-mask**, pertanto viene eseguita senza effettuare **alcun** controllo.

CONTINUA

Nella rivista precedente abbiamo preso in esame la **direttiva .W_ON**, in questo numero le direttive **.ASCII** - **.ASCIZ** - **.DEF**, mentre nelle riviste successive passeremo a considerare tutte le direttive mancanti.

Fig.1 Il programmatore per micro ST62/60-65 è racchiuso dentro un elegante mobile a consolle.

PROGRAMMATORE

Molti softwaristi dopo aver acquistato i programmatori **commerciali** per la nuova famiglia di micro **ST62/60 - ST62/65**, pagandoli più di **650.000 lire**, si sono accorti che oltre ad essere troppo complicati da utilizzare (qualcuno ha bruciato diversi micro), presentano il difetto di **non** funzionare su tutti i computer, tra i quali anche il **Pentium.**

Tutti i softwaristi, che hanno acquistato il nostro programmatore per micro **ST62/10-15-20-25**, sono rimasti a tal punto soddisfatti, da richiedercene uno identico per questa **nuova** famiglia, che sia in grado di funzionare su tutti i tipi di computer **IBM** e compatibili, compreso ovviamente il **Pentium**.

Prima di passare alla descrizione dello schema elettrico vogliamo svelare a coloro che usano qualsiasi tipo di programmatore, compresi i nostri, un piccolo segreto che, da quanto ci risulta, nessuno ha mai reso pubblico.

Ormai tutti sanno che i computer hanno due **porte parallele** denominate **LPT1 - LPT2**, ma nessuno si è mai preoccupato di precisare che il **programmatore** deve essere obbligatoriamente collegato sulla **porta LPT1**.

Quindi se il vostro computer ha la **stampante** collegata sulla **porta LPT1**, dovrete spostarla sulla **porta LPT2**.

Lasciando infatti la **stampante** sulla porta **LPT1** e collegando il **programmatore** sulla porta **LPT2**, potreste **non** riuscire a farlo funzionare.

A coloro che ci hanno chiesto se questa **nuova** famiglia di microprocessori sostituirà i precedenti **ST6**, assicuriamo che anche questi continueranno ad essere prodotti, ad **esclusione** del solo **ST62E10 cancellabile**. L'**ST62T10** tipo **OTP** rimane invece in commercio.

Ci è stato inoltre domandato che cosa ha in più questo nuova famiglia **ST62/60** e **ST62/65** rispetto alla precedente. Rispondiamo accennando velocemente alle novità di questi microprocessori:

– Un banco di memoria **RAM** di **128 K**, cioè il doppio dei precedenti **ST6**.

– Un supplementare banco di **128 bytes** di memoria **EEprom** (si pronuncia **E-quadroprom** e la sigla sta per **E**lectrically **E**rasable **P**rogrammable

Read **O**nly **M**emory). Questa memoria è **cancellabile** e **riscrivibile** elettricamente diverse migliaia di volte. Una volta scritti i dati nella **EEprom**, rimarranno **memorizzati** anche se toglieremo la tensione di alimentazione; ovviamente riappariranno quando il micro verrà nuovamente alimentato.
La **EEprom** permette di risolvere molti problemi. Ad esempio, noi stessi ci siamo serviti di questi **128 bytes** in più presenti nel micro **ST62/T65**, per tenere in **memoria** le **posizioni** dei satelliti TV nel kit **Box per posizionare le parabole TV** (kit **LX.1195**) apparso sulla rivista **N.177-178**.

– Una **Interfaccia Seriale SPI** (**S**ynchronous **P**eripheral **I**nterface) in grado di trasmettere e ricevere dei dati **seriali**.

– Un **Timer Autoreload** autoricaricabile che serve anche per gestire la funzione **PWM**.

– Un **Timer** identico ai precedenti micro **ST6**.

– Una funzione **PWM** (**P**ulse **W**idth **M**odulation), che ci permette di ottenere in uscita delle **onde quadre** con **duty-cycle variabile**, utilizzabili per ricavare delle tensioni variabili oppure delle forme d'onda sinusoidali o triangolari con l'impiego di pochi componenti esterni.

Nella **Tabella N.1** riportiamo le caratteristiche più interessanti di questa nuova famiglia.

Per **programmare** la nuova famiglia di microprocessori abbiamo progettato il **programmatore** siglato **LX.1325** (vedi fig.1), che risulta ben diverso dal precedente **LX.1170**.

Il **programmatore LX.1325** serve solo per i micro **ST62/60-65** e poiché il procedimento di program-

per MICRO ST62/60-65

Sulla rivista N.172/173 vi abbiamo presentato un programmatore per i micro della famiglia ST62T10-T15-T20-T25 e ST62E15-E20-E25. Poiché da tempo è uscita la nuova famiglia ST62T60-T65 e ST62E60-E65, in molti ci hanno richiesto un programmatore dalle prestazioni simili a quello già progettato, ma che programmi questi nuovi micro.

Tabella N.1

Micro tipo OTP (NON CANCELLABILI)

sigla micro	memoria program.	memoria RAM	memoria EEPROM	piedini zoccolo	numero Porte A	numero Porte B	numero Porte C
ST62T60	4 K	128 bytes	128 bytes	20	4	6	3
ST62T65	4 K	128 bytes	128 bytes	28	8	8	5

Micro tipo EPROM (CANCELLABILI con lampade ULTRAVIOLETTE)

sigla micro	memoria program.	memoria RAM	memoria EEPROM	piedini zoccolo	numero Porte A	numero Porte B	numero Porte C
ST62E60	4 K	128 bytes	128 bytes	20	4	6	3
ST62E65	4 K	128 bytes	128 bytes	28	8	8	5

In questa tabella sono riportate le caratteristiche più interessanti dei micro ST62.

mazione e il linguaggio **assembler** rimangono gli stessi che abbiamo iniziato a spiegare per gli **ST62T10** ecc., chi ha seguito i nostri precedenti articoli (abbiamo iniziato dalla rivista **N.172**) non incontrerà nessuna difficoltà ad usarlo.

Per questo motivo vi spiegheremo come dovete procedere per le sole funzioni **supplementari**, cioè **Seriale**, **PWM** ed **EEprom**. Se sarà il caso in un prossimo futuro vi prepareremo anche un valido **software simulatore**.

SCHEMA ELETTRICO

Per quel che concerne lo schema elettrico raffigurato in fig.2, vogliamo subito precisare che il **CONN.1**, visibile sul lato sinistro, è un connettore **maschio** a 25 poli, che deve essere collegato tramite un **cavo parallelo** alla porta **parallela LPT1** del vostro computer.
Se acquistate i cavi **paralleli** in un supermercato accertatevi che i piedini **1 - 2 - 3 - 4** ecc. del **maschio** risultino collegati sui piedini **1 - 2 - 3 - 4** ecc. del connettore **femmina**, perché è piuttosto facile trovare cavi paralleli con le connessioni invertite.

Come qualsiasi altro programmatore, anche il nostro **legge** i dati contenuti nel micro e logicamente li **scrive** al suo interno prelevandoli dal computer.
Per programmare i micro **ST62/60-65** è necessario un nuovo **Software** chiamato **ST626xPG** che è totalmente diverso da quello che fino ad oggi avete usato per i micro **ST62/10-15-20-25**.
Poiché non tutti riusciranno a procurarsi questo **nuovo** software, abbiamo ritenuto opportuno fornirvi assieme al kit anche il **dischetto software**.

Come potete notare dalla fig.2, in questo programmatore abbiamo inserito due **zoccoli textool**, uno da **28 piedini** ed uno da **20**, per il semplice motivo che i piedini di programmazione **Vcc - Vpp - OSC.IN. - RESET - GND - PB2 - PB3** non fanno sempre capo agli stessi piedini nei due zoccoli.

La tensione continua di circa **20 volt** che preleviamo dall'alimentatore siglato **LX.1170/B**, passando attraverso il diodo **DS1**, raggiunge l'integrato **IC2**, che provvede a stabilizzarla sul valore di **5 volt**.
Questa tensione alimenta l'integrato **IC1**, un **C/Mos** tipo **SN.74HC14** composto da **6 Inverter** a trigger di **Schmitt**.

La tensione di **20 volt** raggiunge anche il terminale **E**mettitore del transistor **TR1**, un **PNP** utilizzato come **interruttore elettronico** e come **circuito** di **protezione**. Grazie a questo transistor non danneggerete i **micro** se per **errore** li inserirete negli zoccoli **textool** in senso errato.

Fino a quando il computer non invia al programmatore il comando di scrittura o lettura, sul piedino **2** (**D0**) del **CONN.1** troviamo un **livello logico 1** che viene invertito da **IC1/A**. Poiché l'uscita di questo trigger è collegata al transistor **NPN** siglato **TR2**, sulla sua **B**ase viene applicato un **livello logico 0**.
Con questo livello logico il transistor **TR2** non conduce e quindi non riesce a polarizzare la **B**ase del transistor **PNP** siglato **TR1**. La tensione positiva dei **20 volt** applicata sul suo **E**mettitore non può dunque fuoriuscire dal suo **C**ollettore e, di conseguenza, non raggiunge i due integrati stabilizzatori **IC3** ed **IC4** che a loro volta non possono inviare al **micro**, inserito in uno dei due zoccoli **textool**, nessuna tensione di alimentazione.

Solo quando abilitiamo il computer a leggere o scrivere sul **micro**, sul piedino **2** (**D0**) del **CONN.1** troviamo un **livello logico 0**, che porta in conduzione il transistor **TR2**. Automaticamente questo provvede a polarizzare la **B**ase del transistor **TR1** e così la tensione positiva dei **20 volt** applicata sul suo **E**mettitore fuoriesce dal suo **C**ollettore e raggiunge i due integrati stabilizzatori **IC3** ed **IC4** che subito provvedono ad alimentare il **micro** inserito in uno dei due zoccoli **textool**.

L'integrato stabilizzatore **IC3** viene utilizzato per fornire una tensione di **5 volt stabilizzata** sul piedino **Vcc** e, tramite il diodo **schottky** siglato **DS5**, una tensione di poco inferiore sul piedino **Vpp**.

Prima di procedere alla programmazione del micro il computer **testa** tramite i comandi **Blanck Check** e **Read** se il micro non risulti già programmato oppure se non sia vergine o difettoso e, se tutto ciò non bastasse, verifica che tutti i piedini siano inseriti nello zoccolo ed anche che sia stato scelto il micro giusto per il programma che si vuole memorizzare.

Durante queste fasi di controllo l'inverter **IC1/C** provvede ad inviare sui piedini di **RESET** (piedino **22** per gli **ST62/65** e piedino **16** per gli **ST62/60**) una tensione di **5 volt**.

Sebbene l'integrato stabilizzatore **IC4** sia da **5 volt**, dà in uscita una tensione di **13,2 volt** perché sul suo piedino **M** è collegato un diodo zener da **8,2 volt** (vedi **DZ1**): infatti **8,2 + 5 = 13,2 volt**.

La tensione di **13,2 volt** circa raggiunge il solo piedino **Vpp** quando il transistor **PNP** siglato **TR3** vie-

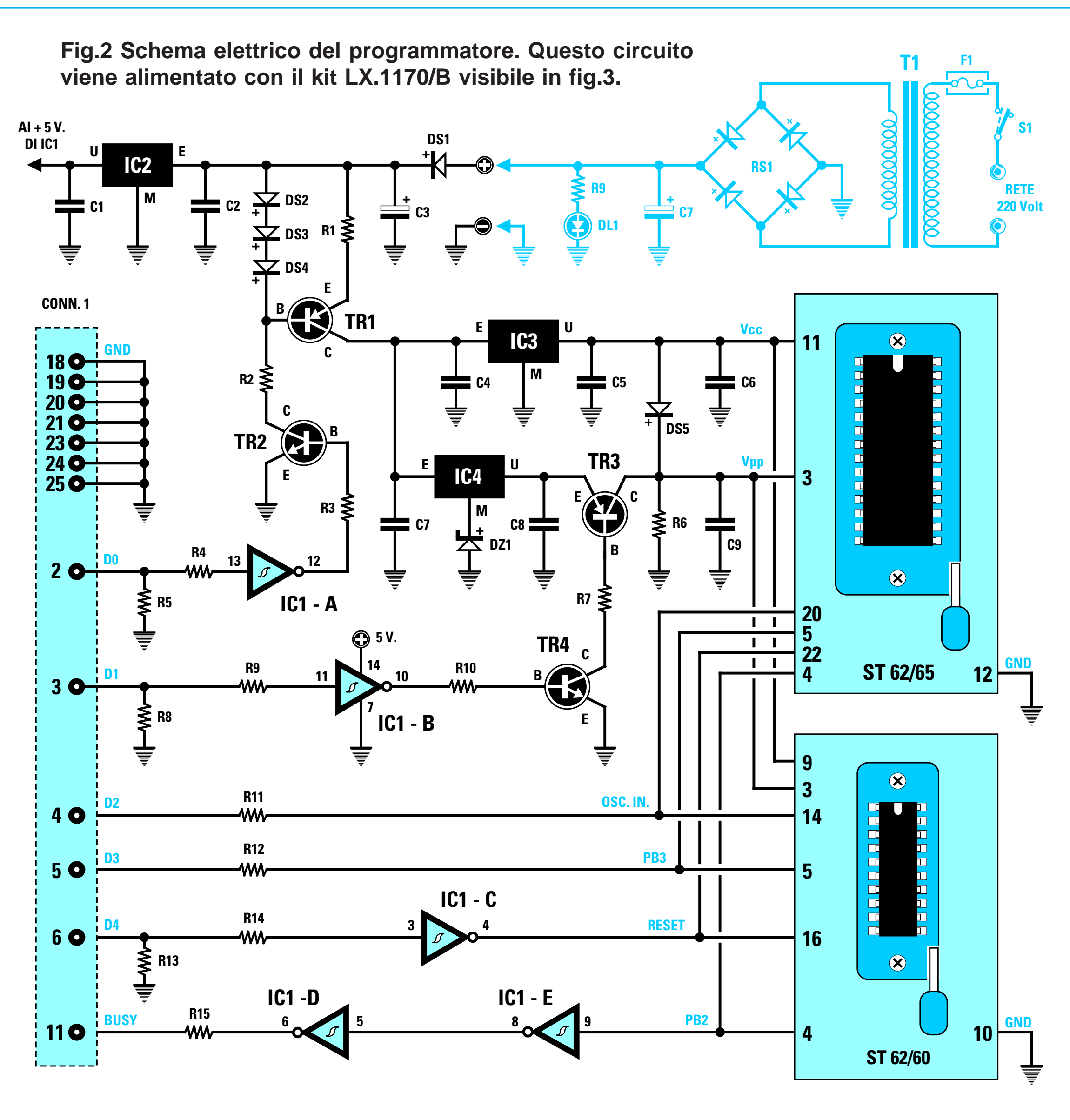
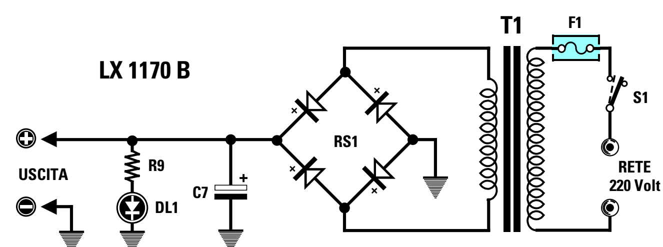


Fig.2 Schema elettrico del programmatore. Questo circuito viene alimentato con il kit LX.1170/B visibile in fig.3.

ELENCO COMPONENTI LX.1325

R1 = 4,7 ohm
R2 = 4.700 ohm
R3 = 10.000 ohm
R4 = 1.000 ohm
R5 = 10.000 ohm
R6 = 560 ohm
R7 = 4.700 ohm
R8 = 10.000 ohm
R9 = 1.000 ohm
R10 = 10.000 ohm
R11 = 1.000 ohm
R12 = 1.000 ohm
R13 = 10.000 ohm
R14 = 1.000 ohm
R15 = 10 ohm
C1 = 100.000 pF poliestere
C2 = 100.000 pF poliestere
C3 = 22 mF elettrolitico
C4 = 100.000 pF poliestere
C5 = 100.000 pF poliestere
C6 = 100.000 pF poliestere
C7 = 100.000 pF poliestere
C8 = 100.000 pF poliestere
C9 = 100.000 pF poliestere
DS1 = diodo tipo 1N.4007
DS2 = diodo tipo 1N.4150
DS3 = diodo tipo 1N.4150
DS4 = diodo tipo 1N.4150
DS5 = diodo schottky BAR.10
DZ1 = zener 8,2 volt 1/2 watt
TR1 = PNP tipo BD.140
TR2 = NPN tipo BC.547
TR3 = PNP tipo BC.328
TR4 = NPN tipo BC.547
IC1 = C/Mos tipo 74HC14
IC2 = uA.78L05
IC3 = uA.78L05
IC4 = uA.78L05
CONN.1 = connettore 25 poli



ELENCO COMPONENTI LX.1170/B

R9 = 1.500 ohm
C7 = 1.000 mF elettrolitico
DL1 = diodo led
RS1 = ponte raddriz. 100 V 1 A
F1 = fusibile 145 mA
T1 = trasform. 3 watt (TN00.01)
sec. 15 volt 0,2 amper
S1 = interruttore

Fig.3 Schema elettrico dello stadio di alimentazione pubblicato sulla rivista N.172/173.

ne posto in conduzione dal transistor **NPN** siglato **TR4**, pilotato dall'inverter **IC1/B** collegato sul piedino **3** (**D1**) del **CONN.1**.

Dopo aver **testato** il micro, se tutto risulta regolare inizia la fase di **programmazione** ed il programma da noi scritto ed assemblato viene trasferito dal computer verso il microprocessore inserito nello zoccolo **textool**.

Durante la fase di programmazione il computer invia sul piedino **3** (**D1**) del **CONN.1** un **livello logico 0** che, raggiungendo l'ingresso dell'inverter **IC1/B**, viene convertito in un **livello logico 1**.
Poiché l'uscita di **IC1/B** risulta collegata sulla **B**ase di **TR4**, questo transistor si porta in conduzione polarizzando la **B**ase del transistor **PNP** siglato **TR3**. In questo modo la tensione positiva di **13,2 volt** applicata sul suo **E**mettitore può fuoriuscire dal suo **C**ollettore raggiungendo il piedino **3** (**Vpp**) del micro.

La tensione di **13,2 volt** non può raggiungere il piedino **Vcc** per la presenza del diodo **DS5**, quindi su questo piedino ritroveremo sempre **5 volt** anche se sul piedino **Vpp** vi sono **13,2 volt**.

Il programma viene inviato dal computer verso il micro in forma **seriale** tramite il piedino **5** (**D3**) del **CONN.1**.

Tramite il piedino **4** (**D2**) del **CONN.1** il computer invia sul piedino **OSC.IN.** del micro un impulso di **clock** che, in fase di **programmazione**, sincronizza i dati inviati sul piedino **5** (**PB3**) del micro.

In fase di **lettura**, che serve per verificare se tutti i dati contenuti nel micro sono stati correttamente memorizzati, i dati vengono prelevati in forma **seriale** dal piedino **4** (**PB2**) ed inviati verso il computer tramite il piedino **11** (**BUSY**) del **CONN.1**.

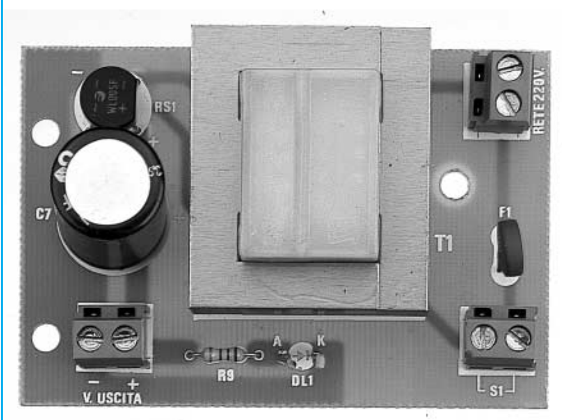


Fig.4 Foto del circuito di alimentazione.

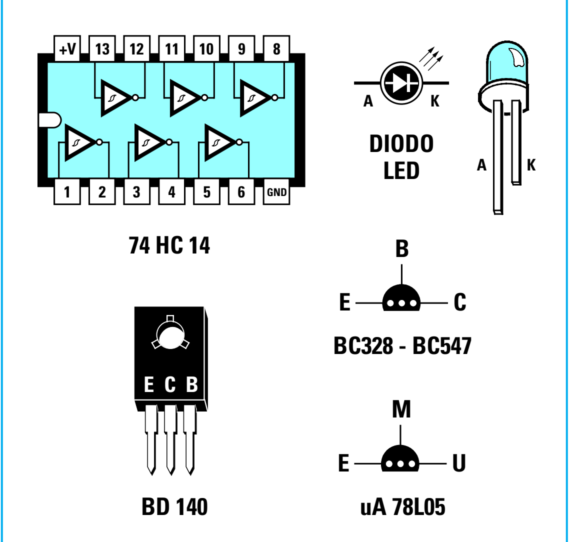


Fig.5 Connessioni dell'integrato 74HC14 viste da sopra e dei transistor BC.328, BC.547 ed integrato 78L05 viste da sotto.

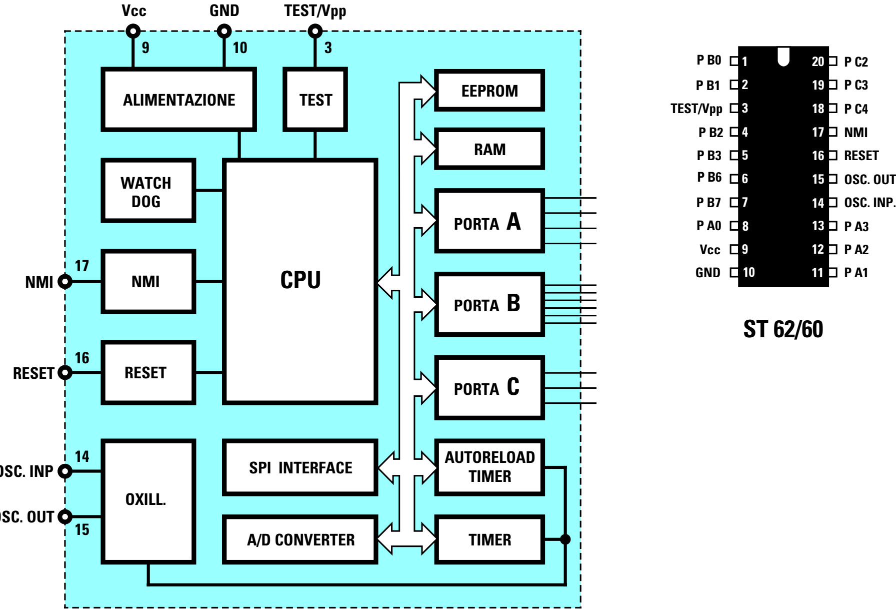


Fig.6 Entrambi i microprocessori tipo ST62T60 (non cancellabili) e gli ST62E60, che risultano cancellabili, hanno 20 piedini. Questi micro hanno 4K di memoria programmabile e 128 bytes di memoria EEprom, più tre porte indicate A-B-C. La porta A ha 4 entrate/uscite, la porta B ha 6 entrate/uscite e la porta C ha 3 entrate/uscite.

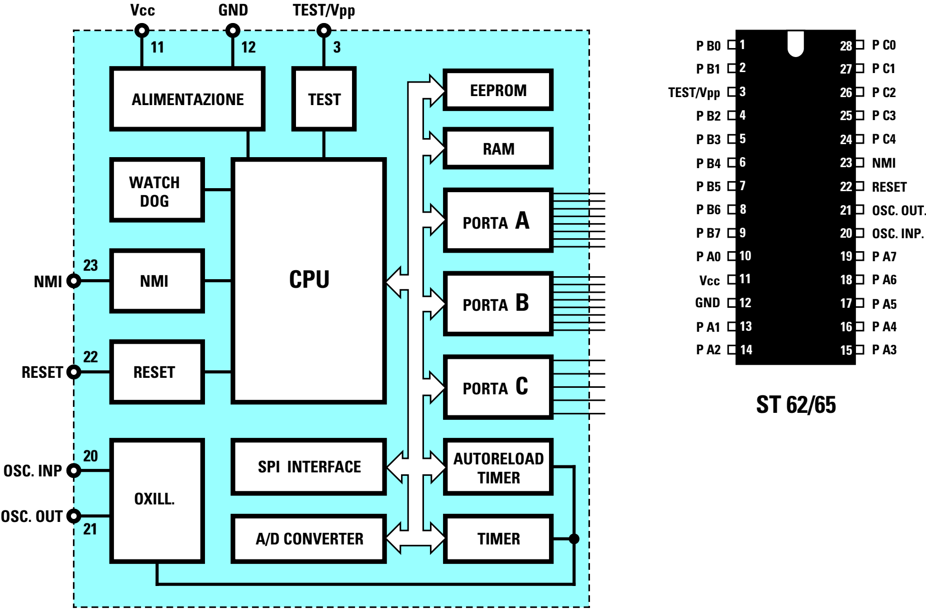


Fig.7 Entrambi i microprocessori tipo ST62T65 (non cancellabili) e gli ST62E65, che risultano cancellabili, hanno 28 piedini. Questi micro hanno 4K di memoria programmabile e 128 bytes di memoria EEprom, più tre porte indicate A-B-C. La porta A ha 8 entrate/uscite, la porta B ha 8 entrate/uscite e la porta C ha 5 entrate/uscite.

REALIZZAZIONE PRATICA

Per realizzare questo programmatore oltre al kit siglato **LX.1325** dovete procuravi anche il kit di alimentazione siglato **LX.1170/B**, che è lo stesso utilizzato per il precedente programmatore presentato sulla rivista N.172/173.

Sul circuito stampato **LX.1325** dovete montare tutti i componenti disponendoli come visibile in fig.10. Quando si effettuano questi montaggi si inizia normalmente dagli **zoccoli** e dai **connettori** perché la vista non è ancora affaticata e pertanto ci possiamo accorgere senza difficoltà se ci siamo dimenticati una stagnatura o se una grossa goccia di stagno ha cortocircuitato assieme due piedini.

Proseguendo nel montaggio inserite tutte le **resistenze** e, dopo queste, tutti i **diodi** rispettando la loro polarità.
La fascia **bianca** del diodo **DS1** con corpo plastico deve essere rivolta verso **sinistra**.
Il diodo **zener DZ1**, riconoscibile perché sul suo corpo è stampigliato il numero **8,2**, deve avere la **fascia bianca** rivolta verso l'**alto**.
La fascia **nera** dei diodi al **silicio** siglati **DS2 - DS3 - DS4** deve essere rivolta come risulta bel visibile nello schema pratico di fig.10.
La fascia **nera** del diodo **schottky DS5**, che si riconosce dagli altri perché il suo corpo è di colore **blu**, va rivolta verso **sinistra**.
Completato il montaggio di questi componenti stagnate tutti i condensatori **poliesteri**, ricordandovi

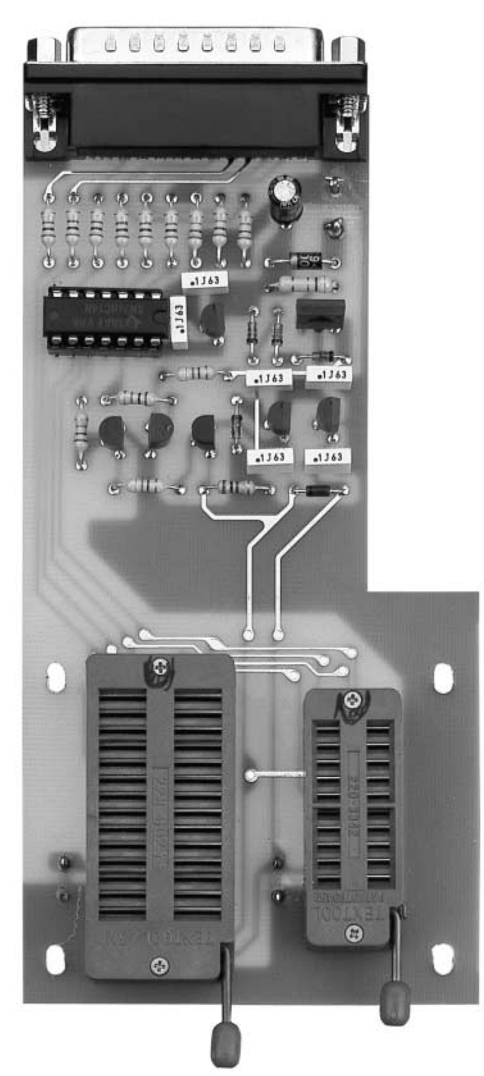


Fig.8 In questa foto è visibile lo stampato del programmatore LX.1325 visto dal lato sul quale sono montati tutti i componenti.

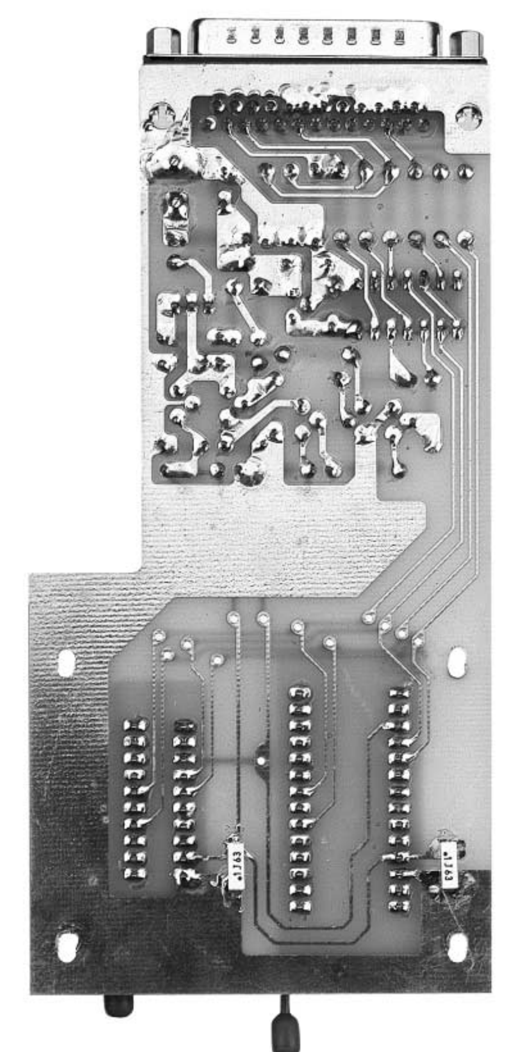


Fig.9 Lo stesso circuito visto dal lato opposto. Notate i due condensatori C6-C9 fissati vicino ai due zoccoli textool.

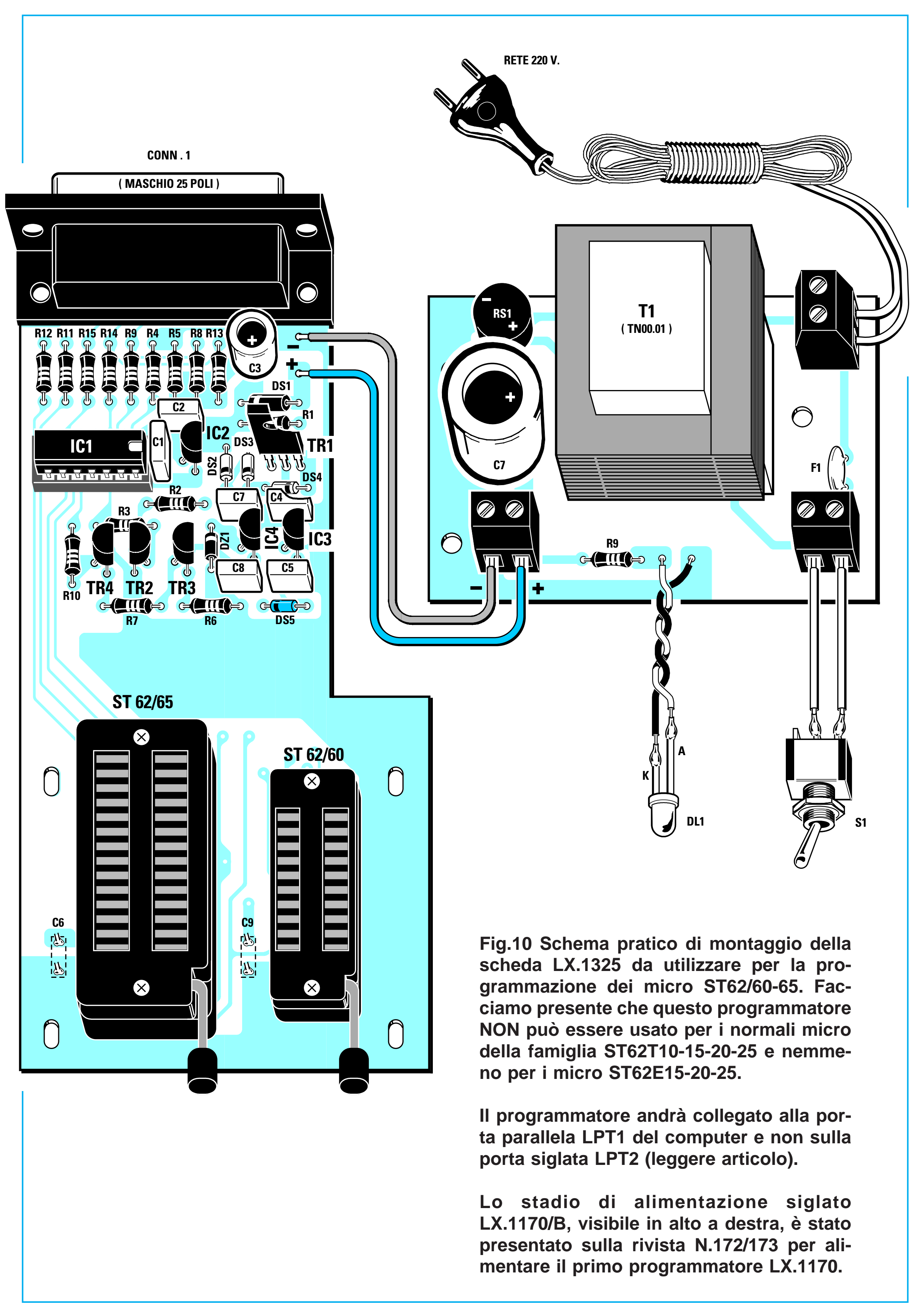


Fig.10 Schema pratico di montaggio della scheda LX.1325 da utilizzare per la programmazione dei micro ST62/60-65. Facciamo presente che questo programmatore NON può essere usato per i normali micro della famiglia ST62T10-15-20-25 e nemmeno per i micro ST62E15-20-25.

Il programmatore andrà collegato alla porta parallela LPT1 del computer e non sulla porta siglata LPT2 (leggere articolo).

Lo stadio di alimentazione siglato LX.1170/B, visibile in alto a destra, è stato presentato sulla rivista N.172/173 per alimentare il primo programmatore LX.1170.

che i due soli condensatori **C6 - C9** devono essere inseriti sul lato opposto del circuito stampato, come risulta ben evidenziato anche in fig.9.
Quando inserite il condensatore **elettrolitico C3** rivolgete il suo terminale **positivo** verso il basso.

A questo punto potete inserire il transistor **TR1**, siglato **BD.140**, rivolgendo il lato del corpo con il **metallo** verso la resistenza **R1**.
Ora prendete i minuscoli integrati stabilizzatori **IC2 - IC3 - IC4**, sul loro corpo c'è la sigla **78L05**, ed inseriteli nelle posizioni visibili nello schema pratico di fig.10, rivolgendo la parte **piatta** del loro corpo verso **destra**.
I transistor **TR2 - TR4**, siglati **BC.547**, vanno stagnati sotto la resistenza **R3**, rivolgendo la parte **piatta** dell'uno verso la parte piatta dell'altro.
Il transistor **TR3**, siglato **BC.328**, va inserito accanto al diodo zener **DZ1** rivolgendo la parte **piatta** del suo corpo verso **destra**.
Fate attenzione a non confondere i transistor **TR2 - TR4** che sono degli **NPN** con il transistor **TR3** che è invece un **PNP**.
Completato il montaggio, potete inserire nel suo zoccolo l'integrato **IC1** rivolgendo la sua tacca di riferimento a forma di **U** verso destra.

Ora potete montare il kit **LX.1170/B** stagnando sul suo circuito stampato tutti i suoi componenti.

Guardando l'eloquente schema pratico di fig.10 riteniamo che nessuno incontrerà difficoltà ad eseguire questo semplice montaggio.

MONTAGGIO nel MOBILE

Il mobile di questo programmatore, di tipo a **consolle** e perfettamente identico a quello del precedente programmatore per **ST6** (vedi fig.14), conferisce al progetto un aspetto decisamente professionale.
Come potete vedere in fig.11, lo stadio di alimentazione **LX.1170/B** va collocato sulla base del semicoperchio con tre distanziatori con base **autoadesiva**, mentre lo stampato **LX.1325** va fissato sul pannello frontale con quattro viti in ferro.
Prima di avvitare questo stampato vi consigliamo di inserire sul piccolo pannello **inclinato** la gemma **cromata** per il **diodo led** e l'interruttore a levetta **S1** dello stadio di alimentazione.
Effettuati i pochi collegamenti richiesti, il programmatore è già pronto per esplicare la sua funzione.

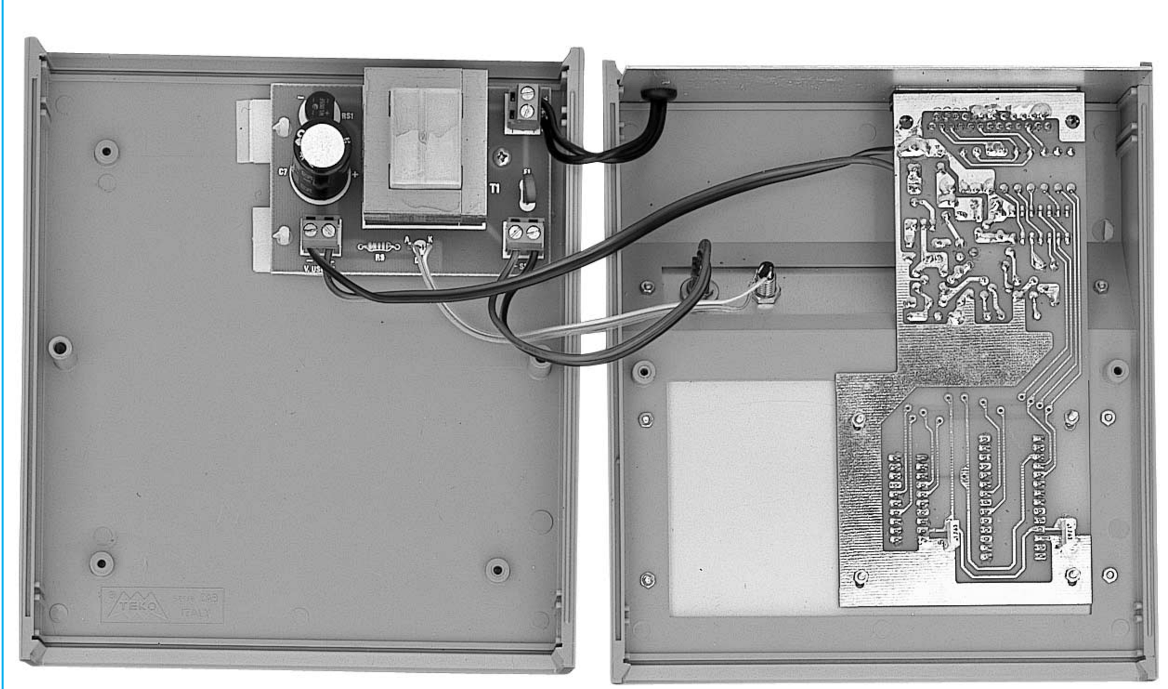


Fig.11 Aperto il mobile plastico a consolle visibile in fig.1, fissate sulla base del semicoperchio lo stadio di alimentazione LX.1170/B con quattro distanziatori autoadesivi, e sull'altro semicoperchio lo stampato LX.1325 avvitandolo con quattro viti.
NOTA = sul pannello frontale fissate le quattro viti con i loro dadi, in modo da creare un piccolo spessore che terrà distanziato lo stampato dal pannello frontale.

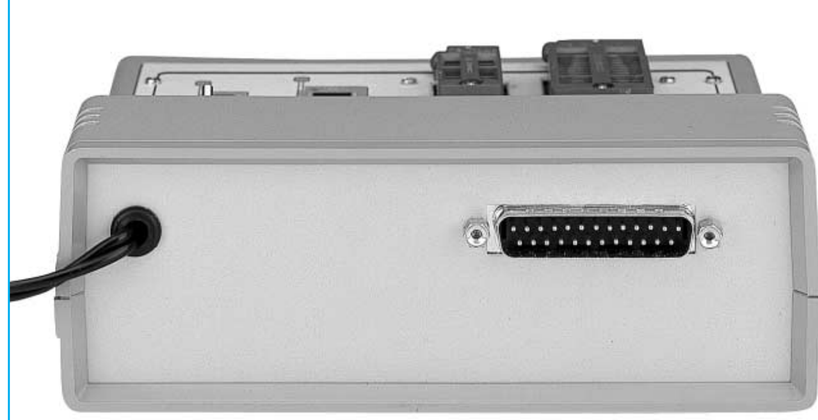


Fig.12 Sul pannello di alluminio posto sul retro del mobile verrà fissato il connettore di uscita per il computer.

COME caricare il PROGRAMMA

Assieme al kit riceverete un dischetto floppy contenente il programma per programmare tutti i microprocessori della famiglia **ST62/60-65**, più sei supplementari programmi che vi aiuteranno a capire come usare le funzioni **PWM** ed **EEprom.**
Per caricare il dischetto nell'**Hard-Disk** seguite le nostre istruzioni.

Inserite il dischetto nell'unità floppy poi digitate:

C:\>A: premete Enter
A:\>installa premete Enter

Il programma una volta caricato occupa circa **1 Megabyte** di memoria.

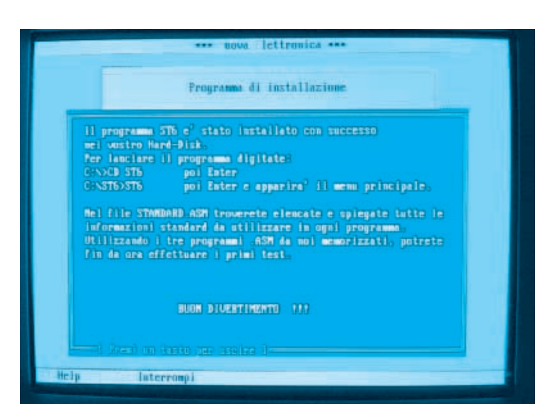


Fig.13 Una volta scompattati tutti i programmi inseriti nel disco floppy, apparirà sul monitor questa scritta. Per proseguire potete pigiare un tasto qualsiasi.

Nota importante: per caricare il programma **usate** le **due** sole istruzioni sopra riportate, perché solo così il programma verrà **scompattato** ed automaticamente verrà creata una **directory** chiamata **ST626** per poterla distinguere da quella del precedente programma che avevamo chiamata **ST6.**
Per copiare il contenuto del dischetto non usate né il **Copy** del **Dos** o altri programmi come il **PCshell - PCtools - Norton** ecc., perché **non** riuscireste a memorizzarlo nel vostro hard-disk.

Durante la scompattazione del programma apparirà sul monitor l'elenco di tutti i files e ad operazione conclusa leggerete la scritta visibile in fig.13.

Completata l'operazione d'installazione potrete togliere il dischetto dal drive e porlo in un cassetto.

PER RICHIAMARE il PROGRAMMA

Per richiamare questo programma dovete semplicemente digitare queste due sole istruzioni:

C:\>CD ST626 premete Enter
C:\ST626>ST6 premete Enter

I 6 PROGRAMMI di TEST

Prima di trasferire all'interno della **memoria vergine** di un micro **ST62/60** o **ST62/65** uno dei sei programmi di **test** che abbiamo inserito nel dischetto, dovete collocare il micro nel suo **zoccolo textool** poi bloccarlo con la sua levetta.
Dopo aver richiamato il programma digitando:

C:\>CD ST626 premete Enter
C:\ST626>ST6 premete Enter

Fig.14 Se avete già costruito il precedente programmatore per gli ST6, siglato LX.1170 (vedi rivista N.172/173), mettetelo a confronto con il modello LX.1325 da utilizzare solo per gli ST62/60-65 e noterete che hanno la stessa estetica. Questi mobili non sfigureranno anche se messi vicino ai più moderni ed eleganti computer.

selezionate quale dei sei programmi-test volete utilizzare pigiando il tasto funzione **F3**.

Sul monitor appariranno i nomi dei files di **test**:

PWM60.ASM
PWM65.ASM
EEPROM60.ASM
EEPROM65.ASM
EEPR60T.ASM
EEPR65T.ASM

NOTE IMPORTANTI

Selezionate il file **PWM60** o **EEPROM60** o **EEPR60T** solo se avete inserito nello zoccolo **textool** il micro **ST62E60** o **ST62T60**.
Selezionate il file **PWM65** o **EEPROM65** o **EEPR60T** solo se avete inserito nello zoccolo **textool** il micro **ST62E65** o **ST62T65**.

Potete memorizzare nel micro **uno solo** di questi sei programmi, perciò se volete testare più programmi dovrete utilizzare un secondo micro oppure **cancellare** con una **lampada ultravioletta** ciò che già avete memorizzato.
Il kit di questa lampada è riportato sulla rivista **N.174** e la sua sigla è **LX.1183**.

Per vedere come funzionano i programmi di test dovete necessariamente realizzare il nuovo Bus **LX.1329**, che può ricevere le schede già usate per il precedente programmatore per **ST6**, cioè:

– scheda **LX.1204**, pubblicata sulla rivista **N.179**, provvista di quattro **display**.

– scheda **LX.1206**, pubblicata sulla rivista **N.180**, provvista di quattro **Triac**.

Il Bus deve essere alimentato con l'alimentatore siglato **LX.1203** pubblicato sulla rivista **N.179**, lo stesso che usate per alimentare il Bus del precedente programmatore **LX.1170**.
Su questa stessa rivista trovate la spiegazione dello schema elettrico e dello schema pratico del Bus **LX.1329** e tutte le istruzioni per eseguire i nostri test.

COSTO di REALIZZAZIONE

Tutti i componenti necessari alla realizzazione del kit **LX.1325** (vedi fig.10) completo di circuito stampato, zoccoli **textool**, transistor, integrati, più il **dischetto** floppy **DF.1325** contenente i programmi per la programmazione degli ST62/60-65 e quelli di test per **EEPROM** e **PWM**, ma **Esclusi** il mobile e l'alimentatore **LX.1170** .... € 51,65

Il solo mobile **MO.1325** completo di due mascherine forate a serigrafate .... € 16,01

Costo dello stadio di alimentazione **LX.1170/B** pubblicato sulla rivista N.172/173 .... € 11,60

Costo del solo stampato **LX.1325** .... € 7,49

Costo di un cavo **parallelo** tipo **CA.05** completo di connettori maschio e femmina .... € 4,13

I prezzi riportati sono compresi di **IVA**, ma non delle spese **postali** che verranno addebitate solo a chi richiederà il materiale in contrassegno.

BUS per TESTARE le

Per impratichirvi con la funzione PWM e la memoria EEPROM presenti nei micro della famiglia ST62/60-65 realizzate il Bus che vi presentiamo ed utilizzatelo insieme alla scheda Display LX.1204 oppure alla scheda Triac LX.1206, che vi sono servite per il precedente programmatore.

Per poter usare al meglio una nuova famiglia di micro è indispensabile fare un po' di pratica, e poiché sappiamo che non troverete in nessun manuale un valido aiuto (quelli da noi personalmente visionati sono pieni di errori o informazioni inesatte) abbiamo ritenuto opportuno aggiungere ai files per la programmazione, che trovate nel dischetto **DF.1325**, i programmi di **test** per provare la memoria **EEprom** e la funzione **PWM**.
Poiché in commercio non è facile trovare Bus per i micro **ST62/60-65** a prezzi contenuti, noi abbiamo risolto questo problema con il kit **LX.1329**, che potete alimentare con il kit siglato **LX.1203**, pubblicato sulla rivista **N.179.**

SCHEMA ELETTRICO del BUS

Lo stadio oscillatore ottenuto con il Nand siglato **IC1/B** ci permette di ottenere la frequenza di **clock** di **8 MHz** che viene trasferita con il Nand **IC1/C** sul piedino **14** del micro **ST62/60** e con il Nand **IC1/D** sul piedino **20** del micro **ST62/65**.
Questi due micro sono quelli che, una volta **programmati**, dovranno essere inseriti nei due zoccoli presenti sullo stampato del **Bus** (vedi fig.1).
Le due tensioni dei **12,6 volt** e dei **5,6 volt** vengono prelevate dall'alimentatore **LX.1203**, pubblicato sulla rivista **N.179**, di cui riportiamo nuovamente in fig.29 l'elettrico nel caso in cui non aveste questo numero della rivista.

REALIZZAZIONE PRATICA

In possesso dello stampato **LX.1329** potete iniziare il montaggio inserendo i due zoccoli da **28** e **20 piedini** per i micro **ST62/65** ed **ST62/60**, poi quello da **14 piedini** per l'integrato **IC1** (vedi fig.4).
Dopo gli zoccoli fissate il connettore a **24 pin** siglato **CONN.1** ed i due connettori a **4 pin** che serviranno per innestare le tre schede sperimentali **LX.1204 - LX.1206 - LX.1329/B**.

Vicino all'integrato **IC1** inserite il quarzo da **8 MHz** fissando il suo corpo in posizione orizzontale.
Sul lato destro dello stampato inserite il pulsante **P1**, poi la **morsettiera** a 3 poli e vicino a questa i due diodi al silicio **DS1 - DS2** rivolgendo la loro fascia di riferimento di colore **bianco** verso **sinistra**.
Completato il montaggio inserite nel suo zoccolo l'integrato **IC1** rivolgendo la tacca di riferimento a forma di **U** verso l'alto.
Se in sostituzione dei **normali** zoccoli per i due micro **ST62/60** e **ST62/65** utilizzate due zoccoli **textool**, risulterà più facile inserirli ed estrarli, ma come saprete, questi zoccoli sono molto costosi.

PER RICHIAMARE il PROGRAMMA

Dopo aver memorizzato nell'hard-disk il programma, per richiamarlo dovete semplicemente digitare due sole istruzioni:

C:\>CD ST626 premete Enter
C:\ST626>ST6 premete Enter

Prima di trasferire all'interno della **memoria vergine** di un micro **ST62/60** o **ST62/65** uno dei **6** programmi di **test**, inserite il micro nel suo **zoccolo textool** poi bloccatelo con la sua levetta.

Quando sul monitor appare la finestra di fig.5 dovete premere il **tasto** funzione **F3**.

Sullo schermo vengono visualizzati i nomi dei nostri programmi di **test** (vedi fig.6).

EEPR60T.ASM	per i micro **ST62/60**
EEPR65T.ASM	per i micro **ST62/65**
EEPROM60.ASM	per i micro **ST62/60**
EEPROM65.ASM	per i micro **ST62/65**
PWM60.ASM	per i micro **ST62/60**
PWM65.ASM	per i micro **ST62/65**

In ogni micro potete inserire **uno solo** programma, quindi per provare un secondo programma dovete **cancellare** la sua memoria con una **lampada ultravioletta** oppure utilizzare un altro micro.

funzioni PWM e EEPROM

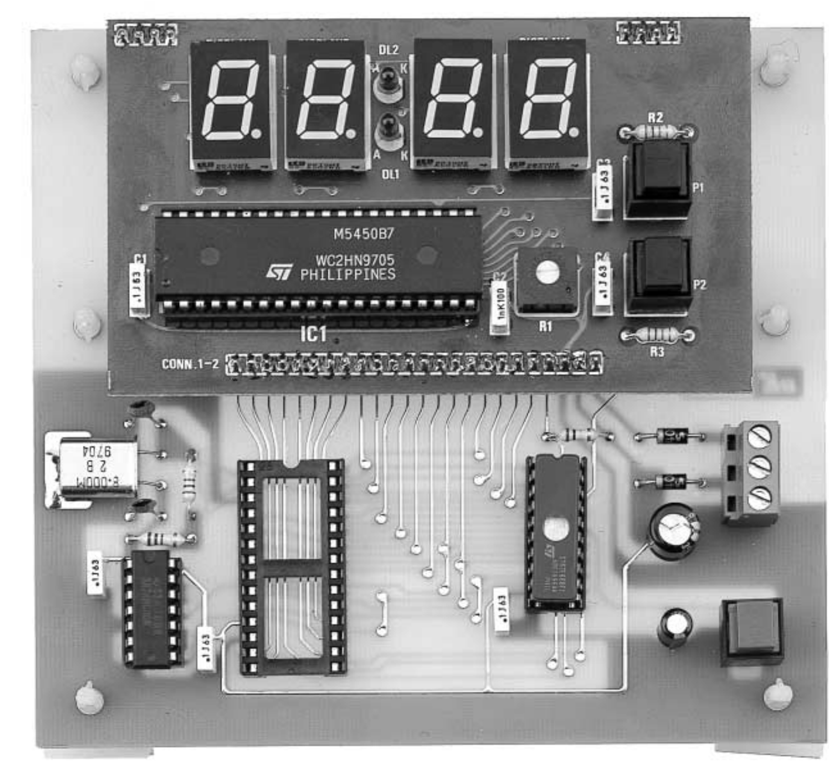
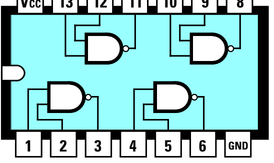


Fig.1 Applicando in questo nuovo Bus siglato LX.1329 le schede Display e Triac già utilizzate per il precedente Bus potrete vedere come funzionano la EEPROM ed il PWM. Negli zoccoli andrà inserito un micro programmato con i nostri programmi di test.

Fig.2 Schema elettrico del Bus LX.1329 da usare per i soli micro della serie ST62/60 e ST62/65. Questo Bus deve essere alimentato con il circuito LX.1203, che abbiamo già presentato sulla rivista N.179, riportato nelle figg.29-31. Se possedete già questo alimentatore non sarà necessario montarne un secondo. I due rettangoli NERI con sopra stampigliato ST62/60-ST62/65 sono i due zoccoli (vedi fig.4) nei quali dovrete inserire i due micro che avrete programmato con i programmi test per EEPROM e PWM.

ELENCO COMPONENTI LX.1329

R1 = 2,2 megaohm
R2 = 1.000 ohm
R3 = 100.000 ohm
C1 = 100.000 pF poliestere
C2 = 100.000 pF poliestere
C3 = 10 pF ceramico
C4 = 10 pF ceramico
C5 = 1 mF elettrolitico
C6 = 100.000 pF poliestere
C7 = 100 mF elettrolitico
XTAL = quarzo 8 MHz
DS1 = diodo tipo 1N.4007
DS2 = diodo tipo 1N.4007
IC1 = C/Mos tipo 74HC00
CONN.1 = connettore 24 poli
P1 = pulsante



74 HC 00

Fig.3 Foto del Bus che ci servirà per ricevere le schede LX.1204 - LX.1206 - LX.1329/B per i test.

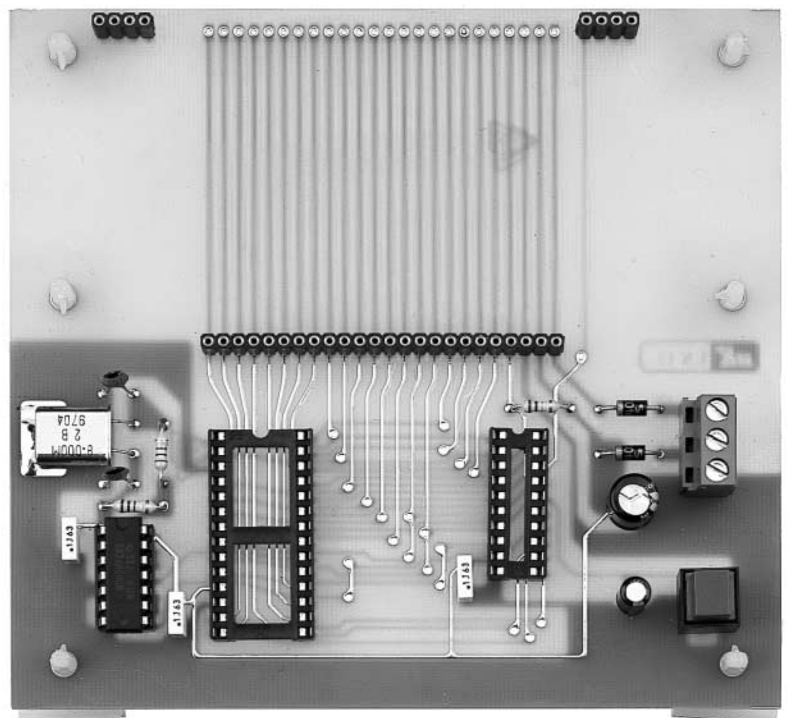


Fig.4 In basso lo schema pratico di montaggio del Bus siglato LX.1329.

CONN. 1
C4
XTAL
R2
C3
R1
C1
IC1
C6
ST 62/65
C2
R3
ST 62/60
DS1
DS2
C7
C5
P1
LX 1329
12,6 V.
5,6 V.
Massa
VERSO LX 1203

TEST con EEPROM tipo ST62E60

Inserite un micro **ST62E60**, cioè del tipo **cancellabile**, nello zoccolo del programmatore **LX.1325**.

Dopo aver richiamato il programma, premete il tasto **F3** in modo da far apparire i nomi dei files e portate il cursore sulla riga:

EEPROM60.ASM premete Enter

Appariranno così le istruzioni del programma con i relativi commenti (vedi fig.7).
A questo punto tenendo pigiato il tasto **ALT** pigiate il tasto **T** per visualizzare la finestra di fig.8.
Premete il tasto **R** e subito apparirà la finestra di programmazione con una infinità di sigle di microprocessori (vedi fig.9).

ST62E60
ST62E60B
ST62T60
ST62T60B
ST62E65
ST62E65B
ST62T65
ST65T65B

Importante: nessuno si è mai preso la briga di precisare se conviene scegliere il micro **senza** la **B** finale oppure quello con la **B** finale, per cui ora vi spiegheremo come sceglierli.

Se sul corpo del vostro micro è stampigliata la sigla **ST62E60/B** dovete selezionare **ST62E60**.

Se sul corpo del vostro micro è stampigliata la sigla **ST62E60/BB** dovete selezionare **ST62E60B**.

Se sul corpo del vostro micro è stampigliato la sigla **ST62T60/B** dovete selezionare **ST62T60**.

Se sul corpo del vostro micro è stampigliata la sigla **ST62T60/BB** dovete selezionare **ST62T65B**.

Quanto detto sopra vale anche per gli **ST62/65**.

Poiché il primo **programma-test** funziona su un micro cancellabile **ST62E60**, prima di scegliere la riga di programmazione controllate attentamente quale sigla è presente sul vostro micro.

Se trovate **ST62E60/B** selezionate **ST62E60**.
Se trovate **ST62E60/BB** selezionate **ST62E60B**.

A questo punto pigiate **L = Load** e nella finestra che appare (vedi fig.10) scrivete il nome del pro-

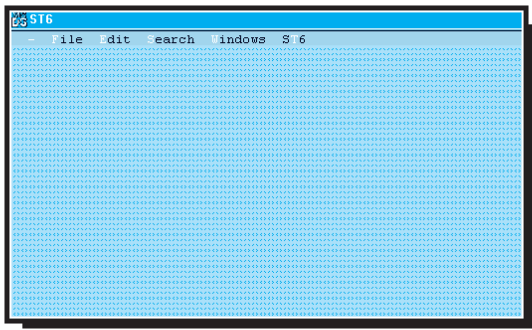


Fig.5 Per trasferire un programma dal computer ad un micro dovete usare il programmatore LX.1325. Dopo aver richiamato il programma, quando appare questa finestra premete il tasto F3.

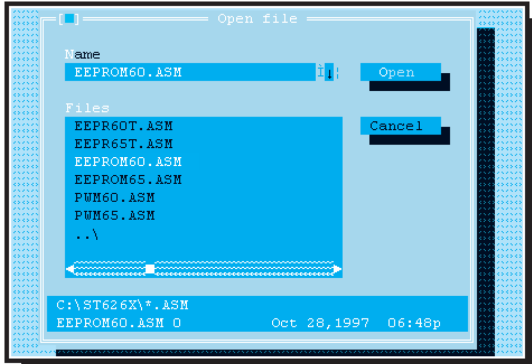


Fig.6 Pigiando F3 apparirà la finestra con i nomi dei 6 files dei programmi di test. In un micro potete inserire 1 solo programma, quindi per provarne un secondo dovrete prima cancellare la sua memoria.

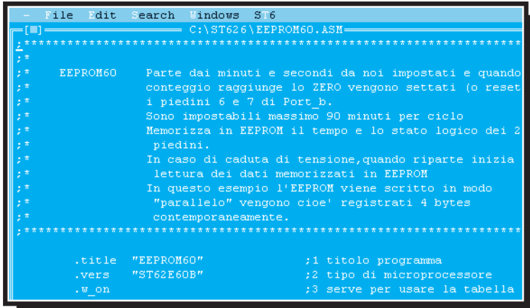


**Fig.7 Scelto il programma che volete memorizzare, sul video appariranno tutte le istruzioni con il relativo commento.
Leggendo queste istruzioni potete imparare come va impostato un programma.**

gramma, cioè **EEPROM60** tralasciando **.ASM**, quindi premete Enter e quando appare la finestra con la scritta **File checksum** premete Enter.

Apparirà così una finestra bianca. Se ora premete il tasto **P = Prg** e di seguito il tasto **N**, dopo pochi secondi apparirà la scritta:

Verifying the target chip ... Please Wait
Verifica chip da programmare ... attendi

Se tutto risulta regolare apparirà la scritta:

Programming the targhet chip ... Please Wait
Programmazione in corso ... attendi

L'operazione di scrittura dal computer verso il micro richiede circa **10 - 14 secondi**.

A programmazione completata sul monitor appare questa scritta:

The device is succesfully programmed
Il micro è stato correttamente programmato

Ora potete estrarre dal **programmatore** il micro già **programmato** per posizionarlo nello zoccolo della scheda Bus **LX.1329**, a cui avrete collegato la scheda con i quattro display siglata **LX.1204**, pubblicata sulla rivista **N.179**.

Sulla morsettiera del Bus dovete applicare due tensioni di **12,6** e **5,6 volt** più il filo di **massa**, che prelevate dal kit **LX.1203**.

Il programma **EEprom** che avete memorizzato nel micro è un **timer** con **4** diversi **cicli** che contano all'indietro.

Se sul Bus risultasse collegata in parallelo alla scheda **LX.1204** anche la scheda del **relè** siglata **LX.1205**, pure questa pubblicata sulla rivista **N.179**, con il primo ciclo vedreste apparire **00:20** e contemporaneamente eccitarsi il relè **RL1**, che si disecciterà quando il conteggio arriverà a **00:00**.

Con il secondo ciclo apparirà sui display **01:30**, e se fosse presente la scheda **LX.1205** vedremmo eccitarsi il **RL2**, che si disecciterà quando il conteggio arriverà a **00:00**.

Il terzo ciclo partirà dal numero **00:47** ed il quarto ciclo dal numero **03:00**.

Quello che abbiamo voluto mettere in evidenza con questo programma è la funzione della **memoria EEprom**.

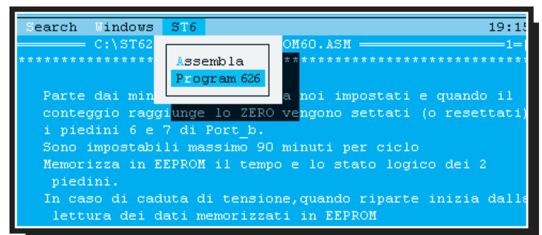


Fig.8 Pigiando ALT e T vedrete apparire sullo schermo questa piccola finestra.

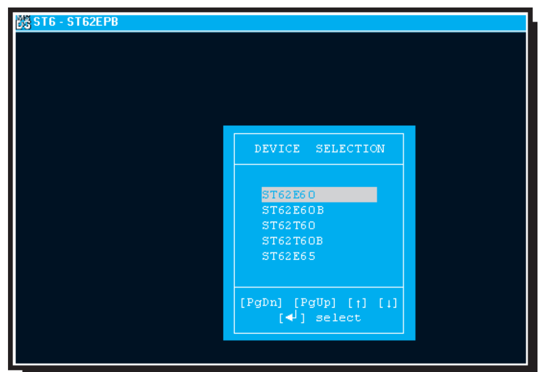


Fig.9 Premendo R vedrete apparire tutte le sigle dei micro che potete programmare.

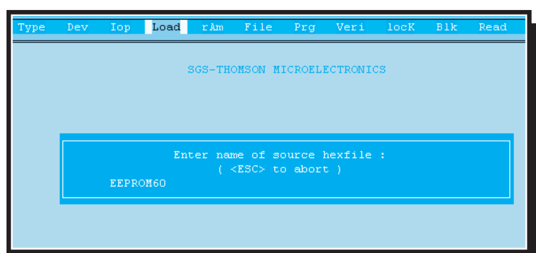


Fig.10 Scelto il micro pigiate L e digitate il nome del programma da trasferire.

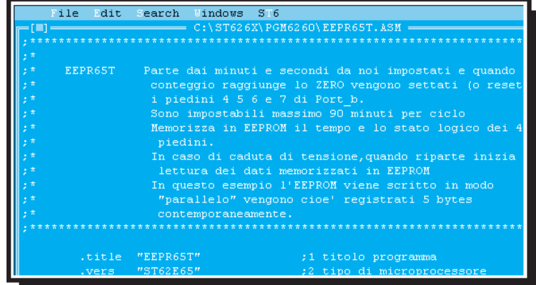


Fig.11 Il programma EEPR65T.ASM deve essere memorizzato nei micro ST62/65.

Se togliamo di proposito la tensione di alimentazione al Bus quando sui display appare il numero **01:34** o qualsiasi altro numero, il numero rimarrà comunque **memorizzato** nella **EEprom**.
Infatti alimentandolo nuovamente dopo **10 minuti**, oppure dopo **3 ore** o anche dopo **1 mese**, vedrete riapparire sui display lo stesso numero che risultava presente al momento dello spegnimento e, da questo numero, ripartirà il conteggio all'indietro.

Per variare i tempi da noi prefissati, nel programma dovrete modificare queste righe:

1° ciclo = righe **626-627**
2° ciclo = righe **635-636**
3° ciclo = righe **643-644**
4° ciclo = righe **651-652**

Portiamo qualche esempio.
Se volete che il **1° ciclo** abbia una durata di **2 minuti** e **30 secondi** dovete inserire nel riga 626 i **secondi** e nella 627 i **minuti**:

```
ldi   stsex,30    ; 626 tempo in secondi
ldi   stmix,2     ; 627 tempo in minuti
set   6,port_b    ; setta l'uscita 6 di port B a 1
```

La riga 628 serve per portare a **livello logico 1** il piedino **6** della **porta B**, così da poterlo utilizzare per eccitare un relè oppure dei Triac tramite un circuito pilota.

Dopo **2 minuti** e **30 secondi**, il nuovo tempo da voi impostato, il programma passa al **2° ciclo** che porta a **livello logico 0** il piedino **6** (riga 637) e a **livello logico 1** il piedino **7** (riga 638).

Se volete modificare i tempi del **2° ciclo** per portarlo ad esempio a **45 secondi**, dovete modificare la riga 635 inserendo questo numero e scrivere nella riga 636 **0 minuti**:

```
ldi   stsex,45    ; 635 tempo in secondi
ldi   stmix,0     ; 636 tempo in minuti
res   6,port_b    ; resetta l'uscita 6 di port B a 0
set   7,port_b    ; setta l'uscita 7 di port B a 1
```

Se nel **1° ciclo** voleste portare il piedino **6** a **livello logico 0** per la durata di **2 minuti** e **30 secondi** modificate il programma come qui sotto riportato:

```
ldi   stsex,30    ; 290 tempo in secondi
ldi   stmix,2     ; 291 tempo in minuti
res   6,port_b    ; resetta l'uscita 6 di port B a 0
```

Per portare a **livello logico 1** il piedino **6** della por-

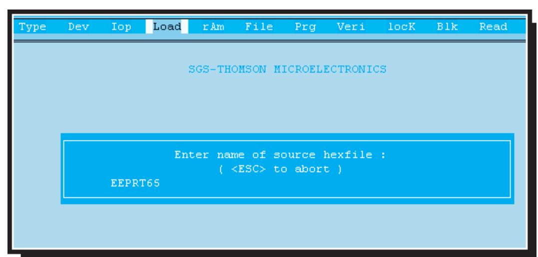


Fig.12 Per vedere il contenuto di una EEPROM pigiate sulla casella LOAD.

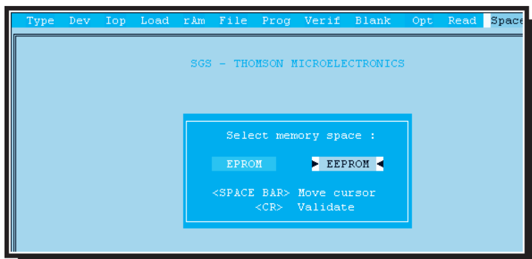


Fig.13 Quando appare questa finestra pigiate Barra e Enter per andare su EEprom.

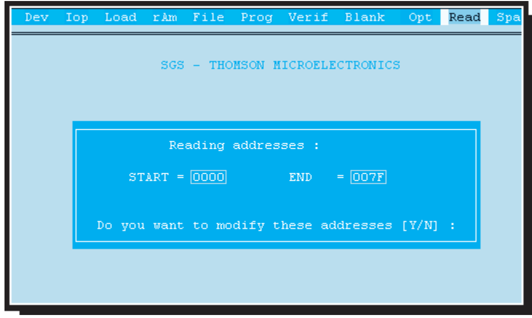


Fig.14 Portate il cursore sulla finestra in alto con scritto READ poi pigiate Y.

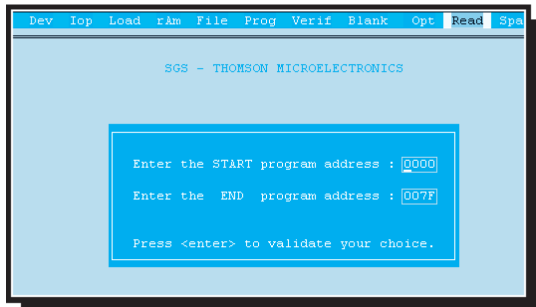


Fig.15 Quando apparirà questa finestra pigiate 2 volte di seguito il tasto Enter.

ta **B** nel **2° ciclo**, modificate il programma così:

ldi **stsex,45** **; 296** tempo in **secondi**
ldi **stmix,0** **; 297** tempo in **minuti**
set **6,port_b** **; setta l'uscita 6 di port B a 1**

I PULSANTI P1 - P2

I pulsanti **P1 - P2** sulla scheda display permettono di bloccare, far ripartire e resettare il conteggio.

Premendo in successione **P1** bloccate e fate ripartire il conteggio dal numero sul quale era stato fermato. Dopo aver bloccato il conteggio con **P1**, premendo il tasto **P2** il programma verrà **resettato**. In questo caso quando premerete **P1** il conteggio ripartirà **sempre** dal **1° ciclo**.

TEST con EEPROM tipo ST62E65

Inserite un micro **ST62E65** nello zoccolo del programmatore **LX.1325** e, dopo aver richiamato il programma, premete il tasto **F3** in modo da far apparire i nomi dei files.
Portate il cursore sulla riga:

EEPR65T.ASM premete Enter

In questo modo appaiono le istruzioni del **programma** con i relativi commenti (vedi fig.11).
A questo punto tenendo pigiato il tasto **ALT** pigiate il tasto **T** ed apparirà la finestra di fig.8.
Premete il tasto **R** e quasi subito apparirà la finestra di programmazione con tutte queste sigle.

ST62E60
ST62E60B
ST62T60
ST62T60B
ST62E65
ST62E65B
ST62T65
ST65T65B

Se sul corpo del vostro micro è stampigliata la sigla **ST62E65/B6** dovete selezionare **ST62E65**.

Se sul corpo del vostro micro è stampigliata la sigla **ST62E65/BB6** dovete selezionare **ST62E65B**.

Pigiate **L = Load** e nella finestra che appare (vedi fig.12) digitate il nome del programma, cioè **EEPR65T** tralasciando **.ASM**, quindi premete Enter e quando appare la finestra con la scritta **File checksum** premete Enter.
Apparirà una finestra bianca e a questo punto potete pigiare il tasto **P = Prg** poi pigiate il tasto **N**.

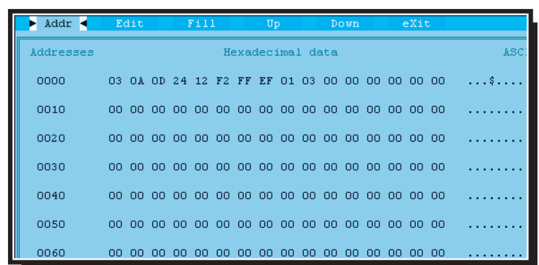


Fig.16 Andate sulla scritta rAm poi pigiate Enter e sullo schermo apparirà il contenuto della EEprom. Notate nella prima riga i dati memorizzati nella EEprom.

Fig.17 Se nella EEprom non è memorizzato nessun dato, vedrete tutti 00 oppure tutti FF. Normalmente FF è presente nei soli micro OTP non cancellabili.

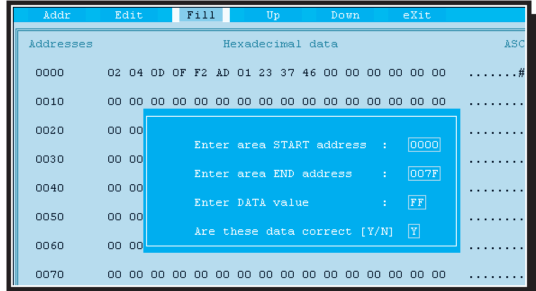


Fig.18 Per ripulire una EEprom andate sulla riga FILL e quando appare questa finestra scrivete nella terza riga FF, poi andate sull'ultima riga e pigiate il tasto Y.

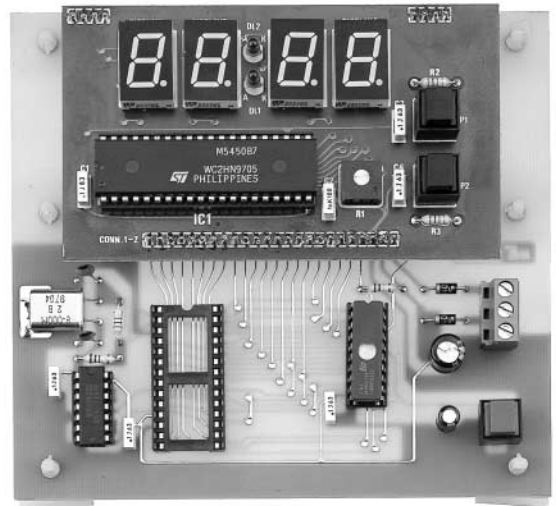


Fig.19 Inserendo nel bus la scheda con display LX.1204 ed anche un micro ST62/60 programmato con il programma Test EEPROM60, potete vedere come un dato memorizzato nella EEprom non si cancelli anche quando si toglie la tensione di alimentazione al Bus.

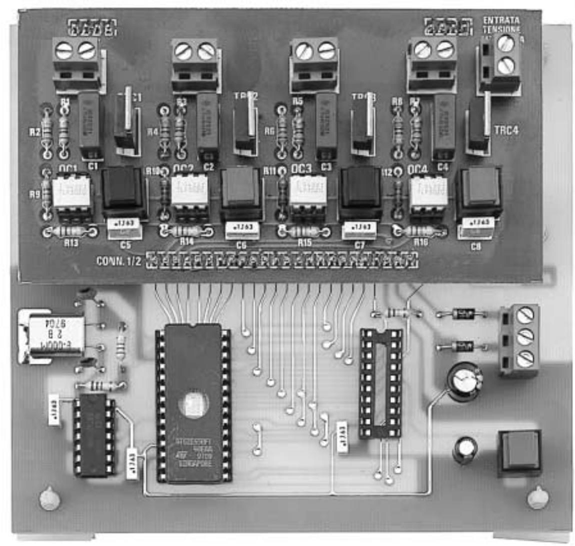


Fig.20 Inserendo nel bus la scheda con Triac LX.1206 ed anche un micro ST62/65 programmato con il programma Test EEPR65T, togliendo la tensione di alimentazione e poi reinserendola il programma ripartirà sempre dalla lampada che risultava accesa in precedenza.

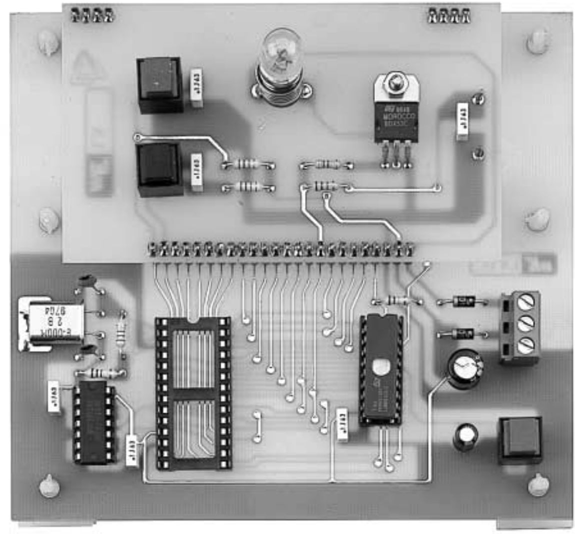


Fig.21 Inserendo nel bus la scheda LX.1329/B (vedi fig.28) potrete vedere come funziona il PWM.
Modificando le righe del programma come spiegato nell'articolo riuscirete a fare un po' di pratica che vi sarà molto utile per poter usare correttamente la EEprom e il PWM.

Quando il micro risulterà programmato sul monitor apparirà la scritta:

The device is succesfully programmed
Il micro è stato correttamente programmato

A questo punto potete estrarre dal **programmatore** il micro già **programmato** per inserirlo nella scheda Bus **LX.1329** alla quale dovete collegare la scheda del kit **LX.1206**, pubblicata sulla rivista **N.180**, provvista di quattro **Triac**.

Nel programma **EEPR65T** è inserito un **timer** che provvede ad **accendere** e **spegnere** a ciclo continuo le quattro lampade collegate ai Triac.

Anche questo programma utilizza la **memoria EEprom**. Infatti se togliete la tensione di alimentazione al Bus quando una delle lampade è accesa, alimentandolo nuovamente dopo **10 minuti** oppure dopo **3 ore** o anche dopo **1 mese**, si riaccenderà la lampada che risultava accesa al momento dello spegnimento, perché questo dato è stato memorizzato nella **EEprom**.

Se volete variare i tempi di accensione e spegnimento delle lampade dovete modificare queste righe del programma:

1° ciclo = righe **626-627**
2° ciclo = righe **635-636**
3° ciclo = righe **643-644**
4° ciclo = righe **651-652**

Se volete che il **1° ciclo** abbia una durata di **5 secondi**, modificate nel modo seguente le righe 626 - 627 del programma:

```
ldi    stsex,5     ; 626 tempo in secondi
ldi    stmix,0     ; 627 tempo in minuti
set    6,port_b    ; setta l'uscita 6 di port B a 1
```

La riga 628 del programma serve per mantenere accesa la lampada per il tempo prefissato, dopodiché il programma passa al **2° ciclo** .

Per modificare i tempi di accensione dovete apportare modifiche su ognuno dei **4 cicli**. Per queste modifiche vi aiuteranno i commenti da noi riportati per ogni riga di programma.

NOTE IMPORTANTI

La lampada ad ultravioletti riesce a **cancellare** il programma memorizzato nel micro, ma **non** cancella il contenuto della **memoria EEprom**.

Per controllare il contenuto di questa memoria e cancellarla procedete come segue:

– Inserite il micro nel **programmatore**.

– Posizionatevi nella **directory C:\ST626>** e digitate quanto sotto riportato:

C\ST626>ST626xPG   premete Enter

– Nella finestra che appare (vedi fig.9) scegliete il tipo di **micro** inserito nel programmatore poi pigiate il tasto Enter.

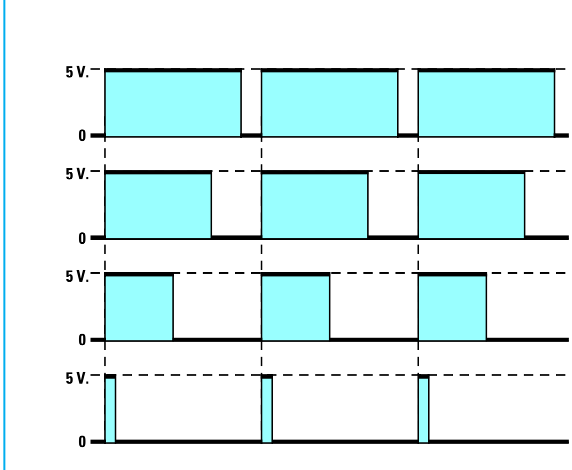


Fig.22 Dal piedino d'uscita del PWM non esce una tensione continua variabile da 0 a 5 volt, ma solo delle onde quadre con un livello logico 0-1.
Modificando tramite programma il duty-cycle di queste onde quadre, vale a dire il tempo che queste rimangono a livello logico 0 e a livello logico 1, riuscirete ad ottenere una tensione efficace che da un minimo di 0 volt potete elevare fino a un massimo di 5 volt.

– Nella pagina che appare (vedi fig.10) andate sulla scritta **Load** e pigiate Enter.

– Digitate il nome dal **file** utilizzato, ad esempio **EEprom60**, poi pigiate Enter.

– Andate sulla scritta **Space**, posizionata nella riga in alto, poi pigiate Enter.

– Nella finestra di fig.13 andate sulla riga **EEPROM** pigiando il tasto della Barra ed Enter.

– Per vedere il contenuto della **EEPROM** andate sulla scritta **Read** poi digitate **Y** (vedi fig.14) e nella finestra che appare (vedi fig.15) pigiate Enter **due volte**.

– Andate sulla scritta **rAm** poi pigiate Enter. Vedrete così sullo schermo il contenuto della **EEprom** (vedi fig.16).

– Se la EEprom **non** è programmata vedrete tutte le celle su **FF** (vedi fig.17).

Per Cancellare o Modificare la EEPROM

– Andate sulla scritta **Fill** poi pigiate Enter.

– Nella **terza** riga della finestra che appare (vedi fig.18) scrivete **FF**, poi andate con il cursore sull'ultima riga in basso, quindi pigiate **Y** per confermare la modifica. Automaticamente tutte le celle si caricheranno con **FF**.

– In sostituzione di **FF** potreste anche scrivere **00**, ma è consigliabile usare **FF** perché le celle dei micro **OTP** sono tutte **FF**.

Nota: se volete cancellare o modificare una **sola cella** dovete selezionare **Edit**.
Spostando il cursore potrete così portarvi sulla cella che vi interessa modificare.

– Per uscire pigiate **Escape** poi **X**.

Tenete presente che le modifiche appena apportate **non** vanno automaticamente a ripulire le **EEprom**, in cui rimarranno memorizzati i vecchi dati.

Per memorizzare i dati cambiati nelle memorie **EEprom** dovete andare sulla scritta **PROG**, poi pigiare **due volte** Enter.

Queste note, che non troverete in nessun manuale, vi saranno molto utili perché vi permetteranno di **vedere** concretamente e, volendo di **modificare** manualmente, il contenuto delle **EEprom**.

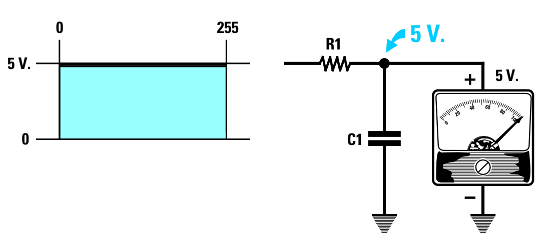


Fig.23 Se l'onda quadra generata dal PWM rimane a livello logico 1 per i suoi totali 256 step, in uscita otterrete 5 volt.

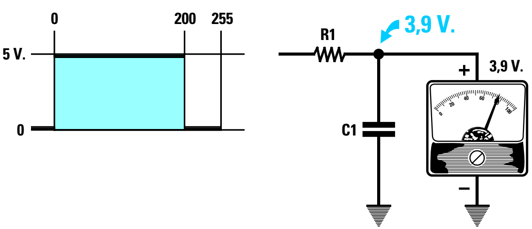


Fig.24 Se l'onda quadra rimane a livello logico 1 per 200 step su 256 step totali, otterrete una tensione di soli 3,9 volt.

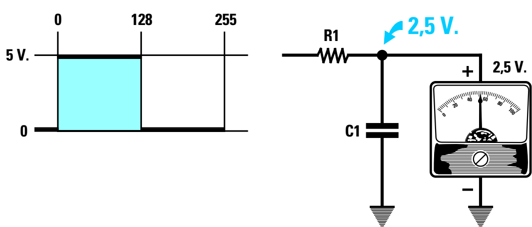


Fig.25 Se l'onda quadra rimane a livello logico 1 per 128 step, otterrete in uscita una tensione di soli 2,5 volt.

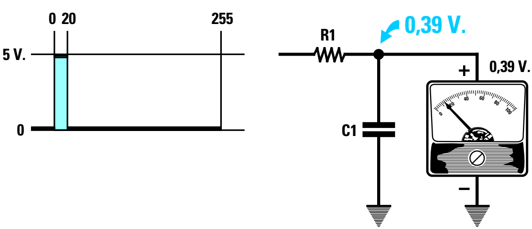


Fig.26 Riducendo a soli 20 step il livello logico 1 dell'onda quadra, otterrete in uscita una tensione di soli 0,39 volt.

TEST PWM con ST62/60-65

Per questo **programma-test** si può usare sia un micro **ST62E60** sia un micro **ST62E65**.

Prima di presentarvi il nostro programma, è necessario spiegare come si fa con la tecnica **PWM** a trasformare i **livello logici 1 - 0** in un valore di tensione continua **variabile**.

Come già sapete il **livello logico 1** corrisponde ad una tensione di **5 volt positivi** ed il **livello logico 0** ad una tensione di **zero volt**.

Poiché il minimo numero decimale che possiamo usare è **0** ed il massimo numero **255**, agendo sul registro **PWM** tramite software si potrà decidere di quanti step da **0** a **255** il segnale dovrà rimanere a **livello logico 1** e di quanti step dovrà rimanere a **livello logico 0**.

Se programmate il **registro PWM** in modo che rimanga a **livello logico 1** da **0** fino a **255**, in uscita otterrete la tensione massima di **5 volt** (vedi fig.23).

Se programmate il **registro PWM** in modo che rimanga a **livello logico 1** dallo step **0** fino a **200** e rimanga da questo numero fino a **255** a livello **logico 0**, in uscita otterrete una tensione di soli di **3,9 volt** (vedi fig.24).

Se programmate il **registro PWM** in modo che rimanga a **livello logico 1** dallo step **0** fino a **128** e rimanga da questo numero fino a **255** a livello **logico 0**, in uscita otterrete **metà** tensione, cioè **2,5 volt** (vedi fig.25).

A questo punto è abbastanza intuitivo che se programmate il **registro PWM** in modo che rimanga a **livello logico 1** dallo step **0** fino a **20** e rimanga da questo numero fino a **255** a livello **logico 0**, in uscita otterrete una tensione di soli **0,39 volt** (vedi fig.26).

Anche se il segnale ad onda quadra che fuoriesce dalla **porta PB7** raggiunge sempre un picco massimo di **5 volt**, dovete considerare il valore dei **volt efficaci**, che risultano proporzionali al tempo che l'onda quadra rimane a **livello logico 1** e a **livello logico 0**.

In linea di massima si potrebbe calcolare il valore di questa tensione dividendo i **5 volt** per i **256 livelli** (da **0** a **255** i livelli sono **256**), poi moltiplicare il risultato per il numero degli step in cui l'onda quadra rimane a **livello logico 1**.

Considerando i valori riportati nelle figg.23-26 otterrete queste esatte tensione:

(5 : 256) x 256 = 5,0 volt
(5 : 256) x 200 = 3,9 volt
(5 : 256) x 128 = 2,5 volt
(5 : 256) x 20 = 0,39 volt

Il condensatore **C1** posto dopo la resistenza **R1** permette di ottenere una tensione **continua efficace** del treno di onde quadre con il **duty-cycle** variabile che fuoriesce dal **PWM**.

Vi starete chiedendo ora a cosa serve una tensione variabile da **0** a **5 volt** se a lato pratico serve una tensione variabile da **0** a **24 volt** oppure da **0** a **220 volt**.

Anche se vi servisse una tensione variabile da **0** a **5 volt** per accendere una piccola **lampadina** non potremmo mai utilizzarla perché la tensione fornita dal **PWM** non ha potenza.

Come potete vedere anche dalla nostra scheda sperimentale siglata **LX.1329/B**, che andrà innestata nel Bus, i **5 volt** vengono utilizzati per pilotare la **B**ase di un transistor di potenza sul cui **C**ollettore abbiamo inserito una lampadina da **12 volt 3 watt**.

Inserendo questa scheda nel Bus, nel quale andrà inserito anche un micro **ST62E60** programmato con il programma:

PWM60.ASM

noterete quanto segue:

– Alimentando il Bus la lampadina si accenderà per un **50%** della sua **luminosità**.

– Ogni volta che premete il tasto **P2** la luminosità della lampadina si **attenua**, perché la tensione scende di volta in volta di **0,5 volt**.

– Ogni volta che premete il tasto **P1** la luminosità della lampadina **aumenta**, perché la tensione sale di volta in volta di **0,5 volt**.

Per modificare il valore del **salto** di luminosità ogni volta che si premono i due pulsanti, dovete variare questi righe di programma:

STARTPW	**.EQU**	**5**	riga **41**
CAPTPW	**.EQU**	**130**	riga **42**
MINPW	**.EQU**	**30**	riga **43**
STEPW	**.EQU**	**25**	riga **44**

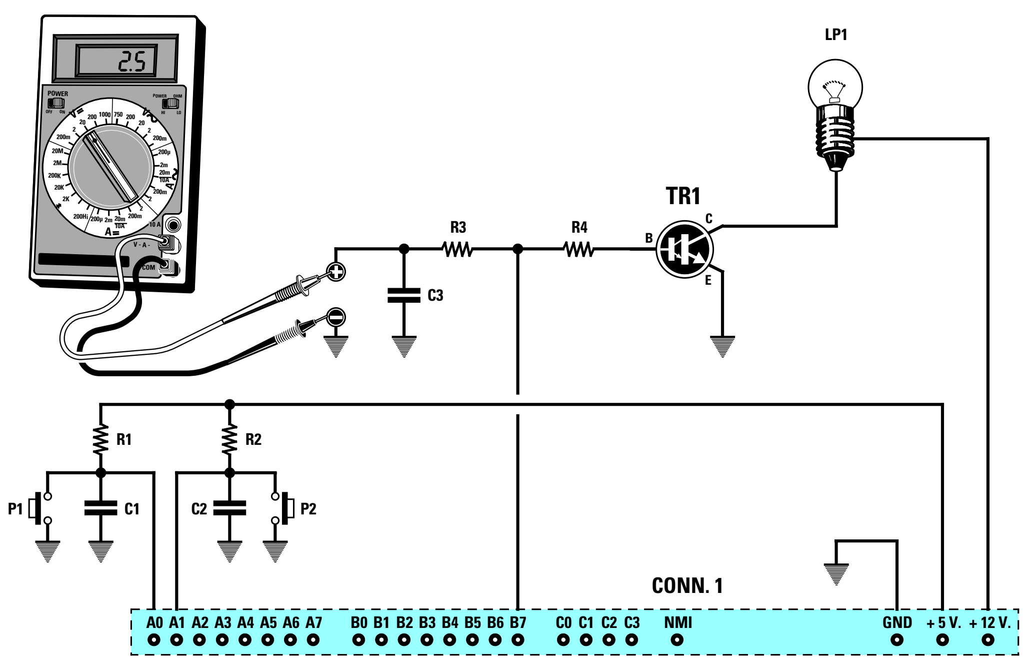


Fig.27 Schema elettrico della scheda siglata LX.1329/B. Come potete vedere in figura la tensione variabile da 0 a 5 volt presa su B7 del CONN.1 si applica sulla Base del transistor TR1 che provvede ad accendere dal suo minimo al suo massimo una lampadina da 12 volt. Collegando un Tester sui terminali +/– potrete leggere la tensione fornita dal PWM.

ELENCO COMPONENTI LX.1329/B

R1 = 10.000 ohm
R2 = 10.000 ohm
R3 = 22.000 ohm
R4 = 4.700 ohm
C1 = 100.000 pF poliestere
C2 = 100.000 pF poliestere
C3 = 100.000 pF poliestere
TR1 = NPN darlington BDX.53C
P1 = pulsante
P2 = pulsante
LP1 = lampada 12 volt 3 watt

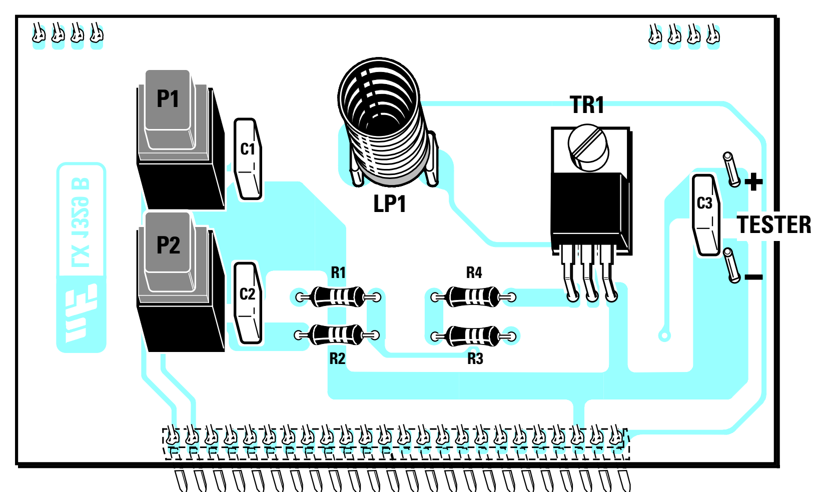


Fig.28 Schema pratico di montaggio della scheda LX.1329/B da usare per i test PWM.

In questo programma abbiamo diviso la frequenza del quarzo da **8 MHz** per **3** utilizzando il **registro ARS2**, definito nella locazione **0D7H** ottenendo così una frequenza base di **2,667 MHz**.

Quando l'**Auto-Reload Timer** arriva al numero **255**, ricarica il **timer** con il numero che abbiamo messo nel registro **ARRC**, che si trova nella locazione di memoria **0D9H**.
Se carichiamo il registro **ARRC** con il numero **0**, il timer partirà da **0** per arrivare a **255** e raggiunto questo valore massimo ripartirà da **0**.

Se carichiamo il registro **ARRC** con il numero **127**, il timer partirà da **127** per arrivare a **255** e raggiunto questo valore massimo ripartirà da **127**.

Poiché nel nostro programma abbiamo caricato il registro **ARRC** con il numero **5**, il timer ripartirà sempre da questo valore e per arrivare a **255** noi avremo disponibili **255 – 5 = 250** step.

Questo significa che per **ogni step** potremo incrementare il valore efficace dei nostri **5 volt** di:

5 : 250 = 0,02 volt

Conoscendo il valore degli **step** (**250**) e la **frequenza base** (**2,667 MHz**) possiamo ricavare la frequenza di lavoro del **PWM**, che nel nostro caso sarà pari a:

2,667 : 250 = 0,0106 MHz (**10,6 KHz** circa)

Se nel registro **ARRC** avessimo messo il numero **127**, il timer sarebbe ripartito sempre da questo valore, quindi per arrivare a **255** avremmo avuto disponibili **255 – 127 = 128 step**.

Vale dire che per **ogni step** avremmo incrementato il valore efficace dei nostri **5 volt** di:

5 : 128 = 0,039 volt

Conoscendo il valore degli **step** (**128**) e la **frequenza base** (**2,667 MHz**), possiamo ricavare la frequenza di lavoro del **PWM** che nel nostro caso sarà pari a:

2,667 : 128 = 0,020 MHz (**20 KHz** circa)

Quindi **riducendo** il numero degli **step** otterremmo un **aumento** della **frequenza** di lavoro.

Di seguito spieghiamo il significato di alcune righe di programma.

STARTPW .EQU 5 (riga **41**) = è il valore degli **step** definito in **Auto - Reload - Timer**, che come già abbiamo visto corrispondono ad un valore di tensione minima di **0,02 volt**.

CAPTPW .EQU 130 (riga **42**) = è il valore del **comparatore** interno che utilizziamo per stabilire da quale **valore** di tensione desideriamo partire.
Poiché lo **StartPW** ha un valore di **5**, noi partiamo da un valore di tensione pari a:

(130 – 5) x 0,02 = 2,5 volt

MINPW .EQU 5 (riga **43**) = definisce il valore **minimo** a cui vogliamo arrivare con la tensione.
Sottraendo a **5** il valore dello **StartPW** noi riusciamo a scendere fino ad un valore di:

(5 – 5) = 0 volt

STEPW .EQU 25 (riga **44**) = in questa riga abbiamo inserito il numero di **salti** di tensione che vogliamo ottenere ogni volta che andiamo a pigiare i pulsanti **P1** o **P2**.
Questo numero va moltiplicato per il valore di tensione corrispondenti ad uno step, cioè a **0,02 volt**.
Con il nostro programma facciamo dei salti di:
25 x 0,02 = 0,5 volt

Se ad esempio volessimo fare dei salti di **1 volt** anziché di **0,5 volt**, partendo dal valore **minimo** di **1 volt** dovremmo modificare le righe **42-43-44** come qui sotto riportato:

```
CAPTPW   .EQU   55   riga 42
MINPW    .EQU    5   riga 43
STEPW    .EQU   50   riga 44
```

Se volessimo fare dei salti di soli **0,04 volt** partendo sempre da un valore minimo di **0 volt**, dovremmo modificare così le righe:

```
CAPTPW   .EQU   5   riga 42
MINPW    .EQU   5   riga 43
STEPW    .EQU   2   riga 44
```

MEMORIZZARE un vostro PROGRAMMA

Fino a qui vi abbiamo spiegato come trasferire i nostri programmi di **test** nella memoria del micro.
Ovviamente i softwaristi vorranno memorizzare nel micro i loro personali programmi e quindi non ci rimane che darvi qualche piccola nota di aiuto.

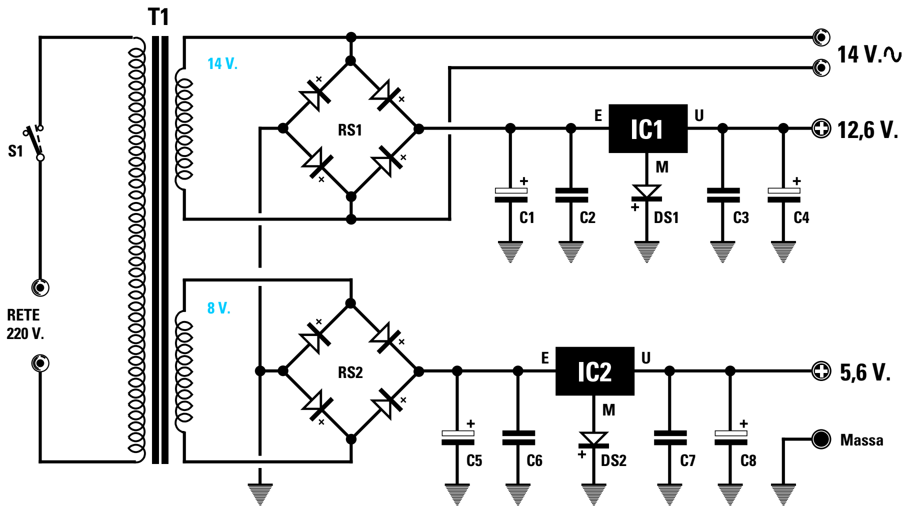
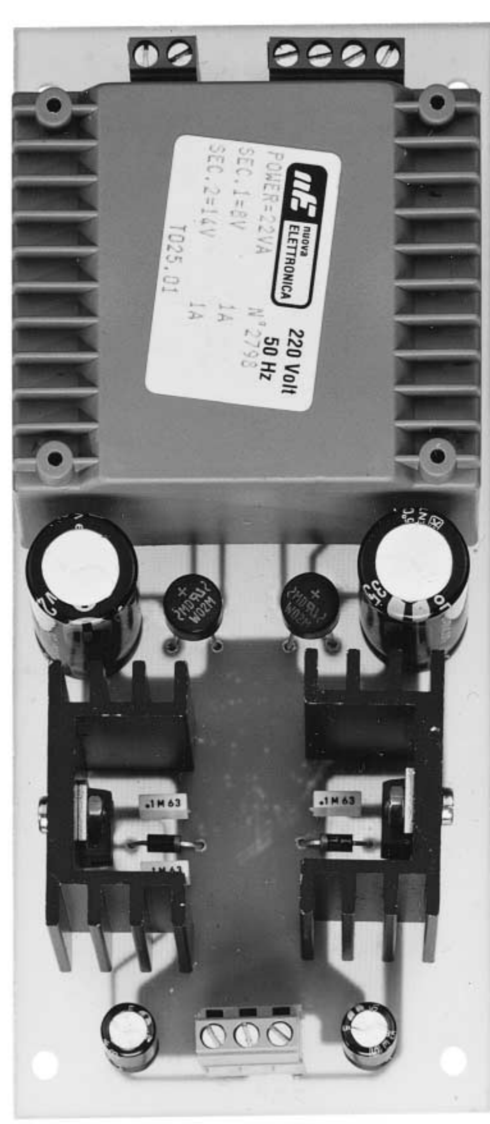


Fig.29 Schema elettrico dello stadio di alimentazione da usare per alimentare il Bus LX.1329. Chi ha già realizzato il Bus per il precedente programmatore per ST6 potrà usare l'alimentatore che già possiede anche per questo Bus.



ELENCO COMPONENTI LX.1203

C1 = 2.200 mF elettr. 35 volt
C2 = 100.000 pF poliestere
C3 = 100.000 pF poliestere
C4 = 100 mF elettr. 35 volt
C5 = 2.200 mF elettr. 35 volt
C6 = 100.000 pF poliestere
C7 = 100.000 pF poliestere
C8 = 100 mF elettr. 35 volt
DS1 = diodo 1N.4007
DS2 = diodo 1N.4007
RS1 = ponte raddriz 100 V 1 A
RS2 = ponte raddriz 100 V 1 A
IC1 = uA.7812
IC2 = uA.7805
T1 = trasform. 25 watt (T025.01)
sec. 14 V 1A – 8 V 1 A
S1 = interruttore

Fig. 30 Foto dello stadio di alimentazione già presentato sulla rivista N.179 perché usato per alimentare il Bus LX.1202 per i normali micro ST6 senza EEprom e PWM.

Innanzitutto precisiamo che l'**EDIT** da noi inserito all'interno del floppy **DF.1325** assieme ai programmi di **test** è molto limitato. Non accetta infatti programmi maggiori di **30 Kilobyte**. Se andrete a salvare dei programmi che occupano uno spazio maggiore, tutto quello che eccede i **30 Kilobyte** verrà inesorabilmente **cancellato**.

Per modificare o salvare programmi che occupano più di **30 Kilobyte** dovete obbligatoriamente utilizzare l'**EDITOR** del **DOS** presente nel vostro computer.
Per entrare nel menu principale dell'editor digitate queste istruzioni:

C:\>CD ST626 premete Enter
C:\ST626>Edit premete Enter

Dopo aver corretto o modificato il vostro programma, prima di trasferirlo nella memoria del micro dovete sempre **assemblarlo**, quindi uscite dal programma pigiando **ALT + F**, poi **ALT + X** e così apparirà sul monitor:

C:\ST626>

Ora dovete digitare:

C:\ST626>AST6 –S –L Pluto premete Enter

Nota: dove noi abbiamo scritto **Pluto** voi dovete scrivere il **nome** del vostro programma.

Dopo diversi secondi sul monitor apparirà:

*****SUCCESS*****

a conferma che l'**assemblaggio** è stato completato senza riscontrare **nessun errore**.

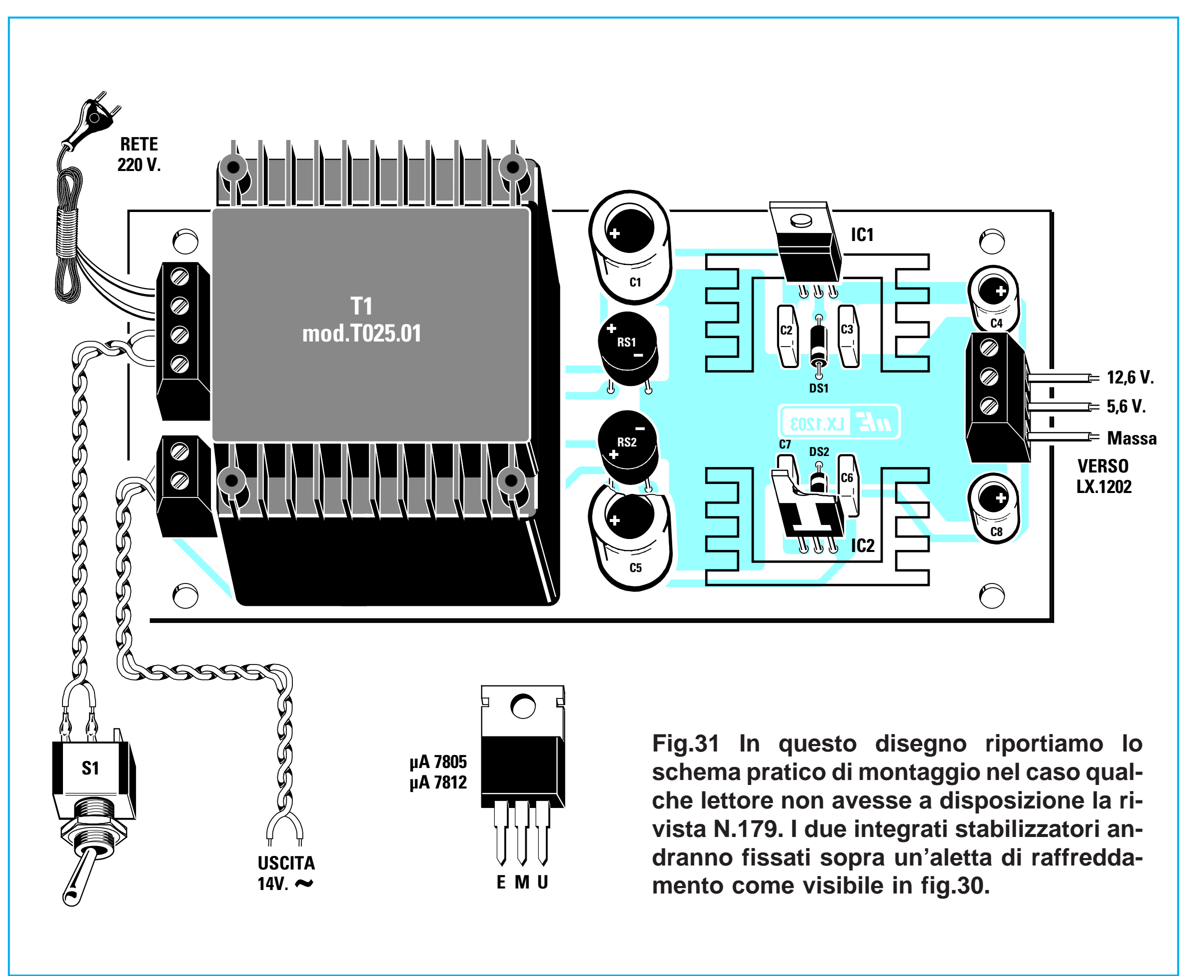


Fig.31 In questo disegno riportiamo lo schema pratico di montaggio nel caso qualche lettore non avesse a disposizione la rivista N.179. I due integrati stabilizzatori andranno fissati sopra un'aletta di raffreddamento come visibile in fig.30.

Se al posto di questa scritta dovesse apparirne una di **errore**, ad esempio:

ERROR C:\ST626\pluto.ASM 151:

significa che nella **riga 151** esiste un errore, quindi rientrate all'interno del vostro programma e correggete l'istruzione in tale riga.

Per fare questa correzione dovete nuovamente richiamare l'**Editor**, andare sulla **riga 151** e dopo aver corretto l'istruzione dovete **riassemblare** il programma procedendo come vi abbiamo appena spiegato.

Poiché in fase di compilazione abbiamo usato le opzioni **–L –S**, verranno generati questi 4 files:

Pluto.DSD
Pluto.HEX
Pluto.SYM
Pluto.LIS

Il file **Pluto.SYM** e il file **Pluto.DSD** serviranno per i programmi di **simulazione**, già conosciuti con i nomi di **DSE622** e di **ST622**.

Il file **Pluto.LIS** contiene il listato completo del programma che potrà risultarvi utile per una consultazione o come **copia** di salvataggio.

Quando sul monitor vi appare:

*****SUCCESS*****

proseguite digitando:

C:\ST626>ST626xPG premete Enter

In questo modo apparirà la finestra di fig.9 e a questo punto procedete con le istruzioni riportate a pag.110, che vi spiegano come trasferire il programma dal **computer** verso il **micro**.

NOTA IMPORTANTE

Usando il sistema operativo **Windows 3.1** non incontrerete nessun problema, ma lo stesso non si può dire con **Windows 95**.
Se usando **Windows 95** riscontrate dei problemi nel lanciate il programma **ST626xPG**, VI consigliamo di inserire nell'ultima riga del file **CONFIG.SYS** questa opzione utilizzando il programma **Edit** oppure **Notepad** o **Write** o se siete esperti, il comando **Sysedit**:

SWITCHES /C

Ora **salvate** il file, quindi **spegnete** il computer e **riaccendetelo**. A questo punto non dovreste più incontrare nessun problema ad utilizzare il programma:

ST626xPG

Con tutte queste spiegazioni ed esempi vogliamo sperare di aver dissipato buona parte dei dubbi che avevate sulle **EEprom** e sul **PWM**.

COSTO di REALIZZAZIONE

Tutti i componenti necessari per la realizzazione del Bus **LX.1329** (vedi figg.2-3-4) completo di circuito stampato e del **74HC00** .... € 19,60

Tutti i componenti necessari per la realizzazione dell'interfaccia **LX.1329/B** (vedi figg.27-28) completa di circuito stampato e di una lampadina da 12 volt per testare il **PWM** .... € 8,80

Costo del solo stampato **LX.1329** .... € 11,10
Costo del solo stampato **LX.1329/B** .... € 4,29

Costo di un **ST62E60** cancellabile .... € 20,66
Costo di un **ST62T60** non cancellabile ..... € 12,39

Costo di un **ST62E65** cancellabile .... € 18,08
Costo di un **ST62T65** non cancellabile ..... € 14,98

COSTO dei precedenti KIT per ST6

Costo dello stadio di alimentazione **LX.1203** (vedi fig.30) pubblicato sulla rivista N.179, **Escluso** il mobile plastico MTK06.22 .... € 25,80

Costo del mobile plastico **MTK06.22** per lo stadio di alimentazione .... € 6,97

Costo del Kit della scheda **Display LX.1204** pubblicata sulla rivista N.179 .... € 18,60

Costo del Kit della scheda **Triac LX.1206** pubblicata sulla rivista N.180 .... € 18,60

I prezzi riportati sono compresi di **IVA**, ma non delle spese **postali** che verranno addebitate solo a chi richiederà il materiale in contrassegno.

LE DIRETTIVE dell'assembler ST6

In questo articolo spieghiamo in maniera dettagliata la direttiva .BYTE, usata per la definizione di dati nell'area del programma, e le direttive .EQU e .SET, che servono per la definizione delle costanti simboliche.

LA DIRETTIVA chiamata .BYTE

La direttiva **.byte** viene utilizzata per definire in **Program Space** una successione di bytes contenenti valori binari ai quali si possono associare eventuali **Etichette**.
Ogni tentativo di inserire questa direttiva nella **Data Space** darà un **errore** in Compilazione.
Come per le direttive **.ascii** e **.asciz**, i valori definiti in **Program Space** non sono modificabili durante il corso del programma.
Per utilizzare i valori definiti con la direttiva **.byte** bisogna prima **caricarli** in **Data Rom Window** utilizzando le stesse modalità e gli stessi accorgimenti già spiegati nel capitolo riguardante la direttiva **.w_on** (vedi rivista **N.190**).

In fase di stesura del programma bisogna attenersi a quanto riportato nel paragrafo riguardante la direttiva **.block** (vedi rivista **N.190**).

L'utilizzo della direttiva **.byte** ci permette di definire una notevole quantità di valori binari in **Program Space** senza **riempire** inutilmente l'area di **Data Space** che è di soli **60 bytes**, che potrà così essere utilizzata per la dichiarazione delle **Variabili** del programma tramite la direttiva **.def**.

Il formato logico della direttiva **.byte** è il seguente:

[etichetta] .byte espress[,espress]

Nota: gli operandi posti fra **parentesi quadre** sono opzionali quindi possono essere omessi.

[etichetta] = va inserito il nome dell'etichetta che vogliamo associare alla locazione di **Program Space** del **1°** valore definito. Questo nome è opzionale quindi può essere omesso.

espress[,espress] = possono essere uno o più valori espressi in **Decimale**, **Binario** o **Esadecimale** separati ognuno da una **virgola** e non devono mai superare la capacità di **8 bits**; oppure possono essere delle **espressioni** (vedi rivista **N.189**) il cui **risultato finale** non deve comunque mai superare

la capacità di **8 bits**, che corrisponde ad un valore di **255**.

L'impiego della direttiva **.byte** risulta particolarmente utile per effettuare conversioni, trasposizioni, sostituzioni di valori o per realizzare delle tabelle di comparazione.

1° Esempio

Con questo esempio vi insegniamo ad utilizzare i numeri **decimali - esadecimali - binari** o le **espressioni** per definire una serie di **tabelle** in Program Space.
Poiché l'esempio è stato definito correttamente, in fase di **Compilazione** non si presenteranno errori.

```
elisto    .def     086h
costan    .set     025h
step01    .equ     020h
          .block   64-$%64
tabval1   .byte    10,15,18,23,45,78,109
tabval2   .byte    010h,015h,018h,023h
tabval3   .byte    00100000b,01010111b
tabval4   .byte    costan*2,elisto+10
tabval5   .byte    step01+18,step01+31
```

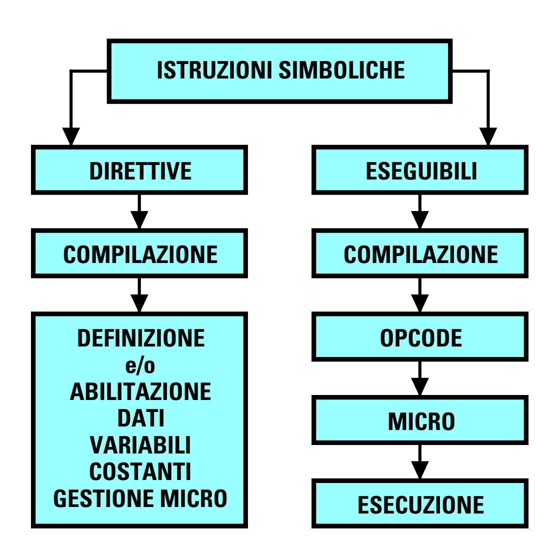


Fig.1 Semplificando possiamo definire le direttive come disposizioni generali atte a fissare le caratteristiche di fondo che necessitano al programma. Ad esempio: definizione di dati, variabili, macro, costanti simboliche, abilitazione dell'area di memoria Data Rom Window, ecc. Perciò, come le istruzioni eseguibili, vengono compilate, ma non generano una opcode, cioè un codice operativo eseguibile.

Il significato di queste istruzioni è il seguente:

elisto .def 086h = definisce la variabile **elisto** all'indirizzo di memoria **086h** di **Data Space**.

costan .set 025h = associa il valore **025h** alla etichetta **costan** senza occupare nessuna aera di **Program Space**.

step01 .equ 020h = associa il valore **020h** all'etichetta **step01** senza occupare nessuna aera di **Program Space**.

.block 64-$%64 = questa funzione è stata già spiegata nelle riviste **N.189** e **N.190**.

tabval1 .byte 10,15,18,23,45,78,109 = definisce in un indirizzo di memoria di **Program Space** una sequenza di **7 bytes** contenenti i valori **decimali** sopra riportati ed associa al primo byte l'etichetta **tabval1**. Poiché i numeri separati dalle **virgole** non superano **255** il compilatore non segnalerà **errore**.

tabval2 .byte 010h,015h,018h,023h = definisce in un indirizzo di memoria di **Program Space** una sequenza di **4 bytes** contenenti i valori **esadecimali** sopra riportati ed associa al primo byte l'etichetta **tabval2**.
Poiché i valori separati dalle **virgole** non superano **0FFh** (che equivale a **255** decimale) il compilatore non segnalerà **nessun errore**.

tabval3 .byte 00100000b,01010111b = definisce in un indirizzo di memoria di **Program Space** una sequenza di **2 bytes** contenenti i valori **binari** sopra riportati ed associa al primo byte l'etichetta **tabval3**. Poiché i valori separati dalle **virgole** non superano **11111111b** (che equivale a **255** decimale) il compilatore non segnalerà **nessun errore**.

tabval4 .byte costan*2,elisto+10 = definisce in un indirizzo di memoria di **Program Space** una sequenza di **2 bytes** contenenti il valore risultante dalle espressioni **costan*2** ed **elisto+10**.
Poiché **costan** è stato definito **025h**, moltiplicandolo per **2** otteniamo **04Ah**.
Infatti **025h** corrisponde al numero **decimale 37**, che moltiplicato per **2** da **74**, che corrisponde al numero **esadecimale 04Ah**.
Poiché **elisto** è stato definito **086h**, che convertito in **decimale** corrisponde al valore **decimale 134**, sommando a questo **10** otteniamo **144**, che corrisponde al numero esadecimale **090h**.
Poiché entrambi i numeri non superano **255 decimale** o **0FFh esadecimale** il compilatore non segnalerà nessun errore.

tabval5 .byte step01+18,step01+31 = definisce in un indirizzo di memoria di **Program Space** una sequenza di **2 bytes** contenenti il valore risultante dalle espressioni **step01+18** e **step01+31**.
Poiché **step01** è stato definito **020h**, che convertito in **decimale** corrisponde al valore **decimale 32**, sommando a questo **18** e **31** otteniamo:

32 + 18 = 50 corrispondente a **032h**
32 + 31 = 63 corrispondente a **03Fh**

Una volta compilate, le **tabelle** si troveranno memorizzate una di seguito all'altra in **Program Space** ed occuperanno un totale di **17 bytes**:

7 bytes per **tabval1**
4 bytes per **tabval2**
2 bytes per **tabval3**
2 bytes per **tabval4**
2 bytes per **tabval5**

Per utilizzarle dovete procedere come già spiegato nella Rivista **N.190**, al capitolo relativo alla **direttiva .w_on**.

2° Esempio

In questo esempio abbiamo inserito un **errore** che verrà segnalato in fase di **Compilazione**.

```
elisto   .def    086h
costan   .set    025h
step01   .equ    $+20
         .block  64-$%64
tabval1  .byte   10,15,18,23,45,78,109
tabval2  .byte   010h,015h,018h,023h
tabval3  .byte   00100000b,01010111b
tabval4  .byte   costan*2,elisto+10
tabval5  .byte   step01+18,step01+31
```

La **3° riga** dell'esempio precedente era:

step01 .equ 020h

In questo **secondo** esempio è stata sostituita con:

step01 .equ $+20

Nel capitolo riguardante le **espressioni** abbiamo detto che il Compilatore sostituisce al simbolo **$** il valore del **Program Counter Relativo**. Ammesso quindi che nella Compilazione la definizione:

step01 .equ $+20

venga a trovarsi nella locazione di memoria **Program Space 0A13h**, che corrisponde al numero decimale **2.579**, quando il compilatore andrà ad eseguire l'ultima istruzione, cioè:

tabval5 .byte step01+18,step01+31

segnalerà subito questo errore:

Error on (8) bits Overflow

Infatti la Costante Simbolica **step01** che equivale a **2.579 decimale** supera già il massimo consentito di **255 decimale** quindi non riesce a sommare come richiesto **step01+18,step01+31**.

Se l'ultima istruzione fosse una sottrazione:

tabval5 .byte step01–2360,step01–2500

il compilatore eseguirebbe correttamente questa istruzione **senza segnalare** nessun errore, perché il risultato delle sottrazioni non supera **255**, infatti:

2.579 – 2.360 = 219 (**0DBh**)
2.579 – 2.500 = 79 (**04Fh**)

Poiché durante la simulazione sul monitor appaiono sempre dei valori espressi in **esadecimali**, per evitare errori vi consigliamo di consultare le **Tabelle** riportate a **pag.381** del nostro volume intitolato **Nuova Elettronica HANDBOOK**.

LA DIRETTIVA chiamata .EQU

La direttiva **.equ** viene utilizzata per associare un valore **numerico**, che può essere ricavato anche dal risultato di una **espressione**, ad una **Etichetta** senza **sprecare** nessun byte di memoria del micro e questo la rende molto interessante.

Usando l'**Etichetta** in sostituzione di valori **anonimi** viene facilitata la lettura ed anche l'interpretazione di un programma sorgente (**.asm**), persino a distanza di mesi dalla sua compilazione.
La direttiva **.equ** deve essere inserita **sempre** prima di quella istruzione o di quella routine che utilizza l'**Etichetta**.
Non è possibile definire la stessa **Etichetta** più di una volta, mentre è possibile associare **Etichette** diverse allo stesso valore.

Il formato logico della direttiva **.equ** è il seguente:

[etichetta] .equ [operando]

[etichetta] = va inserito il nome dell'etichetta da associare al **valore numerico** definito nell'operando.

[operando] = va inserito il **valore numerico** o il risultato di una **espressione** da associare all'etichetta. Questo numero, che può essere espresso in **esadecimale**, **binario** o **decimale** non deve mai superare la capacità di **2 bytes** vale a dire:

FFFFh in **esadecimale**
1111111111111111b in **binario**
65535 in **decimale**

Per chiarire eventuali dubbi sull'uso della direttiva **.equ** vi proponiamo alcuni semplici esempi.

1° Esempio

Con questo esempio vi facciamo vedere come la direttiva **.equ** faciliti la lettura del programma:

```
scrivi   .equ   014h
rout00   ldi    a,scrivi
         call   maiusc
rout01   ldi    a,scrivi
         call   minusc
rout02   ldi    a,scrivi
         call   corsivo
```

Nella prima istruzione l'etichetta **scrivi** è stata associata al valore **014h**, che equivale al numero **decimale 20**.
Le tre routine **rout00**, **rout01**, **rout02** caricano, per prima cosa, nell'accumulatore "**a**" il valore associato all'etichetta **scrivi**, poi eseguono le **subroutine** chiamate **maiusc**, **minusc**, **corsivo**.

Queste **subroutine** potrebbero risultare utili per far apparire sul monitor solo **20 caratteri** (**014h**) in **maiuscolo** oppure in **minuscolo** o **corsivo**.

In pratica noi abbiamo scritto:

```
scrivi   .equ   014h
rout00   ldi    a,scrivi
         call   maiusc
```

ma più semplicemente potevamo scrivere:

```
rout00   ldi    a,014h
         call   maiusc
```

Per la logica e l'esecuzione del programma non cambia assolutamente nulla, ma in questo secondo caso rileggendo il programma a distanza di tempo potremmo non ricordare a cosa serve questa istruzione.
Usando la direttiva **.equ** invece sapremo subito che il valore caricato nell'accumulatore "**a**" serve per scrivere **20** caratteri sul monitor in **maiuscolo**.

Se per qualche motivo volessimo modificare il valore da caricare nell'accumulatore "**a**" in modo da scrivere **30** caratteri anziché **20**, sarà sufficiente modificare la **direttiva** come sotto riportato:

```
scrivi   .equ   01Eh
```

Questa direttiva semplifica notevolmente il nostro lavoro perché se nell'esempio riportato le routines sono poste una di seguito all'altra e quindi facilmente individuabili e modificabili, immaginatevi un programma molto più complesso che utilizzi più routines situate in punti diversi e distanti tra loro.
In questo caso si perderebbe tempo a scorrere tutto il programma nella ricerca del valore **014h** per modificarlo in **01Eh** e si potrebbero introdurre involontariamente degli errori.

2° Esempio

La direttiva **.equ** può risultare molto utile quando si vogliano associare valori **diversi** partendo da un **numero fisso** ed utilizzando i simboli matematici per fare una **somma**, una **moltiplicazione** o una **sottrazione**.

```
ritardo  .equ   150
rout01   ldi    a,ritardo
         call   ritardo1
rout02   ldi    a,ritardo+50
         call   ritardo2
rout03   ldi    a,ritardo*4
         call   ritardo3
rout04   ldi    a,ritardo–83
         call   ritardo4
```

In questo esempio all'etichetta **ritardo** è stato associato il valore fisso **150**.

La routine **rout01** carica nell'accumulatore "**a**" il valore associato all'etichetta **ritardo**, cioè **150**, quindi esegue la subroutine **ritardo1**.

La routine **rout02** carica nell'accumulatore "**a**" il valore **150 + 50 = 200**, poi esegue la subroutine **ritardo2** con questo numero.

La routine **rout03** carica nell'accumulatore "**a**" il valore **150 x 4 = 600**, poi esegue la subroutine **ritardo3** con questo numero.

La routine **rout04** carica nell'accumulatore "**a**" il valore **150 – 83 = 67**, poi esegue la subroutine **ritardo4** con questo numero.

Se per qualche motivo volessimo modificare il valore da caricare nell'accumulatore "**a**", così da al-

lungare o accorciare il ritardo in modo proporzionale, sarebbe sufficiente modificare la sola direttiva **ritardo .equ** con il numero desiderato.

Avrete notato che in entrambi gli esempi abbiamo utilizzato l'istruzione **ldi** (load immediate) e non **ld** per caricare nell'accumulatore "**a**" il valore associato alle etichette **scrivi** e **ritardo**.

Se avessimo utilizzato **ld** avremmo caricato nell'accumulatore "**a**" il valore **memorizzato** all'indirizzo di **memoria 014h** e **150** e non il numero **014h** e **150** che a noi serve.

LA DIRETTIVA chiamata .SET

Questa direttiva è simile alla precedente con la sola differenza che con **.set** noi possiamo definire all'interno del programma più **Etichette** con lo stesso nome, ma con associati valori diversi.

Il formato logico della direttiva **.set** è il seguente:

[etichetta] .set [operando]

[etichetta] = va inserito il nome della etichetta da associare al **valore numerico** definito nell'operando.

[operando] = va inserito il **valore numerico** o il risultato di una **espressione** da associare all'etichetta. Questo numero, che può essere espresso in **esadecimale**, **binario** o **decimale** non deve mai superare la capacità di **2 bytes**, cioè:

FFFFh in **esadecimale**
1111111111111111b in **binario**
65535 in **decimale**

1° Esempio

Per questo esempio abbiamo scelto due istruzioni **.set** con due diversi valori: **150** e **40**.

```
ritardo  .set   150
rout01   ldi    a,ritardo
         call   ritardo1

rout02   ldi    a,ritardo+15
         call   ritardo2
```

seguono righe del programma, quindi:

```
ritardo  .set   40
rout06   ldi    a,ritardo
         call   ritardo6
rout07   ldi    a,ritardo+40
         call   ritardo7
```

Nella prima istruzione all'etichetta **ritardo** viene associato il valore **150.**

In **rout01** viene caricato nell'accumulatore "**a**" il valore associato all'etichetta **ritardo** cioè **150.**

In **rout02** viene caricato nell'accumulatore "**a**" il risultato dell'espressione **ritardo+15** cioè **165.**

Proseguendo nella stesura del programma abbiamo previsto di aver bisogno di una nuova **direttiva .set** associata sempre all'etichetta **ritardo**, ma con un diverso valore che nel nostro esempio è **40.**

In **rout06** viene caricato nell'accumulatore "**a**" il valore associato all'etichetta **ritardo** cioè **40**

In **rout07** viene caricato nell'accumulatore "**a**" il risultato dell'espressione **ritardo+40** cioè **80.**

In qualche manuale sull'**ST6** abbiamo riscontrato un uso **errato** della direttiva **.set** dal quale vogliamo mettervi in guardia perché, non essendo segnalato dal compilatore, potrebbe mettere in un mare di guai un programmatore poco esperto.

Vi abbiamo più volte avvisato sul fatto che il **Compilatore Assembler** non esegue il programma, ma lo traduce solamente in codice **Intel.Hex**, sostituendo alle istruzioni le relative **opcode** e agli operandi i relativi valori o gli indirizzi di memoria, e controllando unicamente l'integrità di ogni singola istruzione.

Per spiegarvi gli **errori** in cui si può involontariamente incappare riscriviamo il nostro precedente esempio secondo i consigli dati in alcuni manuali e vi spieghiamo dove e perché sono **scorretti.**

```
ritardo  .set   150
         call   rout01
ritardo  .set   40
         call   rout06

rout01   ldi    a,ritardo
         call   ritardo1
rout02   ldi    a,ritardo+15
         call   ritardo2
         ret

rout06   ldi    a,ritardo
         call   ritardo6
rout07   ldi    a,ritardo+40
         call   ritardo7
         ret
```

Le istruzioni relative **rout01** e **rout02** sono state raggruppate in un unica subroutine chiamata

rout01, mentre le istruzioni relative a **rout06** e **rout07** sono raggruppate nella subroutine chiamata **rout06**.
Abbiamo quindi posto sotto le due direttive associate a **ritardo** le due rispettive **call**.
Da un punto di vista logico il programma sembra corretto, in realtà è completamente sbagliato.

Per capire il perché analizziamo ciò che avviene quando tentiamo di **compilarlo**.

Quando il Compilatore incontra la direttiva:

```
ritardo   .set      150
```

la esegue ed associa il valore **150** all'etichetta **ritardo**, quindi passa alla successiva istruzione:

```
call      rout01
```

Controlla che sia stata scritta correttamente (non **cals** o **catl** o altro) e che l'operando **rout01** sia un'etichetta di **Program Space** esistente.
Se tutto risulta **ok** la compila.

A questo punto però non salta alla subroutine **rout01** (come avviene in esecuzione), ma passa semplicemente alla istruzione successiva che nel nostro caso è la **direttiva**:

```
ritardo   .set      40
```

ed associa il valore **40** all'etichetta **ritardo**.
Ne consegue che **ritardo** non vale più **150** ma **40**.

E qui sta l'**errore**. Infatti quando il Compilatore, proseguendo in sequenza, arriva alle istruzioni della subroutine **rout01**:

```
rout01    ldi       a,ritardo
          call      ritardo1
rout02    ldi       a,ritardo+15
          call      ritardo2
          ret
```

non utilizza il valore **150** come dovrebbe, ma considera il valore **40** e, di conseguenza, carica nell'accumulatore "**a**" questo valore.
Poiché a questo viene **sommato 15**, avremo **55** (**40 + 15**) anziché **165** (**150 + 15**).

Quando in seguito arriverà alle istruzioni relative a **rout06**, le compilerà in modo corretto perché nell'accumulatore "**a**" verrà caricato il valore **40** e, come richiesto, avremo **40 + 15 = 55**.

OPZIONI del compilatore Assembler

Proseguiamo i nostri articoli esplicativi sul linguaggio di programmazione Assembler per i micro ST6 illustrando le Opzioni del compilatore.

Prima di proseguire con la spiegazione delle **direttive** dell'**assembler** per **ST6** dobbiamo soffermarci sulle **opzioni** del compilatore assembler.
Come abbiamo più volte ricordato, durante la compilazione il **compilatore assembler** genera sempre due file, entrambi con lo stesso nome del programma sorgente: uno con estensione **.HEX** in formato **intel** eseguibile e l'altro con estensione **.DSD** non eseguibile.
Il file con estensione **.dsd** è utile perché contiene tutte le informazioni di **Debug** che verranno poi utilizzate durante la simulazione del programma.

Il compilatore assembler è inoltre dotato di una serie di **opzioni** che, se inserite quando si lancia la compilazione, generano, oltre ai due già descritti, altri tipi di file che ci mettono a disposizione dati supplementari ed ulteriori funzioni di controllo sul programma sorgente.

Supponendo di dover compilare il programma sorgente chiamato **TESTER** e di voler aggiungere le opzioni **–L** e **–S** dobbiamo digitare:

ast6 –L –S TESTER.ASM

Innanzitutto, vi facciamo notare che davanti alla **lettera** che contraddistingue le **opzioni**, in questo caso **L** ed **S**, bisogna sempre inserire il segno **–**, distanziando inoltre le diverse opzioni da uno spazio. Questo è il solo modo corretto di scrittura.

Le opzioni del compilatore assembler sono:

–L –X –M –S –O –E –D –F –W

Premettiamo che utilizzando una qualsiasi di queste opzioni verrà generata una supplementare estensione, oltre alle due **.hex** e **.dsd** già esistenti. Rimanendo nell'esempio sopra riportato, noi avremo un file **.LIS** ed uno **.SYM**.

Di seguito vi spieghiamo a cosa servono le nove opzioni sopra riportate.

OPZIONE –L

Aggiungendo questa opzione il compilatore genera un file con lo stesso nome del programma sorgente, ma con estensione **.LIS** al cui interno viene memorizzato il listato completo del programma.

In fig.1 riportiamo un esempio del listato del file **tester.lis** generato dalla compilazione del programma **tester.asm**.
Sulla **sinistra** troviamo dei valori **numerici** e sulla parte **destra** le **istruzioni** del programma in formato simbolico.

Le istruzioni simboliche sono quelle che abbiamo scritto realizzando il programma, quindi non hanno bisogno di ulteriori spiegazioni.
E' invece importante chiarire il significato dei numeri che appaiono sulla sinistra.

Poiché ciò che diremo verrà successivamente ripreso ed approfondito, ci limitiamo ora a fornirvi una spiegazione molto condensata.

Per rendere più comprensibile la spiegazione, abbiamo aggiunto nella prima riga in alto di fig.1 una serie di **sigle** corrispondenti ai dati incolonnati.
Ovviamente queste sigle **non** appaiono mai nei listati.

Analizziamo ora la prima riga:

LIST = 479

Il numero **479** è il numero della riga del listato del programma e, generalmente, corrisponde alla riga del programma con estensione **.asm**.

STY = P00

Indica in quale numero di sezione/pagina di **Program Space** si trova "memorizzata" l'istruzione dopo la compilazione.
Nel nostro esempio l'istruzione verrà **memorizzata** alla **Pagina 0** di **Program Space**.
Normalmente le sezioni/pagine sono così siglate:

Pnn
Snn
Wnn

Pnn – La lettera **P** sta per **Program Page**, cioè **Pagina** di area di **Programma**, ed **nn** è il numero di pagina in cui si trova l'istruzione.
Questa pagina viene generata quando si compila in assembler e si vuole ottenere un programma eseguibile in formato **.hex**.

Normalmente una **Program Page** è di **2 kbytes** (**2048 bytes**) per i micro **ST6210 - ST6220** e di **4**

```
LIST  STY   SCOU  OPCODE    ST2   SCO2  NLEV  SNU   LABEL     INSTR    OPERAND          COMMENT

479   P00   02DB  C92C      P00   02DB        479             jp       ciclo1           ; salta
480                                           480
481                                           481   ;====================================
482                                           482   ;= DEFINIZIONE DI TABELLE IN PROGRAM S
483                                           483   ;====================================
484                                           484
485                                           485             .block   64-$%64          ;452 ta
486   P00   0300            P00   0300        486   masc01                              ;453 et
487                                           487             .ascii   " VOLT "         ;454 ca
488   P00   0306  7E        P00   0306        488             .byte    01111110b        ;
489   P00   0307  20        P00   0307        489             .byte    32,32,32         ;
490   P00   0308  20        P00   0308        489
491   P00   0309  20        P00   0309        489
492   P00   030A  7F        P00   030A        490             .byte    01111111b        ;
493   P00   030B            P00   030B        491   masc02                              ;455 et
494                                           492             .ascii   "        max5V"  ;456 ca
495                                           493
496                                           494             .block   64-$%64          ;457 ta
497   P00   0340            P00   0340        495   cdgramd                             ;458 et
498                                           496             .input   "TB_CGR02.ASM"   ;459 DI

--- SOURCE FILE : TB_CGR02.ASM ---

499                                     1     1     ;+-----------------------------------------
500                                     1     2     ;| TB_CGR02 Tabella dei caratteri per
501                                     1     3     ;+-----------------------------------------
502                                     1     4     ;
503   P00   0340  00        P00   0340  1     5               .byte    0,0,0,0          ;0
504   P00   0341  00        P00   0341  1     5
```

Fig.1 Esempio del file tester.lis generato dalla compilazione con l'opzione –L.

kbytes (**4096 bytes**) per i micro **ST6215- ST6225**, e corrisponde sempre al numero **P00**.
Per i micro da **4 Kbytes** esiste la possibilità di suddividere la **Program Page** in due pagine, ognuna di **2 Kbytes**, inserendo la direttiva **.pp_on** nel programma.
In questo caso il compilatore divide l'area in due sezioni da **2kbytes** cadauna (**2048 bytes**) ed assegna il numero partendo da **0**, quindi:

P00 = parte da **0** e finisce a **7FFh**
P01 = parte da **800h** e finisce a **FFFh**

Nel nostro esempio l'istruzione, una volta compilata, partirà dall'indirizzo di **program space** che appare nella terza colonna, sotto la scritta **Scou**, cioè da **02DBh**, inserita nella pagina **P00**.

Snn – La lettera **S** sta per **Program Section**, cioè **Sezione** di area di **Programma**, ed **nn** è il numero di sezione in cui si trova l'istruzione.
La sezione viene generata quando si compila in **assembler** un programma **rilocabile**, vale a dire **non eseguibile**, in formato **.obj** (vedi opzione **–O**).
In questo caso il programma sorgente dovrà contenere la direttiva **.section** e opzionalmente la direttiva **.pp_on**.
Sono previste **33** sezioni di Program Space a partire dalla sezione **0**, pertanto inserendo nel programma **.section 1**, poi **.section 2** ecc. il **compilatore** suddivide l'area di **program space** in **1** oppure **2** ecc. sezioni.
La **33°** sezione, che corrisponde alla direttiva **.section 32**, serve **solo** per inserire le istruzioni inerenti alla gestione dei **Vettori di Interrupt**.
Normalmente una **sezione** di Program space è di **2 kbytes** (**2048 bytes**) per i micro **ST6210 - ST6220** e di **4 kbytes** (**4096 bytes**) per i micro **ST6215 - ST6225**.
Per i micro da **4 Kbytes** esiste la possibilità di suddividere la **Program space** in due **sezioni** di **2 Kbytes** inserendo la direttiva **.pp_on** nel programma.

Wnn – Significa **Window Section Number** e viene generata quando si compila un programma che contiene la direttiva **.window/.windowend**.
Serve quando si utilizza il **Linker** per assemblare più programmi rilocabili (**.obj**) che contengano ognuno delle aree di **dati** definiti in **Program Space** (con **.byte .ascii** ecc.) e che utilizzino quindi la **Data Rom Window**.

SCOU = 02DB

Indica l'indirizzo di **Program Space** in cui l'istruzione viene memorizzata dopo la **compilazione**.
Se abbiamo suddiviso il programma in **Program Section** (vedi Snn) o in **Window Section** (vedi Wnn), questo indirizzo corrisponderà all'indirizzo di memoria **relativo** alla **sezione** o alla **finestra**. Se abbiano suddiviso il programma in **Page Section**, corrisponderà all'indirizzo **assoluto** di memoria del microprocessore.
Ad esempio, se nel listato leggessimo:

S01 0034h

significa che l'istruzione relativa all'indirizzo **0034h** si trova nella **Sezione 01** di **Program Space**.

Se nel listato del nostro programma leggessimo:

P00 0034h

significa che l'istruzione relativa all'indirizzo **0034h** si trova nella **Pagina 0** di **Program Space**.

OPCODE = C92C

Il numero **C92C** è la codifica **esadecimale** dell'istruzione **jp ciclo1** dopo la compilazione del programma **tester.hex**.

Sulla rivista **N.185** trovate l'elenco completo di tutte le istruzioni dell'assembler con le relative **OPCODE** e semplici istruzioni per decodificarle.

ST2 = P00

Come avrete notato, questo numero è equivalente a quanto riportato sotto la sigla **STY**, per cui rimandiamo a quanto già spiegato.

SCO2 = 02DB

Questo numero è equivalente a quanto riportato sotto la sigla **SCOU** ed anche in questo caso rimandiamo a quanto già detto.

NLEV SNU

Sotto la colonna **NLEV** il compilatore inserisce un valore che segnala il livello che ha l'istruzione che sta compilando. Se non c'è **nessun** numero, significa che l'istruzione fa parte del programma principale (nel nostro esempio **tester.asm**).
Se c'è il numero **1** significa che l'istruzione fa parte di un **programma** o di un **modulo** che a sua volta viene inserito in fase di compilazione nel programma principale.

Se c'è il numero **2** significa che l'istruzione fa parte di un **programma** o di un **modulo** che a sua vol-

ta viene inserito in un altro **programma** o **modulo** che in fase di compilazione viene inserito nel programma principale.

Sotto la colonna **SNU** c'è il numero **479**, che corrisponde al numero di riga che ha l'istruzione nel file **tester.asm**.

A questo proposito non è inutile ricordare che nella colonna **List** è riportato il numero della riga del listato, cioè del file con estensione **.LIS**, mentre sotto la colonna **SNU** il numero della riga che ha l'istruzione nel programma sorgente (**.ASM**).

Questi due numeri non sempre corrispondono: ad esempio le righe **489 - 490 - 491** della colonna **List** corrispondono alle righe **489 - 489 - 489** della colonna **SNU**.

Il perché è presto detto: l'istruzione **.byte 32,32,32** è definita nel programma sorgente **.asm** alla riga **489**, ma siccome definisce 3 bytes, il compilatore prosegue nella numerazione per altri due numeri.

Per questo motivo la riga **492** della colonna **List** corrisponde alla riga **490** della colonna **SNU**.

Spostatevi ora in basso, alla riga **498** della colonna **List**, che corrisponde alla direttiva:

.input "TB_CGR02.ASM"

Quando il compilatore trova la direttiva **.input**, carica il file riportato nelle **virgolette** (nel nostro esempio **"TB_CGR02.ASM"**) e lo assembla inserendolo all'interno del programma principale e segnalandolo nel listato con la dicitura:

SOURCE FILE : TB_CGR02.ASM

Ora ignoriamo le righe **499** fino alla **502**, che sono dei **commenti**, e passiamo direttamente alla riga **503** della colonna **List** relativa alla direttiva:

.byte 0,0,0,0

A proposito di questa direttiva è importante rilevare innanzitutto che sotto la colonna **NLEV** c'è il numero **1**, quindi questa istruzione **non** è contenuta nel programma principale **TESTER.ASM**, ma nel file **TB_CGR02.ASM**.

Inoltre sotto la colonna **SNU** troviamo il numero **5**, che ci dice che la direttiva **.byte 0,0,0,0** è posizionata nella riga **5** del file **TB_CGR02.ASM**.

Tutte le istruzioni contraddistinte sotto la colonna **NLEV** con il numero **1** fanno parte del file **TB_CGR02.ASM**, quindi nel caso volessimo modificarle **non** dovremmo ricercarle nel programma principale **TESTER.ASM**.

OPZIONE –X

Il compilatore genera un file con lo stesso nome del programma sorgente, ma con estensione **.X**, contenente l'elenco di tutte le **etichette** e di tutte le **variabili** del programma e con l'indicazione di tutte le righe in cui queste vengono utilizzate.

Ad esempio nel file **tester.x** possiamo trovare:

drw 43* 254 300
dsend 164 166 168 170 172 185 191 202*
dvolt 67* 68 375 386 388 404 445

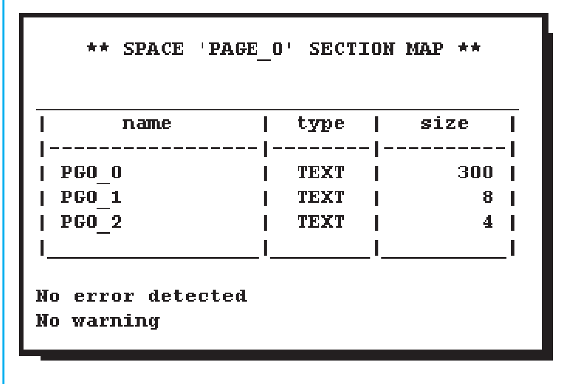
A sinistra è riportato l'elenco in ordine alfabetico di **variabili**, **costanti simboliche** ed **etichette** utilizzate nel programma ed in corrispondenza di ogni voce dell'elenco abbiamo una serie di numeri, uno dei quali contraddistinto da un asterisco.

I **numeri** corrispondono alle righe del programma **.asm** in cui **variabili - costanti - etichette** vengono utilizzate e sono quelli che poi appaiono sotto la colonna **LIST** del file **tester.lis** (vedi fig.1).

Il numero seguito da un asterisco (*) ci segnala la riga del programma sorgente in cui variabili, costanti ed etichette vengono **definite**.

OPZIONE –M

Genera una **mappa** della memoria del programma compilato e la riporta in coda al listato nel file con estensione **.LIS** (vedi fig.2).
Come avrete già intuito, quando si usa questa opzione deve esserci anche l'opzione **–L**, altrimenti il compilatore segnala **errore**.



```
   ** SPACE 'PAGE_0' SECTION MAP **

 ______________________________________________
| name             | type   | size     |
|------------------|--------|----------|
| PG0_0            | TEXT   |      300 |
| PG0_1            | TEXT   |        8 |
| PG0_2            | TEXT   |        4 |
|__________________|________|__________|

No error detected
No warning
```

Fig.2 Mappa della memoria nel file tester.lis.

La scrittura corretta è:

ast6 –L –M TESTER. ASM

Nella parte superiore della mappa appare questa scritta:

**** SPACE 'PAGE_0' SECTION MAP ****

che significa che la mappa stampata riguarda la **Program Page 0** (**P00**), di cui abbiamo già parlato nel paragrafo dedicato all'opzione **–L**.
Nella colonna **name** della mappa vengono riportate **3** aree di **Program Page 0** con a fianco il tipo di istruzioni (**text**) e l'aera occupata in bytes espressa in **esadecimale**.
La **Program Page 0** è suddivisa in tre aree di memoria non consecutive, perché all'interno del programma **tester** abbiamo utilizzato la direttiva **.org**, tre volte in punti non consecutivi, per posizionarci all'interno della **Program Space**.
Più precisamente:

PG0_0 è un area di **Program Page 0** che contiene **300h** bytes di istruzioni in formato eseguibile che corrispondono a **768** byte in **decimali**,
PG0_1 è un area di **Program Page 0** che contiene **08h** bytes di istruzioni in formato eseguibile che corrispondono a **8** byte in **decimale**,
PG0_2 è un area di **Program Page 0** che contiene **04h** bytes di istruzioni in formato eseguibile che corrispondono a **4** byte in **decimale.**

Sommando i numeri **decimali** abbiamo

768 + 8 + 4 = 780 in decimale

Convertendo il risultato in **esadecimale** otteniamo **30Ch**, che è lo spazio occupato dalle sole **istruzioni** del programma.
Lo spazio occupato dall'intero programma **tester.asm** risulterà maggiore, perché queste tre aree non sono consecutive l'una all'altra.

OPZIONE –S

Genera un file con lo stesso nome del programma, ma con estensione **.SYM**, contenente un elenco delle **etichette** definite in **Program Space** e delle **costanti simboliche** utilizzate nel programma.
Digitando:

ast6 –S TESTER.ASM

viene generato il file **tester.sym**.

Di seguito vi riportiamo qualche riga di esempio del file **tester.sym**:

```
serout    :  EQU   00966H P
outstart  :  EQU   00090H C
addr_10   :  EQU   00915H P
eti       :  EQU   00006H C
asci_r    :  EQU   00072H C
asci_w    :  EQU   00077H C
STOPBITS  :  EQU   00001H C
```

Analizziamo la prima riga:

serout : EQU 00966H P

La lettera **P** indica che **serout** è un etichetta definita in **Program Space** ed il numero che si trova dopo **EQU**, cioè **00966H**, è il suo l'indirizzo.

Nella seconda riga:

outstart: EQU 00090H C

la lettera **C** indica che **outstart** è una **costante** simbolica definita nel programma tramite l'utilizzo della direttiva **.set** o **.equ** ed il valore **00090H**, indicato dopo **EQU**, è il valore a lei associato.
Il file **tester.sym** è di vitale importanza per la fase di **Debug** del programma, perché viene utilizzato, assieme al file **.dsd**, dal software di simulazione per rendere "leggibile" il programma da testare.

```
Disassembler
    Add Opcode       Label      Mnemonic
    8B8 0D D8 FE                ldi wdog,FEH
    8BB 03 C0 07                jrr 0,port_a,08C5H
    8BE 13 C1 09                jrs 0,port_b,08CAH
    8C1 1B C1                   set 0,port_b
    8C3 A9 8C                   jp 08CAH
 => 8C5 03 C1 02                jrr 0,port_b,08CAH
    8C8 0B C1                   res 0,port_b
    8CA 0D D8 FE                ldi wdog,FEH
    8CD 83 C0 07                jrr 1,port_a,08D7H
    8D0 93 C1 09                jrs 1,port_b,08DCH
```

Fig.3 Programma compilato senza l'opzione –S.

```
Disassembler
    Add Opcode       Label      Mnemonic
    8B8 0D D8 FE     ripeti     ldi wdog,FEH
    8BB 03 C0 07                jrr 0,port_a,main00
    8BE 13 C1 09                jrs 0,port_b,mains1
    8C1 1B C1                   set 0,port_b
    8C3 A9 8C                   jp mains1
 => 8C5 03 C1 02     main00     jrr 0,port_b,mains1
    8C8 0B C1                   res 0,port_b
    8CA 0D D8 FE     mains1     ldi wdog,FEH
    8CD 83 C0 07                jrr 1,port_a,main01
    8D0 93 C1 09                jrs 1,port_b,mains2
```

Fig.4 Programma compilato con l'opzione –S.

Se in fase di simulazione venisse caricato il solo programma in formato eseguibile, cioè il programma **tester.hex** che contiene le sole **opcode**, anche i programmatori molto esperti avrebbero parecchie difficoltà di lettura.
Utilizzando il file **tester.sym**, il software di simulazione trasforma le **opcode** eseguibili in istruzioni leggibili, rendendo la fase di **Debug** molto più semplice.

In fig.3 potete vedere l'esempio di un programma di simulazione durante la fase di **Debug** del programma **tester.asm** che è stato compilato senza inserire l'opzione **–S**.
In fig.4 riportiamo lo stesso programma compilato con l'opzione **–S**.

Come potete vedere, mancando il file **tester.sym** in fig.3 non appaiono tutte le etichette di salto che sono invece presenti nella fig.4.

OPZIONE –O

Questa opzione serve per generare un programma **rilocabile** non eseguibile in formato **.OBJ**.
Usando questa opzione non vengono generati i file **.hex**, **.dsd** e **.sym**.

Questa opzione si utilizza quando si devono compilare programmi contenenti delle **macroroutine**, che possiamo **unire** in seguito ad altri programmi tramite il **Linker** per ottenere un **unico** programma eseguibile.
In pratica creiamo delle **librerie** utilizzabili ogniqualvolta ne avremo bisogno.

Ma che cosa significa programma **rilocabile**?
Quando si assembla un programma, il compilatore assegna ad ogni **variabile** un indirizzo di **Data Space** e ad ogni **istruzione** un indirizzo di **Program Space**.

Se si è compilato un programma **eseguibile**, che come sappiamo genera i due file **.hex** e **.dsd**, gli indirizzi di Program Space e Data Space assegnati dal compilatore si posizionano all'interno del microprocessore esattamente nel punto di memoria indicato (indirizzamento **assoluto**).

Se si è compilato un programma **rilocabile** tramite l'opzione **–O**, si ottiene un file **.obj** e alle sue **variabili** ed **istruzioni** viene assegnato un indirizzo di memoria (indirizzamento **relativo** a questo **.obj**).
Unendo, tramite il **Linker**, uno o più programmi **rilocabili** otteniamo un file **eseguibile** che può es-

```
LIST STY SCOU OPCODE ST2 SCO2 NLEV SNU    LABEL INSTR    OPERAND
 460                                48   ;****************************
 461                                49          .section 1
 462 S01 0000        S01 0000       50   serin
 463 S01 0000 0D8098 S01 0000       51          ldi      x,in_start
 464 S01 0003        S01 0003       52   get
 465 S01 0003 0DD8FF S01 0003       53          ldi      wdr,0ffh
 466 S01 0006 A101   S01 0006       54          call     get_byte
 467 S01 0008 CBD4   S01 0008       55          res      psi,tscr
```

Fig.5 Il programma rilocabile sub_in.obj generato dall'opzione –O.

```
LIST STY SCOU OPCODE ST2 SCO2 NLEV SNU    LABEL INSTR    OPERAND
 460                                48   ;*************************
 461                                49          .section 1
 462 P01 09CA        S01 0000       50   serin
 463 P01 09CA 0D8098 S01 0000       51          ldi      x,in_start
 464 P01 09CD        S01 0003       52   get
 465 P01 09CD 0DD8FF S01 0003       53          ldi      wdr,0ffh
 466 P01 09D0 419E   S01 0006       54          call     get_byte
 467 P01 09D2 CBD4   S01 0008       55          res      psi,tscr
```

Fig.6 Lo stesso programma di fig.5 dopo l'esecuzione Linker.

sere memorizzato nel micro, ma nell'unione il **Linker** assegnerà ad ogni **istruzione** ed ad ogni **variabile** un nuovo indirizzo di **memoria**.
Come esempio in fig.5 riportiamo alcune istruzioni del listato del programma **sub_in.asm** compilato con l'opzione **–O**.

Come potete vedere alla riga **463** troviamo:

S01 0000 0D8098 S01 0000 51

corrispondente all'istruzione:

ldi x,in_start

S01 0000 è l'indirizzo di **Program Section** dell'istruzione,
0D8098 è l'opcode eseguibile dell'istruzione,
51 è il numero di riga di questa istruzione nel programma sorgente.
In pratica l'istruzione **ldi x,in_start** viene memorizzata nel byte **0** di **Program Section 1**.
Se tramite il **Linker** uniamo questo programma ad un altro programma **.obj**, ad esempio **reg_r.obj**, otteniamo un programma **eseguibile** al quale va assegnato un nome, ad esempio **pluto.hex**.
Il comando di **Linker** utilizzato per eseguire questa unione è il seguente:

Lst6 -l -O PLUTO.HEX REG_R.OBJ SUB_IN.OBJ

A seguito di questa unione viene generato il programma eseguibile **pluto.hex**, composto dai due programmi **reg_r.obj** e **sub_in.obj**.

In fig.6 riportiamo lo stesso listato di **sub_in.obj** dopo l'esecuzione **Linker**.

Come potete vedere, alla riga **463** troviamo ora:

P01 09CA 0D8098 S01 0000 51

P01 09CA è l'indirizzo di **Program Page** dove viene ora **definitivamente** memorizzata l'istruzione e **0D8098** è l'opcode eseguibile dell'istruzione.

Una volta linkata, questa istruzione risulta **memorizzata** definitivamente all'indirizzo **09CAh** di **Program Space** del programma **pluto.hex**.

Vi abbiamo "dimostrato" che unendo i due file con estensione **.obj** vengono modificati gli indirizzi delle **variabili** e delle **opcode**.
Se volete un'ulteriore conferma, confrontate il valore che si trova sotto la colonna **OPCODE** in corrispondenza dell'istruzione **call get_byte** di fig.5 con il rispettivo valore riportato in fig.6.

Il valore dell'opcode che prima del Linker era **A101h** è diventato **419Eh**.
Infatti l'etichetta **get_byte** che prima del linker si trovava all'indirizzo **relativo** di **Program Section 014h**, dopo il Linker è stata memorizzata all'indirizzo di Program Space **9E4h**.

OPZIONE –E

Se compilando il file **TESTER.ASM** digitiamo:

ast6 –E TESTER.ASM

viene generato un file con lo stesso nome del programma, ma con estensione **.ERR**.

Questo file contiene l'elenco di tutti gli **errori** riscontrati durante la compilazione assembler e riporta sul monitor solo l'indicazione (vedi fig.7):

nnn **error detected**
No object created

Il file con gli errori riscontrarti può essere visualizzato e stampato con un qualsiasi Editor.

Questa opzione ci offre molti vantaggi, perché se nel programma vi sono molti errori, è sicuramente molto utile averne a disposizione una **stampa**, anziché dover consultare su video i messaggi di errore con il rischio che qualcuno sfugga.

In fig.8 riportiamo il listato ottenuto con un normale file **tester.err**.
Con questo listato ci sarà possibile modificare e correggere le istruzioni segnalate in modo da ottenere una compilazione corretta.

OPZIONE –D

Se non diversamente specificato, quando si compila un programma ogni byte di area **Program Space** non utilizzata viene riempito dal compilatore con il valore **OFFh**.
Utilizzando l'opzione **–D** seguita da un valore numerico possiamo riempire i byte non utilizzati con un determinato valore.

```
C:\ST6>ast6 -s -l -e -f tester.asm
ST6 MACRO-ASSEMBLER version 4.00  - August 1992
Execution time: 0 second(s)
6 errors detected
 One warning
No object created
```

Fig.7 Segnalazione a video del numero di errori.

Ad esempio, scrivendo:

ast6 –D09 TESTER.ASM

la parte di **Program Space** non utilizzata viene riempita con il valore **09.**
All'atto pratico questa opzione può servire come **chiave** di controllo.
Tenete presente che il numero riportato dopo la **D** deve essere **esadecimale**, diversamente verrà segnalato **errore**. Ad esempio, per inserire il numero **174** dobbiamo digitare **–DAE**.

OPZIONE –F

Inserendo questa opzione, se si verificano errori nella **compilazione**, nel messaggio di errore viene visualizzato l'intero **Pathname** del file contenente il programma sorgente.
Per inserire l'opzione **–F** basta digitare:

ast6 –F TESTER.ASM

Sapere il Pathname completo del programma che ha dato errore è utile nel caso esistano più versioni dello stesso memorizzate in directory diverse o su floppy come copie di sicurezza.

OPZIONE –W

In fase di compilazione possono essere segnalati dal compilatore due diversi tipi di **errori**:

WARNING oppure **ERROR**

La scritta **ERROR** indica che l'errore è molto **grave**, tale da impedire la compilazione in assembler del programma. In questo caso è necessario intervenire nel programma e correggere gli errori segnalati prima di **ricompilare** il programma.

La scritta **WARNING** indica che l'errore riscontrato **non** è grave, quindi la compilazione in assembler riesce a proseguire.

Nel segnalare questo tipo di errore il compilatore gli assegna un numero, **0**, **1** o **2** seguito dal simbolo **>**, che rappresenta la tipologia dell'errore.
E' comunque consigliabile andare a verificare, almeno la prima volta che si compila il programma, anche questo tipo di errore, perché potrebbe compromettere la corretta esecuzione del programma.
In fase di compilazione è possibile comunicare al compilatore quale tipologia di errore **warning** vogliamo che sia segnalata.
Ad esempio se scriviamo:

ast6 –W1 TESTER.ASM

verranno segnalati solo gli errori di tipologia **0** ed **1**, ma non di tipologia **2**.
Nella terza riga di fig.8 è segnalato un errore tipo **warning** con l'indicazione **1>** e la spiegazione dell'errore riscontrato.
Potendo differenziare tre diverse tipologie di errore **warning**, possiamo compilare più volte il programma sorgente dando ogni volta l'opzione **–W** con un diverso numero.
In questo modo potremo controllare prima tutti gli errori **–W0**, poi i **–W1** ed infine i **–W2**.

CONCLUSIONE

Lanciando la compilazione potete caricare più opzioni in una volta, ma tenete presente che alcune opzioni non sono compatibili tra loro.

– Se usate l'opzione **–O** non dovrete usare la **–D**, comunque se la inserite verrà ignorata.

– Se usate l'opzione **–W** non dovrete usare la **–S**, comunque se la inserite verrà ignorata.

– Se usate l'opzione **–D** non potete inserire nel programma la direttiva **.pp_on**.

– Se usate l'opzione **–M** dovrete usare sempre anche la **–L**; se non la inserite il compilatore segnalerà **errore**.

```
Error C:\ST6\tester.asm 60:(77)operand may not reference program space symb
Error C:\ST6\tester.asm 110:(20)operand expected: 3-bit number
Warning C:\ST6\tester.asm 113:(91) 1> simbol declared external but unused
Error  C:\ST6\tester.asm 123:(-1)  syntax error
Error C:\ST6\tester.asm 123:(110)data addresses must be in the range [0..0ffh]
Error C:\ST6\tester.asm 126:(67)undefined macro:  sut
Error  C:\ST6\tester.asm 113:(106)undefined symbol: port_d
```

Fig.8 Esempio di come vengono segnalati gli errori con le opzioni –E e –W.

Le memorie RAM-EEPROM

Continuiamo anche in questo numero le nostre lezioni teorico-pratiche sulla programmazione dei micro ST6. Infatti, contrariamente a quanto supponevamo, i nostri lettori, unitamente a molti Istituti professionali e tecnici e a parecchie piccole e medie Industrie, li aspettano con impazienza perché li trovano molto istruttivi e interessanti.

Dopo l'ultimo articolo dedicato alle **opzioni** del linguaggio Assembler, avremmo dovuto continuare con le lezioni sulle **direttive** per poi arrivare al **linker** e completare così la conoscenza di questo linguaggio di programmazione.
Ma per venire incontro ai molti lettori che ci hanno scritto per avere spiegazioni più dettagliate sulle memorie **Ram-EEprom** dei micro **ST6260** e **ST6265** (vedi nella rivista **N.192** l'articolo "**Bus** per testare le funzioni **Pwm** e **EEprom**"), in questo articolo tratteremo queste memorie.

Prima di entrare nell'argomento vogliamo parlarvi dei **registri** chiamati **Write Only** e **Write Only Bits**, perché se gestiti in maniera **non corretta** possono provocare anomalie anche gravi durante l'esecuzione dei programmi.

I REGISTRI

Con il termine generico di **registri** si intende una serie di **indirizzi** di memoria Ram Data Space, che il micro utilizza per svolgere particolari funzioni.

Per ogni diversa funzione è previsto un apposito **registro**, che si trova in una ben determinata locazione di memoria Data Space. Per facilitare la stesura del programma, ad ogni locazione di memoria viene associata un'**etichetta**.
Per quanto riguarda i micro della famiglia **ST6** da noi finora presi in esame, cioè gli **ST6210-15**, **ST6220-25** e gli **ST6260-65**, nelle **Tabelle N.1** e **N.2** elenchiamo le definizioni di tutti i **registri** ed il loro **indirizzo** di Data Space. Per completezza abbiamo riportato a fianco di ogni registro l'**etichetta** da noi utilizzata nei nostri programmi.

Registri WRITE ONLY

Nelle lezioni sul linguaggio di programmazione per gli **ST6** abbiamo più volte ripetuto che le istruzioni **SET - RES - JRS - JRR** consentono di accedere al **singolo bit** di una variabile o di un registro per settarlo a **0** o a **1** o per interrogare il suo stato.
Se queste istruzioni vengono utilizzate per modificare i singoli bits dei registri **W**rite **O**nly possono

provocare malfunzionamenti del programma, ai quali è difficile risalire.
Infatti, essendo istruzioni formalmente corrette, in fase di compilazione il Compilatore Assembler non segnala **nessuna** anomalia o **errore**.

Non solo, anche testando il programma con i più diffusi software di **simulazione** non viene segnalata nessuna **anomalia**, perché il settaggio del **singolo** bit viene accettato ed eseguito correttamente. Quando però inseriamo il micro sulla sua scheda di utilizzo, il circuito **non** funziona e a questo punto diventa difficile capire perché il micro **non** esegue le istruzioni per cui è stato programmato.

TABELLA N.1

Locazioni dei **registri** dei micro **ST6210-15-20-25**

Data Ram area	etichetta	locazione
X register	x	080h
Y register	y	081h
V register	v	082h
W register	w	083h
port A data register	port_a	0C0h
port B data register	port_b	0C1h
port C data register	port_c	0C2h
port A direction register	pdir_a	0C4h
port B direction register	pdir_b	0C5h
port C direction register	pdir_c	0C6h
Interrupt Option register	ior	0C8h
Data Rom Window register	drw	0C9h
port A option register	pop_a	0CCh
port B option register	pop_b	0CDh
port C option register	pop_c	0CEh
A/D data register	addr	0D0h
A/D control register	adcr	0D1h
Timer Prescaler register	psc	0D2h
Timer counter register	tcr	0D3h
Timer status control register	tscr	0D4h
Watchdog register	wdog	0D8h
Accumulator	a	0FFh

Nota: ricordiamo che nei micro **ST6210-20** non è presente le **porta C**, di conseguenza i registri **port_C - pdir_C** e **pop_C** non sono utilizzabili.
I registri segnalati in **negativo** sono **W**rite **O**nly **R**egister, cioè registri di sola scrittura.

TABELLA N.2

Locazione dei **registri** nei micro **ST6260-65**

Data Ram area	etichetta	locazione
X register	x	080h
Y register	y	081h
V register	v	082h
W register	w	083h
port A data register	port_a	0C0h
port B data register	port_b	0C1h
port C data register	port_c	0C2h
port A direction register	pdir_a	0C4h
port B direction register	pdir_b	0C5h
port C direction register	pdir_c	0C6h
Interrupt Option register	ior	0C8h
Data Rom Window register	drw	0C9h
port A option register	pop_a	0CCh
port B option register	pop_b	0CDh
port C option register	pop_c	0CEh
A/D data register	addr	0D0h
A/D control register	adcr	0D1h
Timer Prescaler register	psc	0D2h
Timer counter register	tcr	0D3h
Timer status control register	tscr	0D4h
AR timer mode control register 1	armc	0D5h
AR timer status/control register 2	ars1	0D6h
AR timer load register	ars2	0D7h
Watchdog register	wdog	0D8h
AR timer reload/capture register	arrc	0D9h
AR timer compare register	arcp	0DAh
AR timer load register	arlr	0DBh
Oscillator control register	ocr	0DCh
Miscellaneous	mis	0DDh
SPI data register	spda	0E0h
SPI divider register	spdv	0E1h
SPI mode register	spmc	0E2h
Data Ram/EEprom register	eedbr	0E8h
EEprom control register	eecr	0EAh
Accumulator	a	0FFh

Nota: i registri segnalati in **negativo**, cioè **ior - drw - ocr - eedbr**, sono **W**rite **O**nly **R**egister.

Nessuno ha mai esplicitamente messo in evidenza che ai registri **Write Only** si può accedere **esclusivamente** con istruzioni che **settano** o **resettano** tutti gli **8 bits** contemporaneamente, vale a dire con le istruzioni tipo **LD - LDI - CLR** ecc.

I registri **Write Only** comuni a tutti i micro della serie **ST6** sono:

Interrupt option register	**0C8h (ior)**
Data rom window register	**0C9h (dwr)**

Nei micro **ST6260-65** abbiamo in più:

Oscillator control register	**0DCh (ocr)**
Data ram/eeprom register	**0E8h (eedbr)**

Quindi anche il **Data Ram-EEprom register** è un registro di sola scrittura.

Per chiarire come vanno utilizzati questi registri facciamo un esempio molto semplice, ma che ci sembra appropriato.
Poniamo il caso di voler scrivere la parola INGRESSO, ma per errore scriviamo INGRASSO.
Se fossimo in un programma di **videoscrittura**, per correggere questo errore basterebbe sostituire la lettera **A** con la lettera **E** e la parola sarebbe **formalmente** corretta.

Per i **registri** di sola scrittura questo non è possibile, perché non possiamo accedere al **singolo bit**, ma solo a tutti gli **8 bits** contemporaneamente.
Nel nostro esempio dovremmo riscrivere l'intera parola **INGRESSO** e non correggere la **A** con la **E**.

Esempio per Interrupt Option Register

Fig.1 Formato del registro IOR

7	6	5	4	3	2	1	0
	LES	ESB	GEN				

Nella quasi totalità dei programmi, gli interrupt del micro sono inizialmente caricati a zero.
Per **disattivare** tutti gli interrupt l'istruzione corretta è la seguente:

```
ldi     ior,00000000b
```

Per attivare l'**interrupt GEN**, cioè il **bit 4** del **registro ior**, verrebbe logico scrivere:

```
set     4,ior
```

Con questa istruzione il programma **non** funziona.
Essendo il registro di Interrupt un registro **Write Only** dobbiamo riscrivere tutti gli **8 bits** utilizzando l'istruzione **LDI**:

```
ldi     ior,00010000b
```

oppure possiamo scrivere:

```
ldi     ior,16
```

in quanto il numero **binario 00010000** corrisponde al numero **decimale 16.** Se anziché scrivere l'istruzione in **binario** o in **decimale** volessimo scriverla in **esadecimale** dovremmo modificarla in:

```
ldi     ior,010h
```

Esempio per Data Ram/EEprom Register

Fig.2 Formato del registro EEDBR

7	6	5	4	3	2	1	0
			EEDBR4			EEDBR1	EEDBR0

Per **attivare una** delle 3 **pagine** aggiuntive di memoria dei micro **ST6260-65**, dobbiamo configurare il registro **EEDBR** come segue:

bit 0: settare a **1** per attivare la **Eeprom Page 0**
bit 1: settare a **1** per attivare la **Eeprom Page 1**
bit 4: settare a **1** per attivare la **Ram Page 2**

In altre parole, essendo anche questo un registro **Write Only**, per attivare la Eeprom Page1 **non** possiamo scrivere:

```
set     1,eedbr
```

ma dobbiamo invece scrivere:

```
ldi     eedbr,00000010b
```

Volendo questa istruzione in **decimale** scriveremo:

```
ldi     eedbr,2
```

Ancora, per disattivare questa **Eeprom Page 1** molti commettono l'**errore** di scrivere:

```
res     1,eedbr
```

invece occorre **necessariamente** scrivere:

```
ldi     eedbr,0
```

In fase di programmazione dovrete sempre ricordarvi di questi **piccoli particolari** per eliminare i problemi che ora potreste riscontrare.

Registri WRITE ONLY BITS

Oltre ai registri **Write Only**, esistono tre registri che hanno solamente alcuni **bits** Write Only.
Anche questi, se utilizzati in maniera impropria, possono creare malfunzionamenti nel programma.

Il registro **Write Only Bits** comune ai micro ST6 è:

A/D converter control register 0D1h (adcr)

Nei micro **ST6260-65** abbiamo in più:

EEprom control register 0EAh (eecr)
AR timer mode control register 0D5h (armc)

In questi registri ci sono dei bits che **non** possiamo mai interrogare con istruzioni tipo **JRS** e **JRR**, perché, qualunque sia il loro stato logico, ritornano sempre il valore **0**.

Fig.3 Formato del registro ADCR

7	6	5	4	3	2	1	0
EAI	EOC	STA	PDS	D3	D2	D1	D0

In questo registro il **Write Only Bit** è il **5**, che troviamo siglato **STA**.
Quando è settato a **1** indica l'inizio della conversione Analogico/Digitale. Se, dopo aver attivato l'A/D Converter con l'istruzione:

```
ldi     adcr,00110000b
```

scriviamo:

```
jrs     5,adcr,start_c
```

il programma non salterà **mai** a **start_c**, perché, essendo il **bit 5** di sola scrittura, e non di lettura, non riesce a vederlo settato e quindi il risultato della interrogazione sarà sempre **0**.

Fig.4 Formato del registro EECR

7	6	5	4	3	2	1	0
D7	EEOFF	D5	D4	EEPAR1	EEPAR2	EEBUSY	EEENA

Nel registro **EECR** vi sono tre bits **Write Only**:

EEENA bit **0**
EEPAR1 bit **3**
EEOFF bit **6**

Naturalmente anche per questi bits vale quanto detto sopra. Nel prossimo paragrafo spiegheremo l'utilizzo completo di questo registro.

Fig.5 Formato del registro ARMC

7	6	5	4	3	2	1	0
TLCD	TEN	PWMODE	EE	CPIE	OVIE	ARMC1	ARMC011

Nel registro **ARMC** c'è un solo bit **Write Only** e precisamente il bit **7** siglato **TLCD**.
Quando questo bit è settato a 1 ricarica il contatore del timer con il valore di base.
Questo particolare registro è stato trattato nei programmi di esempio del **PWM**, nella Rivista **N.192**.
Nel dischetto **DF.1325** da noi fornito troverete una serie di semplici programmi corredati di **note** coi quali sarà semplice capire come usare il **PWM**.

Le MEMORIE EEprom e RAM addizionale

Quando abbiamo presentato il programmatore **LX.1325** per i micro della serie **ST626065** (vedi rivista N.192), ci siamo anche preoccupati di spiegarvi con semplici esempi la logica del **PWM** e della memoria **EEprom**.
Proprio per questo motivo nel dischetto allegato al kit (siglato **DF.1325**) abbiamo fornito una serie di programmi elementari, corredati di note a fianco di ogni istruzione, per rendere più comprensibili l'utilizzo del **PWM** e della memoria **EEPROM**.
Data la novità dell'argomento sono giunte in redazione richieste di approfondimento soprattutto sull'utilizzo e la gestione della memoria EEprom.
Per venire incontro a questa esigenza, affrontiamo in questo paragrafo le memorie **EEprom** e **RAM addizionale** dei micro **ST6260** e **ST6265**.
Nella fig.6 è riportato il diagramma a blocchi dei micro **ST6210-15-20-25**, mentre nella fig.7 potete vedere quello relativo ai micro **ST6260-65**.
Confrontando le due figure potete notare che i micro **ST6260-65**, riportati in fig.7, possiedono in più la funzione **Autoreload Timer**, un **SPI** (**S**erial **Pe**ripheral **I**nterface), una **Data Ram** di **128** bytes ed una **Data EEprom** di **128** bytes.
Specifichiamo subito, per evitare equivoci, che la **DATA RAM** dei micro **ST6260-65** è data da un banco di memoria **RAM** aggiuntivo di **64** bytes che, sommato ai **64** bytes che tutti i micro della classe ST62 possiedono (all'indirizzo **84h-BFh**), fa appunto un totale di **128** bytes di RAM.
Per quanto riguarda invece la memoria **EEPROM** si tratta di due banchi di memoria di **64 bytes.**
I banchi aggiuntivi **RAM** ed **EEPROM** vengono convenzionalmente definiti "**pagine**" e possono essere selezionati ed utilizzati solo uno per volta:

Eeprom Page 0 corrisponde al primo banco aggiuntivo di memoria **EEprom**,
Eeprom Page 1 corrisponde al secondo banco ag-

giuntivo di memoria **EEprom**,
Ram Page 2 corrisponde al banco di memoria **RAM** aggiuntivo.

Come abbiamo già avuto modo di ricordare, togliendo la tensione di alimentazione al microprocessore, la memoria **EEprom** mantiene memorizzati i dati in essa contenuti per circa 10 anni e per questo motivo si usa molto frequentemente.

Ciò procura evidentemente dei vantaggi e perciò si tende normalmente a sfruttare frequentemente questa importante caratteristica.

Dobbiamo comunque precisare che le memorie **EEprom** non hanno vita infinita, infatti la Casa Costruttrice dà una vita media di circa **1.000.000** di cicli di scrittura o cancellazione.

Inoltre la fase di scrittura in una memoria **EEPROM** richiede una certa frazione di tempo (in condizioni ottimali dai **10** ai **20 millisecondi**), perché prima di registrare un dato viene effettuata la **erase**, cioè la **cancellazione** dei dati che erano stati in precedenza memorizzati.

La **Ram Page 2** non presenta nessun problema in fase di gestione, perché, una volta selezionata con il registro **EEDBR**, si può usare come una normale area **Data Ram** per la gestione delle variabili.

LE 3 PAGINE di MEMORIA

Vediamo ora di spiegare come utilizzare in maniera ottimale le tre pagine di memoria aggiuntiva dei micro **ST6260-65**.

La caratteristica comune a queste **3 pagine** di memoria è quella di avere una **dislocazione parallela**, vale a dire che sono poste una sopra l'altra come le pagine di un libro, e ognuna di queste pagine possiede un'area di **64 bytes** che inizia dall'indirizzo di memoria **000h** e termina con l'indirizzo di memoria **03Fh**.
Come in un libro per indicare un capitolo dobbiamo anche specificare in quale **pagina** si trova, così per leggere e per scrivere in queste memorie dobbiamo indicare l'indirizzo dei bytes che ci interessano e la loro **pagina** (**Eeprom** o **Ram**).
Per effettuare la selezione della pagina di memoria che si vuole utilizzare si usa un registro apposito che noi abbiamo chiamato **EEDBR**.
Il Registro **EEDBR** o **D**ata **R**am-**EE**prom **R**egister è, come abbiamo già detto, un registro **Write Only** definito all'indirizzo **0E8h** di **Data Space**.

Come potete vedere in fig.2, per selezionare le pagine di **memoria** occorre usare:

bit 0 per la **Eeprom Page 0**
bit 1 per la **Eeprom Page 1**
bit 4 per la **Ram Page 2**

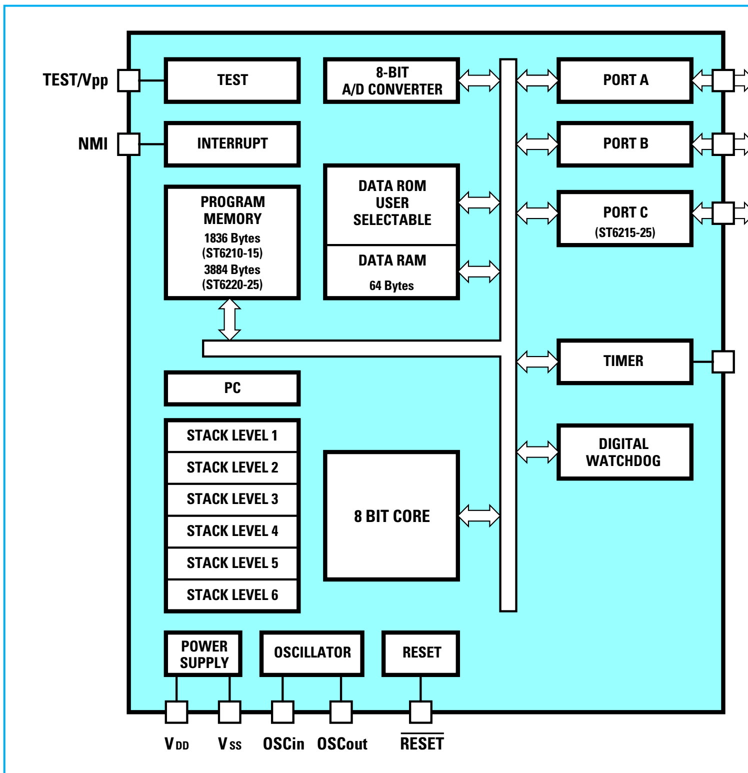


Fig.6 Diagramma a blocchi dei microprocessori ST6210-15-20-25. Come evidenziato anche nella figura, la Port C è presente solo nei micro ST6215-25. Di conseguenza i registri port_C, pdir_C e pop_C non sono utilizzabili con i micro ST6210-20.

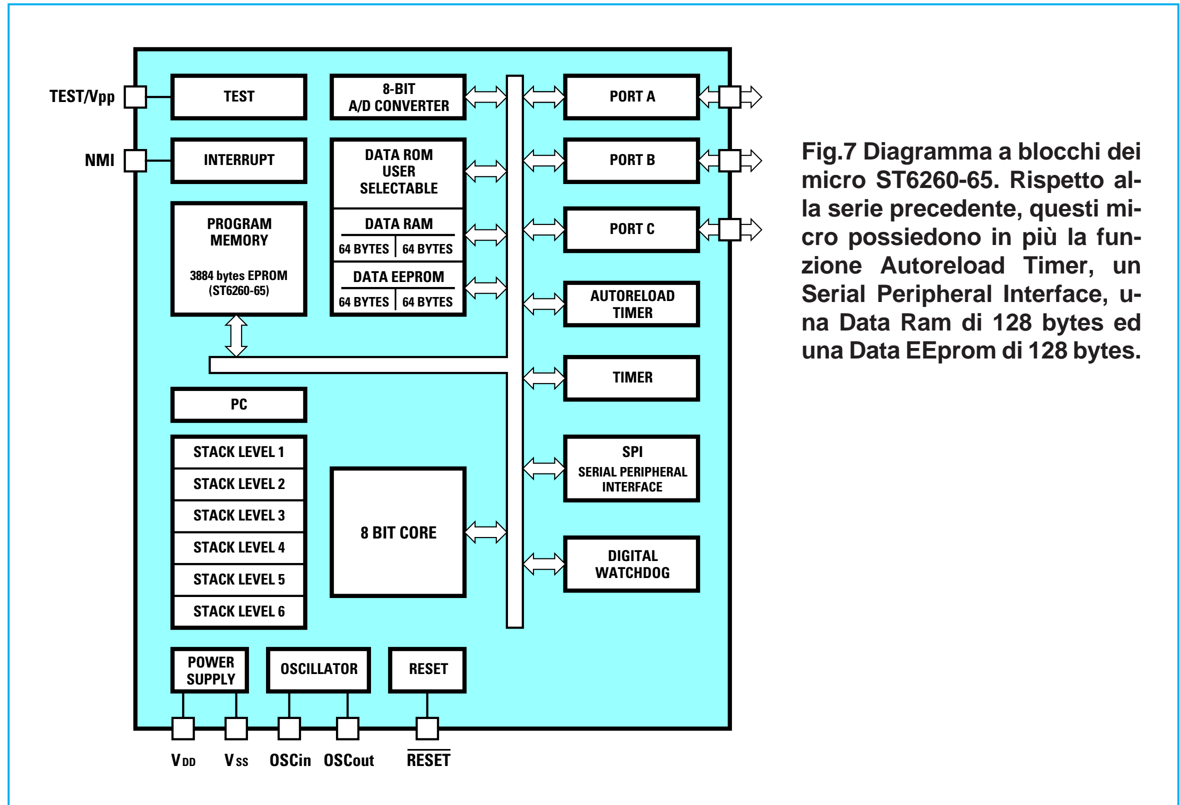


Fig.7 Diagramma a blocchi dei micro ST6260-65. Rispetto alla serie precedente, questi micro possiedono in più la funzione Autoreload Timer, un Serial Peripheral Interface, una Data Ram di 128 bytes ed una Data EEprom di 128 bytes.

Naturalmente potendo utilizzare una pagina di memoria alla volta, non è possibile settare contemporaneamente più di un bit e se lo farete il programma segnalerà **errore**.

Quindi se scrivete:

```
ldi     eedbr,011h
```

fate un errore perché avete settato contemporaneamente il **bit 0** e il **bit 4**.
Infatti **011h** corrisponde in binario a **00010001**.

Per selezionare la **Ram page 2** dovete settare il solo **bit 4** scrivendo:

```
ldi     eedbr,010h
```

infatti **010h** corrisponde al binario **00010000**.

Con il registro **EEDBR** siamo quindi in grado di dire al programma quale pagina di memoria aggiuntiva vogliamo utilizzare.

Nella stesura del programma è inoltre possibile associare agli indirizzi di queste pagine delle etichette con l'istruzione **.def**, tenendo sempre presente però che queste etichette identificano un indirizzo **comune** a tutte e **tre** le pagine.

Ad esempio se scriviamo:

```
pippo   .def    000h
gatto   .def    001h
```

associamo l'etichetta **pippo** all'indirizzo di memoria **000h** e l'etichetta **gatto** all'indirizzo **001h**.
Essendo l'area di memoria comune a tutte e tre le pagine, gli indirizzi **000h** e **001h** costituiscono il **byte 0** e il **byte 1** sia della **Eeprom Page 0** sia della Eeprom **Page 1** sia della **Ram Page 2**.
Quindi **pippo** e **gatto** definiscono il **primo** ed il **secondo** byte di tutte e **tre** le **pagine**.
Se ora riprendiamo l'istruzione:

```
ldi     eedbr,010h
```

che seleziona la **Ram page 2** e successivamente scriviamo:

```
ld      a,pippo
```

carichiamo nell'accumulatore **"a"** il valore corrispondente all'etichetta **pippo**, cioè il valore contenuto nel **byte 0** della memoria **Ram Page 2**.
Se invece scriviamo:

```
ldi     eedbr,001h
ld      a,pippo
```

selezioniamo la **Eeprom Page 0** e carichiamo nell'accumulatore "**a**" il valore contenuto nel **byte 0** di questa memoria.

Come sempre, queste gestioni richiedono un po' di attenzione nella stesura del programma.

E' possibile poi, durante l'esecuzione del programma, "spostarsi" da una pagina all'altra di queste tre memorie tenendo però sempre presente che è meglio utilizzare e soprattutto scrivere nelle **Eeprom Page** solamente quando effettivamente necessita, per "allungare" così la loro vita il più possibile. Conviene perciò, dove naturalmente sia fattibile, che il programma, una volta che si è posizionato in una **Eeprom Page**, non elabori i dati direttamente lì, ma li trasporti in una o più variabili definite nella normale memoria **Data RAM.**
Qui sarà possibile elaborarli tranquillamente e solamente quando necessario il programma li riscriverà nella **Eeprom Page** di partenza.

LA SCRITTURA nella MEMORIA EEPROM

La fase di scrittura delle **Eeprom Page** può essere effettuata in due modalità:

Modalità byte o **Byte mode**
Modalità parallela o **Parallel Mode**

Nella scrittura in **modalità byte** i bytes utilizzati dal programma vengono scritti **uno** alla volta all'interno delle **Eeprom Page**.
Si tratta di una modalità da utilizzare solo nel caso i bytes siano pochissimi o addirittura **1** solo.
Infatti ogni ciclo di scrittura dura una certa frazione di tempo **T**, tipicamente **10 millisecondi**, quindi se i bytes sono molti, ad esempio **7**, la durata della fase completa di scrittura è data da **7 x T**, cioè:

7 x 10 = 70 millisecondi

Nella scrittura in modalità **parallela** invece i bytes vengono scritti all'interno delle **Eeprom Page** contemporaneamente, cioè **8** alla volta.
Con questa modalità si risparmia notevolmente tempo, perché la durata della fase di scrittura è di un **T** per tutti gli **8 bytes**.

Il microprocessore è in grado di **posizionarsi** automaticamente nel punto in cui vogliamo che abbia inizio la registrazione e da quel punto scrive 8 bytes per riga.

Infatti scrivendo in modalità parallela anche se i bytes da scrivere nelle **Eeprom Page** sono minori di 8 o non sono multipli esatti di 8 (ad esempio **3** o **15** o **27**) vengono sempre scritti a 8 bytes alla volta per ogni ciclo di scrittura.

Nella fig.8 è riportata la suddivisione logica delle Eeprom Page in "righe" di 8 bytes nel caso di scrittura in modalità parallela.
Nel caso di **10 bytes** servono almeno **2 cicli** di scrittura; se dovessimo scrivere **64 bytes** dovremmo effettuare **8** cicli di scrittura parallela, infatti **64 bytes : 8 bytes alla volta = 8 cicli**.

Quindi se i **10 bytes** da scrivere in **Eeprom Page** iniziassero all'indirizzo **0**, basterebbero **2 cicli** di **T** per scrivere **10 bytes**: i primi 8 bytes con un ciclo T e i restanti 2 con un altro ciclo T.

Ma cosa succede se dobbiamo iniziare a registrare i **10 bytes** ad esempio dall'indirizzo **01Fh**, che equivale a **31** decimale?

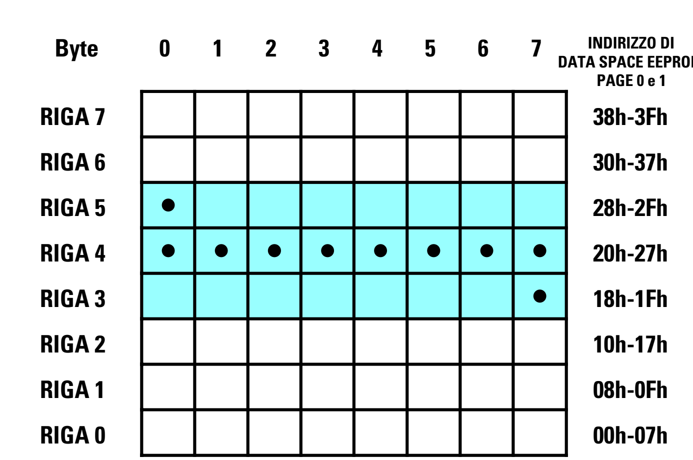


Fig.8 Con la scrittura in modalità parallela i bytes vengono scritti all'interno delle Eeprom Page a 8 alla volta. Qui a fianco potete vedere la suddivisione logica delle Eeprom Page in righe di 8 bytes ciascuna. Come spiegato nell'articolo, per scrivere ad esempio 10 bytes a partire dall'indirizzo 01Fh (vedi pallini in nero) dovremo dare tre istruzioni di scrittura, perché i 10 bytes occupano tre righe di 8 bytes delle Eeprom Page.

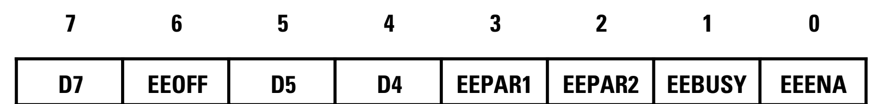


Fig.9 Formato logico del registro EECR definito all'indirizzo 0EAh di Data Space. Questo registro è adibito alla gestione della scrittura nelle Eeprom Page.

Come abbiamo detto, il microprocessore, una volta avviata la procedura di scrittura, si posiziona all'indirizzo **01Fh**, che si trova nella terza riga (vedi in fig.8) e da qui inizia a scrivere.

Poiché deve scrivere **8 bytes** alla volta in **ogni riga**, non gli bastano più 2 T per scrivere **10 bytes**, ma ne impiega **3**, infatti:

Il **primo** dei **10 bytes** viene scritto con un ciclo di scrittura (**1° T**) nella riga che contiene l'indirizzo **01Fh** da cui vogliamo che parta, cioè la terza.
Gli altri **8 bytes** vengono scritti con un altro ciclo di scrittura (**2° T**) nella quarta riga, arrivando così all'indirizzo **27h**.
L'**ultimo** dei 10 bytes viene scritto con un terzo ciclo di scrittura (**3° T**) nella quinta riga.

Dunque nella modalità parallela è necessario che il programmatore tenga conto non solo di quanti bytes vanno scritti, ma anche della posizione in cui i bytes vanno scritti per sapere **quante istruzioni** di **scrittura** in modalità parallela deve dare.

Più avanti troverete altri esempi su questa modalità di scrittura, ma ora continuiamo con la spiegazione della gestione di queste memorie aggiuntive.

Risulta oramai evidente infatti che con il solo registro **EEDBR** non è possibile gestire la fase di scrittura delle **Eeprom Page**, perché non possiede nessun bit che permetta di selezionare e gestire queste due modalità di scrittura.
Adibito a questa funzione c'è un secondo registro, che noi abbiamo chiamato **EECR**.
Il registro **EECR** o **E**eprom **C**ontrol **R**egister, si trova definito all'indirizzo **0EAh** di **Data Space**.
Nella fig.9 potete vedere il suo formato, di cui analizziamo ora ogni singolo bit.

Bit 7 non è utilizzato.

Bit 6, siglato **EEOFF**, è un bit **Write Only** e viene definito **Stand-by Enable bit**.
Quando è **settato** la memoria **EEPROM** è disabilitata, di conseguenza non è possibile leggere o scrivere in questa pagina; quando è **resettato** la **EEprom** è abilitata.

Si può utilizzare questa opzione nel caso il programma da eseguire non debba usare mai le **Eeprom Page**.
Attenzione, essendo un bit **Write Only** non è consigliabile interrogare il suo stato logico.

Bit 5 è riservato e deve sempre essere a **0**.

Bit 4 è riservato e deve sempre essere a **0**.

Bit 3, siglato **EEPAR1**, è un bit **Write Only** definito **Parallel Start Bit**.
Quando è **settato** il micro **scrive** in modalità parallela **8 bytes** adiacenti nelle **Eeprom Page 0** o **1**.
Fintanto che questo bit rimane **settato** non è possibile effettuare altre istruzioni di scrittura.
Quando la fase di scrittura è terminata il microprocessore **resetta** automaticamente questo bit.

Bit 2, siglato **EEPAR2**, è un bit **Write Only** definito **Parallel Mode Enable Bit**.
Questo bit serve solamente per selezionare la **modalità** di scrittura.
Se **settato** attiva la modalità di scrittura **parallela**, se **resettato** attiva la modalità **byte**.
Quando la fase di scrittura parallela è terminata viene automaticamente **resettato**.

Bit 1, siglato **EEBUSY**, viene definito **EEprom Busy bit**, cioè **bit di EEprom occupata**.
Questo bit viene gestito direttamente dal microprocessore che lo **setta** ogniqualvolta si lancia un ciclo di scrittura nelle **Eeprom Page** e lo **resetta** quando questa fase è terminata.
La funzione di questo bit è quella di permettere a chi scrive i programmi di poter interrogare la fine della fase di scrittura, perché fintanto che è in esecuzione non è consigliabile né lanciare un'altra fase di scrittura né tantomeno selezionare una diversa pagina di memoria.

Bit 0, siglato **EEENA**, è un bit **Write Only** definito **EEprom Enable Bit**.
Questo bit serve **solo** per abilitare la modalità **scrittura**. Solo quando risulta **settato** è possibile scrivere nelle **Eeprom Page**. Se **resettato** ogni tentativo di scrittura sarà ignorato.

Ora passiamo ad una serie di esempi per completare e chiarire quanto detto sopra.
Il primo esempio riguarda la gestione della scrittura in **modalità parallela**, il secondo è un esempio di scrittura in **modalità byte**, mentre il terzo è un esempio sulla **gestione** del **tempo** durante la fase di scrittura nella memoria EEprom.
Per vostra comodità, oltre a spiegare istruzione per istruzione, abbiamo riportato l'intero listato di ogni esempio nelle figg.10-12.

ESEMPI

1° Esempio: Scrittura in Modalità Parallela

Dobbiamo scrivere un programma che all'inizio legga **14** bytes della **Eeprom Page 0** a partire dall'indirizzo **0** e poi li muova in altrettanti bytes della **Data Ram** per poterli elaborare.
Dopodiché li deve scrivere con modalità **parallela** nella **Eeprom Page 1** memorizzandoli dall'indirizzo **012h** in poi.
In fig.10 riportiamo il listato del programma di cui ora diamo una spiegazione dettagliata.

Per comodità associamo all'indirizzo **0** l'etichetta **beep0** e all'indirizzo **012h** l'etichetta **sceep1**.

```
beep0   .def    000h
sceep1  .def    012h
```

A questo punto definiamo tutte le nostre variabili e l'indirizzo di memoria **Data Ram** che ci serve per memorizzare i **14** bytes letti da **Eeprom Page 0**:

```
stramx  .def    084h
```

Dopo la definizione delle porte utilizzate dal programma e la gestione degli eventuali **interrupt**, il programma arriverà alla gestione della lettura della **Eeprom Page 0**.
Assegniamo quindi innanzitutto l'etichetta **leepr0** a questa fase, poi ricarichiamo il **Watchdog**:

```
leepr0  ldi     wdog,0ffh
```

Ora attiviamo la memoria aggiuntiva **EEPROM**:

```
        ldi     eecr,0
```

e selezioniamo la **Eeprom Page 0**:

```
        ldi     eedbr,1
```

Dopo questa istruzione nell'area di memoria dall'indirizzo **000h** all'indirizzo **03Fh** sono contenuti i valori della **Eeprom Page 0**.

Siccome sono **14** i bytes da "trasferire" dalla **Eeprom Page 0** alla **Data Ram** e da memorizzare a partire dall'indirizzo **stramx**, carichiamo i registri necessari.
Nel registro **w** carico il numero **14** per effettuare 14 cicli di "trasferimento":

```
        ldi     w,14
```

Nel registro **x** carichiamo l'indirizzo di **beep0**, che corrisponde in questo caso al primo byte di **Eeprom Page 0**:

```
        ldi     x,beep0
```

Nel registro **y** carico l'indirizzo di **stramx** che corrisponde alla locazione iniziale di memoria **Data Ram** dove verranno "trasferiti" i **14** bytes:

```
        ldi     y,stramx
```

Assegniamo a questa fase l'etichetta **ciclor** e ricarico il **Watchdog**.

```
ciclor  ldi     wdog,0FFh
```

Ora tramite l'accumulatore **a** trasferiamo i dati da **Eeprom Page 0** a **Data Ram** un byte alla volta:

```
        ld      a,(x)
        ld      (y),a
```

Decrementiamo quindi il registro **w** di uno:

```
        dec     w
```

Quando arriva a **zero** abbiamo completato il trasferimento dei 14 bytes quindi usciamo da questa fase saltando alla etichetta **fineep**:

```
        jrz     fineep
```

Se invece il trasferimento non è stato ancora completato, ci posizioniamo al byte successivo sia nella **Eeprom Page 0** sia nella **Data Ram**.

```
        inc     x
        inc     y
```

ed eseguiamo di nuovo il ciclo:

```
        jp      ciclor
```

Il trasferimento del contenuto dei **14** bytes ora è completato e quindi possiamo per ora disattivare la memoria **EEPROM**:

```
    ldi     eecr,01000000b
```

Fig.10 Listato 1° Esempio.

```
beep0    .def    000h
sceep1   .def    012h
stramx   .def    084h
......   ...     ...
......   ...     ...
leepr0   ldi     wdog,0ffh
         ldi     eecr,0
         ldi     eedbr,1
         ldi     w,14
         ldi     x,beep0
         ldi     y,stramx
ciclor   ldi     wdog,0ffh
         ld      a,(x)
         ld      (y),a
         dec     w
         jrz     fineep
         inc     x
         inc     y
         jp      ciclor
fineep
         ldi     eecr,01000000b
         call    rou_add
         call    rou_clc
         call    rou_str

wrieep   ldi     eecr,0
         ldi     eedbr,2
         ldi     eecr,00000101b
         ldi     w,14
         ldi     v,6
         ldi     x,stramx
         ldi     y,sceep1
ciclos   ldi     wdog,0ffh
         ld      a,(x)
         ld      (y),a
         dec     w
         jrz     finwrp
         inc     x
         inc     y
         dec     v
         jrnz    ciclos
         ldi     eecr,00001101b
         jrs     1,eecr,$
         ldi     eecr,00000101b
         ldi     v,8
         jp      ciclos
finwrp
         ldi     eecr,00001101b
         jrs     1,eecr,$

         ldi     eecr,0
```

Questo esempio riguarda la gestione della scrittura in Modalità Parallela.

Fig.11 Listato 2° Esempio.

```
wrieep   ldi     eecr,0
         ldi     eedbr,2
         ldi     eecr,00000001b
         ldi     w,14
         ldi     x,stramx
         ldi     y,sceep1
ciclos   ldi     wdog,0ffh
         ld      a,(x)
         ld      (y),a
         jrs     1,eecr,$
         dec     w
         jrz     finwrp
         inc     x
         inc     y
         jp      ciclos
finwrp
         ldi     eecr,0
```

Questo esempio riguarda la gestione della scrittura in Modalità Byte. Per la prima parte del programma, cioè fino all'elaborazione dei dati nelle routine rou_add, rou_clc e rou_str, si può fare riferimento all'esempio riportato in fig.10.

Fig.12 Listato 3° Esempio

```
wrieep   ldi     eecr,0
         ldi     eedbr,2
         ldi     eecr,00000001b
         ldi     w,14
         ldi     x,stramx
         ldi     y,sceep1
ciclos   ldi     wdog,0ffh
         jrs     1,eecr,$
         ld      a,(x)
         ld      (y),a
         dec     w
         jrz     finwrp
         inc     x
         inc     y
         jp      ciclos
finwp
         jrs     1,eecr,$
         ldi     eecr,0
```

Questo esempio riguarda la gestione del tempo durante la fase di scrittura nelle Eeprom Page. Notate il posizionamento dell'istruzione JRS 1,EECR,$ con cui si evita che qualche comando inerente alla memoria EEprom venga attivato prima che la fase di scrittura sia terminata.

A questo punto i dati così caricati in **Data Ram** vengono elaborati da una serie di routines che nel nostro esempio sono **rou_add**, **rou_clc** e **rou_str**:

```
        call    rou_add
        call    rou_clc
        call    rou_str
```

Finita questa fase di elaborazione, il programma deve memorizzare i valori ottenuti nella **Eeprom Page 1**. Riattiviamo perciò la memoria **EEPROM**:

```
wrieep  ldi     eecr,0
```

e ci posizioniamo nella **Eeprom Page 1**:

```
        ldi     eedbr,2
```

Ora dall'indirizzo di memoria **000h** a **03Fh** sono contenuti i dati presenti nella **Eeprom Page 1**.
Attiviamo quindi (non eseguiamo ancora) la scrittura in modalità **parallela**:

```
        ldi     eecr,00000101b
```

Siccome i bytes elaborati da trasferire dalla **Data Ram** alla **Eeprom Page 1** sono **14** e vanno memorizzati a partire dall'indirizzo **sceep1**, carichiamo i registri necessari.
Nel registro **w** carichiamo il numero **14** per effettuare 14 cicli di "trasferimento":

```
        ldi     w,14
```

La scrittura dei 14 bytes deve partire dalla locazione EEprom **012h**, che equivale a **18** in **decimale**, quindi dobbiamo iniziare a scrivere dal **terzo bytes** della **terza riga** (vedi la suddivisione logica delle Eeprom Page in fig.8).
I 14 bytes saranno quindi registrati **6** nella **riga 2** ed i restanti **8** nella riga successiva.
Carichiamo pertanto il valore 6 nel registro **v**:

```
        ldi     v,6
```

Nel registro **x** carichiamo l'indirizzo di **stramx**, che corrisponde alla locazione iniziale di memoria **Data Ram** dove verranno "prelevati" i valori dei **14** bytes:

```
        ldi     x,stramx
```

Nel registro **y** carichiamo l'indirizzo di **sceep1**, che corrisponde all'indirizzo di **Eeprom Page 1** dove verranno "trasferiti" i valori dei **14** bytes:

```
        ldi     y,sceep1
```

A questa fase assegniamo l'etichetta **ciclos** e ricarichiamo il **Watchdog**:

```
ciclos  ldi     wdog,0FFh
```

Ora tramite l'accumulatore **a** trasferiamo (non scriviamo ancora nulla) i dati da **Data Ram** a **Eeprom Page 1** un byte alla volta:

```
        ld      a,(x)
        ld      (y),a
```

e decrementiamo il registro **w** di uno:

```
        dec     w
```

Quando arriva a **zero** abbiamo completato il trasferimento dei 14 bytes ed usciamo da questa fase saltando alla etichetta **finwrp**:

```
        jrz     finwrp
```

Se invece non è stato ancora completato, ci posizioniamo al bytes successivo sia nella **Eeprom Page 1** sia nella **Data Ram**:

```
        inc     x
        inc     y
```

e decrementiamo il registro **v**:

```
        dec     v
        jrnz    ciclos
```

Quando arriva a **0** significa che la prima volta ha terminato di caricare i **6 bytes**, perciò eseguiamo il **primo ciclo** di scrittura e attendiamo che sia terminata la scrittura interrogando il bit **1 EEBUSY**:

```
        ldi     eecr,00001101b
        jrs     1,eecr,$
```

Riattiviamo quindi la modalità **parallela**:

```
        ldi     eecr,00000101b
```

perché restano ancora 8 bytes da trasferire. Ricarichiamo quindi il registro **v** con questo valore:

```
        ldi     v,8
```

e saltiamo all'etichetta ciclos:

```
        jp      ciclos
```

Quando il programma arriva a **finwrp** è terminato il trasferimento dei **bytes**, pertanto eseguiamo il **se-**

condo ciclo di scrittura e attendiamo che sia terminata la scrittura interrogando il bit **EEBUSY**:

```
        ldi     eecr,00001101b
        jrs     1,eecr,$
```

Finita la fase di scrittura parallela il bit **3** ed il bit **2** del registro **eecr**, denominati rispettivamente **EEPAR1** e **EEPAR2**, vengono automaticamente resettati, mentre rimane settato solo il bit **0** denominato **EEENA**. Non ci rimane dunque che disattivare subito la modalità scrittura per evitare il pericolo di sporcare i dati appena scritti.

```
  ldi     eecr,0
```

Il perché di questa ultima affermazione lo capirete meglio con il secondo esempio.

2° Esempio: Scrittura in Modalità Byte

Dobbiamo scrivere un programma che all'inizio legga **14** bytes della **Eeprom Page 0** a partire dall'indirizzo **0** e poi li muova in altrettanti bytes della **Data Ram** per poterli elaborare. Dopodiché li deve scrivere con modalità **byte** nella **Eeprom Page 1** memorizzandoli dall'indirizzo **012h** in poi.
Abbiamo volutamente ripetuto il precedente esempio cambiando solamente la modalità di scrittura per fare risaltare maggiormente le differenze di gestione delle due modalità di scrittura.
E' perciò evidente che la parte iniziale del programma è identica al precedente esempio quindi ci pare inutile rispiegarvela.

Iniziamo dunque la spiegazione dal punto in cui si inizia a gestire la scrittura nella Eeprom Page 1 e cioè dall'istruzione con etichetta **wrieep** (fig.11).
Finita questa fase di elaborazione, il programma deve memorizzare i valori ottenuti nella **Eeprom Page 1**. Riattiviamo perciò la memoria **EEPROM**:

```
wrieep  ldi     eecr,0
```

e ci posizioniamo nella **Eeprom Page 1**:

```
        ldi     eedbr,2
```

Ora **attiviamo** la **scrittura** in modalità **byte**:

```
  ldi     eecr,00000001b
```

Siccome i bytes elaborati da trasferire dalla **Data Ram** alla **Eeprom Page 1** sono **14** e vanno memorizzati a partire dall'indirizzo **sceep1**, carichiamo i registri necessari.

Nel registro **w** carichiamo il numero **14** per effettuare 14 cicli di "trasferimento":

```
        ldi     w,14
```

Nel registro **x** carichiamo l'indirizzo di **stramx**, che corrisponde alla locazione iniziale di memoria **Data Ram** dove verranno "prelevati" i valori dei **14** bytes:

```
        ldi     x,stramx
```

Nel registro **y** carichiamo l'indirizzo di **sceep1**, che corrisponde all'indirizzo iniziale di Eeprom **Page 1** dove verranno "trasferiti" i valori dei **14** bytes:

```
        ldi     y,sceep1
```

A questa fase assegniamo l'etichetta **ciclos** e ricarichiamo il **Watchdog**:

```
ciclos  ldi     wdog,0FFh
```

Ora tramite l'accumulatore **a** trasferiamo e **scriviamo** i dati da **Data Ram** a **Eeprom Page 1** un byte alla volta:

```
        ld      a,(x)
        ld      (y),a
```

In questo momento il dato viene **scritto** nella **Eeprom Page 1**.

Ora gestiamo il tempo di attesa per la scrittura per evitare di attivare la scrittura di un altro byte prima che sia finita la scrittura dell'altro.

```
        jrs     1,eecr,$
```

A questo punto il dato è stato definitivamente scritto nella Eeprom Page 1 ed anche se si verificasse una caduta di tensione non andrebbe perso.
Ora decrementiamo il registro **w** di uno:

```
        dec     w
```

Quando arriva a **zero** abbiamo completato il trasferimento e la **contemporanea scrittura** dei **14** bytes, quindi saltiamo all'etichetta **finwrp**:

```
        jrz     finwrp
```

Se invece non è stato ancora completato, ci posizioniamo al byte successivo sia nella **Eeprom Page 1** sia nella **Data Ram**:

```
        inc     x
        inc     y
```

ed eseguiamo di nuovo il ciclo:

```
        jp      ciclos
```

Una volta finita la fase di scrittura disattiviamo la modalità byte:

```
        ldi     eecr,0
```

Come avete avuto modo di capire con questo esempio, quando attiviamo la scrittura in **modalità byte** ogni variazione che apportiamo ai dati contenuti nella **Eeprom Page** selezionata viene immediatamente scritta nella memoria. Questo è il motivo per cui dopo ogni fase di scrittura vi consigliamo sempre di disattivare la modalità scrittura.

3° Esempio: Gestione del Tempo in Scrittura

Prima di concludere vogliamo portarvi un ultimo semplice esempio per chiarire soprattutto l'aspetto della **gestione** del **tempo** di scrittura dentro le memorie EEPROM.
Nella fig.12 riportiamo le stesse istruzioni di fig.11 con una piccola differenza.
Abbiamo spostato l'istruzione **jrs 1,eecr,$** dopo:

```
ciclos  ldi     wdog,0ffh
```

e poi l'abbiamo ripetuta dopo l'etichetta:

finwrp

Qualcuno a questo punto si domanderà il perché visto che, come abbiamo spiegato, questa istruzione serve per gestire il tempo di scrittura della EEPROM e sembrerebbe logico doverla inserire subito dopo l'istruzione di scrittura.

Con questa istruzione si vuole solamente evitare che venga attivata un'altra fase di scrittura o la selezione di una pagina diversa di memoria prima che sia terminata la fase attuale di scrittura.
Infatti quando si attiva la fase di scrittura nelle memorie EEPROM il micro non sta ad aspettare che questa fase sia terminata, ma continua ad eseguire le istruzioni successive.
Se, per assurdo, le istruzioni successive fossero in numero tale che sommate ci danno un tempo superiore ai **10-20 millisecondi** stimati per la scrittura in EEPROM, in teoria non sarebbe nemmeno necessario utilizzare l'istruzione: **jrs 1,eecr,$**.

La condizione necessaria è che le istruzioni successive non contengano nessun altro comando inerente a queste memorie, perché non sarebbe eseguito. Per non essere perciò costretti a contare i cicli delle istruzioni che seguono la scrittura nelle EEPROM si utilizza l'istruzione:

```
        jrs     1,eecr,$
```

Nella fig.12 abbiamo proposto un diverso modo di gestione della scrittura in EEPROM.
Una soluzione questa che può velocizzare la gestione rispetto a quella di fig.11.
Infatti dopo l'istruzione che "scrive" in **EEPROM**:

```
        ld      (y),a
```

il programma esegue le istruzioni successive, cioè:

```
        dec     w
        jrz     finwrp
        inc     x
        inc     y
        jp      ciclos
ciclos  ldi     wdog,0ffh
```

e solo a questo punto eseguendo l'istruzione:

```
        jrs     1,eecr,$
```

si ferma in attesa che la scrittura nella **EEPROM** sia terminata.
E' evidente che questa attesa sarà sicuramente minore perché parte del tempo è già trascorso con l'esecuzione delle istruzioni precedenti.

Nota: poiché scopo di questi esempi era chiarire come utilizzare le due **modalità** di **scrittura** nella **memoria EEprom**, per non complicare ulteriormente la spiegazione, abbiamo utilizzato le istruzioni **JRZ** e **JRNZ** in maniera impropria, senza tenere conto cioè del fatto che hanno un **salto** condizionato a **– 15/+16 bytes**.
Per l'uso corretto di queste istruzioni rimandiamo a quanto scritto nella rivista **N.185**.

software SIMULATORE

Dopo il successo ottenuto con la prima versione del software simulatore per micro ST6 (vedi rivista N.190), oggi vi presentiamo la nuova versione che permette di simulare anche i micro della serie ST6260-ST6265. Con questo software simulatore potrete leggere e scrivere nelle memorie EEprom e Ram aggiuntiva, verificare la trasmissione Seriale sulla porta Spi, testare il PWM e l'Auto Reload Timer.

A distanza di poco meno di un anno dalla pubblicazione del software simulatore per micro **ST6**, il nostro collaboratore, il Sig. **Ivano Cesarin** di **Porpetto** (**UD**), ha realizzato una nuova versione di simulatore in grado di testare tutti i micro della serie **ST6210-15-20-25** compresi gli **ST6260-65**.

Tutti i softwaristi che per motivi di lavoro o per hobby impiegano questi micro nei loro progetti, riusciranno con questo simulatore a scoprire più facilmente eventuali errori e a correggerli.

Tanto per cominciare vi diciamo che nel **nuovo** simulatore sono state eliminate tutte quelle piccole **anomalie** presenti in fase di apertura dei files contenenti i programmi da simulare.
Rispetto alla versione precedente sono state apportate migliorie e aggiunte altre possibilità di simulazione che troverete molto interessanti, perché vi aiuteranno nei vostri test.
Ad esempio, è stato aggiunto un valido e utile **Help contestuale** con semplici spiegazioni in italiano che vi accompagnano in ogni fase della simulazione.

Come già abbiamo accennato, la novità principale della **nuova** versione riguarda la possibilità di **simulare** non solo i programmi scritti per i micro della serie **ST6210** ecc., ma anche i programmi per la serie **ST6260-65**.
Ciò significa che ora potete simulare le fasi di lettura e scrittura nelle memorie **Eeprom** e **Ram aggiuntiva** (definita dall'Autore **Extra Ram**), verificare la trasmissione seriale sulla porta **Spi**, testare il **PWM** e l'**Auto Reload Timer**.

Per le modalità di **installazione** del programma e per l'uso dei **comandi** identici alla precedente versione, rimandiamo a quanto già scritto nella rivista **N.190**. A questo proposito vi informiamo che se cambierete la **directory** di installazione del simulatore, dovrete modificarla anche nel programma di esempio **pedali.prg** utilizzando un normale editor.

Ora vi spieghiamo solo le **nuove** funzioni presenti in questo simulatore, completandole con le immagini delle **finestre** che appaiono sul monitor, perché una figura a volte chiarisce più di tante parole.

LA VIDEATA PRINCIPALE

Dopo aver memorizzata nell'hard-disk la nuova versione del programma simulatore, quando lanciate il programma appare la finestra di fig.1.

1 – Cliccando su **SimST622** lanciate la simulazione dei programmi scritti per i soli microprocessori della serie **ST6210-15-20-25**.

2 – Cliccando su **SimST626** lanciate la simulazione dei programmi scritti per i soli microprocessori della serie **ST6260-65**.

3 – Cliccando su **Conv > CMD** potete effettuare la conversione di files **.DAT** in files **.CMD**.

4 – Cliccando su **Aiuto** visualizzate le **note** di aiuto del programma (Help contestuale).

per micro ST6

MODIFICA della EEPROM ed EXTRA RAM

Dopo aver lanciato la simulazione di un programma per **ST6260-65**, scegliendo dal menu **File** (vedi fig.2) il comando **Modifica EEProm/Extra RAM** potete accedere alla finestra di dialogo visibile in fig.3, in cui è possibile modificare tutta o una parte sola della memoria **Ram aggiuntiva** (di seguito chiamata Extra Ram) o della memoria **EEprom**.
A questa finestra si accede anche in fase di Creazione o Modifica Progetto cliccando sul pulsante **Precarica EEprom** (vedi fig.4).
Una volta entrati nella finestra visibile in fig.3 è possibile:

1 – Selezionare il tipo di memoria che si vuole modificare o anche solo visualizzare, cliccando nel cerchietto alla destra delle scritte EEprom o Extra Ram. La scelta di una memoria esclude l'altra.

2 – Modificare il contenuto di una **singola** cella cliccandoci sopra 2 volte. In questo caso appare la maschera di fig.5 in cui bisogna digitare il valore che si vuole inserire in quella determinata cella.

3 – Modificare tutto il contenuto della memoria selezionata cliccando sul pulsante **Riempimento**. Appare sempre la maschera di fig.5, ma in questo caso il valore che digitate viene memorizzato in **tutte** le celle della memoria.

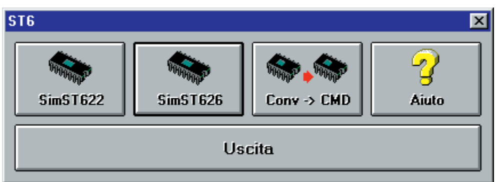


Fig.1 La videata principale della nuova versione del software simulatore per i programmi scritti per i micro ST6.

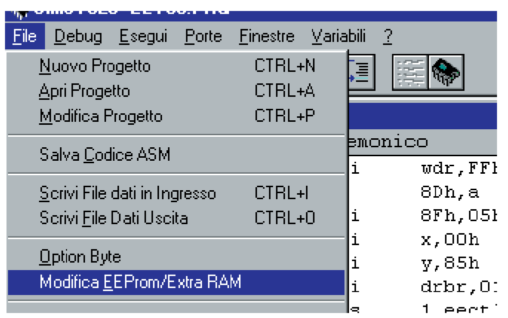


Fig.2 Per modificare la memoria Eeprom o la Ram aggiuntiva, scegliete dal menu File il comando Modifica EEprom/Extra RAM.

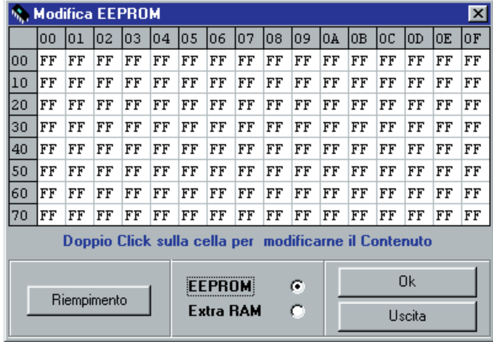


Fig.3 Dopo aver selezionato la memoria, per modificarne l'intero contenuto cliccate sul pulsante Riempimento.

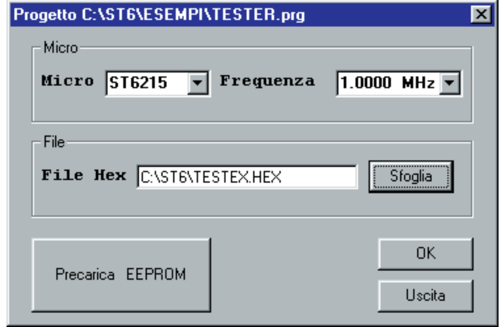


Fig.4 Alla maschera di fig.3 si accede anche in fase di Creazione - Modifica Progetto, cliccando su Precarica EEPROM.

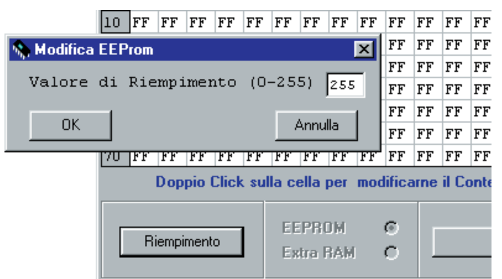


Fig.5 Se avete cliccato sul pulsante riempimento, il valore che digitate verrà memorizzato in tutte le celle.

Fig.6 Con il comando Edit Codice ASM dal menu Debug si può modificare l'istruzione senza interrompere la simulazione.

Fig.7 L'istruzione può essere modificata o eliminata totalmente, inoltre è possibile inserire una nuova istruzione.

Fig.8 Con il pulsante Modifica, l'istruzione è visualizzata suddivisa in più campi corrispondenti alle parti che la compongono.

Fig.9 Se l'istruzione modificata è più lunga o più corta di quella originale, potete rimpiazzare i bytes mancanti con dei NOP.

MODIFICA delle ISTRUZIONI

Grazie alla nuova versione è ora possibile cambiare **in tempo reale** l'istruzione del programma che si sta simulando **senza** dover uscire dalla simulazione e senza dover ricompilare il sorgente.

In questo modo si evitano grosse perdite di tempo e non si deve più impazzire per ritornare nel punto esatto in cui si era dovuta interrompere la simulazione per uscire dal programma, apportare la modifica e ricompilare il sorgente.

Vediamo quindi quali passi bisogna compiere per correggere le istruzioni.

Dal menu Debug cliccate su **Edit Codice ASM** (vedi fig.6) per aprire la finestra di dialogo visibile in fig.7, nella quale è possibile scegliere fra **tre** diversi tipi di digitazione del programma che si testa: **Modifica**, **Inserisci** o **Elimina**.

Fate attenzione perché le variazioni che apportate al sorgente con questi comandi durante la simulazione non vengono memorizzate nel file **.ASM**, ma sono **temporanee** e verranno perse nel caso si chiuda la simulazione e se ne inizi una nuova.
Potrete comunque **memorizzarle** in un file **.TXT**. In seguito vi spiegheremo come questa operazione vi fornirà un'utile traccia per poter in seguito modificare in maniera definitiva il file **.ASM** originale.

Cliccando sul pulsante **Modifica** (vedi fig.7) compare la maschera di fig.8 in cui viene mostrata l'istruzione che stava per essere eseguita suddivisa in **Etichetta - Istruz. - 1° - 2° - 3° Campo**.

Nel caso si sostituisca l'istruzione con un'istruzione più lunga o più corta, prima della conferma delle variazioni il programma vi chiede se volete inserire delle istruzioni **Nop** per "riempire" i bytes restanti.

Nota: ogni istruzione ha una sua precisa lunghezza in **bytes** e a questo proposito consigliamo di consultare le tabelle pubblicate sulla rivista **N.185**.

Nell'esempio visibile in fig.8 abbiamo sostituito l'istruzione **ldi orc,00h**, lunga **3 bytes**, con l'istruzione **reti** lunga **1 bytes** (vedi fig.9).

Pertanto quando diamo l'**OK**, il programma ci chiede se vogliamo che i **bytes** restanti, che, mancando l'**Opcode**, contengono dati non validi, vengano rimpiazzati con dei **Nop**.
Inserendo due **Nop** non viene falsata la numerazione del Displacement e l'istruzione seguente inizia dall'indirizzo di memoria esatto.

Se osservate infatti la fig.10 potete vedere che l'istruzione **ldi** da **3 bytes** è stata sostituita con tre istruzioni da **1 byte** ciascuna:

009A —	**reti**
009B 04	**nop**
009C 04	**nop**

Cliccando sul pulsante **Inserisci** di fig.7, compare la maschera di fig.11 in cui potete scrivere una **nuova** istruzione che automaticamente il programma inserirà **prima** dell'istruzione che stava per essere simulata. Naturalmente in questo caso bisogna fare molta attenzione, perché inserendo una nuova istruzione il Displacement viene variato e gli indirizzi di salto potrebbero essere tutti **falsati**.

Cliccando sul pulsante **Elimina**, sempre in fig.7, compare la maschera di fig.12, in cui si chiede conferma della cancellazione della istruzione.
Se si clicca su **Sì**, compare di seguito la maschera visibile in fig.13 che consente di rimpiazzare i bytes dell'istruzione eliminata con altrettanti **Nop** per **non perdere** il corretto Displacement.

Come abbiamo accennato all'inizio del paragrafo, apportata la modifica potete salvarla in un file **.TXT** così da avere una traccia che vi servirà per cambiare in maniera definitiva anche il file **.ASM**.
Per fare questo, scegliete il comando **Salva Codice ASM** dal menu File (vedi fig.14) e quando appare la finestra di dialogo visibile in fig.15 date un nome al vostro file, quindi cliccate su **OK**.

INSERIRE un'ONDA QUADRA CICLICA

In questo paragrafo vi spieghiamo come procedere per **generare** una forma di **onda quadra** sui **piedini** del microprocessore indicando solamente il **periodo** espresso in numero **cicli** e come visualizzare i **segnali** in **ingresso** e in **uscita** sui piedini del micro anche in modalità **numerica**.

Dopo aver lanciato la simulazione, cliccate sul pulsante **Cronologia Porte** per far apparire la maschera di fig.16 in cui potete scegliere tra due modalità di visualizzazione: **Grafica** e **Numerica**.

Cliccando sul pulsante **Grafica** appare la maschera visibile in fig.17.
Per il nostro esempio abbiamo selezionato la **Porta C** e le funzioni **Edit** e **Limitato** cliccando nelle apposite caselle. A sinistra sono evidenziati i piedini **PC2 - PC3** e **PC4** di **porta C**.
Dopo aver **selezionato** uno dei **piedini** con il cursore, dovete scegliere il tipo di **segnale** in ingresso (la lettera **C** sta per **ciclico**) e di seguito dovete digitare il valore del **periodo** del **ciclo**.

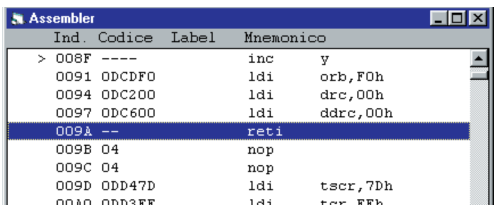


Fig.10 Nell'esempio raffigurato, l'istruzione LDI, lunga tre bytes, è stata sostituita con l'istruzione RETI più due istruzioni NOP.

Fig.11 Inserendo un'istruzione fate attenzione, perché il Displacement viene variato e gli indirizzi di salto vengono falsati.

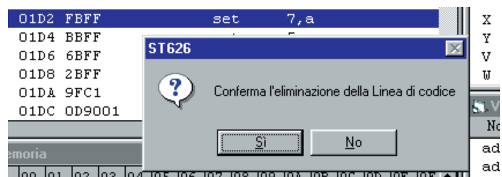


Fig.12 Il comando di eliminazione di un'istruzione dal programma in simulazione va sempre confermato.

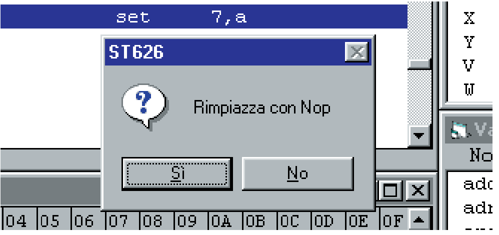


Fig.13 Per non perdere il corretto Displacement, il simulatore vi chiede se deve rimpiazzare l'istruzione con altrettanti NOP.

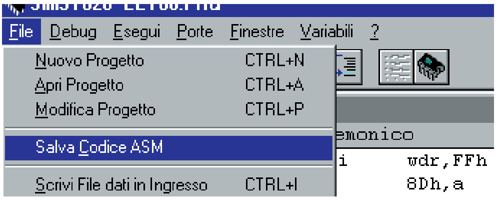


Fig.14 Le variazioni apportate al sorgente durante la simulazione sono temporanee, quindi vanno salvate in un file .TXT.

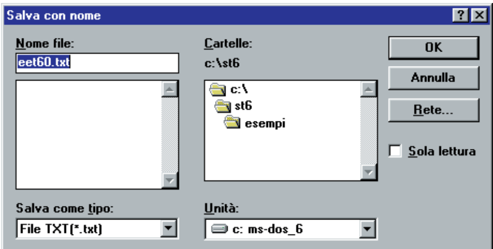


Fig.15 Con il comando Salva Codice ASM dal menu File, si apre questa finestra di dialogo in cui dovete dare un nome al file .TXT che conterrà le variazioni fatte.

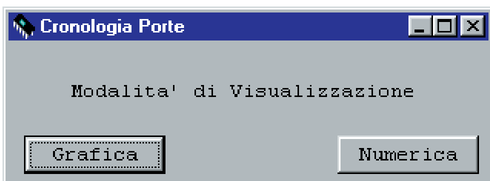


Fig.16 Con questo simulatore è possibile generare una forma d'onda quadra sui piedini del microprocessore e vederla a video in modalità grafica.

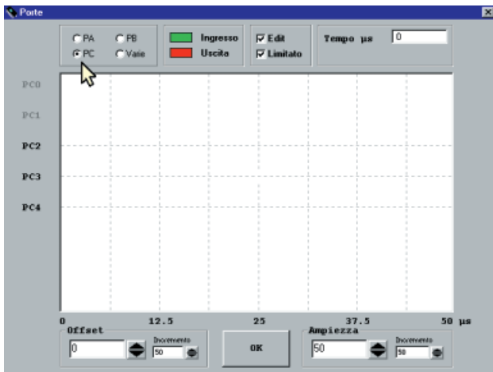


Fig.17 Nell'esempio riportato in questa figura abbiamo immaginato di generare onde quadre sui piedini di PORT C, quindi abbiamo selezionato: Edit, Limitato e PC.

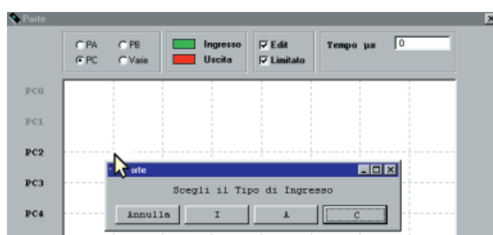


Fig.18 Cliccando su PC2 si attiva la maschera per selezionare il tipo di segnale in ingresso. Per selezionare un segnale ciclico utilizzate il pulsante con la lettera C.

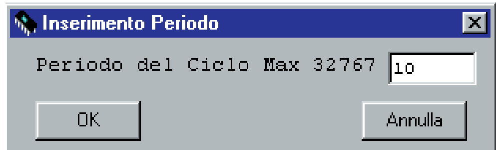


Fig.19 Per generare un'onda quadra con un periodo di 10 cicli è sufficiente digitare il valore 10 e cliccare su OK nella finestra di dialogo qui raffigurata.

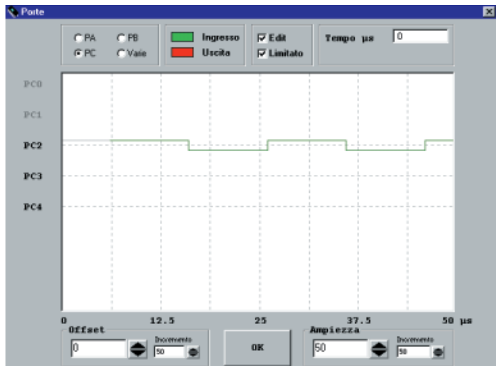


Fig.20 Dopo le operazioni visualizzate nelle due figure precedenti, sul piedino PC2 è ora presente una forma d'onda quadra con un periodo di 10 cicli.

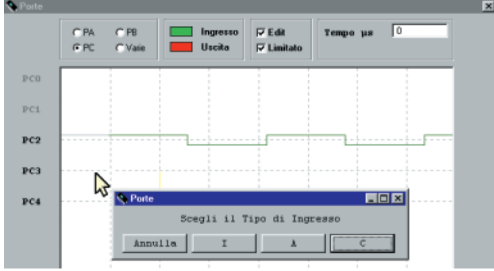


Fig.21 Per generare un altro segnale ad onda quadra clicchiamo sul piedino PC3 e per il tipo di segnale in ingresso scegliamo ancora C, cioè segnale ciclico.

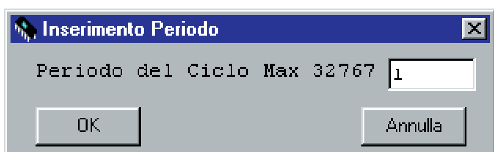


Fig.22 Per generare un'onda quadra con un periodo di 1 ciclo digitiamo 1 in questa finestra di dialogo e clicchiamo su OK. La rappresentazione grafica appare in fig.23.

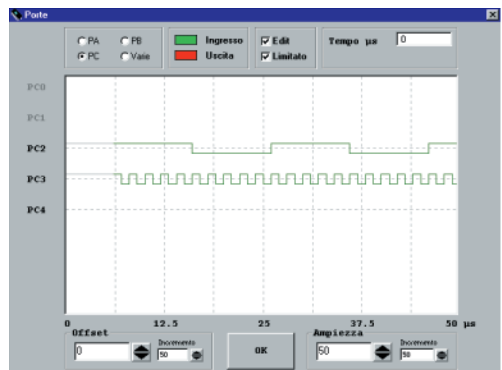


Fig.23 In questa figura potete vedere la rappresentazione grafica dei segnali ad onda quadra con differenti periodi generati sui piedini PC2 e PC3 di port C.

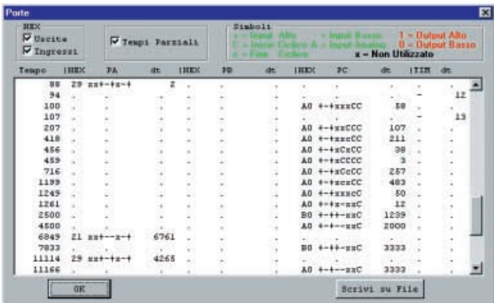


Fig.24 In questa figura potete vedere un esempio di visualizzazione numerica dei segnali immessi o rilevati sui piedini del micro durante la simulazione.

Variabili

Nome	Ind.	HEX	Dec
pdir_a	C4	0	0
pdir_b	C5	3	3
port_a	C0	10	16

Fig.25 Cliccando sul menu Variabili potete visualizzare il contenuto delle variabili utilizzate dal programma che si sta simulando in codifica esadecimale o binaria.

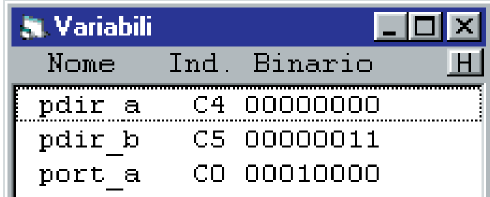


Fig.26 Per passare dalla codifica esadecimale a quella binaria cliccate sulla lettera visibile in alto a destra: H per esadecimale e B per binario (vedi fig.25).

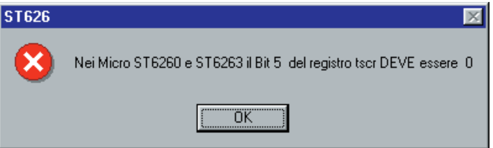


Fig.27 Esempio di uno dei numerosi controlli inseriti dall'Autore per annotare gli errori. In questo caso il registro TSCR è stato caricato con un dato non valido.

Fig.28 Il simulatore ha rilevato un'altra istruzione formalmente corretta, ma logicamente errata: non è possibile scrivere nella EEprom se il bit 2 è uguale a zero.

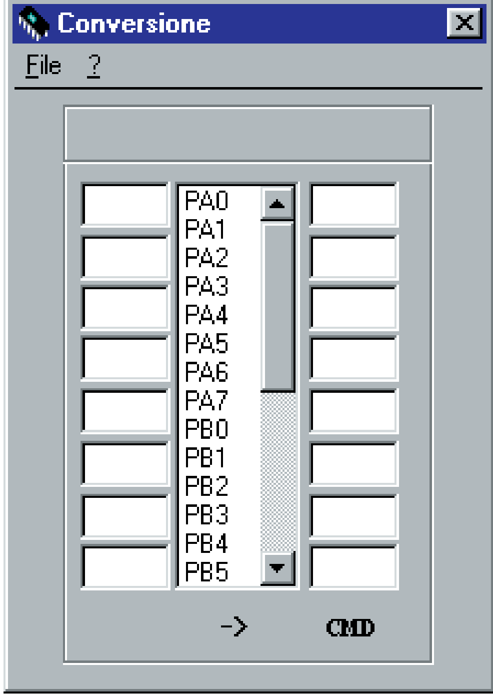


Fig.29 Cliccando sul pulsante Conv>CMD nella videata principale si attiva la Conversione dei segnali in uscita, memorizzati in un file .DAT, in segnali in ingresso, da memorizzare in un file .CMD. In questo modo, partendo da un solo file, si possono generare numerosi files di segnali in ingresso per le successive simulazioni.

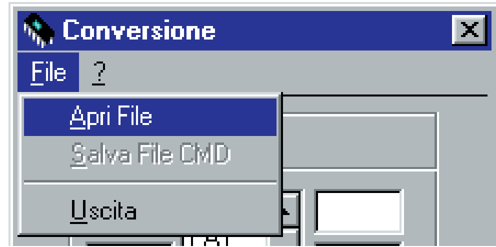


Fig.30 Per aprire un file .DAT, che contiene i segnali in uscita ottenuti durante una simulazione, cliccate sul menu File e scegliete Apri File.

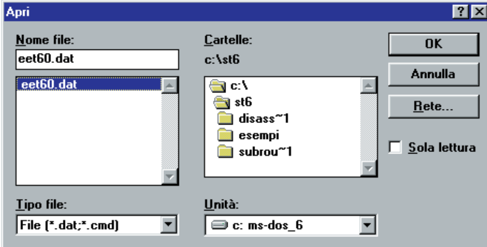


Fig.31 Nella finestra di dialogo visibile in questa figura è necessario selezionare il nome del file di cui vogliamo convertire i segnali. Nel nostro caso è il file eet60.dat.

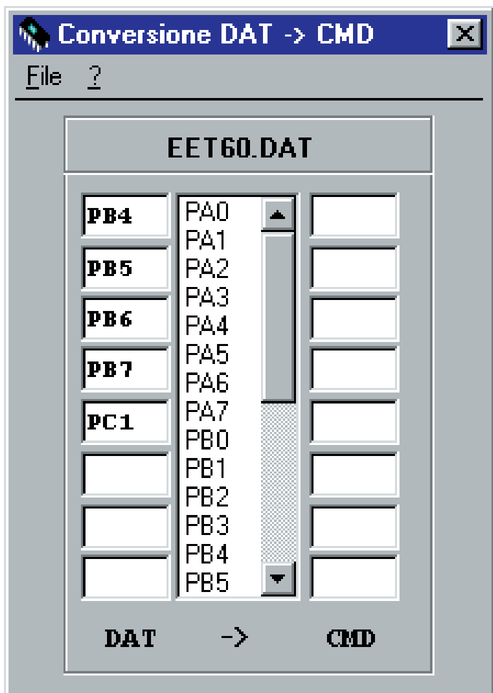


Fig.32 Viene così generata questa maschera in cui a sinistra ci sono i piedini con i segnali da convertire e al centro l'elenco completo dei piedini del micro.

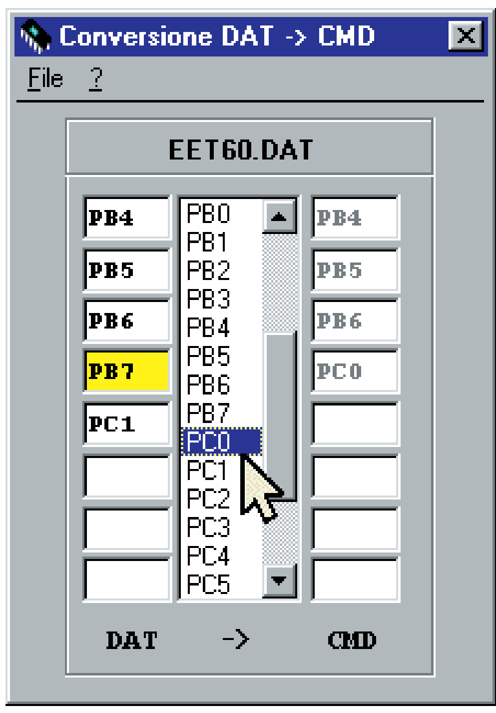


Fig.33 Per effettuare la conversione cliccate prima a sinistra e poi indicate nella colonna centrale su quale piedino deve essere trasferito il segnale.

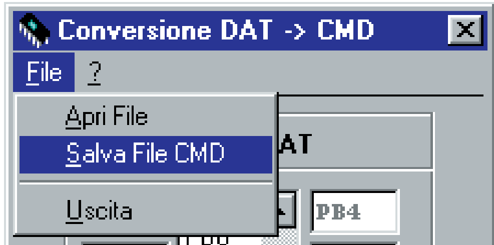


Fig.34 Completato il trasferimento dei segnali in uscita in segnali in ingresso, scegliete Salva File CMD dal menu File per memorizzare la conversione.

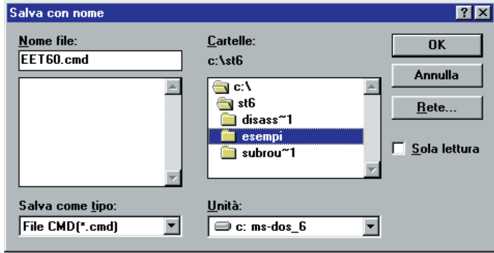


Fig.35 Nella finestra di dialogo visibile in questa figura è necessario scrivere il nome del file che contiene la conversione. Nel nostro caso è il file eet60.cmd.

Le operazioni per generare una forma d'onda quadra con un periodo di **10 cicli** sul piedino **PC2** sono visibili in sequenza nelle figg.18-20.
Le figg.21-22 illustrano invece le operazioni eseguite per generare una forma d'onda quadra con un periodo di **1 ciclo** sul piedino **PC3**.
Infine in fig.23 potete vedere il risultato delle due operazioni in forma grafica.

Per visualizzare i segnali in modalità **Numerica** cliccate sull'apposito pulsante visibile nella maschera di fig.16. Si apre così la maschera di fig.24, in cui è riportato l'esempio di una serie di segnali, rappresentati per l'appunto in forma numerica, rilevati dopo la simulazione di un programma.
Per la spiegazione dei singoli simboli e numeri che appaiono in questa figura, rimandiamo alla consultazione dell'**Help contestuale** fornito con questo software simulatore.

VISUALIZZARE le VARIABILI

Il contenuto delle **variabili** utilizzate nel programma che si sta simulando può essere visualizzato sia in codifica **esadecimale** sia in codifica **binaria**.

Cliccando sul menu **Variabili** si apre una finestra di dialogo che mostra, come nell'esempio di fig.25, l'elenco di variabili inserite nella lista di visualizzazione durante la simulazione del programma.
Come noterete subito il valore che contengono è espresso in Esadecimale sotto la colonna **HEX** e in Decimale sotto la colonna **Dec**.
Se a questo punto cliccate sopra il pulsante contrassegnato dalla lettera **B**, in alto a destra, lo stesso elenco viene visualizzato con il valore espresso in **Binario** (vedi fig.26).
Per tornare alla situazione di fig.25 bisogna cliccare sul pulsante contrassegnato dalla lettera **H**.

RILEVAMENTO degli ERRORI

Per facilitare il rilevamento di **errori** durante la simulazione sono stati aggiunti **numerosi controlli** sulla **validità** delle istruzioni che vengono via via eseguite e sono state previste **segnalazioni** apposite che informano il programmatore delle **anomalie** riscontrate.

A titolo di esempio, nelle figg.27-28 abbiamo riportato solo due delle molteplici indicazioni di anomalie previste dal simulatore, che, nel caso specifico, il **Compilatore Assembler** non avrebbe potuto segnalare dal momento che, pur errate logicamente, le istruzioni sono **formalmente corrette**.
Cliccando su **OK** è possibile continuare nella simulazione e verificare così fino in fondo l'esattezza del programma.

SEGNALI in USCITA e in INGRESSO

Da ultimo, è stata aggiunta la possibilità di **convertire** i **segnali** in **uscita**, ottenuti durante una simulazione e memorizzati in un file **.DAT**, in **segnali** in **ingresso**, memorizzandoli in un file **.CMD** da utilizzare in una successiva simulazione.
E' inoltre possibile redirezionare i segnali da un piedino ad un altro sia nei files **.DAT** sia nei **.CMD**.
Questa possibilità di conversione è molto utile perché permette, partendo da un unico file contenente dei segnali, di generare **numerosi files** di segnali in ingresso che potranno così essere utilizzati in numerose simulazioni.

Cliccando su **Conv > CMD** nella videata principale (vedi fig.1) si attiva la funzione di Conversione ed appare la maschera visibile in fig.29.
Per selezionare il file da "convertire", cliccate su **Apri File** dal menu **File** di fig.30.
Nel nostro esempio abbiamo selezionato il file **eet60.dat** (vedi fig.31) generato da una precedente simulazione.
Cliccando su **OK** appare la maschera di fig.32, generata dalla selezione del file.
Sulla sinistra sono riportate tante caselle all'interno delle quali sono elencate le sigle dei piedini che contengono i segnali da convertire.
Al centro è visibile una barra di scorrimento con l'elenco completo di tutti i piedini del micro, mentre sulla destra vedete tante caselle vuote.
Cliccando rispettivamente nella casella di sinistra per selezionare il piedino e poi nella barra di centro per indicare su quale piedino devono essere "trasferiti" i segnali, si effettua la conversione.
Se osservate la fig.33 vedrete che, per quanto riguarda i primi tre piedini, abbiamo trasferito i segnali da **.DAT** a **.CMD** sugli stessi piedini, mentre per quanto concerne il quarto piedino, cioè **PB7** (evidenziato in giallo dal programma stesso), abbiamo trasferito i suoi segnali sul piedino **PC0**.
Completato il trasferimento dei segnali, è sufficiente scegliere il comando **Salva File CMD** dal menu File (vedi fig.34) per memorizzare la conversione in un file che, sempre nel nostro esempio, abbiamo chiamato **eet60.cmd** (vedi fig.35).

COSTO del PROGRAMMA

Questo nuovo ed aggiornato **software simulatore**, che sostituisce la precedente versione pubblicata sulla rivista N.190, è inserito nei **2 dischetti** floppy siglati **ST626/1 - ST626/2**.

Costo dei due dischetti con IVA inclusa ..... € 10,32

LA funzione SPI

Tutti i micro ST6260-65 possiedono una Serial-Synchronous Peripheral Interface conosciuta più comunemente come SPI, che può essere utilizzata per lo scambio dati tra due micro o per dialogare con una EEprom ecc. In questo articolo vi spieghiamo come settare correttamente i piedini e i registri coinvolti nella trasmissione e ricezione dei dati.

La SPI o Serial Peripheral Interface

La **SPI** è uno standard di trasmissione e ricezione **dati** in modalità **seriale sincrona** che può essere utilizzato, con opportune istruzioni di programma, per dialogare con una Eeprom esterna, con uno Shift register, con un Micro, per pilotare dei display ecc.
Uno dei vantaggi che offre la **SPI** riguarda il fatto che essendo la trasmissione e la ricezione dei dati completamente **automatica**, il microprocessore può continuare ad eseguire altre istruzioni.
È inoltre possibile effettuare una ricezione di dati da un integrato e ritrasmetterli ad un terzo pressoché simultaneamente, senza mai uscire cioè dalla stessa routine.

Ovviamente questa funzione viene attivata tramite il settaggio di particolari registri; in caso contrario i piedini coinvolti continueranno a svolgere i normali compiti per cui sono stati programmati.

È quindi importante conoscere bene le specifiche della funzione **SPI**, che può avere ben **6** differenti **configurazioni** o **modalità**.

– One wire Autoclocked Mode: viene utilizzato un solo piedino per l'invio dei dati ed il clock di trasmissione è prestabilito.

– Two wire Half Duplex Mode: vengono coinvolti 2 piedini: uno definisce il clock di trasmissione, l'altro, alternativamente, serve per la trasmissione e per la ricezione.

– Tree wire Half Duplex with Master/Slave select: è la modalità utilizzata dagli **ST6260/65** e vi sarà spiegata nel corso di questo articolo. Per il momento ci limitiamo a dire che in questa modalità vengono utilizzati **3 piedini** indicati con le sigle **Sin** (ingresso), **Sout** (uscita) e **Sck** (clock).

– **Tree wire Full Duplex Mode**: sono coinvolti 3 piedini ed è possibile la contemporanea trasmissione e ricezione dei dati.

– **Tree wire Full Duplex Mode with Clock Arbitration**: è il risultato della fusione delle modalità **2** e **4** e implica l'uso di 3 piedini.

– **Four wire Full Duplex Mode with Master/Slave select**: ha origine dalla fusione delle modalità **3** e **4** ed impiega 4 piedini.

L'interfaccia SPI negli ST6260 - ST6265

I microprocessori della serie **ST6260 - ST6265** utilizzano **solo** la configurazione:

Tree wire Half Duplex with Master/Slave select

Questa configurazione impiega **tre piedini** di **Porta C**: uno per la ricezione (**Sin**), uno per la trasmissione (**Sout**) ed uno per inviare o ricevere il segnale di clock (**Sck**) per il sincronismo dei dati.

Nel caso specifico dei micro **ST6260** ed **ST6265** i dati da trasmettere o da ricevere si trovano memorizzati in un apposito **shift register** da **1 byte**.
Ciò porterebbe a concludere che sia possibile trasmettere o ricevere un massimo di **8 bits** per **ciclo**, ma vedremo in seguito che non è proprio così.
Se si devono trasmettere più dati occorre scrivere una **subroutine** che esegua tanti cicli di trasmissione quanti sono i bytes da inviare; ad esempio, se volessimo trasmettere **32 bytes**, la subroutine dovrebbe eseguire **32 cicli** di trasmissione.

I termini **Master Mode** indicano che i dati vengono **inviati** dal **micro** ad un integrato esterno, uti-

per lo scambio DATI

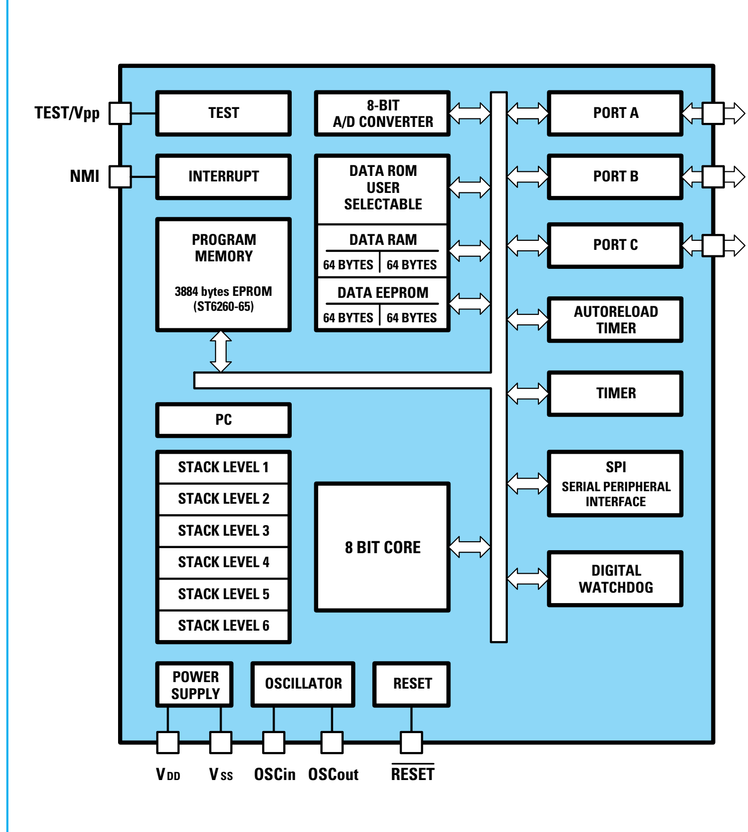


Fig.1 Schema a blocchi dei micro ST6260-65. I micro di questa serie possiedono al loro interno una comoda e pratica Serial-Synchronous Peripheral Interface, comunemente chiamata SPI, che consente lo scambio di dati tra due o più micro o tra un micro e altri integrati. La configurazione usata dagli ST6 permette di attivare una comunicazione Half Duplex su tre fili con selezione dispositivo Master e Slave.

lizzando il **clock** generato dallo stesso **micro**.
I termini **Slave Mode** indicano che i dati vengono **inviati** da un integrato esterno al micro, utilizzando il **clock** generato dall'integrato esterno.
Cercate di ricordare la differenza tra **Master** e **Slave**, perché nell'articolo citeremo frequentemente queste due modalità di trasmissione.

Vi sono altre parole che troverete spesso nella documentazione relativa ai microcontrollori e, tra queste, vale la pena spiegare subito i termini **Rising edge** e **Falling edge**.
Rising edge indica il **fronte di salita** dell'onda quadra del **clock** di trasmissione.
Falling edge indica il **fronte di discesa** dell'onda quadra del **clock** di trasmissione.

Tenete presente che per utilizzare al meglio lo standard SPI sarebbe preferibile che tutti gli integrati o i micro con i quali desiderate dialogare disponessero di tale funzione.
In teoria si può dialogare anche con integrati o micro sprovvisti della funzione **SPI**; in questi casi però potrebbe essere necessario utilizzare un piedino in più per attivare un eventuale segnale di conferma trasmissione o ricezione o per memorizzare i dati trasmessi (**latch**, **strobe**, **ecc.**).
Inoltre potrebbe essere necessario realizzare un certo numero di subroutines, perdendo così il vantaggio dell'esecuzione automatica della **SPI**.

PIEDINI e REGISTRI della SPI

Per attivare la funzione **SPI** sui piedini **PC2-PC3-PC4** di **Port_C** dei micro **ST6260-ST6265** occorre settare **4 registri**, diversamente questi tre piedini svolgeranno le normali funzioni di **I-O**.
Prima però di fornire le necessarie spiegazioni per la loro configurazione, dovete prendere confidenza con i termini e le abbreviazioni utilizzate.

Sin = Serial Input. È il piedino **PC2** di **Port_C** utilizzato per la **ricezione** dati.
Sout = Serial Output. È il piedino **PC3** di **Port_C** utilizzato per la **trasmissione** dati.
Sck = Serial Clock. È il piedino **PC4** di **Port_C** utilizzato per il segnale di clock di trasmissione o ricezione dati.

spmc = Spi Mode Register. È il registro che controlla tutta l'interfaccia **SPI**. È lungo **1 byte** ed è definito all'indirizzo **0E2H**. I suoi 8 **bits** da **7** a **0** verranno sempre indicati con le seguenti sigle.

7	6	5	4	3	2	1	0
Sprun	Spie	Cpha	Spclk	Spin	Spstrt	Efilt	Cpol

spdv = Spi Divide Register. È il registro che gestisce la velocità di trasmissione e il numero di bit da inviare e ricevere. E' lungo **1 byte** ed è definito all'indirizzo **0E1H**. I suoi 8 **bits** da **7** a **0** verranno sempre indicati con le seguenti sigle.

7	6	5	4	3	2	1	0
Spint	Div6	Div5	Div4	Div3	CD2	CD1	CD0

spda = Spi Data Register. È il registro nel quale vengono **memorizzati** i dati ricevuti o da trasmettere. Si tratta di uno **shift register** quindi la ricezione-trasmissione dei dati viene effettuata shiftando di un **bit** verso sinistra ad ogni fronte del clock. E' lungo **1 byte** ed è definito all'indirizzo **0E0H**. I suoi 8 **bits** da **7** a **0** verranno sempre indicati con le seguenti sigle.

7	6	5	4	3	2	1	0
D7	D6	D5	D4	D3	D2	D1	D0

misc = Miscellaneous Register. Solitamente questo registro, lungo **1 byte** e definito all'indirizzo **0DDH**, contiene dati per **settare** diverse funzioni. Nel nostro caso viene utilizzato solo il **bit 0**.

7	6	5	4	3	2	1	0
							M0

CONFIGURAZIONE dei PIEDINI

Ora cercheremo di spiegarvi nel modo più semplice possibile la configurazione iniziando da alcune note generali riguardanti i tre piedini di **Port_C**.
Innanzitutto va ricordato che i piedini **PC2 Sin** e **PC4 Sck** vengono utilizzati come normali piedini standard **I-O** quando il bit **4** del registro **spmc** siglato **Spclk** è a **0**. Lo stesso dicasi per il piedino **PC3 Sout** se il bit **0 M0** del registro **misc** è a **0**.
Se tramite i registri standard per la gestione di **port_c**, **pdir_c**, **popt_c** (vedi rivista **N.175/176**), il piedino **PC2 sin** viene configurato in **input**, qualsiasi segnale che entra sul piedino **PC2** (non importa se si è in **Master Mode** o in **Slave Mode**) viene automaticamente memorizzato nello **shift register spda**, indipendentemente dallo stato logico del bit **Spclk** del registro **spmc**.

Se settiamo a **1** il bit **M0** del registro **misc**, il piedino **PC3 Sout** viene configurato come **SPI push-pull output**, indipendentemente dai settaggi presenti sui registri **port_c**, **pdir_c** e **popt_c**.

Per trasmettere il clock (**Master Mode**), il piedino **PC4 Sck** deve essere settato come **push-pull output** nei registri **port_c**, **pdir_c** e **popt_c**, inoltre, va settato a **1** il bit **Spclk** del registro **spmc**.

Per ricevere il clock (**Slave Mode**), il piedino **PC4 Sck** deve essere settato come **input** nei registri **port_c**, **pdir_c e popt_c**, inoltre, deve essere settato a **0** il bit **Spclk** del registro **spmc**.
Con quest'ultima configurazione il piedino **PC4** può essere usato anche come piedino in **input**.

SINCRONISMO SPI

Nel paragrafo successivo chiariremo bit per bit il settaggio dei registri coinvolti nella gestione della **SPI**, ma prima di continuare è necessario illustrare con l'aiuto di qualche disegno, cosa significano i termini **rising edge**, **falling edge**, **polarità** e **fase**, perché la combinazione di questi dati ci consente di dialogare con la quasi totalità degli integrati che dispongono della funzione **SPI**.

Nelle figg.2-5 potete vedere i **4** tipi di **diagramma** di sincronismo **SPI** in cui è stata ipotizzata una trasmissione-ricezione di **8 bits**.

In queste figure sono richiamati i piedini **PC4-PC3** di **Port_C**, che abbiamo indicato con le sigle **Sck** e **Sout**, ed il bit **7 Sprun** del registro **spmc**. A proposito di questo bit è il caso di anticipare che **Sprun** sta per **Spi run**; in altre parole questo bit è lo start della funzione **SPI**.
Quando **Sprun** viene posto a **1** inizia la trasmissione o la ricezione dei dati, completata la quale il bit va automaticamente a **0**.

Per semplificare il disegno non abbiamo riportato il piedino **PC2 Sin**; d'altra parte la logica descritta per la trasmissione è identica in caso di ricezione.

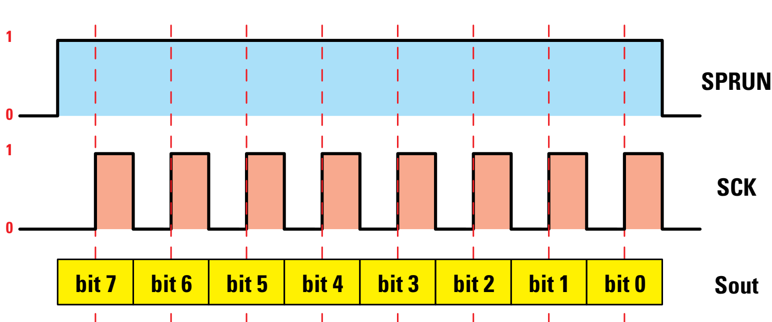


Fig.2 Diagramma di sincronismo SPI in cui è stata ipotizzata una trasmissione di 8 bits dal piedino Sout. Gli 8 cicli di trasmissione sono visibili sul piedino Sck, sul quale il segnale di clock dal livello logico 0 si porta al livello logico 1 e poi torna sul livello logico 0 per 8 volte. Quando la forma d'onda quadra è 0-1-0 la POLARITA' del clock è NORMALE e poiché la trasmissione inizia sul primo fronte di clock (fronte di salita), anche la FASE è NORMALE. Per avere una trasmissione con queste caratteristiche bisogna settare a 0 sia il bit cpol sia il bit cpha del registro spmc.

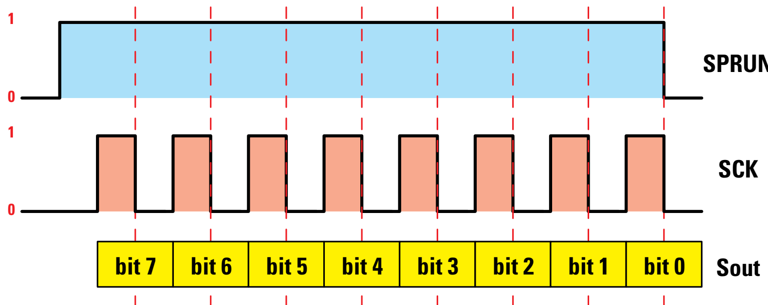


Fig.3 Diagramma di sincronismo SPI in cui è stata ipotizzata una trasmissione di 8 bits dal piedino Sout. Gli 8 cicli di trasmissione sono visibili sul piedino Sck, sul quale il segnale di clock dal livello logico 0 si porta al livello logico 1 e poi torna sul livello logico 0 per 8 volte. Quando la forma d'onda quadra è 0-1-0 la POLARITA' del clock è NORMALE e poiché la trasmissione inizia sul secondo fronte di clock (fronte di discesa), si ha uno SLITTAMENTO di FASE. Per avere una trasmissione con queste caratteristiche bisogna settare a 0 il bit cpol e settare a 1 il bit cpha del registro spmc.

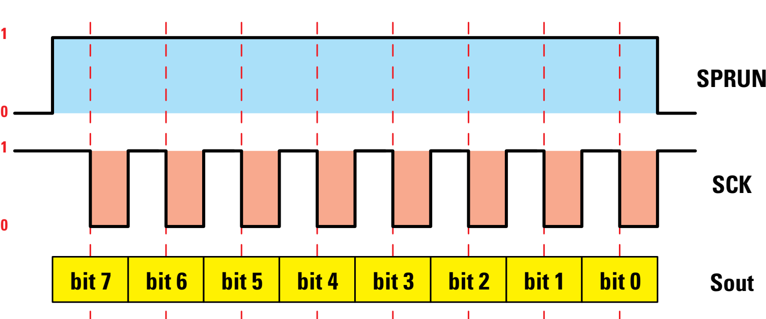


Fig.4 Diagramma di sincronismo SPI in cui è stata ipotizzata una trasmissione di 8 bits dal piedino Sout. Gli 8 cicli di trasmissione sono visibili sul piedino Sck, sul quale il segnale di clock dal livello logico 1 si porta al livello logico 0 e poi torna sul livello logico 1 per 8 volte. Quando la forma d'onda quadra è 1-0-1 la POLARITA' del clock è INVERTITA e poiché la trasmissione inizia sul primo fronte di clock (fronte di discesa), la FASE è NORMALE. Per avere una trasmissione con queste caratteristiche bisogna settare a 1 il bit cpol e settare a 0 il bit cpha del registro spmc.

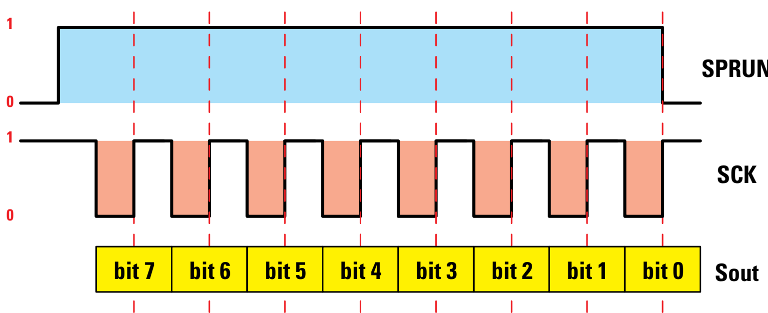


Fig.5 Diagramma di sincronismo SPI in cui è stata ipotizzata una trasmissione di 8 bits dal piedino Sout. Gli 8 cicli di trasmissione sono visibili sul piedino Sck, sul quale il segnale di clock dal livello logico 1 si porta al livello logico 0 e poi torna sul livello logico 1 per 8 volte. Quando la forma d'onda quadra è 1-0-1 la POLARITA' del clock è INVERTITA e poiché la trasmissione inizia sul secondo fronte di clock (fronte di salita), si ha uno SLITTAMENTO di FASE. Per avere una trasmissione con queste caratteristiche bisogna settare a 1 sia il bit cpol sia il bit cpha del registro spmc.

Osservando le figg.2-5 si può innanzitutto notare che il bit **Sprun** passa dallo stato logico **0** allo stato logico **1** (inizio della trasmissione o ricezione) e mantiene questo stato per **8 cicli** di **trasmissione-ricezione** dati. Gli 8 cicli di **clock** sono visibili come altrettante onde quadre sul piedino **Sck.**

Nelle figg.2-3 il segnale di **clock** parte con un **livello logico 0**, si porta a **1**, mantiene questo stato per un breve periodo dopodiché si riporta a **0.**

Quando si è in presenza di questa forma d'onda quadra (**0 - 1 - 0**) si parla di **polarità** di **clock normale**. Ogni volta che il clock dal **livello logico 0** si porta sul **livello logico 1** si ha un fronte di **salita** ed ogni volta che dal **livello logico 1** si porta sul **livello logico 0** si ha un fronte di **discesa**.

Quando la trasmissione e/o ricezione avviene sul **1° fronte** di **clock** (che in polarità normale è il fronte di **salita**) si parla di **fase normale**, quando avviene sul **2° fronte** di **clock** (che in polarità normale è il fronte di **discesa**) si parla di **shift** di fase.
Se siamo in trasmissione viene prelevato il livello logico contenuto nel registro **spda** partendo dal **bit 7** ed inviato sul piedino **Sout**. Se siamo in ricezione viene letto il livello logico presente sul piedino **Sin** e memorizzato nel registro **spda** partendo dal bit **0**.
Dopo **8 cicli**, quando il piedino **Sck** passa dal **livello logico 1** al **livello logico 0**, automaticamente termina la trasmissione o ricezione dei **dati**.

Nella fig.2 la trasmissione-ricezione dei dati avviene in **polarità normale** e ha inizio sul **1° fronte** di **clock**, quindi è in **fase normale**.

Ora osserviamo la fig.3 dove, come abbiamo appena detto, il segnale di **clock** sul piedino **Sck** parte con un **livello logico 0** e si porta a livello logico **1** (polarità normale), ma la trasmissione-ricezione non inizia sul **1° fronte**, bensì sul **2° fronte** di **clock**, è cioè shiftata di fase.
Se siamo in trasmissione viene prelevato il livello logico contenuto nel registro **spda** partendo dal **bit 7** e inviato sul piedino **Sout**. Se siamo in ricezione viene letto il livello logico presente sul piedino **Sin** e memorizzato nel registro **spda** partendo dal bit **0**.
Dopo **8 cicli**, quando il piedino **Sck** passa dal **livello logico 1** al **livello logico 0**, automaticamente termina la trasmissione o ricezione dei **dati**.

Nella fig.3 la trasmissione-ricezione dati avviene in **polarità normale**, ma ha inizio sul **2° fronte** di **clock** quindi è in **slittamento** di **fase**.

Ora osserviamo le figg.4-5, in cui il segnale di **clock** sul piedino **Sck** parte dal **livello logico 1**, si porta a **0** e poi ritorna a **1**. Quando si è in presenza di questa forma d'onda quadra (**1 - 0 - 1**) si parla di **polarità** di **clock invertita**.
Ogni volta che il clock dal **livello logico 1** si porta sul **livello logico 0** si ha un fronte di **discesa** ed ogni volta che dal **livello logico 0** si porta sul **livello logico 1** si ha un fronte di **salita**.

Quando la trasmissione e/o ricezione avviene sul **1° fronte** di **clock** (che in polarità invertita è il fronte di **discesa**) si parla di **fase normale**, quando avviene sul **2° fronte** di **clock** (che in polarità invertita è il fronte di **salita**) si parla di **shift** di fase.
Se siamo in trasmissione viene prelevato il livello logico contenuto nel registro **spda** partendo dal **bit 7** e inviato sul piedino **Sout**. Se siamo in ricezione viene letto il livello logico presente sul piedino **Sin** e memorizzato nel registro **spda** partendo dal bit **0**.
Dopo **8 cicli**, quando il piedino **Sck** passa dal **livello logico 0** al **livello logico 1**, automaticamente termina la trasmissione o ricezione dei **dati**.

Nella fig.4 la trasmissione-ricezione dati avviene in **polarità invertita** e ha inizio sul **1° fronte** di **clock**, quindi è in **fase normale**.

Per finire passiamo alla fig.5 dove il segnale di **clock** sul piedino **Sck** parte sempre con un **livello logico 1** e si porta a livello logico **0**, ma la trasmissione-ricezione non inizia sul 1° fronte, bensì sul **2° fronte** di **clock**, è cioè shiftata di fase.
Se siamo in trasmissione viene prelevato il livello logico contenuto nel registro **spda** partendo dal **bit 7** e inviato sul piedino **Sout**. Se siamo in ricezione viene letto il livello logico presente sul piedino **Sin** e memorizzato nel registro **spda** partendo dal bit **0**.
Dopo **8 cicli**, quando il piedino **Sck** passa dal **livello logico 0** al **livello logico 1**, automaticamente termina la trasmissione o ricezione dei **dati**.

Nella fig.5 la trasmissione-ricezione dati avviene in **polarità invertita**, ma ha inizio sul **2° fronte** di **clock**, quindi è in **slittamento** di **fase**.

CONFIGURAZIONE dei REGISTRI

Il registro **spmc** (**Spi Mode Register**) è quello che in pratica controlla tutta la gestione **SPI**.

7	6	5	4	3	2	1	0
Sprun	Spie	Cpha	Spclk	Spin	Spstrt	Efilt	Cpol

Sprun bit 7 = **Spi run**. Quando viene posto a **livello logico 1** ha inizio la trasmissione dati (**Master Mode**) o la ricezione dati (**Slave Mode**). Alla fine della trasmissione o della ricezione questo bit si porta automaticamente a **livello logico 0**.
Se viene forzato a **livello logico 0** dal programma, si interrompe la trasmissione o la ricezione.
Quando va a **0** può generare una richiesta di **Interrupt** se il bit **6** (**Spie**) è settato a 1 ed è stata attivata la routine di Interrupt di SPI nel registro ior.
Utilizzato assieme al bit **2** (**Spstrt**) stabilisce una condizione di **start** in ricezione o trasmissione.
In questo caso la trasmissione-ricezione **dati** ha inizio solo se viene rilevato un segnale esterno con un fronte di salita (**rising edge**) sul piedino **PC2**.

Spie bit 6 = **Spi Enable Interrupt**. Quando questo bit è settato a **1** abilita l'interrupt **SPI**; quando è resettato, cioè posto a **0**, lo disabilita.

Cpha bit 5 = **Clock Fase Selection.** Quando è settato a **0** si ha una **fase normale** di clock (vedi figg.2 e 4), quando è settato a **1** si ha lo slittamento di fase (vedi figg.3 e 5).

Spclk bit 4 = **Base Clock Selection**. Questo bit selezione il **clock**. In pratica dice al microcontrollore se il **clock** sarà **interno** o **esterno**.
Se è settato a **0** e nel contempo il **PC4 Sck** è configurato **input**, viene attivata la **ricezione** (**Slave Mode**) pertanto il clock viene prelevato **esternamente** dall'integrato che invia i dati.
Se invece è settato a **1** e contemporaneamente il **PC4 Sck** è stato configurato in **output push-pull**, viene attivata la **trasmissione** (**Master Mode**) pertanto il clock risulta **interno**.
In questo caso il clock viene ricavato dalla frequenza del **quarzo** diviso **13** ed ulteriormente diviso per il valore contenuto in alcuni bits del registro **spdv**, come spiegheremo più avanti.

Spin bit **3** = **Input Selection**. Questo bit gestisce la selezione di **input**. Se è settato a **1** abilita il trasferimento dei dati ricevuti da **PC2 Sin** nello shift register **spda** e quindi al termine della ricezione questo registro conterrà i **dati** ricevuti.
Se è settato a **0** il trasferimento viene disabilitato e i dati letti su **PC2 Sin** dovranno essere trattatati direttamente dalle istruzioni di programma. In questo caso **PC2 Sin** si comporta praticamente come un normale piedino.

Spstrt bit **2** = **Start Selection**. Questo bit viene utilizzato per gestire la selezione di **Start**, possibilità questa che può risultare molto utile in determinati casi. Infatti, se questo bit è settato a **0**, la fase di trasmissione o di ricezione **SPI** ha inizio quando viene posto a **1** il bit **Sprun**.
Se invece viene posto a **1** e contemporaneamente si setta a **1** anche il bit **Sprun**, la **ricezione** o la **trasmissione** ha inizio solamente quando viene ricevuto un fronte di **salita** esterno su **PC2 Sin**, cioè un segnale **rising edge**.
In questo modo è possibile pilotare esternamente l'inizio di una trasmissione-ricezione **SPI**. Una volta che è iniziata, la trasmissione-ricezione continua anche se il segnale su **PC2 Sin** viene resettato.

Efilt bit **1** = **Enable Filter**. Questo bit serve per abilitare o disabilitare un filtro anti-rumore sui piedini **PC2 Sin** e **PC4 Sck**. Se è settato a **0** il filtro è disabilitato, se è settato a **1** è abilitato.
In fase di ricezione dati capita di frequente che sui piedini interessati si trovino disturbi di qualsiasi natura che potrebbero falsare i dati ricevuti.
Quando è abilitato, questo filtro elimina ogni impulso rilevato che sia più piccolo di **1-2** periodi del clock principale del micro.
In pratica ad ogni clock interno del micro viene letto una prima volta il dato sul piedino, il clock successivo viene riletto e se il dato è lo stesso viene accettato. Se alla seconda lettura il dato risulta invece diverso, vengono ignorati entrambi perché considerati disturbi.
Così, ad esempio, se il micro **ST6265** lavora ad una frequenza di **8 MHz**, avremo un filtraggio ogni **125 nanosecondi** ed un possibile ritardo sulla conferma di un segnale fino a **250 nanosecondi**.

Cpol bit **0** = **Clock Polarity**. Questo bit gestisce la polarità del **clock** sul piedino **Sck**. Se è settato a **0** la **polarità** è **normale** (vedi figg.2-3), se è settato a **1** la polarità è **invertita** (vedi figg.4-5).

Il registro **spdv** o **Spi Divide Register** è il registro che gestisce il numero dei bits da inviare-ricevere e che permette di configurare la frequenza di trasmissione. Non è possibile scrivere o variare i valori in questo registro quando **Sprun** è settato a **1**, vale a dire quando è attiva la trasmissione o la ricezione dei dati.

7	6	5	4	3	2	1	0
Spint	Div6	Div5	Div4	Div3	CD2	CD1	CD0

Spint bit **7** = **Input Flag**. Questo bit è un **read an clean only**, ciò significa che lo possiamo solo resettare a **0** o leggere. Infatti viene settato a **1** dal micro solo quando viene riscontrata la fine della ricezione o della trasmissione **SPI** ed è stata attivata una richiesta di **Interrupt**, come abbiamo spiegato nel registro **spmc** a proposito del bit **6**.
Questo bit deve poi essere azzerato dal programma una volta che è sia stata eseguita la sub-routine attivata dall'Interrupt sopracitato.

Div6-Div3 bits **6-5-4-3** = **Burstmode Bit Clock Period**. Questi bits servono per configurare il numero dei **bits** per ogni ciclo di **SPI** da ricevere o trasmettere. Naturalmente, siccome la trasmissione-ricezione avviene sul fronte del clock, in pratica si configura così anche il numero dei clock per quel ciclo di trasmissione-ricezione.
Nella **Tabella N.1** è riportata la loro configurazione. Per ogni ciclo è possibile trasmettere-ricevere un massimo di **8 bits** in quanto il registro dal quale vengono trasmessi è lungo solo **1 byte**.
È però possibile configurare la tabella per una trasmissione fino a **15 clock** per **ciclo**, ma in questo caso sui fronti di clock eccedenti verranno inviati i **livelli logici 0** subentrati ai valori presenti nel registro **spda** per effetto dello shiftamento durante la trasmissione.

Ad esempio, se configuriamo questi piedini per inviare **11 clock** per ciclo, con i primi **8 clock** verranno trasmessi i primi **8 bits** così come si trovano nel registro **spda** (vedi fig.6), mentre per i successivi **3 clock** verranno trasmessi i **livelli logici 0**.
Questo significa che se il contenuto del **registro spda** del micro che **trasmette** era:

7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	1

dopo **11 clock** di trasmissione, il registro **spda** del micro che ha ricevuto i dati conterrà questi valori:

7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0

perché, come abbiamo avuto già modo di ricordare e come spiegheremo più dettagliatamente in seguito, i bits, man mano che vengono ricevuti, shiftano verso sinistra nel registro spda.

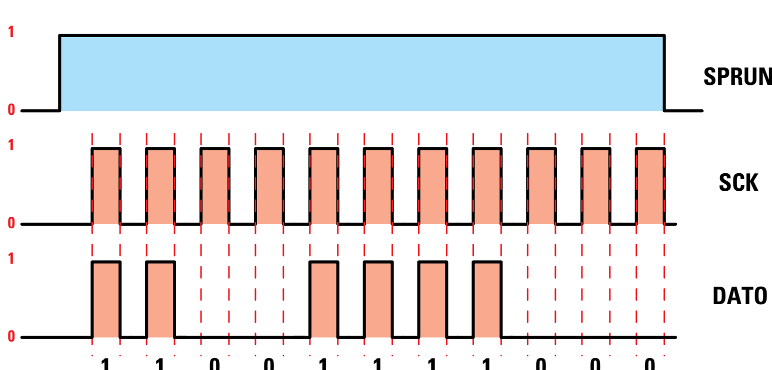


Fig.6 Diagramma di sincronismo SPI in cui abbiamo ipotizzato una trasmissione di 11 clock per ciclo. Con i primi 8 clock vengono trasmessi gli 8 bits contenuti nel registro spda, mentre per i successivi tre clock vengono trasmessi dei livelli logici 0. Poiché i dati, man mano che vengono ricevuti, shiftano nel registro spda del micro ricevente verso sinistra, dopo 11 clock il registro spda conterrà solo gli ultimi 8 bits trasmessi. La quantità dei bits da inviare deve essere identica alla quantità dei bits da ricevere, cioè i registri spdv del Master e dello Slave devono avere la stessa configurazione.

Nota: ovviamente il registro **spdv** deve avere la stessa configurazione sia in **master** sia in **slave**, cioè il numero dei bits da inviare e ricevere deve essere lo stesso.

TABELLA N.1

DV6	DV5	DV4	DV3	numero bits
0	0	0	0	riservato
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	1	14
1	1	1	1	15

Nota: La quantità di bits da trasmettere o ricevere ad ogni ciclo è legata naturalmente al tipo di problematica che si vuole risolvere ed al tipo di integrati con i quali si vuole dialogare. È perciò necessario consultare sempre i **data-sheet** degli integrati o dei micro per non trovarsi poi in situazioni imprevedibili da cui non si sa più come uscire.

CD2-CD0 bits **2-1-0** = **Clock Rate Selection**. Questi bits servono per ottenere il numero da utilizzare per configurare la frequenza di clock di trasmissione. In sostanza dalla configurazione appropriata di questi tre bits (vedi **Tabella N.2**) otteniamo il **divisore**.

TABELLA N.2

CD2	CD1	CD0	DIVISORE
0	0	0	divide x 1
0	0	1	divide x 2
0	1	0	divide x 4
0	1	1	divide x 8
1	0	0	divide x 16
1	0	1	divide x 32
1	1	0	divide x 64
1	1	1	divide x 256

Dividendo la **frequenza** del **quarzo** utilizzato dal **micro** prima per **13** poi per questo **divisore** si ottiene la frequenza di clock di trasmissione dati:

Fclock = (Fquarzo in Hz : 13) : N.Divis.

dove:

Fclock è la frequenza del clock di trasmissione,
Fquarzo è la frequenza del quarzo in hertz,
13 è un numero fisso,
N.Divis. è il numero del divisore (vedi Tabella n.2).

Poniamo ad esempio il caso che un programma richieda una frequenza di clock (**Fclock**) approssimativa di **9600 bit rate**.
Per trasmettere i **dati** da un micro **ST6265** che utilizza un quarzo da **8 MHz** ad un dispositivo esterno, potremo calcolare il **numero** del **divisore** utilizzando questa seconda formula:

N.Divis. = (Fquarzo in Hz : 13) : Fclock
(8.000.000 : 13) : 9600 = 64,10256

Poiché i decimali non vanno considerati, per poter ottenere una frequenza approssimativa di clock di **9600 bit rate** dovremo dividere la frequenza del quarzo per **13** e successivamente per **64**.
Consultando la **Tabella N.2** siamo ora in grado di sapere che per ottenere il **divisore 64**, i tre bits devono essere così settati:

CD2	CD1	CD0
1	1	0

Nota: anche in questo caso, come nel precedente, sarà necessario consultare attentamente i data-sheet dei dispositivi usati per poter selezionare correttamente la frequenza di clock ottimale o necessaria con cui operare la trasmissione dati.

Il registro **spda** o **Spi Data Register** è il registro in cui vengono memorizzati i dati ricevuti e i dati da trasmettere.

7	6	5	4	3	2	1	0
D7	D6	D5	D4	D3	D2	D1	D0

Poiché è uno shift register, i dati vengono trasmessi e ricevuti a cominciare sempre dal **Msb**, cioè dal bit col valore significativo più alto.
I dati vengono ricevuti e/o trasmessi da questo registro ad ogni fronte (**edge**) di clock compatibilmente a quanto settato come **polarità** e **fase** nei bits **Cpol** (**0**) e **Cpha** (**5**) del registro **spmc**, di cui già abbiamo parlato. Non è possibile modificare il contenuto di questo registro quando è attiva una trasmissione o una ricezione.

D7-D0 bits **7-6-5-4-3-2-1-0** = **Data Bits**. Questi bits contengono i valori ricevuti o da trasmettere.

Poiché il registro **spda** è uno shift register, è necessaria una certa cautela nell'utilizzarlo.
Mettiamo ad esempio il caso che si vogliano trasmettere ad un altro micro solo **4 bits** e che il valore contenuto in **spda** prima della trasmissione sia **179.** La rappresentazione binaria è:

D7	D6	D5	D4	D3	D2	D1	D0
1	0	1	1	0	0	1	1

Quando, al primo fronte di clock, inizia la trasmissione, il registro shifta di un bit verso sinistra, quindi **D0** assume valore **0** ed il valore contenuto in **D7** viene inviato sul piedino **PC3 sout**, che lo trasmette al micro slave.

Questo micro lo riceve sul piedino **PC2 sin** e lo memorizza nel suo registro **spda**, partendo dal bit **D0.** Dopo la trasmissione del primo bit, la rappresentazione binaria del registro spda del micro **master** diventerà:

D7	D6	D5	D4	D3	D2	D1	D0
0	1	1	0	0	1	1	0

Mentre quella del registro spda del micro **slave** è:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	1

Al secondo fronte di clock, il registro shifta nuovamente verso sinistra di un bit, quindi **D0** assume nuovamente valore **0** e **D7** (che aveva assunto il valore di **D6** dopo il primo fronte di clock) viene inviato per essere trasmesso al piedino **PC3 sout.**
Il micro slave riceve il dato sul piedino **PC2 sin** e lo memorizza nel registro **spda** sempre a partire dal bit **D0**, shiftando in **D1**, cioè verso sinistra, il valore prima contenuto in **D0.**
Il ciclo descritto per la trasmissione dei primi due bits si ripete anche per i rimanenti 2 bits, come esemplificato in fig.7.

A fine trasmissione il registro spda del micro master ha questa rappresentazione binaria:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	1	0	0	0	0

Come potete vedere il contenuto di questo registro è ora **48.** Avendo trasmesso **4 bits**, il registro è shiftato di 4 posizioni verso sinistra e i bits a destra sono stati riempiti con degli **0.**

A sua volta il registro **spda** del **slave** ha la seguente configurazione binaria:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	0	1	1

Avendo ricevuto **4 bits**, i primi quattro bits a destra hanno il valore indicato, che equivale a **11.**

Nel caso in cui si trasmetta un numero inferiore a **8 bits**, ricordate di fare molta attenzione perché, come avete visto dall'esempio, i 4 bits trasmessi avevano un valore di **88**, mentre i 4 bits ricevuti hanno un valore di **11.**

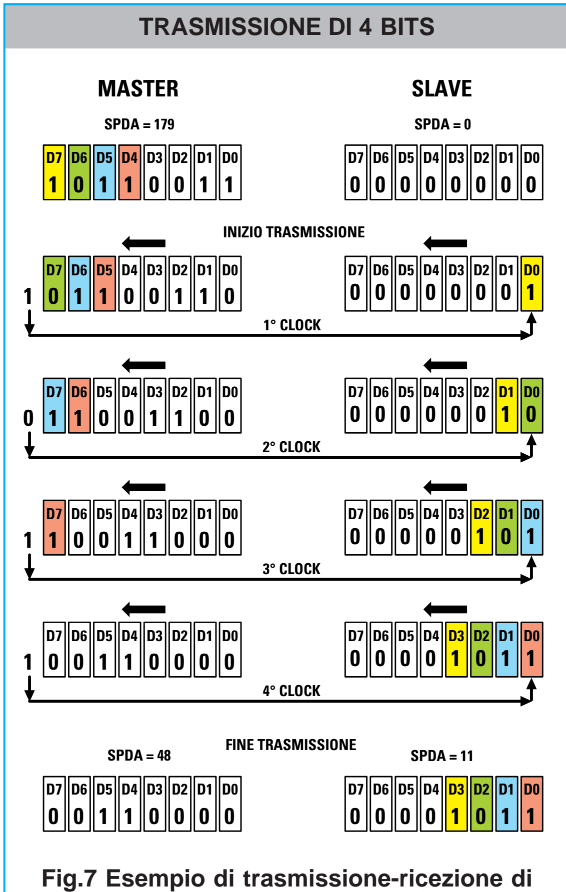


Fig.7 Esempio di trasmissione-ricezione di 4 bits. I dati vengono inviati a partire dal bit 7 e ricevuti a partire dal bit 0. Ad ogni clock i dati shiftano verso sinistra.

Per concludere, nel caso in cui siate in modalità master e dobbiate caricare un valore nel registro **spda**, non usate l'istruzione **LDI**, perché non verrebbe caricato nulla e trasmettereste **8 bits** a zero. E' invece necessario caricare prima il valore nell'accumulatore **A** e successivamente muovere il contenuto dell'accumulatore in **spda**.
Se ad esempio volessimo trasmettere **139**, la sequenza esatta delle istruzioni sarebbe:

```
ldi     a,139
ld      spda,a
```

Il registro **misc** o **Miscellaneous Register** è un registro comune a molti livelli di micro **ST6** e quindi contiene dati per settare varie funzioni.
Per la **SPI** viene utilizzato solo il bit 0.

7	6	5	4	3	2	1	0
							M0

M0 bit 0 = **Mode Sout**. Se il piedino **PC3** viene settato a **1** come **Sout** per la funzione **SPI** sarà attivo per la trasmissione dati. Se settato a **0** il **PC3** diventa un normale piedino di **I-O** di **Port_C**.

A questo punto, conclusa la trattazione teorica sulla SPI, non ci rimane che suggerirvi di realizzare subito le **tre** semplici **interfacce periferiche** pubblicate in questo stesso numero per poter provare i **programmi dimostrativi**, da noi appositamente scritti, sulla trasmissione e ricezione di dati con lo standard **SPI** utilizzato dai micro **ST62/65**.

Fig.8 Sulla rivista N.192 abbiamo presentato un valido programmatore per i micro ST62/60 e ST62/65 che vi servirà per programmare questi nuovi microprocessori.

Sappiamo per esperienza che le spiegazioni puramente teoriche sono solitamente molto noiose e quasi sempre difficili da capire e da assimilare.
Per questo, quando è possibile, cerchiamo di affiancare ad esse la realizzazione di circuiti pratici che, consentendoci di **vedere** quello che la teoria ci spiega, rendono tutto più comprensibile.

Abbiamo quindi scritto alcuni **programmi dimostrativi** sulla trasmissione e ricezione di dati con lo standard **SPI** utilizzato dai micro **ST62/65**, che si possono **vedere** in funzione montando **3** semplici **interfacce periferiche**.

La **prima** interfaccia, siglata **LX.1380**, va innestata nella scheda **bus** siglata **LX.1329** (vedi fig.1), apparsa sulla rivista N.192, che molti tra voi avranno sicuramente già montato per poter testare le funzioni **PWM** ed **EEprom**.

quarzo da **8 MHz**, un **pulsante**, un **deviatore**, un **trimmer** e **8 diodi led**, che vi consentiranno di sapere quale livello logico **0-1** è presente sulle uscite del **microprocessore** che trasmette i dati.
A seconda del programma che memorizzerete nel micro, potrete effettuare una trasmissione o una ricezione dati tra due micro **ST62/65**.

Sulla **terza** interfaccia siglata **LX.1382** (vedi fig.11), che deve essere collegata tramite una **piattina** all'interfaccia siglata **LX.1380**, dovrete montare quattro **shift register HC/Mos** tipo **HCF.4094** o **MC.14094** (vedi **IC1-IC2-IC3-IC4**), le reti resistive siglate **R1-R2-R3-R4**, tre **display** e otto **diodi led**.
Sui **display** apparirà il **dato** ricevuto espresso col sistema **decimale** e sui **diodi led** il corrispondente valore espresso col sistema **binario**.
Quando il numero **binario** corrisponde a **255** decimale tutti i diodi led sono **accesi**, quando corrisponde a **0** tutti i diodi led sono **spenti**.

CIRCUITI test per la SPI

Grazie ai cinque programmi dimostrativi, che vi forniamo su richiesta insieme alle tre semplici interfacce presentate in queste pagine, non solo potrete sperimentare subito la trasmissione-ricezione dati con lo standard SPI utilizzato dai micro ST62/65, ma avrete anche a disposizione delle utili "schede di valutazione" per testare immediatamente se i programmi scritti da voi trasmettono e ricevono i dati correttamente.

Come potete vedere dalla fig.4, l'interfaccia **LX.1380** è molto semplice: sul suo circuito stampato vanno infatti montati un solo **dip-switch** provvisto di **8** levette (vedi **S1**), un **pulsante** e due **connettori** maschi a **5+5** terminali che vi serviranno per collegare, con le apposite piattine, le due interfacce **LX.1381 - LX1.382**.

Se non disponete ancora del **bus** siglato **LX.1329**, potrete richiederlo al nostro indirizzo assieme alla rivista **N.192**, nella quale potete trovare anche il **Programmatore** per i **micro ST62/60 - ST62/65**.

La **seconda** interfaccia, siglata **LX.1381**, va collegata, sempre tramite la **piattina** che vi forniamo già cablata e completa di connettori femmina, sulla prima interfaccia siglata **LX.1380**.

Come potete vedere dalla fig.7, sull'interfaccia **LX.1381** vanno montati due zoccoli, uno per il micro **ST62/65** e l'altro per la rete resistiva **R3**, un

Con questa scheda i dati trasmessi dal micro **ST62/65** inserito nel **bus LX.1329** o nell'interfaccia **LX.1381** vengono ricevuti dai quattro shift register e visualizzati sui display e sui diodi led.

REALIZZAZIONE PRATICA LX.1380

Sul circuito stampato siglato **LX.1380** dovete montare tutti i componenti come disposti nel disegno visibile in fig.4.
Per iniziare vi consigliamo di inserire sul lato opposto dello stampato in **basso** il **CONN.1** a 1 fila 24 terminali e in **alto** i due connettori a 1 fila 4 terminali, che vi consentiranno di collegare in modo stabile questa scheda all'interfaccia bus **LX.1329**.
Proseguendo inserite anche i due connettori maschi a 5+5 terminali (vedi **CONN.2**) rivolgendo l'**asola** di riferimento verso l'alto.
Al centro stagnate il pulsante **P1** e sotto questo il **dip-switch** siglato **S1**, rivolgendo il lato del corpo che riporta la scritta **ON** verso l'alto (vedi fig.4).

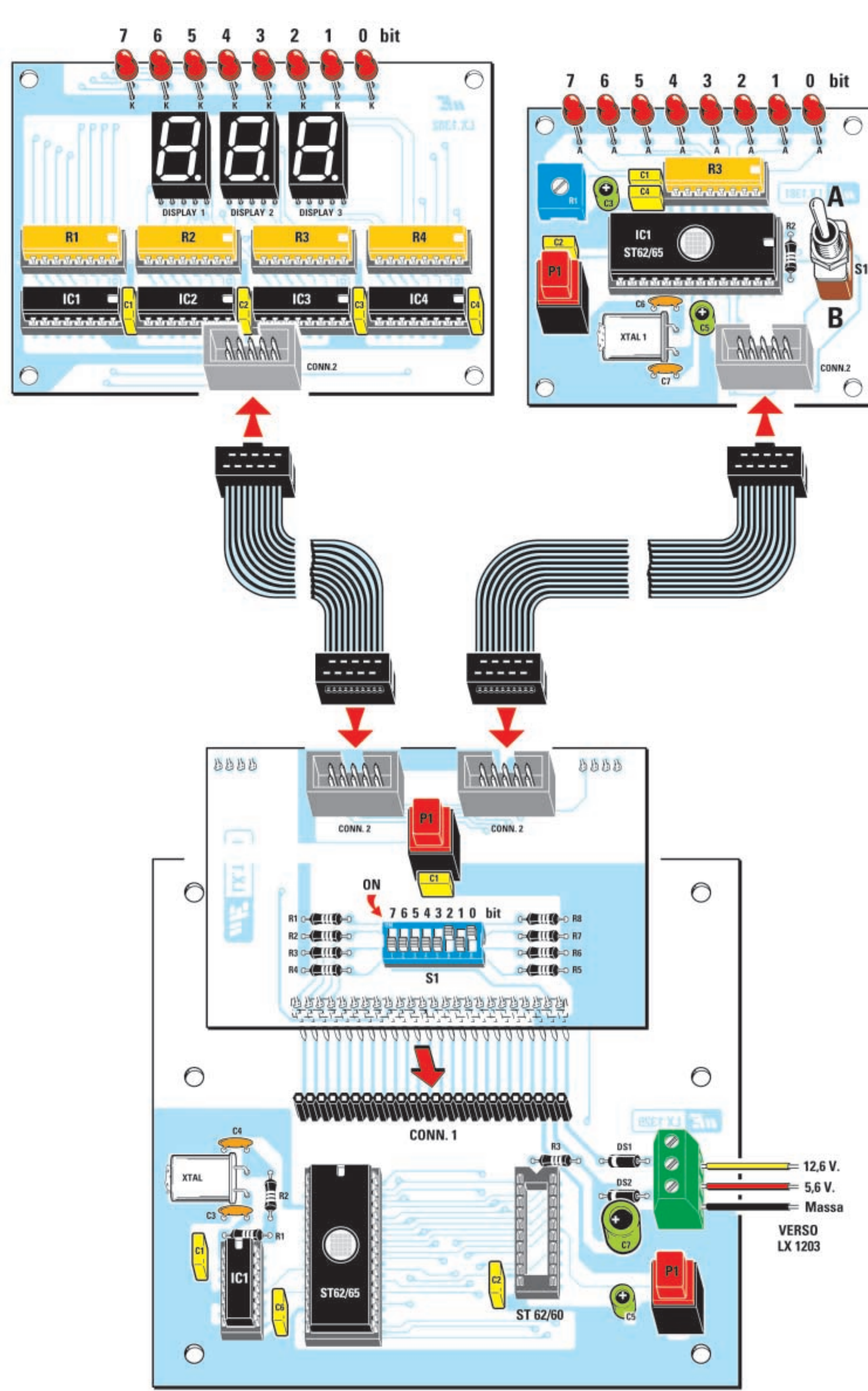


Fig.1 Sulla scheda bus LX.1329 presentata nella rivista N.192, che sicuramente avrete già montato, dovete inserire un micro ST62/65. Sopra a questa scheda andrà innestata l'interfaccia LX.1380, che vi permetterà di dialogare con le altre due interfacce siglate LX.1381-LX.1382. La scheda LX.1329 va alimentata tramite la scheda LX.1203.

Le 8 levette del dip-switch equivalgono agli 8 bits da configurare per la trasmissione dati.
A questo proposito ricordate che la levetta all'estrema destra rappresenta il **bit 0** e quella all'estrema sinistra il **bit 7**.

REALIZZAZIONE PRATICA LX.1381

Sul circuito stampato siglato **LX.1381** vanno montati tutti i componenti come visibile in fig.7.
Iniziate inserendo i due zoccoli per la rete resisteva **R3** e per il micro **ST62/65** (vedi **IC1**), poi in basso a destra inserite il **CONN.2** rivolgendo la sua **asola** di riferimento verso l'alto.
In alto stagnate gli 8 **diodi led** rivolgendo il terminale **più lungo** (vedi **A**nodo) verso **R3**.

I dati ricevuti dal micro vengono visualizzati tramite gli 8 diodi led e, come già spiegato a proposito dell'interfaccia **LX.1380**, tenete presente che il diodo led più a **destra** corrisponde al **bit 0** e il diodo led più a **sinistra** al **bit 7**.
Quindi il diodo led all'estrema destra visualizzerà il dato configurato con la levetta più a destra del dip-switch della scheda **LX.1380**, e così via.

Per completare il circuito stagnate il deviatore a levetta **S1**, il pulsante **P1**, il quarzo, il trimmer **R1**, i pochi condensatori e l'unica resistenza, così come appare nel disegno in fig.7.
Ricordatevi che le tacche di riferimento a **U** della rete resistiva e del micro vanno rivolte a **destra**.

Il trimmer **R1** è stato inserito per applicare sull'ingresso analogico **PA0** un valore di tensione variabile da **0** a **5 volt**, in modo da farvi vedere il valore da **0** a **255** della conversione eseguita dall'**A/D converter** in un numero **binario**.

REALIZZAZIONE PRATICA LX.1382

Sul circuito stampato siglato **LX.1382** inserite come primi componenti i 4 zoccoli per le **reti resistivi** e i 4 zoccoli per gli integrati **shift register**.
In basso al centro inserite il **CONN.2** rivolgendo la sua **asola** di riferimento verso l'alto.
In alto stagnate gli 8 **diodi led** rivolgendo il terminale **più corto** (vedi **K**) verso i display.
Come già spiegato a proposito del circuito precedente, anche in questo caso il diodo led più a **destra** visualizza il dato corrispondente al **bit 0** ed il diodo led più a **sinistra** quello del **bit 7**.
Per completare il montaggio innestate i tre display rivolgendo il **punto decimale** verso il basso.

I PROGRAMMI nel DISCHETTO DF.1380

Il dischetto siglato **DF.1380**, che forniamo a parte su richiesta, contiene **5 programmi**:

PROG01 – questo programma contiene un esempio di trasmissione dati tramite **SPI** da un micro **ST62/65** a **4 shift register** a **8 bits**. Il micro in cui è stato caricato questo programma deve essere innestato nella scheda **LX.1329** e i dati trasmessi vengono visualizzati sulla scheda **LX.1382**.

TXPG02-RXPG02 – questi programmi contengono un esempio molto **semplice** di trasmissione dati fra 2 micro ST62/65 tramite funzione **SPI**. Il micro in cui è stato caricato il programma **Master** (**TXPG02**) va inserito nella scheda **LX.1329**, mentre il micro in cui è stato caricato il programma **Slave** (**RXPG02**) nella scheda **LX.1381**.

TXPG03-RXPG03 – questi programmi contengono un esempio abbastanza **complesso** di trasmissio-

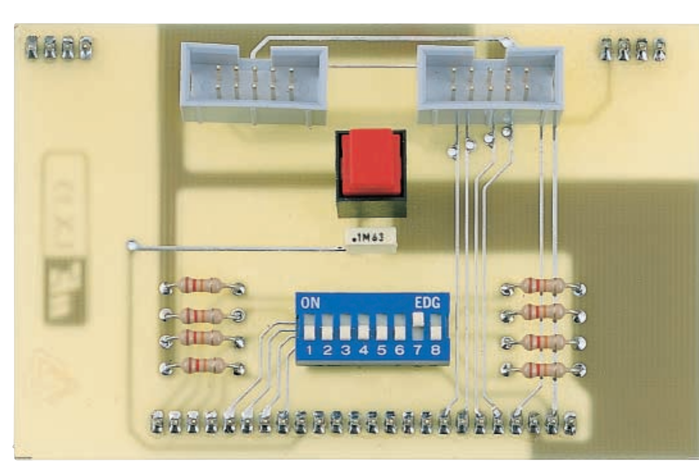


Fig.2 Questa è la foto della scheda LX.1380 provvista di un dip-switch, un pulsante e due connettori maschio per poter collegare le schede LX.1381-LX.1382 (vedi fig.1).

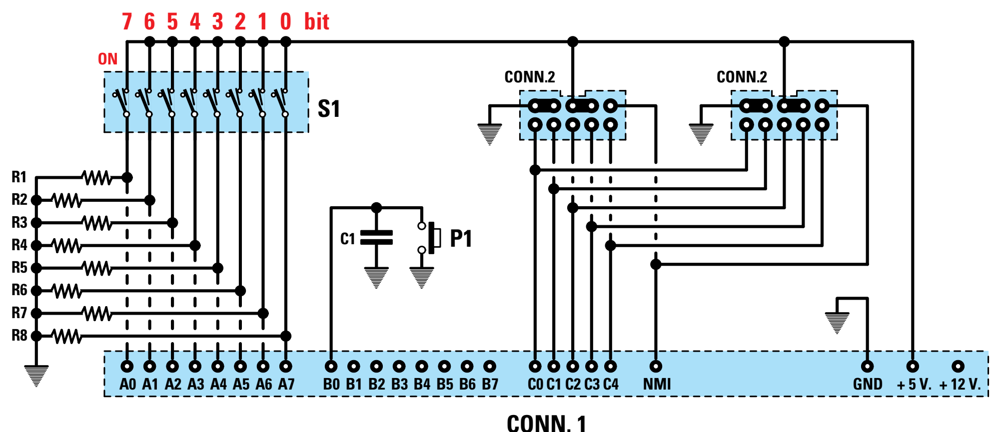


Fig.3 Schema elettrico dell'interfaccia LX.1380. Quando le levette del dip-switch S1 sono poste in posizione ON (vedi fig.4), commutano sul livello logico 1 i piedini A0-A7 di porta A.

ELENCO COMPONENTI LX.1380

R1 = 22.000 ohm
R2 = 22.000 ohm
R3 = 22.000 ohm
R4 = 22.000 ohm
R5 = 22.000 ohm
R6 = 22.000 ohm
R7 = 22.000 ohm
R8 = 22.000 ohm
C1 = 100.000 pF poliestere
P1 = pulsante
S1 = dip-switch 8 posizioni
CONN.1 = connettore 24 poli
CONN.2 = connettore 5+5 poli

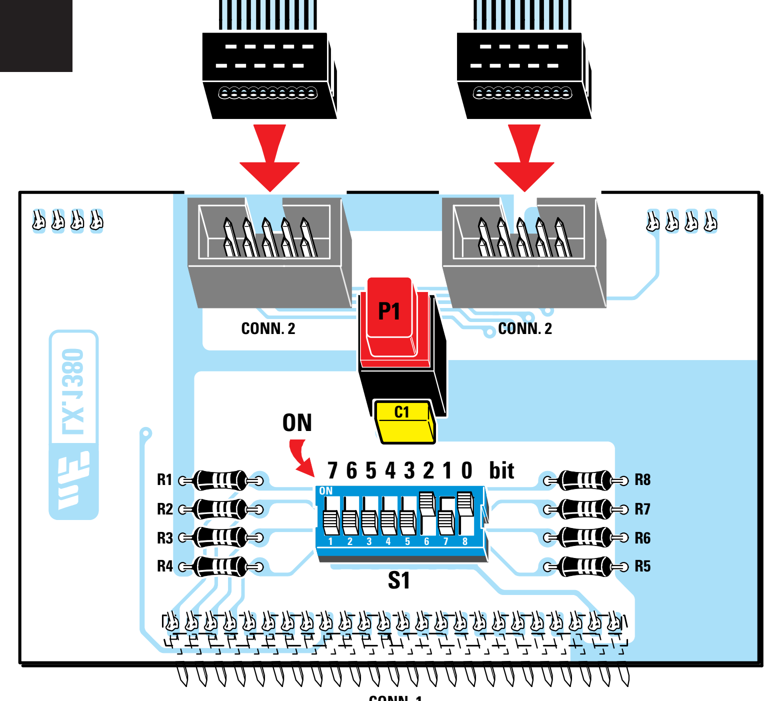


Fig.4 Schema pratico della scheda LX.1380. Le asole dei connettori CONN.2 vanno rivolte verso l'alto. Le levette di S1 corrispondono ai piedini di porta A (vedi fig.3) secondo la numerazione presente sulla serigrafia dello stampato, quindi non considerate la numerazione presente sul dip-switch.

ne/ricezione dati tramite **SPI** che coinvolge due micro ST62/65 e 4 shift register.
In sostanza il micro contenente il programma **Slave** (**RXPG03**), innestato nella scheda **LX.1329**, riesce a ricevere i dati dal micro contenente il programma **Master** (**TXPG03**), innestato nella scheda **LX.1381**, e li ritrasmette, sfalsati da un solo ciclo di trasmissione, ai 4 shift register che si trovano sulla scheda **LX.1382** sfruttando lo stesso segnale di clock trasmesso dal Master. Naturalmente l'unica condizione è che i tre dispositivi abbiano il segnale del clock SPI in comune.

Poiché nel dischetto **DF.1380** tutti questi programmi sono in formato **.ASM**, dovrete necessariamente **assemblarli** in modo da ottenere dei files in formato **.HEX** (vedi rivista N.179), prima di poterli caricare sui micro **ST62/65** tramite il programmatore **LX.1325** descritto nella rivista N.192.

Accanto ad ogni istruzione di programma abbiamo inserito un **commento** chiarificatore, quindi se avete qualche dubbio potete aprire i files con un qualsiasi **editor** e leggere le spiegazioni.

È sottinteso che per effettuare questi **test** è consigliabile usare dei micro ST62**E**65 provvisti di **finestra** perché si possono **cancellare** e quindi riutilizzare, mentre i micro ST62**T**65 si possono programmare una volta sola.

La scheda **LX.1380** va innestata sulla scheda **bus LX.1329** e dovrete necessariamente alimentarla con una tensione stabilizzata di **5 volt**.
La fig.1 potrà chiarire su quale dei tre poli presenti sulla morsettiera dovrete inserire il positivo ed il negativo dei **5 volt**.

Il programma PROG01

Dopo aver assemblato il file **PROG01.ASM** in **PROG01.HEX**, caricate questo programma su un micro ST62**E**65 che inserirete nella scheda bus siglata **LX.1329**.
Eseguita questa operazione collegate la scheda **LX.1380** alla scheda bus **LX.1329** tramite il connettore **CONN.1** e la scheda **LX.1382** alla scheda **LX.1380** tramite piattina utilizzando a vostro piacere il **CONN.2** a destra o quello a sinistra.

Spostate a vostro piacere una o più leve del **dip-switch** presente nella scheda **LX.1380** e non appena premerete il pulsante **P1** la configurazione selezionata sul **dip-switch** verrà inviata tramite la **SPI** alla scheda siglata **LX.1382**.

Sui **diodi led** apparirà il valore **binario** degli **8 bits** selezionati tramite **dip-switch** e sui **display** apparirà l'equivalente valore **decimale**.

Il programma **PROG01** è un esempio di come risulti possibile trasmettere dei **dati** tramite **SPI** da un micro **ST62/65** a **4 shift register** a **8 bits** di tipo **HC/Mos 4094** collegati in serie.

Gli **shift register** (vedi **IC1-IC2-IC3-IC4**) pur non disponendo della funzione **SPI** ricevono i **dati** in modo **seriale** sul **piedino 2** ed il segnale di **clock** sul piedino **3** (vedi fig.9).
I dati ricevuti vengono successivamente memorizzati e visualizzati in questi registri solo inviando un segnale di **latch** (high) sul piedino **1**.
Infatti, solamente quando questo piedino passa dallo stato logico **0** allo stato logico **1**, i dati presenti in quell'istante nel registro vengono memorizzati e contemporaneamente inviati in modalità parallela su 8 dei suoi piedini (per la precisione i piedini **4-5-6-7-14-13-12-11**) per essere visualizzati sui display e sui led della scheda.
Il collegamento in serie degli integrati **4094** è stato ottenuto collegando il piedino **9** del primo dispositivo al piedino **2** del secondo e così via.
I dati inviati passano perciò di volta in volta dal primo registro al secondo fino a quando non viene inviato il segnale di **latch**.

I programmi TXPG02 - RXPG02

Dopo aver assemblato il programma **TXPG02.ASM** ottenendo **TXPG02.HEX** ed il programma **RXPG02.ASM** ottenendo **RXPG02.HEX**, dovrete caricarli su due micro cancellabili tipo ST62**E**65.

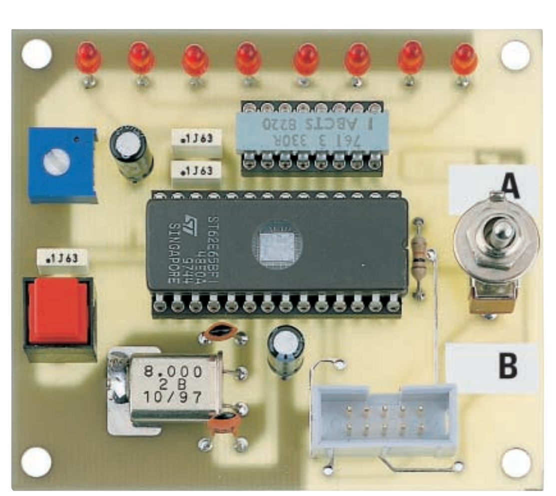


Fig.5 Foto della scheda LX.1381 sulla quale dovrete inserire un micro ST62/65 per poter dialogare con il micro inserito nella scheda LX.1329 (vedi fig.1).

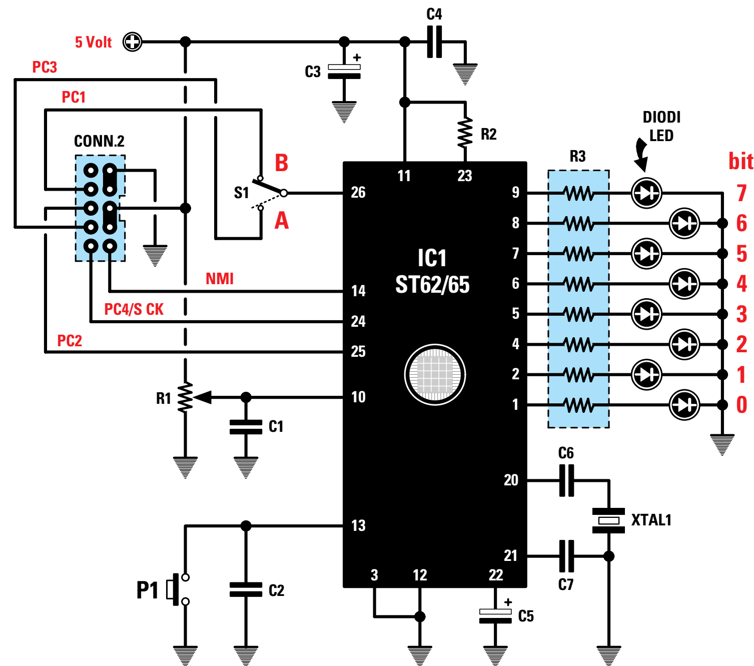
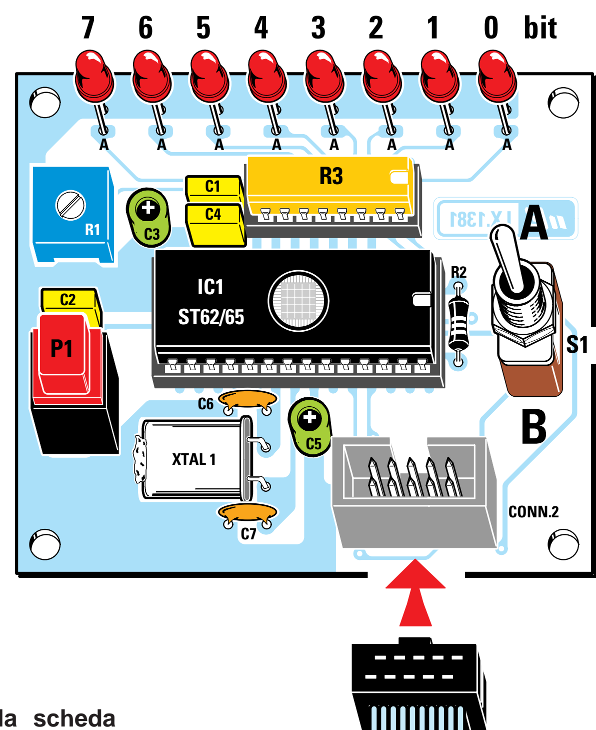


Fig.6 Schema elettrico del circuito LX.1381. Gli 8 diodi led collegati al micro si accenderanno con lo stesso codice binario impostato col dip-switch S1 montato sull'interfaccia periferica LX.1380 (vedi fig.4).

ELENCO COMPONENTI LX.1381

R1 = 10.000 ohm trimmer
R2 = 10.000 ohm
R3 = 330 ohm rete resist. x 8
C1 = 100.000 pF poliestere
C2 = 100.000 pF poliestere
C3 = 22 mF elettrolitico
C4 = 100.000 pF poliestere
C5 = 1 mF elettrolitico
C6 = 22 pF ceramico
C7 = 22 pF ceramico
DL1-DL8 = diodi led
IC1 = micro ST62/65
XTAL1 = quarzo 8 MHz
P1 = pulsante
S1 = deviatore
CONN.2 = connettore 5+5 poli



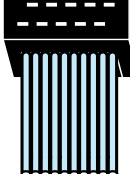


Fig.7 Schema pratico di montaggio della scheda LX.1381. Ricordate che la levetta del deviatore S1 deve essere rivolta su A quando userete i programmi TXPG02-RXPG02 e su B quando userete i programmi TXPG03-RXPG03 (leggete il testo).

Sul micro in cui avete caricato il programma **TXPG02** attaccate un'etichetta con la scritta **TX02** (master), mentre sul micro in cui avete caricato il programma **RXPG02** attaccate un'etichetta con la scritta **RX02** (slave).
Il micro **TX02** va innestato nella scheda **LX.1329**, mentre il micro **RX02** nella scheda **LX.1381**.
Eseguite queste operazioni collegate la scheda **LX.1380** alla scheda bus **LX.1329** tramite il connettore **CONN.1** e la scheda **LX.1381** alla scheda **LX.1380** tramite piattina utilizzando a vostro piacere il **CONN.2** a destra o quello a sinistra.

IMPORTANTE: il deviatore **S1** presente sul circuito **LX.1381** va posizionato verso **A** in modo da collegare il piedino **26** (**PC2**) del micro Slave con il piedino **25** (**PC3**) del micro Master, diversamente non verrà effettuata nessuna trasmissione **dati**.
Nella fig.8 potete vedere la piedinatura elettrica e logica del micro **ST62/65**.

Questi due programmi offrono un esempio di **trasmissione dati** tra **2 micro ST62/65** tramite la funzione **SPI**.
Vi facciamo notare che il pulsante **P1** che si trova sulla scheda **LX.1380** è **inattivo**, mentre è **attivo** il pulsante **P1** ed **inattivo** il trimmer **R1** presenti nella scheda **LX.1381**.
Più avanti spiegheremo nei particolari le istruzioni specifiche della **SPI** presenti in questi programmi.
Voi stessi leggendo i **sorgenti** potrete verificare che la trasmissione e la ricezione dei **dati** non è continua, ma avviene solo quando il programma **RXPG02** (Slave) ne fa richiesta.

Per il momento vi diciamo solo di concentrare l'attenzione sulla configurazione del registro **spmc**, tramite il quale è stata attivata la **Start Condition**.

Compito del programma **TXPG02-Master** è leggere la configurazione del **dip-switch** presente nella scheda **LX.1380** per trasmetterla al programma **RXPG02-Slave**.
Questo programma, dopo aver ricevuto i **dati**, li **visualizza** con gli **8 diodi led** presenti sul circuito.
Nell'esempio che abbiamo scritto, la trasmissione **dati** avviene soltanto quando viene premuto il pulsante **P1** presente nella scheda **LX.1381**.
In sostanza dunque è il programma **Slave** che, rilevando la pressione esercitata sul pulsante, invia al programma **Master** la richiesta di trasmissione.

I programmi TXPG03 - RXPG03

Dopo aver assemblato il programma **TXPG03.ASM** ottenendo **TXPG03.HEX** ed il programma **RXPG03.ASM** ottenendo **RXPG03.HEX**, dovrete caricarli su due micro cancellabili tipo ST62E65.
Sul micro in cui avete caricato il programma **TXPG03** applicate un'etichetta con la scritta **TX03** (master), mentre nel micro in cui avete caricato il programma **RXPG03** applicate un'etichetta con la scritta **RX03** (slave).

Il micro **RX03** va inserito nella scheda **LX.1329**, mentre il micro **TX03** nella scheda **LX.1381**.
Eseguite queste operazioni collegate la scheda **LX.1380** alla scheda bus **LX.1329** tramite il connettore **CONN.1** e la scheda **LX.1381** alla scheda

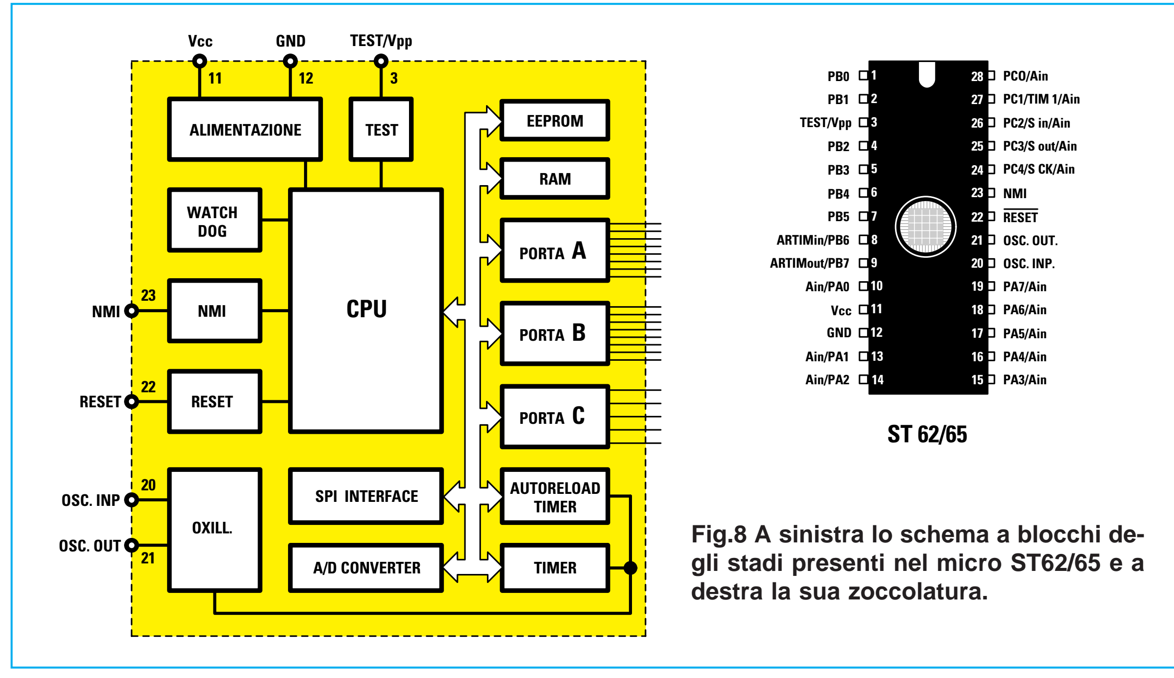


Fig.8 A sinistra lo schema a blocchi degli stadi presenti nel micro ST62/65 e a destra la sua zoccolatura.

LX.1380 tramite piattina utilizzando a vostro piacere il **CONN.2** a destra o quello a sinistra.
Per finire collegate sempre tramite piattina anche la scheda **LX.1382** alla scheda **LX.1380** sul connettore rimasto libero.

IMPORTANTE: il deviatore **S1** presente sul circuito **LX.1381** va posizionato verso **B** in modo da collegare il piedino **26** (**PC2**) del micro **TX03** con il piedino **27** (**PC1**) del micro **RX03**, diversamente non avverrà nessuna trasmissione **dati**.

Anche in questo caso il pulsante **P1** presente sulla scheda **LX.1380** è **inattivo** e lo stesso dicasi per il pulsante **P1** presente sulla scheda **LX.1381**.
Quello che risulta **attivo** è il solo trimmer **R1** che ci serve per variare la **tensione** sul piedino **10** utilizzato come **A/D converter**.
Noi abbiamo utilizzato un trimmer, ma potrete entrare su questo piedino con qualsiasi tensione continua da **0** a **5 volt massimi** prelevabili da una sorgente qualsiasi, una **fotoresistenza**, una resistenza **NTC**, un **alimentatore** ecc.
In pratica il programma Master rileva il valore di tensione leggendolo sul piedino **10** del micro **TX03** e, passando attraverso il micro **RX03 Slave** situato sulla scheda **LX.1329**, lo **visualizza** sui **diodi led** della scheda **LX.1382** con un codice **binario** e sui tre **display** in un valore **decimale** da **0** a **255**.

In questo caso la trasmissione **dati** tra due micro e tra il micro e gli shift register avviene utilizzando lo stesso **clock** del **micro Master**.
Unica condizione è, ovviamente, che i piedini **PC4 Sck** dei micro ed i piedini **3** degli shift register siano collegati insieme.

Il micro **TX03** legge per **32 volte** la tensione presente sul piedino dell'**A/D converter**, ne fa il **totale** ed il numero **binario** che ne risulta lo **divide** per **32** in modo da ottenere un valore **medio**.
Questo valore **medio** viene poi convertito in un codice **BCD** da **3 bytes**, che, inviato alla scheda **LX.1382**, ci servirà per far apparire sui **display** un numero **decimale**.

A differenza dei programmi precedenti, in questo esempio non abbiamo attivato la **Start Condition** tramite il registro **Mode spmc**, ma viene invece effettuato un controllo sul piedino **PC2** di **Port_C** tramite l'istruzione **JRR**.
Infatti solo quando il micro **TX03** riceve un impulso di **reset** sul suo piedino **26** (**PC2 Sin**), inizia a trasmettere i **dati** per un totale di **5 cicli** di trasmissione di **8 bits** ciascuno.
Per tutta la durata della trasmissione si **accende** il diodo led posto a sinistra.

Lo stesso micro controlla inoltre che il numero **binario** non superi una soglia che possiamo prefissare tra **1** e **255**: nel nostro esempio abbiamo prefissato la soglia a **230**.
Se questo numero viene superato, il Master invia un segnale di **allarme** sul piedino **14**, corrispondente al piedino logico **PA2** di **porta A** del micro **TX03** (vedi fig.8), collegato al piedino **NMI** del micro **RX03**, presente nella scheda **LX.1329**, e carica un **livello logico 1** nella variabile **nonesi**, che è normalmente a **livello logico 0**.

Nel programma è stata inserita una routine che si attiva quando viene letto un dato superiore a 230.
Con questa routine viene abbassato il tempo di richiesta invio dati da **10 secondi** ad **1 secondo**, fino a che il valore medio rilevato sul trimmer non torna sotto il limite dei 230.

Il superamento del livello di soglia è per noi anche visivo perché il diodo led posto a sinistra comincia a lampeggiare molto più velocemente, all'incirca **1** volta al **secondo**.

Il programma **RX03**, presente sulla scheda bus **LX.1329**, utilizza un orologio interno generato tramite la funzione **Timer** e ogni **10 secondi** invia al micro **TX03**, presente sulla scheda **LX.1381**, una richiesta di invio dati relativa appunto alla codifica digitale della tensione rilevata sul trimmer **R1**.

Pur essendo **RX03** settato in **ricezione**, avendo caricato il valore **1** sul registro **misc**, il suo piedino **25** diventa un piedino settato in trasmissione come **PC3 Sout**.
Come già sapete, questo significa che in presenza di un **clock** SPI, il valore presente di volta in volta sul **bit 7** del registro **spda** viene trasmesso su **PC3 Sout** e nel nostro caso inviato agli shift register montati sulla scheda **LX.1382**.
Quindi ogni **10 secondi** si ha una ricezione **dati** suddivisi in **5 cicli** di 8 bits ciascuno.

Poiché qualcuno si chiederà perché occorrono **5 cicli** cercheremo di spiegarvelo:

– al **1° ciclo** gli **8 bits** presenti nel registro **spda** del micro **TX03** vengono trasferiti nel registro **spda** del micro **RX03** e qui rimangono parcheggiati.

– al **2° ciclo** i successivi **8 bits** presenti nel registro **spda** del micro **TX03** vengono inviati nel registro **spda** del micro **RX03**, mentre gli 8 bits del **1° ciclo** vengono inviati tramite PC3 Sout all'integrato **IC1** della scheda **LX.1382**.
Nel registro **spda** del micro **RX03** risultano ora parcheggiati gli **8 bits** del **2° ciclo**.

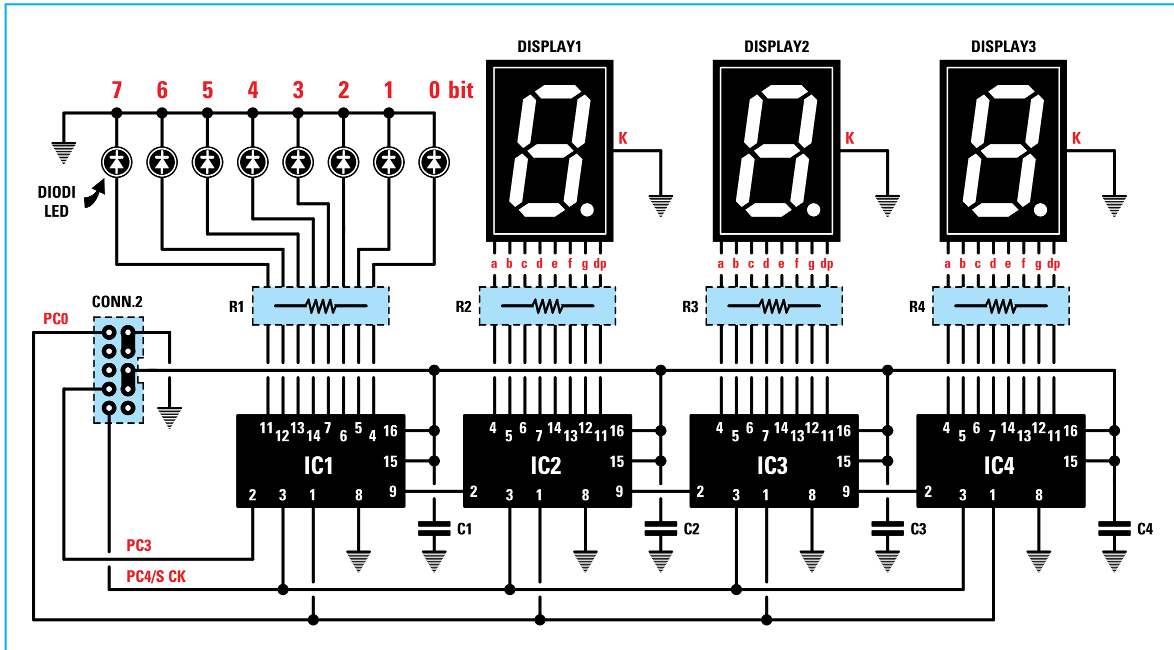


Fig.9 Schema elettrico del circuito LX.1382. Con il programma PROG01 la configurazione selezionata con il dip-switch S1 presente sulla scheda LX.1380 verrà inviata su questa scheda per essere visualizzata tramite gli otto diodi led e i tre display.

ELENCO COMPONENTI LX.1382

R1 = 330 ohm rete
R2 = 330 ohm rete
R3 = 330 ohm rete
R4 = 330 ohm rete
C1 = 100.000 pF poliestere
C2 = 100.000 pF poliestere
C3 = 100.000 pF poliestere
C4 = 100.000 pF poliestere
DL1-DL8 = diodi led
DISPLAY1 = display TIL.702
DISPLAY2 = display TIL.702
DISPLAY3 = display TIL.702
IC1 = C/Mos tipo 4094
IC2 = C/Mos tipo 4094
IC3 = C/Mos tipo 4094
IC4 = C/Mos tipo 4094
CONN.2 = connettore 5+5 poli

– al **3° ciclo** i successivi **8 bits** presenti nel registro **spda** del micro **TX03** vengono inviati nel registro **spda** del micro **RX03**, gli **8 bits** del **2° ciclo** vengono inviati sempre con PC3 Sout all'integrato **IC1** della scheda **LX.1382** e gli 8 bits del **1° ciclo** vengono trasferiti ad **IC2**.
Nel registro **spda** del micro **RX03** risultano ora parcheggiati gli **8 bits** del **3° ciclo**.

– al **4° ciclo** i successivi **8 bits** presenti nel registro **spda** del micro **TX03** vengono inviati nel registro **spda** del micro **RX03**, gli **8 bits** del **3° ciclo** vengono inviati con PC3 Sout all'integrato **IC1** della scheda **LX.1382**, gli 8 bits del **2° ciclo** vengono trasferiti ad **IC2** e gli 8 bits del **1° ciclo** vengono trasferiti ad **IC3**.

Nel registro **spda** del micro **RX03** risultano ora parcheggiati gli **8 bits** del **4° ciclo**.

– al **5° ciclo** i successivi **8 bits** presenti nel registro **spda** del micro **TX03** vengono inviati nel registro **spda** del micro **RX03**, gli **8 bits** del **4° ciclo** vengono inviati con PC3 Sout all'integrato **IC1** della scheda **LX.1382**, gli 8 bits del **3° ciclo** vengono trasferiti ad **IC2**, gli 8 bits del **2° ciclo** vengono trasferiti ad **IC3** e gli 8 bits del **1° ciclo** vengono trasferiti ad **IC4**.

Se ci fossimo fermati alla trasmissione del **4° ciclo**, questo sarebbe rimasto parcheggiato nel registro **spda** del micro **RX03** e non avrebbe raggiunto la scheda **LX.1382**.

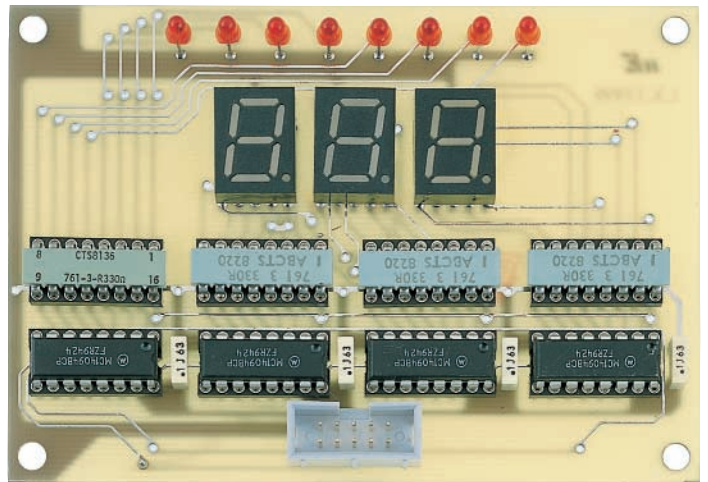
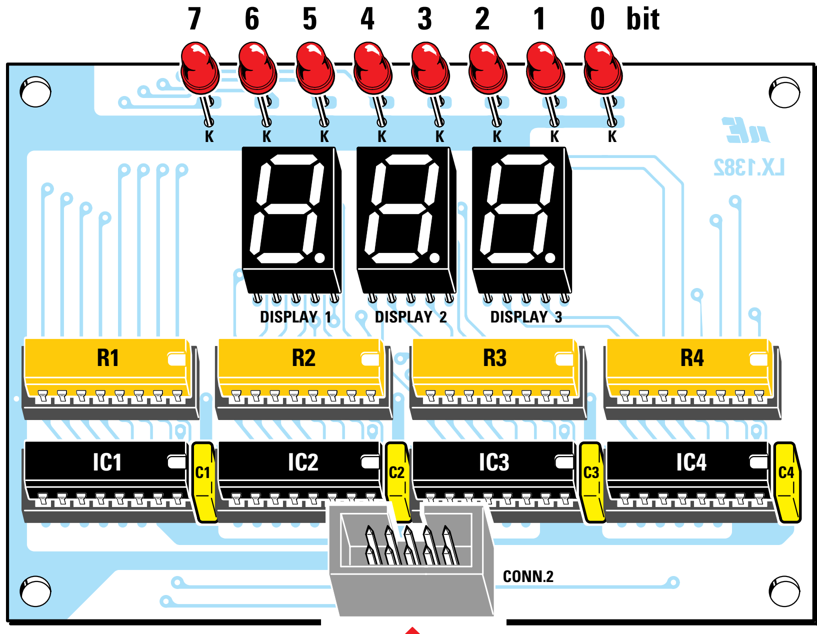


Fig.10 Foto della scheda test LX.1382. Poiché questa è la foto di un prototipo, sul circuito stampato manca il disegno serigrafico.



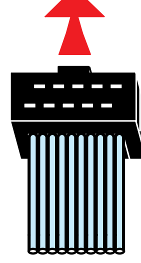


Fig.11 Schema pratico di montaggio della scheda LX.1382. Il terminale più corto dei diodi led (vedi terminale K) va rivolto in basso verso i tre display.

A questo punto il micro **RX03** lancia il segnale di **latch** e tutti i dati presenti nei 4 shift register vengono visualizzati sui display e sui led.

I Programmi TXPG02 e RXPG02 ISTRUZIONE per ISTRUZIONE

Le possibilità offerte dalla funzione **SPI** sono molteplici, ma non potendo fare un articolo fiume che contemplasse tutte le casistiche necessarie ad illustrarle, abbiamo realizzato i programmi di esempio di cui avete appena letto una sintetica descrizione. Oltre a permettervi di sperimentare subito la **SPI**, potranno servirvi per testare un programma scritto da voi.
Ad esempio, caricando sul circuito Master il vostro programma e sul circuito Slave il nostro **RXPG02**, sarete in grado di valutare immediatamente se il vostro programma “trasmette” i dati correttamente. Stessa cosa potrete fare per testare un vostro programma in ricezione.

Proprio perché potete disporre di una sorta di scheda di valutazione, abbiamo pensato di analizzare nei particolari le istruzioni dei programmi denominati **TXPG02** ed **RXPG02**, fermo restando che potrete utilizzare come test anche gli altri programmi scritti da noi.
Inoltre, per focalizzare la vostra attenzione sull’argomento che stiamo trattando, ometteremo di seguito la spiegazione delle istruzioni non inerenti alla funzione **SPI**, ampiamente trattate nel corso delle precedenti lezioni.

Innanzitutto con i programmi **RX** e **TX** noi leggiamo i livelli logici presenti sulla **porta A** del micro **Master** inserito nel **Bus LX.1329** e li inviamo con la funzione **SPI** al micro **Slave** inserito nella scheda **LX.1381**. Quando infatti, il deviatore presente su questa scheda è posizionato su **A**, collega il piedino **26** (**PC2 Sin**) del circuito **LX.1381** al piedino **25** (**PC3 Sout**) del circuito **LX.1329**.
I livelli logici della **porta A** possono essere modificati a piacere tramite il **dip-switch S1**.

Facciamo presente che il micro **Master** invia i dati dei suoi **8 bits** alla velocità di **2.400 bits rate** verso il micro **Slave** ogni volta che premiamo il pulsante **P1** montato sulla scheda **LX.1381**, cioè quando il programma Slave fa una richiesta di trasmissione.

Appena il micro **Slave** riceve i dati dal micro **Master** li carica sulla sua **porta B** e li visualizza sugli **8 led** secondo questa logica:

Livello logico 1 = diodo led **acceso**
Livello logico 0 = diodo led **spento**

Analizziamo ora il programma caricato sul micro **Master** chiamato **TXPG02**. In **Data Space** troviamo le istruzioni dei registri utilizzati per la **SPI**:

```
misc    .def    0ddh
spda    .def    0e0h
spdv    .def    0e1h
spmc    .def    0e2h
```

I piedini della **porta A** gestiti dal **dip-switch** risultano configurati **Input Pull-Up** senza **Interrupt**.

```
        ldi     port_a,00000000b
        ldi     pdir_a,00000000b
        ldi     popt_a,00000000b
```

A questo proposito vi ricordiamo che i piedini di **Port_A** corrispondono alle levette del **dip-switch** come qui sotto riportato:

```
ldi port_a,  0  0  0  0  0  0  0  0b
S1           0  1  2  3  4  5  6  7
```

Ora passiamo ai piedini di **Port_C** che, per gestire in modalità corretta la **SPI**, vanno così configurati:

```
        ldi     port_c,00000100b
        ldi     pdir_c,00011000b
        ldi     popt_c,00011000b
```

Il piedino **2** viene configurato **Input no Pull_up**, mentre i piedini **3-4** come **Output Push_pull**.
In questo esempio di trasmissione i restanti piedini non ci interessano quindi non li abbiamo riportati.
Con le istruzioni appena viste abbiamo solamente predisposto i piedini interessati alla trasmissione, ma non abbiamo ancora attivato la SPI.
In questo programma il piedino **2** è stato configurato come **input**, perché dovrà ricevere dal micro **Slave** il segnale necessario al micro **Master** per iniziare la trasmissione ed avere così una sorta di sincronismo tra i due microprocessori.
Se non è necessario alcun sincronismo, il piedino **2** può essere anche ignorato e non configurato nel programma **Master**.
Dal momento che in questo programma non vengono utilizzati, tutti i registri di **interrupt** sono così configurati:

```
        ldi     armc,00000000b
        ldi     adcr,00000000b
        ldi     tscr,00000000b
        ldi     ior,00000000b
```

Ora passiamo al programma principale e analizziamo, per quanto riguarda la trasmissione dei dati, le istruzioni riga per riga:

```
main    ldi     wdog,0ffh
```

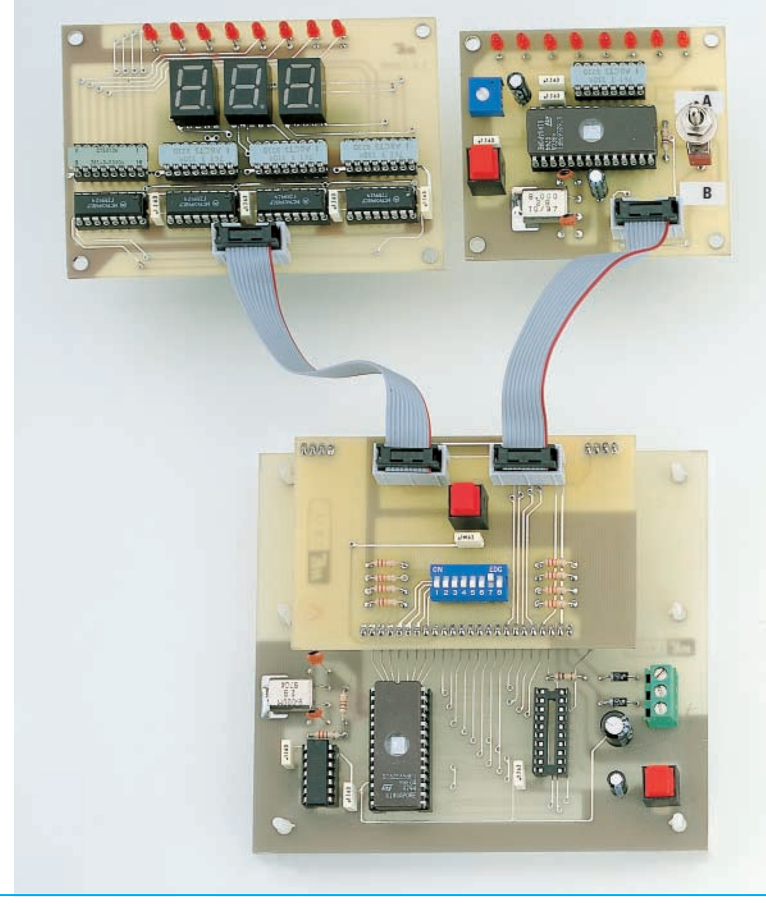


Fig.12 Dopo aver innestato la scheda LX.1380 sulla scheda LX.1329, per collegare le altre due schede potrete usare le piattine cablate che abbiamo inserito nel kit.

Provvede ad assegnare all'etichetta **main** questa istruzione che ricarica il **watchdog**.

ldi misc,1

Come abbiamo spiegato, mettendo a **1** il bit **0** di **misc**, il piedino **3** di **Port_C** passa dallo stato di **I-O** a **PC3 Sout** e diventa il piedino di trasmissione della funzione **SPI**.

Attenzione: vi ricordiamo che se non inserite questa istruzione, anche configurando il registro **spmc** in **Master Mode**, la trasmissione dei dati **non** avverrà mai e il **clock** di trasmissione su **PC4 Sck** non partirà mai.
La successiva istruzione:

ldi spdv,01000111b

serve per configurare il registro **spdv** con la modalità di trasmissione di **8** bits per ciclo alla velocità di **2400 bits rate**.
Se avete letto la spiegazione dei **registri** e avete visto le tabelle riportate a pag.109 di questa rivista, avrete capito perché abbiamo caricato questo valore nel registro **spdv**.

L'istruzione che segue, cioè:

ldi spmc,00010100b

carica nel registro **spmc** i valori di configurazione **Master** per la trasmissione **dati** e seleziona la modalità **Clock Master mode** con **polarità** e fase **normali**. Non è previsto un **filtro** in trasmissione e il **bit 2 Spstrt** posto a **1** serve a gestire assieme al **bit 7 Sprun** la condizione di **Start** trasmissione-ricezione. Inoltre il bit 7 Sprun è stato messo momentaneamente a 0. Infatti ponendolo a 1 avremmo attivato la condizione di Start trasmissione-ricezione e se durante la fase iniziale di configurazione dei due micro fosse stato inviato un falso segnale sul piedino **PC2** del Master, questi avrebbe iniziato a trasmettere con il programma Slave non ancora pronto a ricevere i dati.

Se andate a rileggere quanto spiegato per questo **registro** potrete verificare personalmente quanto detto in proposito.

Ora passiamo alla successiva istruzione:

pippo ldi wdog,0ffh

La label **pippo** viene associata all'istruzione che ricarica il watchdog.

```
        ld      a,port_a
        ld      spda,a
```

Come già ribadito, questa è la sequenza giusta per caricare nel registro **spda** il valore da trasmettere. Nel nostro caso muoviamo il valore logico presente sugli **8** piedini di **Port_A** nel registro **spda**.

```
        set     7,spmc
```

Mettendo a **1** il **7 bit** (**Sprun**) del registro **spmc**, abbiamo predisposto tutto per la trasmissione del valore presente su porta **A**, ma non abbiamo iniziato ancora la trasmissione. Infatti come già ripetuto oramai varie volte, settando **Sprun** e **Spstrt** abbiamo creato la condizione di **start**.

In questa condizione tutto è pronto per la trasmissione, che avviene però solamente quando sul piedino **2** di **Porta C** viene rilevato un fronte di salita o **rising edge**.
Questo segnale verrà generato dal micro **Slave** e vi sarà spiegato nel programma **RXPG02**.

Una cosa che vale la pena sottolineare è che non dovremo inserire nessuna istruzione o routine per verificare la presenza del segnale **rising edge** sul piedino **2** di **porta C**, ma sarà automaticamente rilevato dal micro Master che provvederà, sempre automaticamente, ad iniziare la trasmissione.

Con l'istruzione seguente:

```
pluto   ldi     wdog,0ff
```

abbiamo assegnata la label **pluto** ad un'istruzione che ricarica il watchdog.

```
        jrs     7,spmc,pluto
```

Con questa istruzione il programma esegue un "loop" e salta a **pluto** finché il **bit 7** (**Sprun**) è settato. In pratica il programma rimane in loop finché non è avvenuta la trasmissione. Quando la trasmissione è terminata, il **bit 7** di **spmc** viene automaticamente resettato.

```
        jp      pippo
```

Questa istruzione viene eseguita solo a trasmissione terminata ed il programma salta perciò nuovamente a **pippo** dove ricaricherà un eventuale nuovo valore in **spda** (se sono stati modificati i dip-switch) e si preparerà di nuovo a trasmetterlo.

Avrete già sicuramente notato, ma lo evidenziamo ugualmente, che il programma **TXPG02** così come è stato scritto, invia al micro slave sempre la penultima configurazione presente in porta A, mai l'ultima. Non è un errore, ma solamente la necessità di scrivere un programma semplice e breve.
Una gestione più complessa avrebbe potuto creare altri problemi e non ci avrebbe permesso di focalizzare bene la SPI.

Vediamo ora il programma **RXPG02** caricato sul micro Slave.
Passiamo subito al settaggio delle porte iniziando da **Porta A**:

```
        ldi     port_a,00000000b
        ldi     pdir_a,00000000b
        ldi     popt_a,00000000b
```

Il piedino **0** viene configurato **Input Pull-up** per gestire la pressione del pulsante **P1**.
Questo pulsante servirà per inviare la richiesta al micro Master di inizio trasmissione dati.

A seguire viene configurata **Porta B**:

```
        ldi     port_b,00000000b
        ldi     pdir_b,11111111b
        ldi     popt_b,11111111b
```

Tutti gli 8 piedini di questa porta sono configurati come **Out Push-pull**. A questa porta sono stati collegati **8 leds** per rendere possibile la visualizzazione dei dati ricevuti con la SPI.
Al piedino **1** è stato collegato il led 0, al piedino **2** il led 1 e così via (vedi fig.6).

Infine configuriamo **Porta C**:

```
        ldi     port_c,00010100b
        ldi     pdir_c,00001000b
        ldi     popt_c,00001000b
```

Il piedino **2** viene configurato come **Input No Pull-up No Interrupt** e riceve i dati trasmessi da **PC3 Sout** del Master.
Il piedino **3** serve solo per inviare il segnale di richiesta dati al micro Master e viene perciò configurato come **Out Push-pull**.
Il piedino **4** infine viene configurato come **Input No Pull-up No Interrupt** e riceve il clock di trasmissione dal rispettivo piedino **PC4 Sck** del Master.
Anche in questo caso non abbiamo ancora attivato la SPI, ma solamente predisposto i piedini interessati alla ricezione dati.
In questo programma è prevista la gestione di un interrupt, ma per il momento carichiamo i registri

relativi tutti a zero per evitare in questo modo attivazioni premature:

```
        ldi     armc,00000000b
        ldi     adcr,00000000b
        ldi     tscr,00000000b
        ldi     ior,00000000b
```

L'interrupt da gestire in questo programma è quello su **SPI** per fine ricezione dati (ricordate il bit **6 Spie** del registro **spmc** spiegato nell'articolo teorico a pag.107), quindi prima del programma principale inseriamo la routine per gestire questo interrupt.

```
CS_int  res     3,port_c
        ldi     wdog,0ffh
        ld      a,spda
        ld      port_b,a
        res     7,spdv
        reti
```

Questa routine viene attivata quando la ricezione dati è terminata e svolge la seguente funzione:

```
        res     3,port_c
```

trasmette cioè subito un segnale **falling edge** (fronte di discesa) tramite il piedino **3** di **Port_C** al micro Master, cosicché il relativo piedino **2** si troverà a livello logico 0 e sarà ripristinata la condizione di start di trasmissione già spiegata per il programma **TXPG02**.

```
        ldi     wdog,0ffh
```

Questa istruzione ricarica il watchdog.

```
        ld      a,spda
        ld      port_b,a
```

La ricezione è terminata, quindi il registro **spda** contiene il valore del dato ricevuto e trasmesso dal Master. Per visualizzarlo tramite gli **8 leds** lo dobbiamo caricare su **porta B** e per questo utilizziamo l'accumulatore **a**.

```
        res     7,spdv
```

Il bit 7 del registro **spdv**, come già spiegato, si setta automaticamente a 1 all'attivazione dell'interrupt e quindi prima di uscire dalla routine relativa sarà nostra cura portarlo a **0**.

```
        reti
```

Conoscete oramai tutti la sua funzione.
Definita e spiegata questa routine di interrupt si passa ora al programma principale:

```
main    ldi     wdog,0ffh
```

Assegna come sempre l'etichetta main alla relativa istruzione che ricarica il watchdog.

```
        res     3,port_c
```

Questa istruzione è, come vedete bene, identica a quella inserita nella routine di interrupt, ed ha lo stesso scopo.

```
        ldi     misc,0
```

Mettendo a 0 il bit 0 del registro **misc** noi riportiamo il piedino 3 di porta **C** a normale piedino di I-O e non più **PC3 Sout** di **SPI**.
Se per errore lo avessimo settato a 1 in questo programma specifico, i dati che mano a mano venivano ricevuti su PC2 Sin e caricati bit per bit sul registro spda, con la stessa sequenza sarebbero stati ritrasmessi sul piedino 3 di porta C (PC3 Sout) creando probabilmente un notevole caos.

```
        ldi     spdv,01000111b
```

Con questa istruzione configuriamo il registro **spdv** e quindi la ricezione dati sarà di **8 bits** alla velocità di **2400 bit rate**.

Come avrete notato, abbiamo inserito le identiche modalità del programma **TXPG02**, anche se nel caso della velocità è completamente superfluo dal momento che la ricezione dei dati avviene sul fronte del clock presente sul piedino PC4 Sck e quindi "comanda" sempre la frequenza del Master. Se ad esempio avessimo scritto:

```
        ldi     spdv,01000110b
```

che corrisponde ad una velocità di ricezione di 9600 bits rate (vedi Tabella N.2 nella rivista **N.198**), la ricezione sarebbe avvenuta comunque a 2400 bits rate, dal momento che Master trasmette con un clock di 2400 bits rate. Comunque, nel caso di dialogo tra due microprocessori, per coerenza tra i dati conviene sempre definire un'identica **velocità** di trasmissione e di ricezione.

Per il numero dei bit da ricevere è invece assolutamente necessario definirli sempre uguali al numero dei bit da trasmettere altrimenti potrebbero sorgere grossi problemi di valorizzazione dati.
Infatti, se ricordate, la trasmissione finisce quando sono stati trasmessi un numero di bits pari a quello indicato nel registro **spdv** del programma Master e stessa cosa vale anche per la ricezione dove vengono ricevuti un numero di bits pari a quello indicato nel registro **spdv** del programma Slave.
Questo significa che se i due valori non sono uguali la trasmissione dei dati potrebbe durare più

della ricezione e viceversa e vi lasciamo immaginare quali valori strani potreste ritrovare nel registro **spda** al termine di tutto ciò.

```
        ldi     spmc,01001000b
```

In questo modo il registro **spmc** viene caricato con valori di configurazione Slave per la ricezione dati. Viene selezionata la modalità **Clock Slave mode**, con **polarità** e fase **normali** e senza **filtri** in ricezione. Notate che il bit **7 Sprun** è stato caricato a **0**: questo sta a significare che, per il momento, non abbiamo dato inizio a nessuna ricezione dati.
Inoltre abbiamo attivato la richiesta di interrupt **SPI** settando a 1 il bit **6 Spie**.
Pertanto, tutte le volte che verrà rilevata la fine ricezione, il programma attiverà la richiesta di interrupt su **SPI**, salterà alla locazione di memoria relativa al vettore e cioè **0F4H** dove troverà l'istruzione di salto **jp CS_int**, e attiverà così la routine descritta poco sopra.

Nota: se avete già usato i programmi che forniamo come esempio dovreste già avere questo vettore corretto, in caso contrario all'indirizzo 0F4H inserite l'istruzione **jp CS_int**.

Proseguendo troviamo:

```
        ldi     ior,00010000b
```

con questa istruzione abilitiamo tutti gli interrupt.

Di seguito sono inserite:

```
pippo   ldi     wdog,0ffh
        jrr     3,port_c,res3
        jp      pippo
res3    jrr     0,port_a,rilp1
        jp      pippo
rilp1   jrs     0,port_a,sipl1
        ldi     wdog,0ffh
        jp      rilp1
```

Queste 8 istruzioni hanno il compito di testare se è stato premuto il pulsante P1 e, nel caso, il relativo rilascio evitando così rimbalzi e falsi segnali sul piedino **0** di **Port_A**.
Inoltre si accede alla parte della gestione del pulsante P1 solamente quando il piedino **3** di **Port_C** è a livello logico 0 e cioè solo quando il dato è stato ricevuto e viene visualizzato tramite gli 8 leds (vedi routine **CS_int**).

```
sipl1   set     7,spmc
```

Il programma salta a questa etichetta nel caso sia stato premuto correttamente il pulsante **P1**.

In questo caso il **bit 7 Sprun** di **spmv** viene settato a 1 e ciò dà inizio alla ricezione dati.
Il programma però non riceve ancora nulla, perché, come già detto, il micro Master è in condizione di Start di trasmissione e attende solamente un segnale sul suo piedino **2** di **Port_C** sotto forma di fronte di salita (rising edge) per iniziare ad inviare il clock e i dati.

```
        set     3,port_c
```

Con questa istruzione inviamo finalmente **questo** segnale e a questo punto avrà inizio la trasmissione del Master e la corrispondente ricezione.

```
        jp      pippo
```

Ora il programma ritorna al ciclo di gestione pulsante **P1** per attivare eventualmente altri cicli di ricezione dati.

COSTO di REALIZZAZIONE

Tutti i componenti necessari per realizzare la scheda **LX.1380** visibile in fig.4 .... € 8,30

Tutti i componenti necessari per realizzare la scheda **LX.1381** visibile in fig.7, compresi il quarzo, i diodi led, una piattina cablata completa di due connettori femmina **Escluso** il micro **ST62/65** che potrete richiedere a parte .... € 12,65

Tutti i componenti necessari per realizzare la scheda **LX.1382** visibile in fig.11, compresi i tre display, gli 8 diodi led, 4 integrati 4049 più 8 zoccoli, una piattina cablata completa di due connettori femmina .... € 20,90

Un dischetto floppy **DF.1380** contenente i 5 programmi descritti nel testo .... € 7,75

Su richiesta possiamo fornire anche i microprocessori **ST62/E65** riprogrammabili a .... € 18,08

Costo del solo stampato **LX.1380** .... € 4,29
Costo del solo stampato **LX.1381** .... € 3,41
Costo del solo stampato **LX.1382** .... € 5,68

Nota: se ancora non avete la scheda bus siglata **LX.1329**, pubblicata sulla rivista N.192, ve la possiamo fornire completa di circuito stampato, zoccoli, **quarzo** ed integrato **74HC00** a .... € 19,60

Tutti prezzi sono già **comprensivi** di IVA. Coloro che richiedono il kit in **contrassegno**, dovranno aggiungere le sole spese postali richieste dalle P.T. che si aggirano intorno a € 3,10 per pacco.

COME PROGRAMMARE i

La SGS/Thomson ha cessato di produrre tutta la serie dei micro con le sigle ST62 e li ha sostituiti con la nuova serie ST6/C da programmare in ambiente Windows. Chi possiede il programmatore LX.1325 potrà usarlo anche per gli ST6/C, ma chi possiede solo il vecchio programmatore LX.1170 dovrà completarlo con questa semplice interfaccia.

Da tempo sapevamo che tutti i micro della serie **ST62E10B - ST62E15B - ST62E20B** ecc. seguiti dalle lettere **SWD - HWD** ed anche tutti gli **OTP** della serie **ST62T10B-ST62T15B ST62T20B** ecc., sarebbero stati messi fuori produzione e sostituiti con la nuova serie **C**, che, rispetto alle precedenti, ha in aggiunta l'**option byte**, che permette in fase di programmazione di settare diverse funzioni supplementari.
Con questa nuova serie di micro, siglati **ST62E10C** oppure **ST62T10C**, ecc. (il numero è seguito dalla lettera **C** e non più da **B** o **BB**), è possibile ad esempio selezionare un **watchdog** tipo **hardware** o **software**, mentre con i precedenti micro si doveva necessariamente scegliere un chip con watchdog **SWD** (software) o con watchdog **HWD** (hardware).
La **SGS/Thomson** ha realizzato il **programma** per programmare questa nuova versione **C** solo per ambiente **Windows 3.1- 95 - 98**.

Chi utilizza per la programmazione dei computer che lavorano solo con il sistema operativo **DOS**, sprovvisti cioè di ambiente **Windows**, può ugualmente programmare i nuovi micro **ST6/C** senza bisogno di realizzare l'interfaccia **LX.1430**, ma poiché con il **DOS** non si riesce a modificare l'**option byte** non potrà **proteggerli**, perché questa funzione è presente solo nell'**option byte**.

Importante

Chi utilizza il nostro programmatore **LX.1325** presentato nella rivista N.192, dovrà **solo** caricare nel suo computer il programma che noi forniamo.
Chi utilizza il precedente programmatore siglato **LX.1170**, apparso sulla rivista N.172, oltre a caricare il programma dovrà necessariamente collegare l'**interfaccia LX.1430** tra l'uscita del programmatore e l'ingresso del computer.

SCHEMA ELETTRICO INTERFACCIA

Come potete vedere in fig.1 in questa interfaccia abbiamo un solo integrato siglato **74HC04** provvisto di **6 inverter**, perché il programmatore **LX.1170**, per poter comunicare con il programma **Epromer** che vi forniamo, necessita di alcuni **livelli logici invertiti**.

Nello schema elettrico il **CONN.1** posto sulla sinistra è il connettore **maschio** che andrà collegato all'uscita **parallela** del computer.

Il **CONN.2** posto sulla destra è invece il connettore **femmina** che andrà inserito nell'uscita del programmatore **LX.1170.**

In questo schema elettrico non abbiamo rispettato l'ordine sequenziale dei piedini dei connettori per non ritrovarci con un intreccio di fili difficile da districare, ma abbiamo riportato i relativi numeri.

In fig.6, dove abbiamo disegnato i due connettori **maschio** e **femmina** con vista frontale e posteriore, potete vedere che questi connettori hanno **25 piedini** disposti su due file.

La prima fila è di **13 piedini** e la seconda, sottostante, di **12 piedini**.
Se il connettore **femmina** è visto frontalmente il piedino **1** si trova a **destra**, mentre nel connettore **maschio** si trova a **sinistra**.

L'integrato va alimentato con una tensione di **5 volt**, che preleviamo con i diodi **DS1-DS2** dai piedini **14-16** del connettore **maschio**.

REALIZZAZIONE PRATICA

Per realizzare questa interfaccia abbiamo utilizzato il circuito stampato siglato **LX.1430**, che deve poi essere inserito all'interno del suo minu-

nuovi MICRO serie ST6/C

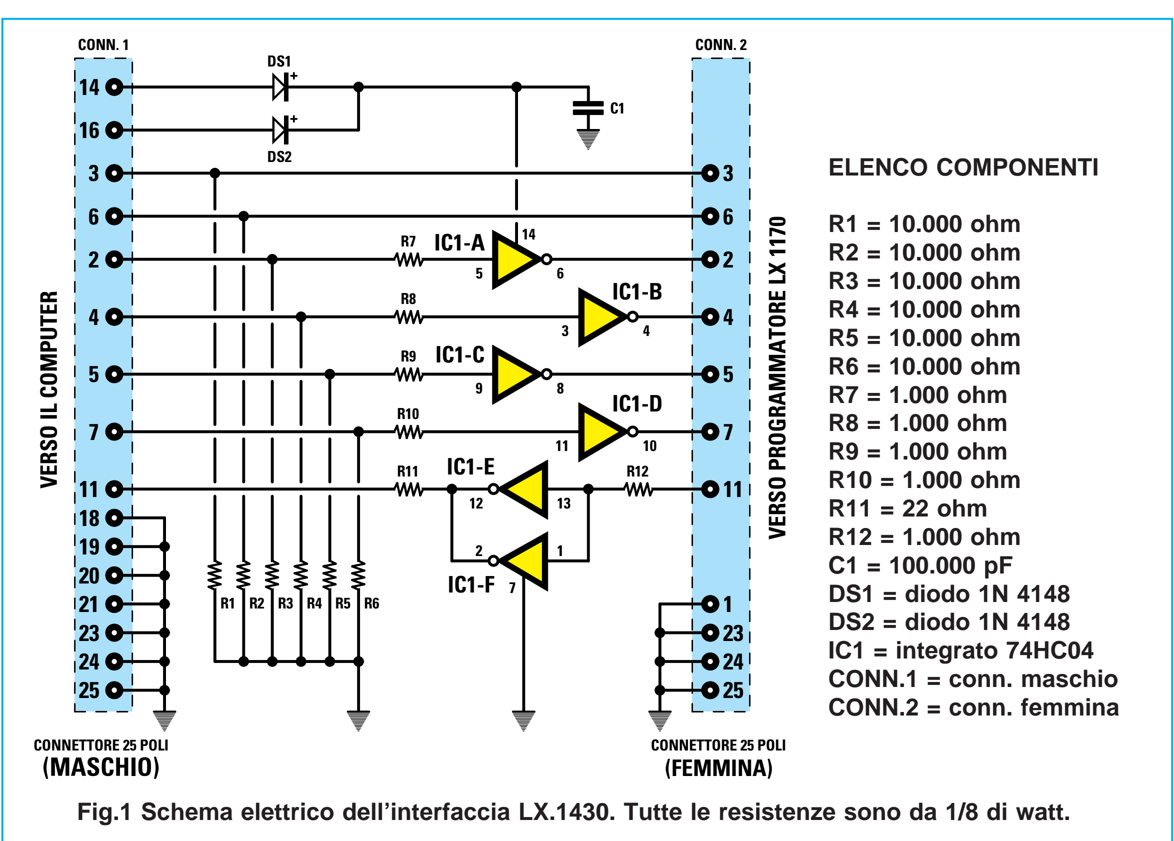


Fig.1 Schema elettrico dell'interfaccia LX.1430. Tutte le resistenze sono da 1/8 di watt.

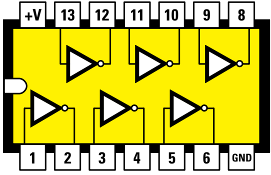


Fig.2 Connessioni del 74HC04 viste da sopra con la tacca a U rivolta a sinistra.

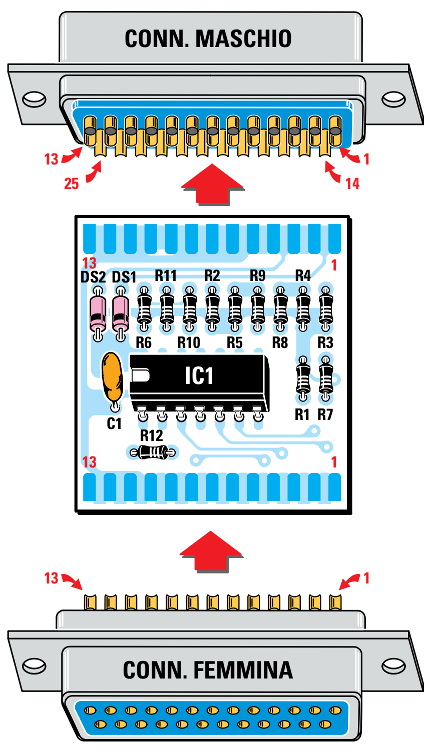


Fig.4 Schema pratico di montaggio della piccola interfaccia. Il connettore femmina va inserito verso la R12.

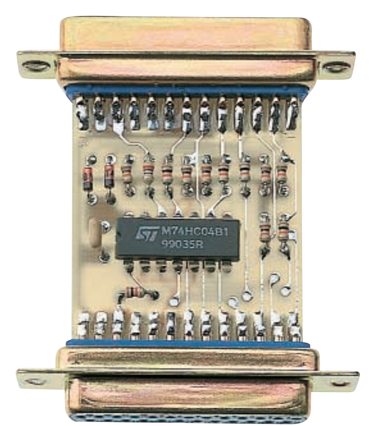


Fig.3 Foto dell'interfaccia già montata e, in basso, già racchiusa all'interno del suo piccolo contenitore plastico.

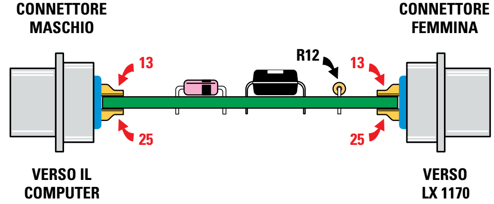


Fig.5 Tra le due file dei terminali dei connettori maschio e femmina dovete innestare il circuito stampato.

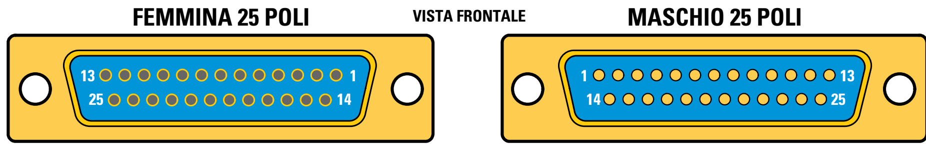


Fig.6 Guardando frontalmente il connettore Femmina, a sinistra è visibile il terminale 13 e a destra il terminale 1. Guardando frontalmente il connettore Maschio, a sinistra è visibile il terminale 1 e a destra il terminale 13.

scolo contenitore plastico (vedi fig.3).
Potete iniziare il montaggio di questa scheda inserendo tutte le **resistenze** da **1/8** di **watt**, dopo aver ovviamente controllato il loro valore ohmico per non inserirle in una posizione errata.

Dopo le resistenze potete montare i due diodi **DS1-DS2** rivolgendo il lato contornato da una **fascia nera** verso il condensatore **C1**, che potete inserire subito dopo nello stampato (vedi fig.4).

Completato il montaggio di questi componenti montate l'integrato **IC1** che, contrariamente ad ogni nostro montaggio, va innestato nello stampato **senza zoccolo**, diversamente non riuscirete a **chiudere** il mobile plastico.

Prima di saldarne i piedini sul circuito stampato, controllate che la sua tacca di riferimento a forma di **U** sia rivolta verso il condensatore **C1**.
A questo punto prendete il connettore **maschio** ed il connettore **femmina** e innestate il circuito stampato tra le due file dei loro terminali come visibile nelle figg.4-5, in modo che sopra ci sia la fila di **13 terminali** e sotto quella di **12 terminali**, poi saldate tutti i **25 terminali** sulle piste del circuito stampato facendo attenzione a non cortocircuitare due piste adiacenti con un eccesso di stagno.
Completate le saldature, potete inserire il circuito stampato dentro i due gusci del mobile plastico e chiuderlo.

CARICARE il PROGRAMMA

Per caricare il programma sul vostro hard-disk inserite il disco siglato **DFST6/C** nel drive floppy.

Se nel vostro computer è installato **Windows 3.1** dovete selezionare nella barra dei menu la scritta **File**; nella finestra che appare andate sulla riga **Esegui ...** e nella nuova finestra che appare dovete digitare **A:setup** poi cliccare su **OK**.

Se nel vostro computer è installato **Windows 95** dovete cliccare con il cursore del **mouse** sulla scritta **Avvio**, posta in basso a sinistra, e nella finestra che appare andate sulla riga **Esegui ...** quindi digitate **A:setup** e cliccate su **OK**.

Se nel vostro computer è installato **Windows 98** dovete cliccare con il cursore del **mouse** sulla scritta **Start**, posta in basso a sinistra, e nella finestra che appare andate sulla riga **Esegui ...** e digitate **A:setup** poi cliccate su **OK**.

In questo modo lancerete l'installazione e verrà creato il gruppo di programmi contenente il file del programmatore.

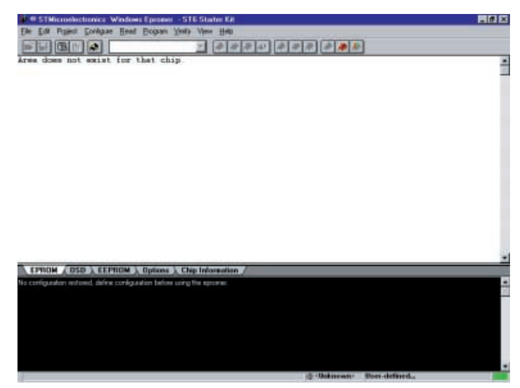


Fig.7 Dopo aver lanciato il file Epromer, sul monitor vi apparirà questa videata.

Fig.8 Dopo aver cliccato sullo scritta Configure, cliccate su Configure Epromer.

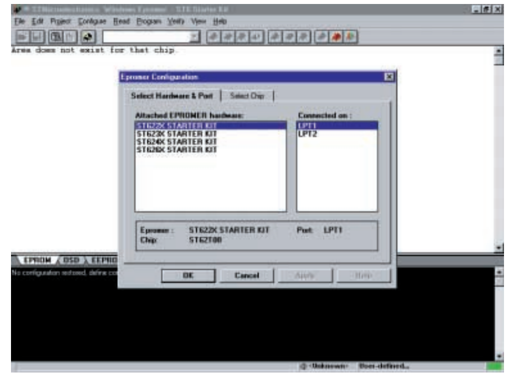


Fig.9 Quando apparirà questa finestra, selezionate la porta parallela e anche il tipo di programmatore (leggere testo).

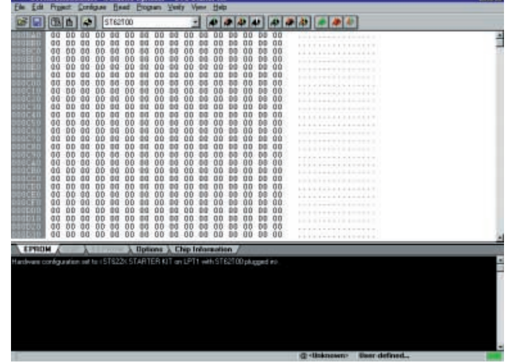


Fig.10 La mappa del micro risulterà vuota cioè con tutti 00, fino a quando non verrà richiamato un programma .Hex.

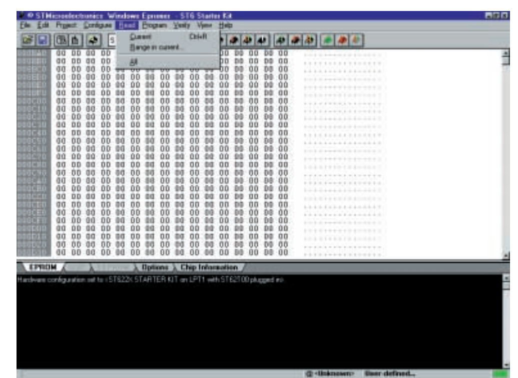


Fig.11 Andando sulla scritta Read del menu, potete leggere il contenuto di un microprocessore già programmato.

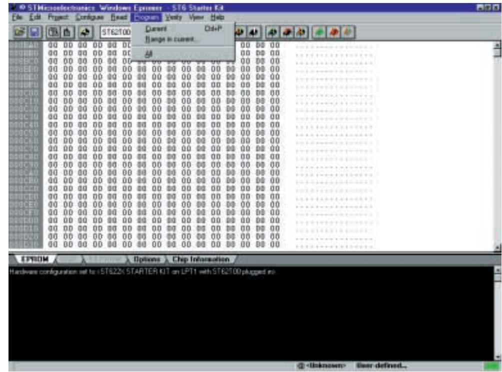


Fig.12 Andando sulla scritta Program, potete trasferire un programma .Hex all'interno del microprocessore.

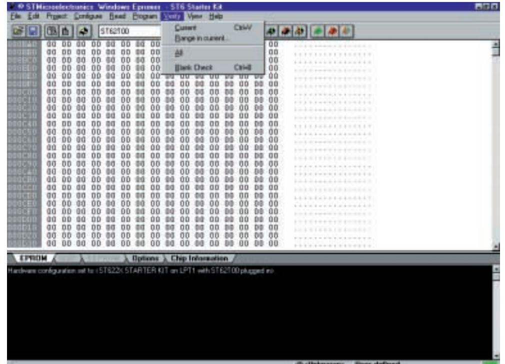


Fig.13 Andando sulla scritta Verify, potete verificare se il micro che desiderate programmare risulta vergine.

Per far partire questo programma è sufficiente cliccare sul nome del programma **Epromer**.

Nella nuova maschera che appare a video cliccate sulla scritta **Configure** e poi su **Configure Epromer** (vedi figg.8-9).

Quando sullo schermo appare la finestra visibile in fig.9 dovete selezionare la porta parallela che volete utilizzare, cioè **LPT1** o **LPT2**.
Se il vostro computer ha la sola **LPT1** e su questa è già collegata la **stampante**, dovete scollegarla ed inserire in sua sostituzione il connettore collegato al nostro programmatore.

Come seconda operazione dovete selezionare nella finestra a sinistra il tipo di **programmatore** che userete.

Se utilizzate il nostro programmatore **LX.1325** dovete selezionare la riga **ST626X**, perché con questo potete programmare tutti i microprocessori della serie **ST6260 -ST6265**, ecc.

Se utilizzate il programmatore **LX.1170**, completo dell'interfaccia **LX.1430**, dovete selezionare la riga **ST622X**, perché con questo potete programmare tutti i micro **ST6210-ST6215**, ecc.

Dopo aver selezionato la porta parallela ed il tipo di programmatore, cliccate prima sul tasto con la scritta **APPLY** e poi sul tasto con la scritta **OK** ed in basso sullo schermo vedrete apparire il tipo di configurazione e la porta selezionata.

Sul monitor apparirà la mappa della memoria relativa al programma da caricare nel micro.
Questa mappa, come visibile in fig.10, risulterà vuota finchè non verrà richiamato il programma **.HEX**.
Come avrete modo di appurare, con questo **nuovo** programma oltre a programmare i nuovi **ST6/C**, riuscirete anche a programmare **tutte** le versioni dei micro **precedenti**.

Infatti, cliccando sulla freccia a **V** posta sulla destra della sigla del micro (vedi fig.15), compariranno a video tutte le **sigle** dei micro che è possibile programmare.

Per selezionare uno dei tanti micro inclusi nella lista basta cliccare **una** volta sola sulla **sigla** del micro desiderato.
Nella riga in basso appariranno queste scritture:

EPROM EEPROM Options Chip Information

che potrete utilizzare per visualizzare a monitor le varie funzioni.

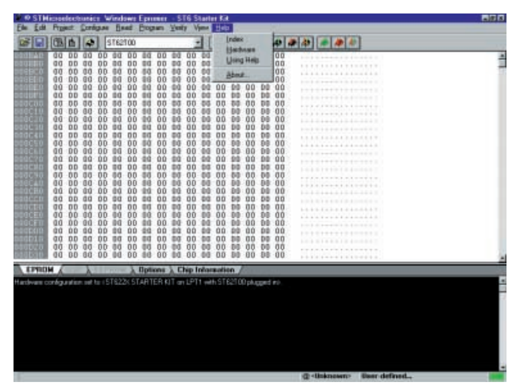


Fig.14 Andando sulla scritta Help, potete accedere ad una guida in linea, che risulta però scritta in inglese.

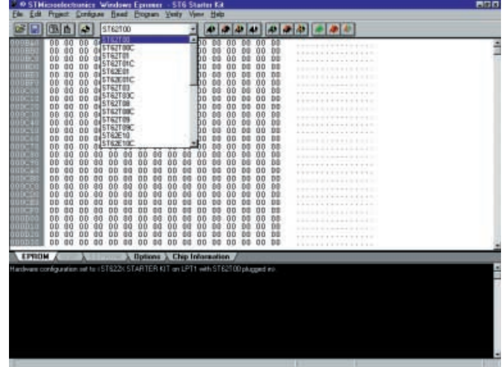
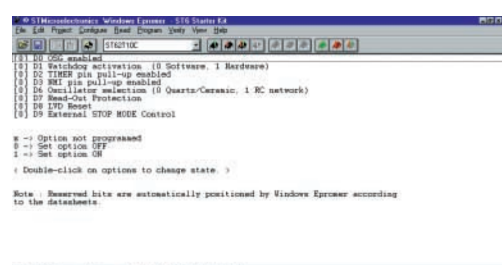


Fig.15 Cliccando sulla freccia a V, posta a destra di ST62T00, vi apparirà la lista dei micro ST6 che potete programmare.



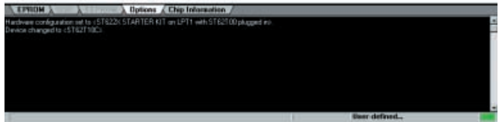


Fig.16 Ammesso di aver scelto il micro ST62T10C, cliccando su Options appariranno tutte le funzioni dell'Option Byte.

Eprom – visualizza il file con estensione **.HEX** da caricare sul micro.

EEprom – visualizza l'area di memoria della **EEprom**; questa funzione è attiva solo per quei micro che dispongono di tale memoria.

Options – visualizza l'**option byte** che potrete settare secondo le vostre esigenze.

Chip Information – visualizza alcune informazioni sul micro selezionato.

OPTION BYTE per micro ST62XX

Se cliccate su **Options** e selezionate ad esempio un micro **ST62T10C** (vedi fig.16), a video appariranno le seguenti righe, che vi permetteranno di settare o resettare le funzioni dell'**option byte**:

[0] D0 OSG enabled
[0] D1 Watchdog activation
[0] D2 TIMER pin pull-up enabled
[0] D3 NMI pin pull-up enabled
[0] D6 Oscillator selection
[0] D7 Read-Out Protection
[0] D8 LVD Reset
[0] D9 External STOP MODE Control

OPTION BYTE per micro ST626X

Se cliccate su **Options** e selezionate ad esempio un micro **ST62T60** della serie **C**, a video appariranno le seguenti righe, che vi permetteranno di settare o resettare le funzioni dell'**option byte**.

[0] D0 OSG enabled
[0] D1 Oscill select
[0] D2 POR delay
[0] D3 Watchdog activation
[0] D4 PB0-1 pins pull-up disabled
[0] D5 PB2-3 pins pull-up disabled
[0] D6 Extern STOP mode enabled
[0] D7 Read-Out Protection
[0] D8 HLVD enabled
[0] D9 NMI pin pull-up enabled
[0] D12 ADC Synchro

Poiché non tutti sapranno già come usare queste nuove funzioni, vi diciamo subito che modificando il numero **0** racchiuso dentro le parentesi quadre con il numero **1** si ottiene quanto segue:

OSG enabled – Tutti i micro della serie **C** dispongono internamente di uno stadio oscillatore di **emergenza** che permette al micro di funzionare con

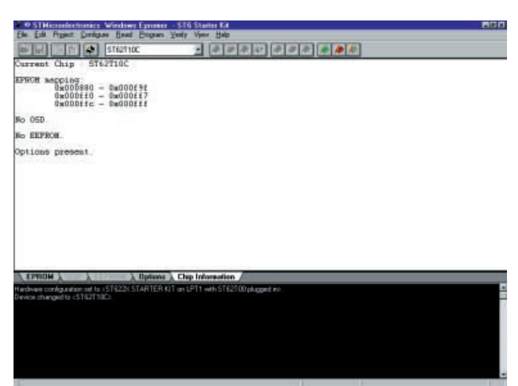


Fig.17 Cliccando sulla scritta Chip Information, potete avere delle utili informazioni sul micro prescelto.

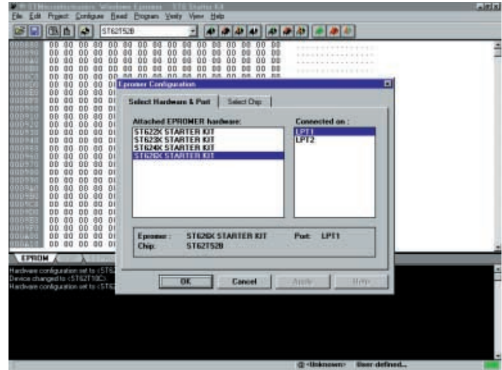


Fig.18 Per programmare un microprocessore ST62T60, dovete usare il programmatore LX.1325 e selezionare ST626X.

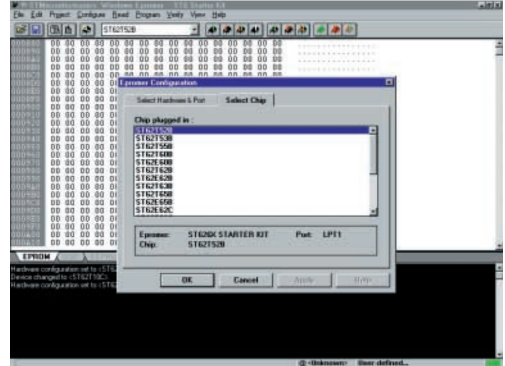


Fig.19 I micro da programmare possono essere selezionati anche cliccando sulla scritta Select Chip di questa finestra.

un clock interno ausiliario minore di **1 MHz**, nel caso venisse a mancare il clock dello stadio oscillatore principale. Se lasciate **[0]** questa funzione risulta **disattivata**, se invece mettete **[1]** questa funzione viene **attivata**.

Oscillator selection – Lasciando **[0]** dovrete utilizzare per lo stadio oscillatore un **quarzo**; se invece mettete **[1]** potrete escludere il quarzo ed inserire tra il piedino **Osc.out** e la **massa** una resistenza il cui valore determinerà la frequenza di clock. Con un valore di **470.000 ohm** otterrete una frequenza di **1 MHz** circa, con un valore di **100.000 ohm** otterrete una frequenza di **3 MHz** circa e con un valore di **47.000 ohm** otterrete una frequenza di circa **5 MHz**. La frequenza generata **non** risulterà però stabile come quella di un quarzo.

POR delay – Lasciando **[0]** avrete un **ritardo** sull'esecuzione della prima istruzione dopo che si è verificato un **reset** di **2.048 cicli**; se mettete **[1]** avrete un **ritardo** sull'esecuzione della prima istruzione di ben **32.768 cicli**.

Watchdog activation – Lasciando **[0]** potrete attivare e disattivare il **watchdog** tramite istruzioni software; se mettete **[1]** il **watchdog** sarà di tipo hardware e non potrete disattivarlo tramite nessuna istruzione software.

PB0-1 – Lasciando **[0]** i pin **PB0** e **PB1** risultano collegati con una resistenza di pull-up al positivo di alimentazione; se mettete **[1]** viene **esclusa** su questi due piedini la resistenza che li collega al positivo di alimentazione.

PB2-3 – Lasciando **[0]** i pin **PB2** e **PB3** risultano collegati con una resistenza di pull-up al positivo di alimentazione; se mettete **[1]** viene **esclusa** su questi due piedini la resistenza che li collega al positivo di alimentazione.

TIMER pin pull-up – Se mettete **[1]** il pin del **timer** viene collegato al positivo di alimentazione tramite una resistenza interna.

NMI pin pull-up – Se mettete **[1]** il pin del **NMI** viene collegato al positivo di alimentazione tramite una resistenza interna.

External STOP – Se mettete **[1]** ed il pin **NMI** si trova a **livello logico 1**, potrete eseguire la funzione di **Stop** anche con il **watchdog hardware**. In passato questa istruzione di **Stop** poteva essere eseguita solo con **watchdog software**.

Read-Out Protection – Se mettete **[1]** proteggerete il micro in lettura.

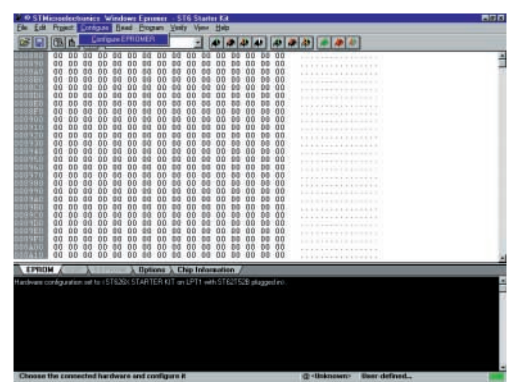


Fig.20 Dopo aver lanciato il programma, cliccate sulla scritta Configure e poi nuovamente su Configure Epromer.

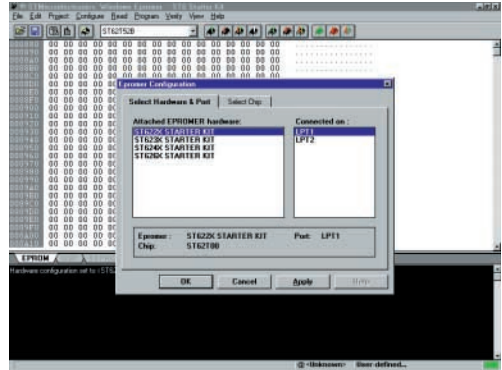


Fig.21 Ammesso di voler programmare un micro ST62E10C, come prima operazione andate sulla prima riga ST622X.

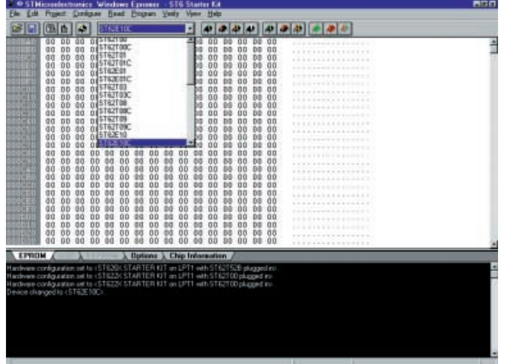


Fig.22 Dopo aver cliccato sulla freccia a V, ricercate la sigla ST62E10C corrispondente al micro da programmare.

LVD (HLVD) enabled – Lasciando **[0]** il **reset** viene attivato solo portando a **livello logico 0** il pin **reset** oppure all'accensione del micro. Se invece mettete **[1]** la condizione di **reset** viene automaticamente attivata quando la tensione di alimentazione scende al disotto dei **3,7-3,5 volt**, e disattivata automaticamente quando la tensione sale oltre i **4 volt**.

ADC Synchro – Lasciando **[0]** la lettura dell'**A/D** viene eseguita non appena si comanda lo **start convertion**. Se mettete **[1]** potete posizionare il micro sulla funzione **Wait** in modo da ridurre il "rumore" durante la lettura **A/D**. Per ottenere questa lettura bisogna eseguire lo **start conversion** dell'**A/D** e poi si deve obbligatoriamente eseguire l'istruzione di **Wait**.
Terminata la conversione, viene generata una richiesta di **interrupt** dell'**A/D** che automaticamente permette l'uscita dalla condizione di **Wait**.

ESEMPIO di programmazione ST62E10C

Supponiamo che abbiate un file già compilato, che potreste aver chiamato ad esempio **Prova.Hex**, e che lo abbiate memorizzato nella directory **C:\ST6**.
Con questo programma desiderate programmare un micro **ST62E10C** utilizzando il nostro programmatore **LX.1170** completo dell'interfaccia **LX.1430**.

Come prima operazione dovete richiamare il programma **Epromer** e, se ancora non è stato configurato, cliccate sul menu **Configure** e selezionate la riga **ST622X**, quindi cliccate sulla piccola finestra **APPLY** e di seguito su **OK** (vedi fig.21).

Dopodiché cliccate sulla freccia a **V** posta sulla destra della sigla del micro (vedi fig.22) per far comparire a video tutte le **sigle** dei micro.

Ora andate sulla scritta **ST62E10C** e con un clic selezionate questo micro.

Per caricare il programma andate sulla scritta **File**, posta in alto nella riga dei menu, e cliccando una sola volta col mouse apparirà una finestra e qui cliccate sulla scritta **Open** (vedi fig.23).
Ora andate nella finestra **C:** e cercate la directory **ST6** e qui cliccate **2 volte**.

Sulla finestra di sinistra appariranno tutti i file **.HEX** e nel nostro esempio selezionate la scritta **Prova.Hex** e poi cliccate su **OK** (vedi fig.25).
In questo modo avrete caricato in memoria il file per programmare questo micro.
Ora potete cliccare su **Options** per far apparire

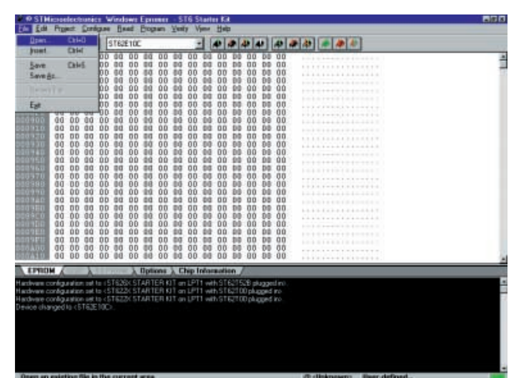


Fig.23 Per caricare il programma in memoria, dovete cliccare sulla riga File posta in alto, poi sulla scritta Open.

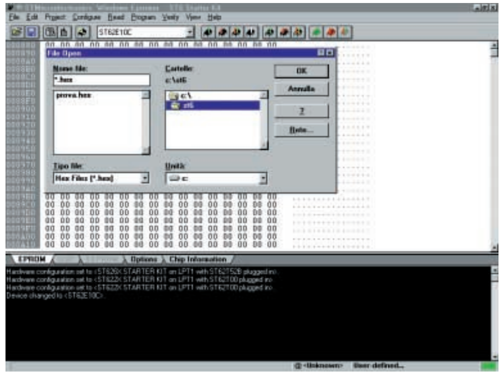


Fig.24 Dopo aver cliccato su Open, apparirà una finestra e qui ricercherete la directory C:\ST6 che contiene il programma.

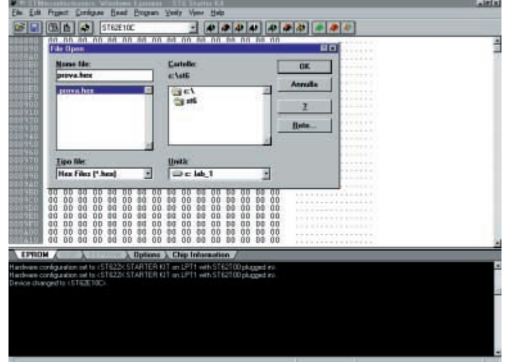


Fig.25 Dopo aver selezionato la directory C:\ST6, nel riquadro di sinistra appariranno tutti i file con estensione .Hex.

le righe dell'**option byte** (vedi fig.27).

[0] D0 OSG enabled
[0] D1 Watchdog activation
[0] D2 TIMER pin pull-up enabled
[0] D3 NMI pin pull-up enabled
[0] D6 Oscillator selection
[0] D7 Read-Out Protection
[0] D8 LVD Reset
[0] D9 External STOP MODE Control

Se volete modificare l'opzione **Watchdog**, portate il cursore sullo **[0]** e cliccate **2 volte**: in questo modo apparirà **[1]**.

Se volete **proteggere** il micro in lettura dovete portare il cursore sullo **[0]** della riga **Read-Out protection** e cliccare velocemente **2 volte** in modo che appaia **[1]**.

Ora tornate sulla riga **Eprom** posta in basso a sinistra e cliccate su questa scritta per far apparire la finestra di fig.26.
Per programmare il micro cliccate su **Program** poi sulla finestra **All**.

Le scritte poste nella parte alta di questa finestra sono già molto intuitive, comunque vi diciamo che andando su **READ** potrete leggere il contenuto del micro a patto che questo **non risulti protetto** in lettura.

Andando sulla scritta **VERIFY** e poi su **Blanck Check** potrete verificare se il micro risulta ancora **vergine** o contiene già un programma. Questa funzione potrebbe risultare utile per verificare se la lampada **ultravioletta** l'ha totalmente cancellato.

ESEMPIO di programmazione ST62E60C

Supponiamo che abbiate un file già compilato, che potreste aver chiamato ad esempio **Prova.Hex**, e di averlo memorizzato nella directory **C:\ST626**.
Con questo programma desiderate programmare un micro **ST62E60C** utilizzando il nostro programmatore **LX.1325**.

Come prima operazione dovete richiamare il programma **Epromer** e se ancora non è stato configurato, andate sul menu e cliccate su **Configure** e selezionate la riga **ST626X**, quindi cliccate sulla piccola finestra **APPLY** e di seguito su **OK**.

Dopodiché cliccate sulla freccia a **V** posta sulla destra della sigla del micro (vedi fig.22) per far comparire a video tutte le **sigle** dei micro.

Ora andate sulla riga **ST62E60C** e con un clic selezionate questo micro.

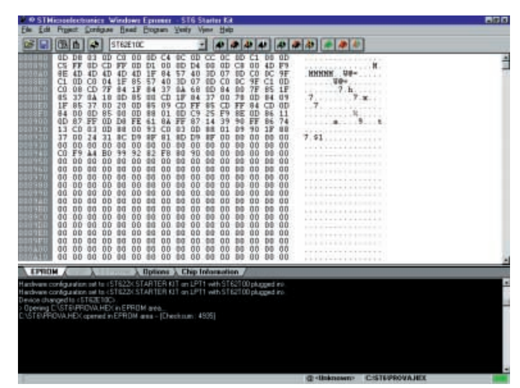


Fig.26 Avendo scelto come esempio il file Prova .Hex, dopo averlo caricato vedrete sul video il contenuto del file compilato.

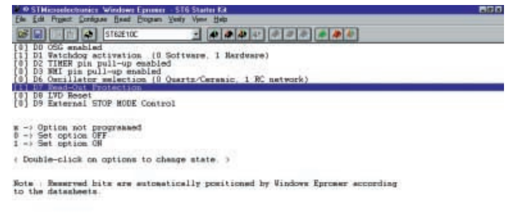


Fig.27 Selezionando Options vedrete le funzioni dell'Option Byte, che potete modificare come spiegato nel testo.

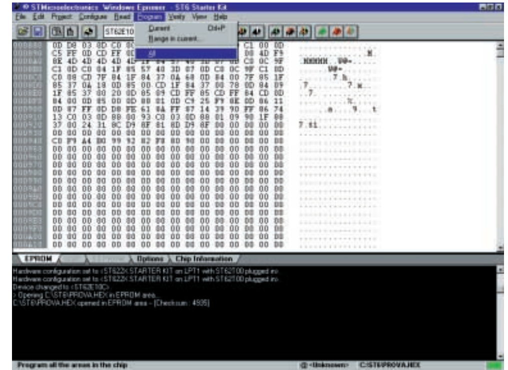


Fig.28 Per poter trasferire i dati del programma all'interno del micro dovrete selezionare Program e poi cliccare su All.

Per caricare il programma andate sulla scritta **File**, posta in alto nella riga dei menu, e cliccando una sola volta col mouse apparirà una finestra (vedi fig.23) dove cliccherete sulla scritta **Open**.
Ora andate nella finestra **C:** e cercate la directory **ST626** per cliccarci sopra **2 volte**.

Sulla finestra di sinistra appariranno tutti i file **.HEX** e nel nostro esempio selezionate la scritta **Prova.Hex** e poi cliccate su **OK**.
In questo modo avrete caricato in memoria il file per programmare questo micro.
Ora potete cliccare su **Options** per far apparire le righe dell'**option byte**.

[0] D0 OSG enabled
[0] D1 Oscill select
[0] D2 POR delay
[0] D3 Watchdog activation
[0] D4 PB0-1 pins pull-up disabled
[0] D5 PB2-3 pins pull-up disabled
[0] D6 Extern STOP mode enabled
[0] D7 Read-Out Protection
[0] D8 HLVD enabled
[0] D9 NMI pin pull-up enabled
[0] D12 ADC Synchro

Se volete **proteggere** il micro in lettura, dovete portare il cursore sullo **[0]** della riga **Read-Out protection** e cliccare velocemente **2 volte** in modo che appaia **[1]**.

Con questo micro potete visualizzare il contenuto della **EEprom** cliccando in basso sulla riga **EEprom**. Nella schermata che appare potete modificare manualmente il contenuto di questa area di memoria.

Ora ritornate sulla riga **Eprom** posta in basso a sinistra e cliccate su questa scritta per far apparire la finestra di fig.26.
Per programmare il micro cliccate su **Program** poi sulla finestra **All**.

COSTO di REALIZZAZIONE

Tutti i componenti necessari per realizzare questa interfaccia **LX.1430** (vedi fig.4), compresi due connettori e un piccolo mobile plastico .... € 5,40

Costo del circuito stampato **LX.1430** .... € 1,03

Programma **DFST6/C** sotto Windows .... € 7,75

Nota = Questa inferfaccia serve solo per usare il programmatore **LX.1170** sotto **Windows**.
Il programmatore **LX.1325** non ha bisogno di questa interfaccia, ma solo del programma **DFST6/C**.

COME UTILIZZARE

Molti consigliano di usare nei programmi per gli ST6 le direttive del linguaggio Assembler, ma pochi spiegano come si fa. E' inutile consigliare di trasformare le istruzioni in una "macro", se non si spiegano quali accorgimenti adottare per evitare errori. In questo articolo vi spieghiamo ciò che occorre sapere per usare correttamente la direttiva .macro.

Se non programmate i micro **ST6**, questo articolo sarà per voi poco interessante, ma per le piccole Industrie che utilizzano questo microprocessore non è così, tant'è vero che insistono affinché vengano dedicate più pagine sulla rivista a questo argomento, perché quello che noi spieghiamo non si trova in nessun manuale.
Oggi li accontenteremo spiegando come si possa trasformare un gruppo di istruzioni in una **macro**.

Nella **stesura** di un programma capita di frequente di dover utilizzare delle sequenze di **istruzioni** che sono già state scritte per programmi precedenti, come ad esempio quelle per eseguire delle **somme**, per configurare le **porte**, per **visualizzare** dei dati sul display ecc.
In questi casi capita spesso che le istruzioni vengano **riscritte** con il rischio di inserire degli **errori**.

Chi ha un po' di esperienza si avvale di un altro metodo, va cioè alla **ricerca** dei programmi in cui sa che ci siano queste istruzioni, poi, adoperando le funzioni dell'**editor** normalmente usate nella videoscrittura, seleziona quelle di cui ha bisogno e le **incolla** direttamente nel nuovo programma, apportando eventuali modifiche per renderle compatibili alle nuove esigenze.

Questa soluzione è sicuramente molto valida, ma può presentare piccoli inconvenienti.
Se dopo avere **incollato** le istruzioni nel nuovo programma ci si accorge che c'è un **errore** oppure si scopre che è possibile perfezionarle, come ricordare in quali altri programmi sono state utilizzate per poterle correggere?

Ebbene, non tutti sanno che c'è un'altra soluzione, sicuramente molto valida, che consiste nel selezionare tutti i **blocchi** di istruzioni che possono servire in altri programmi, per memorizzarli in una istruzione **macro** a cui verrà assegnato un **nome**.

Procedendo in questo modo, ogni volta che si scriverà un **nuovo** programma e serviranno queste i-

struzioni, **non** si dovrà più perdere tempo per andarle a cercare, ma sarà sufficiente inserire nel punto desiderato il **nome** della **macro** che le contiene ed **assemblare** il programma.

Questa soluzione offre molti vantaggi ai programmatori, perché se ci si accorge che nelle istruzioni è presente un **errore** oppure che è possibile **migliorarle**, basta correggere la sola **macro** per avere la certezza che in tutti i programmi in cui è stata utilizzata o che verrà utilizzata sarà perfetta.

Per rendere ancora più agevole l'uso di queste **macro** non va dimenticato di assegnare dei **nomi** che siano il più possibile significativi, così da poter capire immediatamente quali funzioni eseguono.

Di conseguenza se avete una macro che esegue delle **somme**, datele il nome **sommat**, se avete una macro che **configura** le **porte**, datele il nome **defport** e così via.

corrispondenza dell'etichetta **lamp4** e sono:

```
lamp4
            ldi         wdog,0feh
            jrr         4,port_b,lamp0
            res         4,port_b
            jp          lamp1
lamp0       set         4,port_b
lamp1       call        delay
            jp          lamp4
```

Per creare una **macro** da adoperare in altri programmi dobbiamo innanzitutto scegliere un **nome** che ci ricordi quale funzione svolge questo gruppo di istruzioni e poiché fanno **lampeggiare** un diodo **led** potremmo chiamare la **macro**:

ledflash

Ricordatevi sempre che i **nomi** non possono mai superare gli **8 caratteri**.

la DIRETTIVA .MACRO

Inoltre vi consigliamo di inserire sempre un **commento** che spieghi quale funzione esegue la **macro**, perché col tempo è facile dimenticarsene.

Un altro consiglio che vi diamo è quello di creare una **directory**, che potrete ad esempio chiamare **dirmacro**, nella quale memorizzare tutte le vostre **macro**, in modo da avere una **libreria** sempre aggiornata e facile da consultare.

Il modo più semplice per imparare a **creare** e a **utilizzare** una **macro** è sicuramente quello di affidarsi alla pratica, pertanto di seguito troverete alcuni esempi per trasformare una sequenza di istruzioni in una **macro**.

COME creare una MACRO

Supponiamo di avere un semplice programma chiamato **LAMPLED.ASM**, che provvede a far **lampeggiare** un diodo **led** collegato sul **pin 4** di **Porta B** di un **ST6210**.

Nella fig.1, riportata nella pagina seguente, potete vedere il listato completo di questo programma.

Le istruzioni per il lampeggio sono state poste in

Per trasformare queste istruzioni in una **macro** occorre utilizzare due sole direttive:

.macro e **.endm**

che vanno scritte secondo questo formato:

```
.macro    nome [,variab] [,\num] [,?label]
.endm
```

Nel nostro caso la direttiva **.macro** va inserita nella riga precedente al gruppo di istruzioni che vogliamo trasformare in una macro e la **.endm** nella riga successiva al gruppo di istruzioni.

Dopo la direttiva **.macro** scriviamo il **nome** scelto, cioè **ledflash**, e nelle voci tra parentesi quadre **[ ]** racchiuderemo i tre parametri opzionali che potremo inserire prima della **compilazione** per rendere la macro parametrizzabile.

Con il programma di **editor** che utilizziamo normalmente per scrivere i programmi apriamo un nuovo documento e iniziamo a scrivere:

```
            .macro      ledflash
```

Come vedete la prima istruzione è la direttiva **.ma-**

PROGRAMMA per far LAMPEGGIARE un LED sul PIN 4 di PORTA B

```
        .title   "LAMPLED"                  ;1  titolo del programma
        .vers    "ST62E10"                  ;2  tipo di microprocessore
        .romsize 2
;+-----------------------------------------------------------------------+
;|                        VARIABILI DEL MICRO                            |
;+-----------------------------------------------------------------------+
; Attenzione: Le righe da 3 a 24 non vanno mai modificate

a       .def     0ffh                       ;3  accumulatore
x       .def     080h                       ;4  registro x
y       .def     081h                       ;5  registro y
v       .def     082h                       ;6  registro v
w       .def     083h                       ;7  registro w
port_a  .def     0c0h                       ;8  porta A
port_b  .def     0c1h                       ;9  porta B
port_c  .def     0c2h                       ;10 porta C
pdir_a  .def     0c4h                       ;11 direzione porta A
pdir_b  .def     0c5h                       ;12 direzione porta B
pdir_c  .def     0c6h                       ;13 direzione porta C
popt_a  .def     0cch                       ;14 opzioni porta A
popt_b  .def     0cdh                       ;15 opzioni porta B
popt_c  .def     0ceh                       ;16 opzioni porta C
ior     .def     0c8h                       ;17 registro interrupt
addr    .def     0d0h                       ;18 dato convertitore A/D
adcr    .def     0d1h                       ;19 registro convertitore A/D
psc     .def     0d2h                       ;20 registro del timer
tcr     .def     0d3h                       ;21 contatore del timer
tscr    .def     0d4h                       ;22 registro prescaller del timer
wdog    .def     0d8h                       ;23 registro del watchdog
drw     .def     0c9h                       ;24 registro data rom window
;************************************************************************
;***                       Settaggio iniziale                           *
;************************************************************************
        .org     0880h                      ;25 per ST62E10 ST62E15
inizio                                      ;26 etichetta inizio
        ldi      wdog,0ffh                  ;27 ricarica il watch dog

;***    setta la porta B
        ldi      port_b,00010000b           ;36 bit 0 = non usato
        ldi      pdir_b,00010000b           ;37 bit 1 = non usato
        ldi      popt_b,00010000b           ;38 bit 2 = non usato
                                            ;39 bit 3 = non usato
                                            ;40 bit 4 = uscita per led 1
                                            ;41 bit 5 = non usato
                                            ;42 bit 6 = non usato
                                            ;43 bit 7 = non usato


        ldi      adcr,00000000b             ;48 disabilita il convertitore A/D
        ldi      tscr,00000000b             ;49 disabilita il TIMER
        ldi      ior, 00000000b             ;50 disabilita tutti gli interrupt
        reti                                ;51 istruzione da non togliere
        jp       main                       ;52 salta al programma principale

;NOTA  Se usiamo un ST62E20 o un ST62E25 nella riga 25 occorre mettere 080h
```

```
;*************************************************************************
;***                    Gestori di interrupt                           ***
;*************************************************************************
; In questo programma non vengono usati gli interrupt ma bisogna ugualmente
; inserire queste righe

ad_int  reti                            ;53 interrupt del convertitore A/D
tim_int reti                            ;54 interrupt del timer
BC_int  reti                            ;55 interrupt delle porte B e C
A_int   reti                            ;56 interrupt della porta A
nmi_int reti                            ;57 interrupt del piedino NMI
;*************************************************************************
;***                        SUBROUTINE                                 ***
;*************************************************************************
delay   ldi     wdog,0feh               ;58
        ldi     y,20                    ;59
dela1   ldi     wdog,0feh               ;60
        ldi     x,20                    ;61
dela2   dec     x                       ;62
        jrnz    dela2                   ;63
        dec     y                       ;64
        jrnz    dela1                   ;65
        ret                             ;66
;*************************************************************************
;***                   PROGRAMMA PRINCIPALE                            ***
;*************************************************************************
main
        ldi     wdog,0feh               ;67 ricarica il watch dog
        res     4,port_b                ;68 diseccita il led 1 (L1)
lamp4                                   ;69 etichetta per la ripetizione
        ldi     wdog,0feh               ;70 ricarica il watch dog
        jrr     4,port_b,lamp0          ;71 se spento vai a lamp0
        res     4,port_b                ;72 altrimenti lo spengo
        jp      lamp1                   ;73
lamp0   set     4,port_b                ;74 se spento lo accendo
lamp1   call    delay                   ;75 ritardo
        jp      lamp4                   ;76 ritorna
;*************************************************************************
;***                  VETTORI DI INTERRUPTS                            ***
;*************************************************************************
; Attenzione: Le righe da 98 a 109 non vanno mai modificate

        .org    0ff0h                   ;98
        jp      ad_int                  ;99  interrupt del convertitore A/D
        jp      tim_int                 ;100 interrupt del timer
        jp      BC_int                  ;101 interrupt porte B e C
        jp      A_int                   ;102 interrupt porta A
        .org    0ffch                   ;103
        jp      nmi_int                 ;104 interrupt piedino nmi
        jp      inizio                  ;105 salta a inizio al reset

        .end                            ;109 fine del programma
```

Fig.1 Il listato completo del programma sorgente che provvede a far lampeggiare un diodo led collegato sul piedino 4 di porta B di un ST62E10. Compilando questo programma otterrete un programma in formato intel eseguibile con estensione .HEX.

cro seguita dal nome **ledflash**, con il quale d'ora in poi **dovrà** essere richiamata questa **macro** all'interno di un programma.

Tralasciamo per ora i parametri racchiusi tra **[ ]** perché li spiegheremo con gli esempi successivi.

Prima della direttiva **.macro** non va inserita alcuna etichetta perché il **compilatore** la ignorerebbe.

A questo punto inseriamo i comandi che vogliamo raggruppare in una **macro**.

```
            jrr         4,port_b,lamp0
            res         4,port_b
            jp          lamp1
lamp0       set         4,port_b
lamp1
            .endm
```

Come avrete notato nella riga successiva alla etichetta **lamp1** abbiamo inserito la direttiva **.endm**, per segnalare al compilatore la fine delle istruzioni della macro **ledflash**.

La direttiva **.endm** va tassativamente inserita come ultima istruzione di qualsiasi **macro**.

Anche in questo caso davanti alla direttiva **.endm** non scrivete alcuna etichetta, perché il compilatore la ignorerebbe.

Ora che la macro è stata creata dobbiamo salvarla in un file che chiameremo **LEDFLASH.LMA**.

L'estensione **.LMA** sta per **L**ibreria **M**acro **A**ssembler e serve a ricordarci che questo file è una **macro** e non un programma completo.

Ovviamente potrete chiamarla con il nome che ritenete più opportuno e scegliere l'estensione che ritenete più valida, ma non usate mai le estensioni tipo **.EXE**, **.HEX**, **.GIF**, **.HTM** o **.INI** ecc. perché sono **riservate** e potrebbero dar luogo a problemi.

Potrete anche lasciare l'estensione **.ASM** e salvare il file in un'altra directory chiamata **MACRO**.

Creata una **macro**, possiamo inserirla nel programma **lampled** scrivendo semplicemente queste istruzioni:

```
lamp4
            ldi         wdog,0feh
            ledflash
            call        delay
            jp          lamp4
```

Come vedete, rispetto al listato di fig.1 abbiamo sostituito le istruzioni che eseguivano il lampeggio con la sola parola **ledflash**.

Se ora proviamo a ricompilare il programma **LAMPLED.ASM**, avremo però la sgradita sorpresa della segnalazione di questo errore:

Error LAMPLED.ASM
(67) undefined macro : ledflash

Eppure noi abbiamo creato correttamente la macro **ledflash** dentro al file **LEDFLASH.LMA**.
Il compilatore quando arriva alla parola **ledflash** non la riconosce come istruzione assembler e cerca di interpretarla come **macro**, ma non la trova ancora esattamente definita.

Per definirla esattamente dobbiamo inserire nel programma **LAMPLED.ASM** anche l'istruzione:

```
            .input      "ledflash.LMA"
```

Sebbene non ci sia un punto preciso nel quale scrivere questa istruzione, noi consigliamo di inserirla dove c'è la dichiarazione della configurazione delle porte così da essere immediatamente notata.

Conoscete la direttiva **.input** (vedi rivista **N.182**) e perciò sapete che quando il compilatore la incontra inserisce nel programma che sta compilando tutto ciò che trova memorizzato nel file segnalato tra virgolette **" "**.

Nel nostro caso il compilatore cerca il file generato in precedenza e chiamato **ledflash.LMA** e inserisce nel programma **LAMPLED.ASM** tutte le istruzioni lì contenute.

Nel nostro esempio verranno quindi inserite:

```
            .macro      ledflash
            jrr         4,port_b,lamp0
            res         4,port_b
            jp          lamp1
lamp0       set         4,port_b
lamp1
            .endm
```

A questo punto, nel ricompilare il programma, quando il compilatore arriverà all'istruzione:

```
lamp4
            ldi         wdog,0feh
            ledflash
            call        delay
            jp          lamp4
```

capirà che **ledflash** è una **macro** e la sostituirà con le istruzioni prelevate da **LEDFLASH.LMA**.
Il sorgente **LAMPLED.ASM** continuerà ad essere scritto come segue:

```
lamp4
        ldi       wdog,0feh
        ledflash
        call      delay
        jp        lamp4
```

mentre il file **LAMPLED.HEX** conterrà in formato eseguibile le seguenti istruzioni:

```
lamp4
        ldi       wdog,0feh
        jrr       4,port_b,lamp0
        res       4,port_b
        jp        lamp1
lamp0   set       4,port_b
lamp1   call      delay
        jp        lamp4
```

```
Ind.  Codice  Label   Mnemonico
 089D 0DD8FE delay     ldi      wdog,FEh
 08A0 0D8114           ldi      y,14h
 08A3 0D8014 dela1     ldi      x,14h
 08A6 1D     dela2     dec      x
 08A7 F0               jrnz     dela2
 08A8 5D               dec      y
 08A9 C8               jrnz     dela1
 08AA CD               ret
 08AB 0DD8FE main      ldi      wdog,FEh
 08AE 2BC1             res      4,port_b
> 08B0 0DD8FE lamp4    ldi      wdog,FEh
 08B3 23C104           jrr      4,port_b,lamp0
 08B6 2BC1             res      4,port_b
 08B8 C98B             jp       lamp1
 08BA 3BC1   lamp0     set      4,port_b
 08BC D189   lamp1     call     delay
 08BE 098B             jp       lamp4
 08C0 FFFF             dec      a
```

Fig.2 Nella simulazione del programma LAMPLED.HEX, l'istruzione ledflash è stata sostituita con le istruzioni della macro.

A riprova di quanto detto, abbiamo simulato il programma in esecuzione col simulatore ST626 e in fig.2 è visibile la parte in cui avevamo inizialmente sostituito le istruzioni di lampeggio con il comando **ledflash**. Come potete notare, **non** esiste più **ledflash**, perché al suo posto il compilatore ha inserito le istruzioni della **macro**.

Abbiamo poi compilato **LAMPLED.ASM** con l'opzione **–L** ottenendo così anche il listato del programma (**LAMPLED.LIS**) e in fig.3 è visibile il punto in cui è stata inserita la direttiva:

```
        .input        "LEDFLASH.LMA"
```

Nota: nella rivista **N.194** vi abbiamo insegnato come leggere i listati **.LIS**.

Come potete notare, subito dopo questa direttiva è stata inserita la sequenza di istruzioni contenuta nel file **LEDFLASH.LMA**, che non occupa nessuna area di memoria, come visibile nella parte sinistra del tabulato sotto la dicitura:

—— SOURCE FILE : LEDFLASH.LMA ——

In fig.4 è invece visibile il punto del listato in cui avevamo inserito la macro **ledflash**.

Sottolineiamo ancora una volta che il compilatore ha sostituito la **macro** con le relative istruzioni e anche se nella riga **109** il nome **ledflash** è rimasto, non occupa nessuna **area** di memoria come visibile alla sua sinistra.

LE SPECIFICHE della DIRETTIVA .MACRO

È abbastanza intuitivo che la macro **ledflash** così com'è può essere richiamata solo nei programmi che utilizzano il **pin 4** di **porta B**.

Se volessimo utilizzare un qualsiasi **piedino** di una qualsiasi **porta** dovremmo necessariamente

```
  37                              37          .input    "LEDFLASH.LMA"
--- SOURCE FILE : LEDFLASH.LMA ---
  38                           1   1          .macro    ledflash
  39                           1   2          jrr       4,port_b,lamp0
  40                           1   3          res       4,port_b
  41                           1   4          jp        lamp1
  42                           1   5  lamp0
  43                           1   6          set       4,port_b
  44                           1   7  lamp1
  45                           1   8          .endm
```

Fig.3 Parte del listato LAMPLED.LIS generato con l'opzione –L in cui è stata inserita la direttiva .input. Come potete notare, dopo questa direttiva il compilatore inserisce nel programma ciò che trova nel file segnalato tra virgolette, cioè nel file "ledflash.lma".

```
104 P00 08AB        P00 08AB       96  main
105 P00 08AB 0DD8FE P00 08AB       97           ldi      wdog,0feh
106 P00 08AE 2BC1   P00 08AE       98           res      4,port_b
107 P00 08B0        P00 08B0       99  lamp4
108 P00 08B0 0DD8FE P00 08B0      100           ldi      wdog,0feh
109                               101           ledflash
110 P00 08B3 23C104 P00 08B3   1    2           jrr      4,port_b,lamp0
111 P00 08B6 2BC1   P00 08B6   1    3           res      4,port_b
112 P00 08B8 C98B   P00 08B8   1    4           jp       lamp1
113 P00 08BA        P00 08BA   1    5  lamp0
114 P00 08BA 3BC1   P00 08BA   1    6           set      4,port_b
115 P00 08BC        P00 08BC   1    7  lamp1
116                            1    8           .endm
117 P00 08BC D189   P00 08BC      102           call     delay
118 P00 08BE 098B   P00 08BE      103           jp       lamp4
119                               104
```

Fig.4 Parte del listato LAMPLED.LIS generato con l'opzione –L in cui sono state inserite le istruzioni contenute nella macro ledflash (vedi da riga 109 a riga 116). Vi facciamo notare che sebbene il nome ledflash sia ancora presente, non occupa nessuna area di memoria: infatti, a destra del numero 109 non c'è nessuna scritta.

creare una nuova **macro** modificando i soli parametri racchiusi tra parentesi quadre **[ ]**.

Infatti, quando si dichiara una **.macro**, oltre al **nome** possiamo definire **tre** differenti categorie di **parametri** che in fase di compilazione verranno "passate" e sostituite alle istruzioni inserite nella macro stessa rendendola più duttile.
In questo modo potremo modificare la stessa macro a seconda delle necessità e delle circostanze.

Queste tre categorie rappresentano la cosiddetta **COMMON AREA** o area di Link della macro.

Cerchiamo di spiegarci meglio procedendo ancora una volta con degli esempi pratici.

Il parametro [,variab]

Se analizziamo il file **LEDFLASH.LMA** vediamo che le **istruzioni** inserite sono costituite essenzialmente da **4 gruppi**, cioè:

– le **istruzioni** Assembler vere e proprie:
  jrr res jp set

– le **variabili**:
  port_b

– le **costanti** numeriche:
  4

– le **labels** o etichette interne:
  lamp0 lamp1

Con il parametro **[,variab]** possiamo rendere parametrica la variabile **port_b**, riuscendo ad ottenere una macro che esegue il lampeggio sempre sul pin 4, ma di una qualsiasi **porta** del micro, se passiamo questa informazione dal programma principale alla **macro**.

Riprendiamo perciò il nostro file **LEDFLASH.LMA** e modifichiamolo in questo modo:

```
         .macro   ledflash cheporta
         jrr      4,cheporta,lamp0
         res      4,cheporta
         jp       lamp1
lamp0    set      4,cheporta
lamp1
         .endm
```

In altre parole dopo **ledflash** abbiamo inserito uno **spazio** seguito dalla parola **cheporta**, poi abbiamo sostituito tutte le istruzioni che usano **port_b** sempre con la scritta **cheporta**.

In questo modo abbiamo inserito nella **macro** il parametro **[,variab].**

Nota: in questo caso, essendo **cheporta** il primo parametro della macro non si deve inserire alcuna virgola dopo **ledflash**.

Noi abbiamo utilizzato il termine **cheporta**, ma potevamo usare qualsiasi altro nome, ad esempio **pippo**, **finestra** ecc., anche se è consigliabile u-

sare sempre delle parole che identifichino il tipo di **variabile** da modificare in modo inequivocabile.

Ora modifichiamo il programma **LAMPLED.ASM** scrivendo nella riga della **macro**:

ledflash port_

In coda a **ledflash** abbiamo dunque inserito la variabile **port_b**.

Assemblando il programma **LAMPLED.ASM**, quando il compilatore troverà **ledflash port_b**, la sostituirà con le istruzioni di **ledflash.LMA** sostituendo il parametro **cheporta** con **port_b**.

Se anziché scrivere:

ledflash port_b

avessimo scritto:

ledflash port_c

il compilatore avrebbe caricato al suo posto le istruzioni di **ledflash.LMA** sostituendo il parametro **cheporta** con **port_c**.

In questo modo abbiamo usato la **COMMON AREA** della **macro** per passare un parametro **variabile**, ottenendo così una macro che può funzionare indifferentemente su **tutte** le **porte** del micro.

Se nel programma **LAMPLED.ASM** avessimo scritto solamente:

ledflash

dimenticandoci di completarla con **port_b**, il compilatore si sarebbe trovato nell'impossibilità di sostituire **cheporta** ed avrebbe segnalato l'**errore** che abbiamo riportato in fig.5, pertanto quando si utilizzano i **parametri** della direttiva **.macro** bisogna fare molta attenzione.

Il parametro [,\num]

Pur avendo reso parametrica la **porta**, il lampeggio avverrà sempre sul **pin 4** della porta prescelta, quindi per scegliere un diverso **pin**, dovremo variare nella **macro** il numero **4.**

Per scegliere un **pin** diverso si utilizza il parametro **[,\num]** modificando il file **LEDFLASH.LMA** nel modo seguente:

```
        .macro   ledflash cheporta,\chepin
        jrr      chepin,cheporta,lamp0
        res      chepin,cheporta
        jp       lamp1
lamp0   set      chepin,cheporta
lamp1
        .endm
```

Come avrete notato, dopo **cheporta** abbiamo inserito una **virgola**, una **barra rovesciata** ed il parametro **chepin** sostituendolo al numero **4** presente nelle istruzioni.

Importante: la **barra rovesciata** prima di **chepin** va inserita solo nella riga della direttiva **.macro** per indicare che qui verrà inserito un nuovo valore numerico **[,\num]**. Le istruzioni in cui abbiamo inserito **chepin** richiedono infatti, in quella posizione, un **numero** e non il contenuto di una variabile.

In questo caso abbiamo usato il nome **chepin**, ma potevamo chiamarlo con il nome **pinout** ecc.

Ora dobbiamo modificare nel programma **LAMPLED.ASM** la macro **ledflash** come segue:

ledflash port_b,4

Quando il Compilatore assembler troverà **ledflash port_b,4** caricherà al suo posto le istruzioni di **LEDFLASH.LMA** sostituendo il parametro **cheporta** con **port_b** ed il parametro **chepin** con **4.**

```
ST6 MACRO-ASSEMBLER version 4.00  - August 1992
Error  LEDFLASH.LMA 2: (-1)   syntax error
Error LEDFLASH.LMA 2: (110) data addresses must be in the range [0..0ffh]
Execution time: 1 second(s)
2 errors detected
No object created
```

Fig.5 Questo errore è stato generato perché nel file LEDFLASH.LMA è stata parametrizzata una variabile, che però non è stata definita nel programma in formato .ASM. A causa di ciò il compilatore non ha potuto generare il file in formato .HEX.

```
Error LEDFLASH.LMA 2: (20) operand expected: 3-bit number
Warning LEDFLASH.LMA 2: (91) 2> r/w access control not done on data-space operand
Error LEDFLASH.LMA 3: (20) operand expected: 3-bit number
Warning LEDFLASH.LMA 3: (91) 2> r/w access control not done on data-space operand
Error LEDFLASH.LMA 5: (20) operand expected: 3-bit number
Warning LEDFLASH.LMA 5: (91) 2> r/w access control not done on data-space operand
Execution time: 0 second(s)
3 errors detected
 3 warnings
No object created
```

Fig.6 In questo caso l'errore è stato generato perché pur avendo definito nel file in formato .ASM i parametri da associare alla macro ledflash, la variabile port_b e la costante numerica 4 sono state invertite. L'uso dei parametri opzionali rende la macro più duttile e perciò adattabile alle diverse necessità e circostanze, ma è necessario fare molta attenzione quando si definiscono questi parametri nei programmi sorgente.

Se invece avessimo scritto:

```
ledflash port_b,6
```

quando il compilatore avrebbe assemblato il programma avrebbe sostituito i parametri **cheporta** e **chepin** presenti in **LEDFLASH.LMA** con **port_b** e con **6**. Il **lampeggio** in questo ultimo caso sarebbe avvenuto sul pin **6** della porta **B** e non più sul pin **4** della porta **B.**
Come avrete capito bastano poche modifiche per far lampeggiare il diodo led posto su un qualsiasi **piedino** delle **porte** del micro.
Qualcuno certamente si starà chiedendo cosa avviene se per **errore** si scrive:

```
ledflash 4,port_b
```

In questo caso, quando il Compilatore incontra **ledflash** carica le istruzioni relative sostituendo **cheporta** con **4** e **chepin** con **port_b**, quindi segnala l'**errore** visibile in fig.6 perché nelle istruzioni successive la **macro** caricata contiene istruzioni assolutamente sbagliate:

```
        jrr     port_b,4,lamp0
        res     port_b,4
        jp      lamp1
lamp0   set     port_b,4
lamp1
```

È evidente che anche se l'uso delle **macro** è abbastanza facile è necessario prestare sempre molta attenzione nello scrivere questi parametri.

Il parametro [,?label]

A questo punto abbiamo ottenuto una **macro** che setta e resetta un qualsiasi **piedino** di una qualsiasi **porta** del micro.

Esiste però un altro problema dovuto al fatto che all'interno di **ledflash** ci sono le labels **lamp0** e **lamp1**. Infatti ogniqualvolta vorremo utilizzare questa **macro** dovremo assicurarci di **non** avere già utilizzato queste **labels** nel nostro programma principale, perché se esistono il compilatore segnalerà **errore**.

Inoltre se in più punti del programma principale noi inseriamo la macro **ledflash**, il compilatore sostituirà ad ogni **ledflash** che incontra la relativa sequenza di istruzioni ed anche in questo caso le labels **lamp0** e **lamp1** sarebbero doppie, triple ecc.

Conviene quindi sempre parametrizzare anche le **labels** interne di una **macro** utilizzando il parametro **[,?label]**.

Modifichiamo dunque il file **LEDFLASH.LMA** come qui sotto riportato:

```
                .macro  ledflash cheporta,\chepin,?chelab0,?chelab1
                jrr     chepin,cheporta,chelab0
                res     chepin,cheporta
                jp      chelab1
        chelab0
                set     chepin,cheporta
        chelab1
                .endm
```

Nella riga **.macro** abbiamo inserito, oltre ai parametri di cui abbiamo già parlato, anche i parametri **?chelab0** e **?chelab1** e nelle righe successive abbiamo sostituito la label **lamp0** con **chelab0** e la label **lamp1** con **chelab1**.
Il simbolo **?** davanti a **chelab0** e **chelab1** definisce che si tratta di una **label interna** e quindi le **label** situate internamente alla **macro** e gli eventuali salti di programma verranno automaticamente effettuati all'interno della stessa.

Ovviamente dovremo modificare il programma **LAMPLED.ASM** come segue:

ledflash port_b,4,lamp0,lamp1

Assemblando il programma, quando il compilatore incontrerà questa istruzione inserirà la relativa macro, sostituendo **cheporta** con **port_b**, **chepin** con **4**, **chelab0** con **lamp0** e **chelab1** con **lamp1**.
Abbiamo cioè generato una **macro** dove anche le **labels** usate internamente sono parametrizzate, scongiurando così il pericolo che queste possano già esistere nel programma principale ed evitando molti inconvenienti.

Ora dobbiamo soffermarci su un piccolo particolare, perché aumentando i parametri potrebbe diventare più difficoltoso gestire queste **macro**.
In presenza di **macro** molto complesse potrebbe infatti succedere che le **labels** interne siano ben più di **due** ed anche in questo caso aumenterebbero le difficoltà.
Inoltre potrebbe essere necessario richiamare più volte, in punti diversi del programma principale, la stessa macro, obbligando il programmatore ad inserire delle **labels** sempre diverse dalle precedenti.

Esistono comunque **due** soluzioni tra le quali scegliere per semplificare questo problema.

PRIMA SOLUZIONE

Consiste nell'omettere i nomi delle **labels interne** in una **macro** anche se questa li richiede.

Per utilizzare questa soluzione lasceremo invariata la macro così come è scritta in **ledflash.LMA**:

```
        .macro   ledflash cheporta,\chepin,?chelab0,?chelab1
        jrr      chepin,cheporta,chelab0
        res      chepin,cheporta
        jp       chelab1
chelab0
        set      chepin,cheporta
chelab1
        .endm
```

però nel programma **LAMPLED.ASM** dovremo inserire il richiamo della **macro** in questo modo:

ledflash port_b,4

omettendo volutamente le **labels**.

Assemblando il programma **LAMPLED.ASM**, il compilatore inserirà le istruzioni **ledflash**, sostituendo **cheporta** con **port_b** e **chepin** con **4.**

Non trovando nessuna **labels**, le sostituirà automaticamente con **L2$** e **L3$** al posto di **chelab0** e **chelab1** senza segnalare errori.

Il compilatore è quindi in grado di provvedere automaticamente alla codifica delle **label interne** della macro, perché si è utilizzato il parametro labels interne (**?chelab ecc.**), risparmiando così al softwarista l'impegno di inserirle.

Per verificare se questa soluzione risulta valida provate ad inserire in più punti del programma **LAMPLED.ASM** un richiamo della macro **ledflash**, poi riassemblatelo con l'opzione **–L.**

In fig.8 vi riportiamo il punto esatto del listato **.LIS** in cui è stato inserito il file **ledflash.LMA**, affinché possiate constatare da voi la definizione delle **labels** interne **?chelab0** e **?chelab1**.

In fig.9 sono invece riportati i due punti esatti in cui abbiamo inserito, a poca distanza l'una dall'altra, l'istruzione **ledflash port_b,4** omettendo la definizione delle **labels**.

Come potete notare, nella prima **macro** il compilatore ha sostituito **?chelab0** e **?chelab1** con:

L2$** e **L3$

mentre nella seconda le ha sostituite con:

L4$** e **L5$

Questo ci conferma che, nel caso sia previsto nei parametri della **macro** l'utilizzo delle labels interne

Fig.7 Seguendo le lezioni sulle istruzioni del linguaggio Assembler, imparare a programmare i micro ST6 sarà più semplice di quanto possiate immaginare.

(simbolo **?**), il compilatore è in grado di generare automaticamente, in caso di omissione, queste **labels** assegnandole un numero consecutivo.

Unico inconveniente che potremmo riscontrare è che le **labels** così generate siano veramente poco comprensibili rendendo molto difficoltosa la lettura del programma.

SECONDA SOLUZIONE

Consiste nello sfruttare la capacità che possiede la direttiva **.macro** di concatenare due **stringhe** passando una **sola label** interna che diventerà un **suffisso**.

Se seguirete questo nostro esempio scoprirete come in realtà sia semplice questa operazione.

Prendiamo sempre la macro contenuta nel file **ledflash.LMA** e modifichiamola come segue:

Nella riga **.macro** abbiamo inserito **?chext** al posto di **?chelab0** e **?chelab1**, mentre nelle righe successive abbiamo sostituito:

chelab0 con **lamp0'chext**
chelab1 con **lamp1'chext**

Ora prendiamo il nostro programma principale **LAMPLED.ASM** e andiamo a modificare entrambi i richiami a **ledflash**.

Nel primo richiamo scriviamo:

```
ledflash port_b,4,r1
```

e nel secondo richiamo scriviamo:

```
ledflash port_b,4,r2
```

Assemblando il programma **LAMPLED.ASM** il compilatore quando trova il primo **ledflash** inseri-

```
            .macro      ledflash cheporta,\chepin,?chext
            jrr         chepin,cheporta,lamp0'chext
            res         chepin,cheporta
            jp          lamp1'chext
lamp0'chext
            set         chepin,cheporta
lamp1'chext
            .endm
```

sce le istruzioni relative sostituendo **cheporta** con **port_b** e **chepin** con **4**, quindi ricerca le **label** che contengono **'chext** e le sostituisce con **r1** effettuando una concatenazione.
Il risultato è che la label **lamp0'chext** diventerà:

lamp0r1

e la label **lamp1'chext** diventerà:

lamp1r1

Stessa cosa avverrà quando verrà letto il secondo **ledflash** con la differenza che **lamp0'chext** diventerà **lamp0r2** e **lamp1'chext** diventerà **lamp1r2**.

A questo punto abbiamo creato una **macro** che è in grado di settare e resettare un **bit** qualsiasi di una qualsiasi **variabile** e che utilizza le **labels** interne significative parametrizzandole.
Abbiamo quindi una **macro** che può essere richiamata tranquillamente all'interno di qualsiasi programma senza alcuna precauzione.

Vi è solo una piccola regola da rispettare, cioè quando si utilizza la concatenazione di queste **la-**

```
38          38          .input 'LEDFLASH.LMA"
-- SOURCE FILE : LEDFLASH.LMA ---
39        1    1          .macro   ledflash cheporta,\chepin,?chelab0,?chelabl
40        1    2          jrr      chepin,cheporta,chelab0
41        1    3          res      chepin,cheporta
42        1    4          jp       chelabl
43        1    5  chelab0
44        1    6          set      chepin,cheporta
45        1    7  chelabl
46        1    8          .endm
```

Fig.8 Parte del listato LAMPLED.LIS generato con l'opzione –L in cui è stata inserita la definizione delle labels interne ?chelab0 e ?chelab1 (vedi riga 39).

```
119 P00 08B7        P00 08B7      104  ripeti
120 P00 08B7 0DD8FE P00 08B7      105           ldi      wdog,0feh
121                               106           ledflash port_b,4
122 P00 08BA 23C104 P00 08BA   1    2           jrr      4,port_b,L2$
123 P00 08BD 2BC1   P00 08BD   1    3           res      4,port_b
124 P00 08BF 398C   P00 08BF   1    4           jp       L3$
125 P00 08C1        P00 08C1   1    5  L2$
126 P00 08C1 3BC1   P00 08C1   1    6           set      4,port_b
127 P00 08C3        P00 08C3   1    7  L3$
128                            1    8           .endm
129 P00 08C3 718A   P00 08C3      107           call     delay
130 P00 08C5 798B   P00 08C5      108           jp       ripeti
131                               109  ;        ....
132                               110  ;        ....
133                               111           ledflash port_b,4
134 P00 08C7 A3C104 P00 08C7   1    2           jrr      4,port_b,L4$
135 P00 08CA ABC1   P00 08CA   1    3           res      4,port_b
136 P00 08CC 098D   P00 08CC   1    4           jp       L5$
137 P00 08CE        P00 08CE   1    5  L4$
138 P00 08CE BBC1   P00 08CE   1    6           set      4,port_b
139 P00 08D0        P00 08D0   1    7  L5$
140                            1    8           .endm
141                               112
```

Fig.9 Parte del listato LAMPLED.LIS generato con l'opzione –L in cui sono definite dal compilatore, in modo del tutto automatico, le labels ?chelab0 e ?chelab1 con le scritte L2$ (vedi riga 122), L3$ (vedi riga 124), L4$ (vedi riga 134) e L5$ (vedi riga 136).

bels la lunghezza massima delle **labels** ed anche delle **variabili** non deve superare gli **8 caratteri**.
Infatti, se invece di:

```
ledflash port_b,4,r1
ledflash port_b,4,r2
```

avessimo scritto:

```
ledflash port_b,4,ret1
ledflash port_b,4,ret2
```

avremmo generate queste **label**:

lamp0ret1,lamp1ret1
lamp0ret2,lamp1ret2

che come noterete hanno più di **9 caratteri**.
Il compilatore in questo caso tenterà comunque di assemblare il programma, troncando i caratteri eccedenti quindi queste righe diventeranno:

lamp0ret e **lamp1ret**
lamp0ret e **lamp1ret**

e verrà segnalato un **errore** perché le **labels** sono doppie e già definite.

Come per le labels, la proprietà di concatenazione esiste anche per le **variabili**, come riportato nell'esempio che segue.
Modifichiamo **ledflash.LMA** scrivendo:

```
        .macro  ledflash cheporta,\chepin
        jrr     chepin,port_'cheporta,lamp0
        res     chepin,port_'cheporta
        jp      lamp1
lamp0
        set     chepin,port_'cheporta
lamp1
        .endm
```

Avrete dunque notato che nella **2°-3°-6° riga** abbiamo inserito **port_'** prima di **cheporta**.
In questo modo noi possiamo modificare **LAMPLED.ASM** scrivendo solo:

```
ledflash b,4
```

cioè abbiamo sostituito **port_b** con **b**.

Ricompilando il programma **LAMPLED.ASM** il compilatore quando incontra **ledflash** carica le istruzioni relative sostituendo **'cheporta** con **b.**
Quindi **port_'cheporta** diventa **port_b** e la parola **chepin** diventa **4**.
Abbiamo così ottenuto un concatenamento dei parametri della **variabile** definita **port_b**.

CONCLUSIONE

Con questi esempi pensiamo di avere sufficientemente spiegato il procedimento per **creare** ed **utilizzare** una **macro**, comunque per diventare esperti softwaristi dovrete sempre perdere un po' di tempo e fare anche tante prove pratiche.
Se ad esempio inserite nel programma **LAMPLED.ASM** per **due volte** la macro **ledflash port_b,4** in due punti diversi del programma, siccome i parametri sono uguali (**port_b** e **4**) avrete commesso un piccolo **errore**, perché richiamando per **due volte** la stessa **macro** il compilatore inserirà per **due volte** consecutive tutte le istruzioni della **macro**, sprecando così **memoria** preziosa.
Per evitare questi **sprechi** di **memoria** conviene scrivere una **sub-routine**, che potrete ad esempio chiamare **lampeg**:

```
lampeg      ledflash port_b,4
            ret
```

Con questa **sub-routine** anziché scrivere per **due volte** la parola **ledflash** scriverete solo:

```
            call        lampeg
            ….
            ….
            call        lampeg
```

L'utilizzo di **macro** all'interno dei programmi, se da un lato può facilitare e snellire la stesura dei programmi stessi, dall'altro permette anche di proteggerli dalla lettura.
Infatti se qualcuno venisse in possesso di un **sorgente**, dove nei punti principali anziché le istruzioni scritte in modo **chiaro** trovasse delle **macro**, non potrebbe decifrare il programma senza il listato di queste **macro**.

A chi volesse **proteggere** i propri programmi consigliamo di memorizzare queste **macro directory** su supporti esterni come **Floppy Disk**, unità **Zip** o **Hard Disk** removibili da inserire solo al momento della compilazione in **assembler**.
In questi casi dovete ricordarvi di modificare il comando **.input**.

Ad esempio se la macro **ledflash** fosse stata memorizzata su **Floppy Disk** anziché nella directory di **LEDFLASH.LMA**, avremmo dovuto scrivere nel programma questa istruzione:

```
            .input          "a:\ledflash.LMA"
```

In questo modo la **macro** verrà direttamente prelevata dal **Floppy Disk** e solo così il programma riuscirà ad assemblarsi.

Per PROGRAMMARE i

I microprocessori della serie ST6/B sono stati sostituiti dalla nuova serie ST6/C programmabile in ambiente Windows 3.1-95-98. Il linguaggio di programmazione non è cambiato, ma poiché nuove funzioni sono state aggiunte ed altre sono state modificate, in questo articolo ci occuperemo delle novità più rilevanti.

La funzione SPI

Come abbiamo spiegato nella rivista **N.198**, la **S**erial **P**eripheral **I**nterface, meglio conosciuta come **SPI**, consente di mettere in comunicazione il nostro micro con una **EEprom esterna** oppure con uno **Shift register** o con un altro **integrato**, secondo uno standard di **trasmissione** e **ricezione dati** in modalità **seriale sincrona**.

Le possibilità offerte da questa particolare funzione, di cui sono dotati anche i nuovi microprocessori della serie **ST6/C**, sono molteplici e offrono al programmatore non pochi vantaggi, soprattutto considerando il fatto che svolgendosi la trasmissione e ricezione **dati** in maniera del tutto **automatica**, il microprocessore può nel frattempo eseguire le altre istruzioni del programma.

In sostanza, le specifiche della funzione **SPI** in configurazione "Tree wire Half Duplex with Master/Slave select", propria degli **ST6260-65**, permettono di attivare una comunicazione **Half Duplex** su tre fili con selezione **Master** e **Slave**.

Dei **piedini** e dei **registri** coinvolti nella **ricezione - trasmissione** dati ci siamo occupati, con particolare riguardo, nella rivista **N.198**, che vi consigliamo di rileggere.
In questa circostanza è invece utile ricordare che la **SPI** si **attiva** predisponendo adeguatamente certi registri, diversamente i piedini coinvolti continueranno a svolgere le normali funzioni per cui erano stati in precedenza programmati.

Vediamo dunque subito quali differenze ci sono tra la serie **B** e la nuova serie **C** degli **ST6**.
Nella versione degli **ST6/B**, per attivare la **SPI** in **Master Mode** era necessario configurare il piedino **PC4** (**Sck**) di **Porta C** come **Output Push Pull** e settare a **1** il bit **Spclk** del registro **spmc** (**Spi Mode Register**).

Il bit **Spclk**, Base Clock Selection, consente infatti di selezionare il **clock** e informa il microcontrollore se il clock sarà **interno** (bit a **1** e dunque attivazione del **Master Mode**) o **esterno** (bit a **0** e dunque attivazione dello **Slave Mode**).

Nella versione degli **ST6/C**, per attivare la **SPI** in **Master Mode**, i piedini **PC3** (**Sout**) e **PC4** (**Sck**) devono essere configurati in **Reset State**, cioè:

pdir_c = 00000000b
popt_c = 00000000b
port_c = 00000000b

perché è sufficiente settare a **1** il bit **Spclk** del registro **spmc** (**Spi Mode Register**) per configurare **automaticamente** il **PC4** di **Porta C** come **Output Push Pull**.
I bit relativi a **pdir_c** e **popt_c** non devono essere assolutamente modificati e quindi devono rimanere in **Reset State**.

nuovi MICRO serie ST6/C

Quando settiamo a **1** il bit **M0** del registro **misc** (**Miscellaneous**), che attiva la **SPI** per la trasmissione dati, automaticamente il **PC3** di **Porta C** si configura come **Output Push Pull**.

Anche in questo caso i bit relativi a **pdir_c** e **popt_c** non devono essere modificati, cioè devono rimanere in **Reset State**.
Per capire meglio le peculiarità della programmazione della **SPI** nei micro **ST6** della versione **C** vi portiamo un semplice esempio.

Ammesso di voler trasmettere il valore **200** tramite **SPI** da un dispositivo in **M**aster **M**ode ad un dispositivo **S**lave, in modalità **8 bits** alla velocità di **9600 B/Rate**, in **P**olarità e **F**ase **normali**, senza **F**iltro e **I**nterrupt, le istruzioni saranno:

```
        ldi     pdir_c,00000000b
        ldi     popt_c,00000000b
        ldi     port_c,00000100b

        ldi     misc,1

        ldi     spdv,01000110b
        ldi     spmc,00010100b
        ldi     spda,C8h
        set     7,spmc
loop    jrs     7,spmc,loop
```

Ponete particolare attenzione al fatto che nella **terza** riga (vedi **port_c**) abbiamo configurato a **1** il piedino **PC2** (00000**1**00) in modalità **I**nput **N**o **P**ull-**U**p; nella **sesta** riga, avendo settato a **1** il **bit 2** del registro **spmc** (00010**1**00), abbiamo attivato la condizione di **S**tart **S**election (vedi rivista **N.198**) e nella **settima** riga (vedi **spda**) abbiamo inserito il valore **esadecimale C8h** che corrisponde al valore **decimale 200**.

Dopo aver visto come va configurata la funzione **SPI** per attivare il **M**aster **M**ode nei nuovi micro della serie **C**, ora descriviamo alcune importanti **caratteristiche** di questi nuovi microprocessori e le funzioni dell'**option byte**, che abbiamo già avuto modo di presentarvi nella rivista **N.202**.

Il dispositivo LFAO

Nella versione **C** degli **ST62X** è stato inserito un oscillatore ausiliario interno di emergenza siglato **LFAO** (**L**ow **F**requency **A**uxiliary **O**scillator).
Questo oscillatore può essere attivato in sostituzione dell'oscillatore principale **settando** a **1** il bit **2** denominato **OSC. OFF**, cioè **M**ain **O**scillator **O**ff, del registro **adcr** dell'**A/D converter** (vedi fig.1).

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
EAI	EOC	STA	PDS	//	OSC. OFF.	//	//

Fig.1 Gli ST6 della serie C hanno un oscillatore ausiliario interno di emergenza che può essere attivato ponendo a livello logico 1 il bit 2 (vedi OSC. OFF) del registro ADCR dell'A/D converter.

Attivando questo dispositivo si riduce drasticamente la frequenza interna di **clock** ad una frequenza compresa tra **0,8-1 MHz**, che permette al microcontrollore di eseguire tutte le sue funzioni, anche se a **velocità** ridotta. Allo stesso tempo si riduce la corrente di **assorbimento** del micro, che scende ad un valore di circa **1 mA**.

Solitamente questo dispositivo si attiva solo quando **non** necessitano elevate velocità di esecuzione oppure per ridurre il consumo di corrente quando il micro è alimentato da un **Gruppo** di **Continuità**.

Può inoltre risultare utile per **diminuire** il **rumore** durante una conversione **A/D** in concomitanza con l'utilizzo dell'istruzione **wait** (modalità **stand-by**).
Nel paragrafo relativo all'**A/D sync**, riportato in questo articolo, parleremo proprio di questo caso in modo esauriente.

Resettando il bit **OSC. OFF**, ponendolo cioè a **0**, si riattiva automaticamente l'oscillatore principale.

La funzione LVD

Nelle versioni degli **ST6** precedenti alla **C**, la condizione di **Reset** del microprocessore veniva attivata quando si verificava una di queste condizioni:

1 – all'accensione del micro (**P**ower **O**n **R**eset),
2 – quando il **watchdog** si decrementava fino a **0**,
3 – quando il piedino di **Reset** veniva esternamente cortocircuitato a **massa**.

Nella versione **C** è stata prevista una **quarta** condizione di **Reset**, che si può attivare in fase di programmazione settando la funzione **LVD** o **L**ow **V**oltage **D**etector dell'option byte.

Con **LVD** attivato, cioè settato a **1**, quando la tensione di alimentazione, che deve essere di **5 volt**, scende al disotto di **3,7-3,5 volt**, il micro si posiziona automaticamente sul vettore di **interrupt reset**.
In altre parole entra nel cosiddetto **R**eset **S**tatico sospendendo **temporaneamente** ogni attività senza resettarsi. Quando la tensione di alimentazione sale nuovamente sopra i **4 volt**, il micro riparte ed esegue l'eventuale routine legata al vettore di **interrupt RESET**.
In fig.2 è riportato il diagramma di intervento della funzione **LVD**.

C'è anche un altro caso in cui l'attivazione della funzione **LVD** interviene provocando una condizione di Reset temporaneo.

Quando si accende il microcontrollore, si attiva il **Power On Reset** (**POR**) per cui tutte le **porte** di **I/O** sono inizialmente configurate come **input pull-up** e non viene eseguita alcuna istruzione.
Non appena la tensione di alimentazione raggiunge i **2,5 - 3,0 volt**, l'oscillatore inizia a generare la sua frequenza di **clock** e poco dopo il micro inizia ad eseguire la prima istruzione.
Questa fase è abbastanza critica, perché se per vari motivi la tensione non raggiungesse i **4,1 volt** o non fosse **stabile**, il micro potrebbe presentare anomalie di funzionamento nella partenza o durante l'esecuzione del programma.

Attivando la funzione **LVD**, il micro esegue solamente la fase di **POR**, dopodiché attende che la tensione raggiunga un valore superiore ai **4 volt** prima di iniziare ad eseguire tutte le istruzioni del programma (vedi fig.3).
Garantendo una tensione stabile e, di conseguenza una frequenza di clock stabile, si evitano false partenze di programmi e altre possibili anomalie di funzionamento.

In conclusione la funzione **LVD** attiva una condizione di Reset temporaneo in due occasioni:

1 – nella fase di **inizializzazione** o **POR**, fino a quando la tensione non raggiunge un valore superiore a **4 volt**;

2 – durante l'**esecuzione** del programma, quando la tensione scende al disotto di **3,7-3,5 volt**.

In entrambi i casi la condizione di Reset temporaneo permane fino a quando la tensione non si è stabilizzata sopra i **4 volt**.

Lo stadio OSG

All'interno dei nuovi micro della serie **C** è stato inserito uno stadio denominato **OSG** (**O**scillator **S**a-

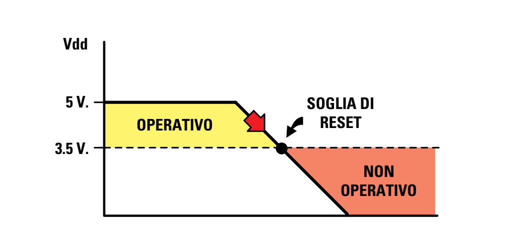


Fig.2 Utilizzando la funzione LVD, viene attivato in modo automatico un reset temporaneo se la tensione di alimentazione dovesse scendere sotto i 3,5 volt.

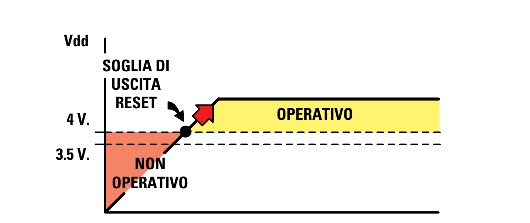


Fig.3 La funzione LVD mantiene un reset temporaneo nella fase di inizializzazione fino a quando la tensione di alimentazione non raggiunge un valore maggiore di 4 volt.

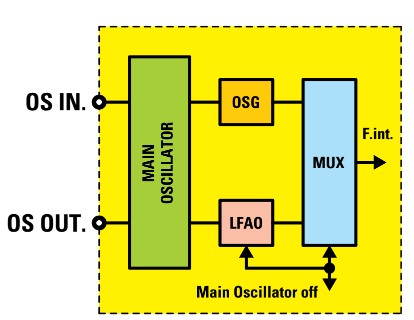


Fig.4 Schema a blocchi dello stadio OSG di cui sono dotati i micro ST6/C. Questo dispositivo deve essere attivato in fase di programmazione settando a 1 l'option byte denominato Osg enable.

fe **G**uard), che si può attivare settando a **1** l'option byte **Osg enable** o disattivare settandolo a **0**.

In fig.4 è riportato lo schema a blocchi di questo stadio che se attivato svolge **3** importanti funzioni:

- **funzione Filtro**
- **attivazione LFAO**
- **limitatore frequenza di clock**

1 – La funzione FILTRO

Quando si utilizza un oscillatore **esterno** per generare la frequenza di **clock** può verificarsi che l'ingresso **Osc in** (ingresso oscillatore) capti degli impulsi **spuri** che potrebbero generare delle frequenze di clock superiori a quelle ottimali o addirittura bloccare le funzioni del micro.

Settando a **1** la funzione **OSG** dell'option byte, si attiva un **filtro digitale** da **8 MHz** che filtra tutti gli impulsi spuri che si verificano entro un tempo di **62,5 microsecondi** dal cambiamento di stato del clock, rendendo in tal modo la frequenza di clock più stabile.

In questo modo la frequenza massima di **clock** non potrà mai superare gli **8 MHz.**
Poiché il suo periodo corrisponde a un tempo di:

1 : 8 = 0,125 millisecondi

pari a **125 microsecondi**, ogni cambio di stato da **1** a **0** o viceversa avviene ogni **62,5 microsecondi.**

Ogni **cambio** di stato del clock viene mantenuto stabile dall'**OSG** per un tempo massimo di **62,5 microsecondi**, pertanto qualsiasi impulso spurio che entrasse sull'ingresso del micro in questo lasso di tempo, verrebbe **ignorato** (vedi fig.5).

2 – L'attivazione del dispositivo LFAO

Se l'**OSG** non rileva la frequenza di **clock** generata dall'oscillatore **principale**, attiva automaticamente il dispositivo **LFAO**, cioè l'oscillatore **interno** di emergenza.
In questo modo viene generata una frequenza di **clock** compresa tra **0,8-1 MHz** che consente al micro di continuare a funzionare anche se più lentamente (vedi fig.6).

Sempre automaticamente, l'**OSG** provvede a disattivare il dispositivo **LFAO** quando l'oscillatore principale torna a generare la sua frequenza di **clock**. Disabilitando il dispositivo **LFAO**, il micro può riprendere a lavorare a velocità regolare.

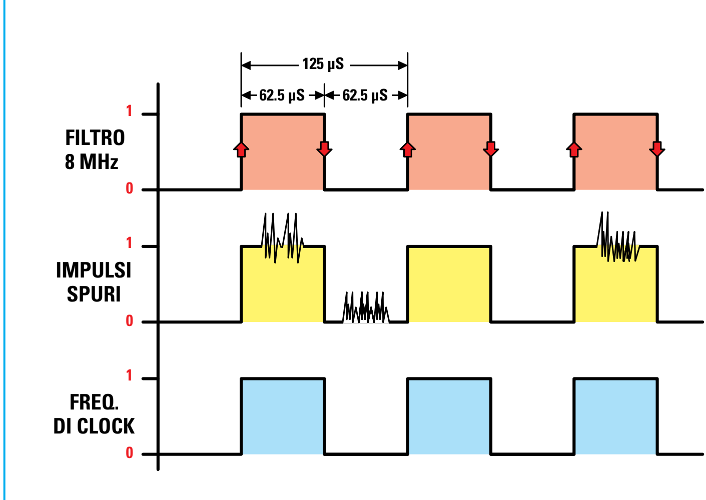


Fig.5 Con l'Osg enable settato a 1, si attiva un filtro digitale da 8 MHz che ha il compito di mantenere bloccato ogni cambio di stato logico del clock per un tempo di 62,5 microsecondi. Quindi se in questo lasso di tempo dovessero entrare degli impulsi spuri (vedi figura al centro), verrebbero ignorati e la frequenza di clock sarebbe perfettamente pulita (vedi figura in basso).

3 – La limitazione della frequenza di clock

L'ultima funzione del circuito **OSG** è quella di diminuire automaticamente la frequenza interna di clock nel caso dovesse abbassarsi la tensione di alimentazione del micro.

Vediamo come ciò avviene in pratica.

Se la **foscn**, cioè la frequenza di clock di **8 MHz**, rimane sempre costante con una tensione compresa tra **4,5-5 volt**, nel momento in cui la tensione di alimentazione scende al disotto di **4,5 volt**, l'**OSG** provvede ad abbassare la frequenza di clock interna a **4 MHz**, anche se il quarzo continua ad oscillare a 8 MHz.
Se questa tensione scende al disotto di **3,5 volt**, l'**OSG** abbassa ulteriormente la sua frequenza interna di clock a **2 MHz**.

Questa funzione è molto utile perché, se per un qualsiasi motivo dovesse abbassarsi la tensione di alimentazione, il micro potrà continuare a lavorare anche se ad una velocità **ridotta**.
È ovvio che nel caso il micro svolgesse funzioni di **timer**, di **orologio** ecc., cioè funzioni legate al tempo, con l'attivazione dell'**OSG** al variare della tensione di alimentazione si avrebbero delle variazioni non regolari sui **tempi** di lavoro.

In questi particolari casi potrebbe convenire al programmatore non limitare la frequenza tramite l'**OSG**, ma, come abbiamo già avuto modo di dire, far operare in automatico un reset temporaneo tramite la funzione **LVD** tutte le volte che la tensione di alimentazione scende sotto i 3,7-3,5 volt.

Qualcuno potrebbe obiettare che non c'è differenza tra l'attivazione del **LFAO**, precedentemente spiegata, e la limitazione della frequenza di clock al variare della tensione.

Quando l'**OSG** rileva che viene a mancare la **frequenza** di **clock** dall'oscillatore **principale**, attiva il dispositivo **LFAO**, cioè l'oscillatore **interno** di emergenza che genera una frequenza di clock compresa tra **0,8 - 1 MHz**.

Se l'**OSG** rileva che l'oscillatore **principale** funziona regolarmente, ma per un motivo qualsiasi la **tensione** di alimentazione scende al disotto di **4,5 volt**, attiva il limitatore di frequenza portando la frequenza di **clock** ad un valore **inferiore**.

WATCHDOG

In questo paragrafo ci occupiamo in special modo di due funzioni dell'**Option Byte** e della loro stretta relazione con le istruzioni **wait** e **stop**.

Watchdog activation

0 = Watchdog Software
1 = Watchdog Hardware

External Stop Mode Control

0 = Disattivato
1 = Attivato

Prima della comparsa sul mercato dei micro **ST6** della serie **C**, se si voleva gestire il **Watchdog** tramite **software** si doveva scegliere il micro siglato **SW** o **SWD**, mentre se si voleva gestire il **Watchdog** tramite **hardware** si doveva scegliere il micro siglato **HW** o **HWD**.
Nella nuova serie **C** il tipo di **Watchdog** è selezionabile tramite l'option byte.

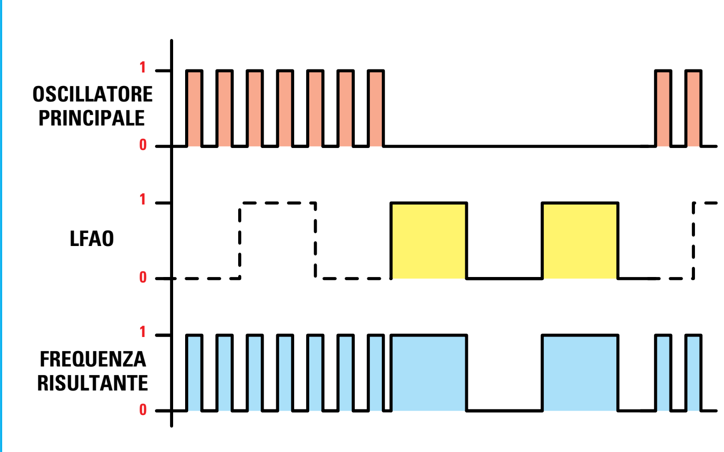


Fig.6 Quando l'OSG non rileva la frequenza di clock generata dall'oscillatore principale, automaticamente attiva l'oscillatore interno di emergenza LFAO che, generando una frequenza a 0,8-1 MHz, consente al micro di continuare a eseguire le istruzioni. Quando l'oscillatore principale torna a generare la sua frequenza di clock, l'OSG disabilita in modo automatico l'oscillatore LFAO.

Watchdog hardware

Settando a **1** la funzione **Watchdog activation**, il watchdog diventa di tipo **HW** (hardware), pertanto **non** si può più disattivare tramite software.
Questo significa ovviamente che non è possibile utilizzare né l'istruzione **wait** né l'istruzione **stop**.

Infatti, l'istruzione **wait** blocca il **program counter** e di conseguenza l'esecuzione del programma, ma non blocca l'oscillatore principale e di conseguenza il **watchdog** si decrementa fino a **0** resettando il microprocessore.

L'istruzione **stop** dovrebbe in teoria bloccare anche l'oscillatore e il **watchdog**, ma essendo quest'ultimo di tipo **HW** non si può disattivare e si ottiene lo stesso risultato dell'istruzione **wait**.

Se però si setta a **1** l'option byte denominato **External Stop Mode Control** e contemporaneamente si setta a **1** anche l'option byte denominato **NMI pin pull-up** a **1** (vedi rivista N.202), si pongono i presupposti necessari per utilizzare l'istruzione **stop** anche con un watchdog di tipo HW.
A queste condizioni, quando il programma incontra l'istruzione **stop**, il **watchdog** viene temporaneamente bloccato e il micro entra nella condizione **halt mode** o **stop mode** fermandosi completamente.
Nella condizione di **stop mode** il micro **blocca** tutte le sue funzioni compreso l'oscillatore lasciando attivo solo l'**interrupt** sul piedino di **NMI**.
Per uscire dalla condizione di **stop mode** si deve far giungere sul piedino **NMI** un **fronte** di **discesa**, attivando così un **interrupt** che potrà essere eventualmente gestito con una routine software.

Watchdog software

Settando a **0** la funzione **Watchdog activation**, il watchdog diventa di tipo **SW** (software), pertanto per attivarlo si dovrà settare a **1** il bit denominato **C** e settarlo a **0** per disattivarlo (vedi fig.7).

Un **Watchdog SW** può essere disattivato tramite il programma e dunque si possono tranquillamente utilizzare le istruzioni **wait** e **stop**.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
T0	T1	T2	T3	T4	T5	SR	C

Fig.7 Se la funzione Watchdog viene settata a 0, il watchdog è di tipo SW (software), quindi per attivarlo si deve settare a 1 il bit C del registro di Watchdog.

Tenete presente che se **disattivate** il watchdog, potrete sempre **riattivarlo**, ma una volta attivato non potrete più disattivarlo.
In questo caso potrà essere gestito come già spiegato nel paragrafo dedicato al **Watchdog HW**.

Per **disattivare** il watchdog dovete scrivere come prima istruzione del programma:

```
ldi          wdog,feh
```

Dopo non è più necessario gestire il **Watchdog** all'interno del programma.

ADC sync

L'**A/D converter** presente nei micro **ST62** è un **convertitore analogico - digitale** a **8 bit** in grado di eseguire una conversione in un tempo di **70 nanosecondi** con una frequenza di clock di **8 MHz.**

La conversione di un segnale **analogico** in un segnale **digitale** viene eseguita con una sequenza di approssimazioni successive, utilizzando per il **clock** la frequenza generata dall'oscillatore principale **divisa** per **12**.

Lavorando quindi per approssimazioni successive, se nel micro sono presenti varie fonti di rumore (timer attivo, Vdd instabile - PWM ecc.), si potrebbe avere sul valore finale un **errore** di **1** o **2 bit**.

Per evitare questo errore in molti nostri programmi (vedi dischetto **DF.1208**) abbiamo utilizzato la **tecnica** del **campionamento**, che consiste nel ripetere la **stessa** conversione **A/D** per **16**, **32**, **64** volte e poi dividere il risultato per il numero scelto.

In questo modo si ottiene una maggiore precisione però si ha lo svantaggio di allungare notevolmente il tempo di esecuzione della routine di conversione.

Per eliminare le fonti di rumore ed ottenere un risultato finale più preciso durante una conversione A/D utilizzando i micro con watchdog **SW**, veniva consigliato di usare l'istruzione **wait** subito dopo l'istruzione **ldi**, come qui sotto riportato:

```
ldi     adcr,10110000b
wait
```

Purtroppo se il programma utilizza più di un **clock** (timer - PWM ecc.), prima che l'istruzione **wait** riesca a mettere il micro in **stand-by** la conversione è già **completata**, quindi si continuano ad avere degli errori sul risultato finale.

Per ovviare a questo problema è stata aggiunta la funzione **ADC Syncro** nell'option byte dei micro della serie **C**.
Attivando a **1** questa funzione, la conversione **A/D** parte non con l'istruzione **ldi**, ma con l'istruzione **wait**, quando cioè il micro è in condizione di **stand-by**.
Chiaramente l'istruzione **wait** deve essere posta subito dopo l'istruzione **ldi adcr,10110000b**, altrimenti la conversione **A/D** non risulterà corretta.

Di seguito riportiamo un piccolo esempio di una routine di conversione **A/D** per **ST62/65C** con la modalità **ADC synchro** attivata. Ovviamente il watchdog è di tipo **SW** ed è disattivato.

```
addr    .def 0d0h        ;dati a/d conv.
adcr    .def 0d1h        ;registro a/d
        ........

;——- Inizio programma
inizio  ldi   wdog,0feh ; disattivo wdog
        ......
        ......
        ldi   ior,10h    ;Global Interrupt On
        .........
```

```
;——- routine A/D
adcon ldi   adcr,10110000b
        wait              ;stand-by
        nop               ;va sempre inserito
        ld    a,addr      ;risultato a/d in a
        ......
        ......
;——-subroutine di interrupt A/D
tad_int  ldi  adcr,00010000b
         reti

;——-vettori di interrupt
        .org 0ffh
        jp    tad_int     ;interrupt a/d e timer
        jp    art_int
        jp    CS_int
        jp    AB_int
        .org 0ffch
        jp    nmi_int
        jp    inizio
        .end
```

Utilizzando la funzione **ADC Syncro** riuscirete ad ottenere una conversione **A/D** molto precisa.

LA DIRETTIVA .IFC

Obiettivo di questo articolo è spiegarvi l'utilizzo di un gruppo di **direttive** in uso nel linguaggio **Assembler** per **ST6** che, utilizzate durante la stesura di **programmi** e **macro**, vengono elaborate già in fase di compilazione snellendo l'esecuzione del programma o della macro stessa.

Non dobbiamo infatti dimenticare che la quasi totalità dei programmi contiene al suo interno **istruzioni**, **sub-routine**, **moduli** che effettuano delle **scelte** in base al **valore** di variabili, costanti, espressioni o condizioni logiche.
In base ai valori riscontrati o ottenuti, si attivano **altre** istruzioni o sub-routine o si effettuano salti di programma oppure si richiamano moduli ecc.

Quando si **compila** un programma per generare il file in formato **.HEX**, anche le **istruzioni**, le **subroutine** e i **moduli** che comportano una scelta generano un codice eseguibile.
Questo significa che le **scelte** vengono effettuate durante l'esecuzione del programma, cioè quando il programma sta effettivamente funzionando.
In alcuni casi però ciò porta a un **appesantimento** del programma o dà luogo a una maggiore difficoltà durante la fase di **test** e di **simulazione**.

Esistono tuttavia delle **direttive** che, in molti casi, possiamo utilizzare per effettuare queste **scelte** in modo automatico durante la compilazione del programma, in modo da ottenere un completo programma eseguibile già **parametrizzato**.

Queste direttive sono:

.ifc	= direttiva che equivale a **se**
.else	= direttiva che equivale ad **altrimenti**
.endc	= direttiva che equivale a **fine**
.mexit	= direttiva che equivale a **uscita forzata**
.error	= direttiva che mostra un messaggio di **errore** impedendo al **compilatore** di generare il programma **.HEX**
.warning	= direttiva che pur mostrando un messaggio di **errore** consente al compilatore di generare il programma **.HEX**
.display	= direttiva che provvede a visualizzare sul monitor un **messaggio**

Per farvi capire a cosa servono le tre **direttive**:

.ifc
.else
.endc

analizziamo insieme una **situazione** che, verificandosi spesso, vi è sicuramente nota.

dell'ASSEMBLER per ST6

In questo articolo ci occupiamo di un gruppo di direttive del linguaggio Assembler per ST6 che, opportunamente utilizzate, vi consentono di ottenere con la compilazione un programma eseguibile parametrizzato.

Tutte le volte che si fa la spesa al supermercato, dopo aver riempito il carrello ci si avvia alla **cassa** e per pagare la merce acquistata si può scegliere fra diverse **modalità**, tutte però subordinate a precise condizioni:

– **se** la cassa **accetta** assegni
– **pago** con assegno
– **altrimenti**
– **se** la cassa **accetta** la carta di credito
– **pago** con carta di credito
– **altrimenti**
– **se** la cassa **accetta** il bancomat
– **pago** con bancomat
– **altrimenti**
– **pago** in **contanti**
– **fine** delle possibilità

Nel nostro esempio le parole "**se accetta**" rappresentano la condizione, mentre "**assegno - carta di credito - bancomat**" sono i suoi oggetti.

Quindi l'azione "**pago** con **assegno**" si può eseguire solo nel caso risulti **vera** la condizione in cui la cassa **accetti** gli **assegni**.

Se questa condizione **non** è **vera** si passa all'altra possibilità verificando la seconda condizione:

– **altrimenti**
– **se accetta** la carta di **credito**

Quindi l'azione "**pago** con carta di **credito**" si può eseguire solo nel caso risulti **vera** la condizione in cui la cassa **accetti** la carta di **credito**.

Se anche questa condizione **non** è **vera**, si passa alla terza possibilità verificando la terza condizione:

– **altrimenti**
– **se accetta** il **bancomat**

e se anche questa ultima condizione **non** risulta **vera** si passa all'ultima possibilità e quindi non rimane altro che:

– **altrimenti**
– **pago** in **contanti**
– **fine** delle possibilità.

Le tre direttive **.ifc**, **.else**, **.endc** devono dunque essere utilizzate in questo ordine:

.ifc: dopo aver definito la condizione e i suoi oggetti, vanno inserite le istruzioni da assemblare solo in presenza di una condizione **vera**.

.else: di seguito vanno inserite le istruzioni da assemblare solo se la condizione precedente **non** risulta **vera**.

.endc: stabilisce la fine delle scelte.

Le **condizioni** vengono espresse con queste sigle:

eq = significa **è uguale** a **0**

ne = significa **non è uguale** a **0**

gt = significa **è maggiore** di **0**

lt = significa **è minore** di **0**

le = significa **è minore** o **uguale** a **0**

ge = significa **è maggiore** o **uguale** a **0**

df = significa **è definita**

ndf = significa **non è definita**

È ovvio che ognuna di queste **condizioni** risulterà **vera** oppure **non vera** rispetto alla definizione del suo oggetto.

Cercheremo perciò ora di spiegarvi in maniera dettagliata ed esauriente come usare le **condizioni** della direttiva **.ifc**.

eq = condizione uguale a 0

Se ad esempio scriviamo:

```
.ifc    eq valx
ldi     coms,8
.else
ldi     coms,13
.endc
```

Quando il **compilatore** incontra:

```
.ifc    eq valx
```

verifica se il valore di **valx** è uguale a **0** ed assembla l'istruzione:

```
ldi     coms,8
```

solo se la condizione è **vera**.
Se invece **valx** e diverso **0**, noi abbiamo una condizione **non vera** quindi assembla l'istruzione che si trova **dopo** la direttiva **.else**:

```
.else
ldi     coms,13
```

Ricordate che per diverso da **0** si intende un valore che può essere **maggiore** o **minore** di **0**.
La direttiva **.endc** segnala al **compilatore** la **fine** del blocco della condizione e deve essere sempre inserita come **ultima istruzione**.
A questo punto vi chiederete come fa **valx** a contenere un valore uguale a **0** o diverso da **0**.
Gli esempi che seguono chiariranno ogni dubbio.

ne = condizione non uguale a 0

Si tratta della condizione opposta alla precedente, per cui se sostituiamo **eq** con **ne**:

```
.ifc    ne valx
ldi     coms,8
.else
ldi     coms,13
.endc
```

Il **compilatore** assembla:

```
ldi     coms,8
```

solo nel caso in cui **valx** risulti diverso da **0** altrimenti assembla:

```
ldi     coms,13
```

poi passa alla direttiva **.endc**, che gli segnala la **fine** del blocco della condizione.

gt = condizione maggiore di 0

Se inseriamo **gt** prima di **valx**:

```
.ifc    gt valx
ldi     coms,8
.else
ldi     coms,13
.endc
```

Il **compilatore** assembla:

```
ldi     coms,8
```

solo nel caso **valx** risulti **maggiore** di **0**; se invece è **uguale** o **minore** di **0** assembla:

```
ldi     coms,13
```

È sottinteso che, in questo caso, il **compilatore** è in grado di riconoscere anche un valore **negativo** come risultato di una espressione.
Nel secondo esempio che chiude questo articolo, avremo modo di spiegarvi come ciò accada.

lt = condizione minore di 0

Se inseriamo **lt** prima di **valx**:

```
.ifc    lt valx
ldi     coms,8
.else
ldi     coms,13
.endc
```

Il **compilatore** assembla:

```
ldi     coms,8
```

solo nel caso in cui **valx** risulti **minore** di **0**; se invece risulta **uguale** o **maggiore** di **0** assembla l'istruzione:

```
ldi     coms,13
```

quindi passa alla direttiva **.endc**, che gli segnala la **fine** del blocco della condizione.

le = condizione minore o uguale a 0

Se inseriamo **le** prima di **valx**:

```
.ifc    le valx
ldi     coms,8
.else
ldi     coms,13
.endc
```

Il **compilatore** assembla:

```
ldi        coms,8
```

solo se **valx** risulta **minore** o **uguale** a **0**.
Se **valx** è **maggiore** di **0** assembla l'istruzione:

```
ldi        coms,13
```

quindi passa alla direttiva **.endc**, che gli segnala la **fine** del blocco della condizione.

ge = condizione maggiore o uguale a 0

Se scriviamo **ge** prima di **valx**:

```
.ifc       ge valx
ldi        coms,8
.else
ldi        coms,13
.endc
```

Il **compilatore** assembla:

```
ldi        coms,8
```

solo nel caso **valx** risulti **maggiore** o **uguale** a **0**.
Se **valx** è **minore** di **0** assembla l'istruzione:

```
ldi        coms,13
```

quindi passa alla direttiva **.endc**, che gli segnala la **fine** del blocco della condizione.

df = condizione definita

Nel caso si volessero inserire più **macro** o più **moduli** all'interno di un programma principale, la condizione **df** ci permetterà di controllare e quindi di gestire (ad esempio per segnalare un errore, per modificare un valore, ecc.) se esiste una variabile, un'etichetta o una costante **già definita** in altri punti del programma con lo stesso **nome**.

```
.ifc       df pippo
.display   "pippo già definito"
.endc
```

Solo se **pippo** risulta già definito, il **compilatore** assembla:

```
.display   "pippo già definito"
```

facendo apparire sul monitor il messaggio **"pippo già definito"**.

ndf = condizione non definita

È la condizione opposta alla precedente, per cui la condizione **ndf** ci permetterà di controllare e quindi di gestire (ad esempio per segnalare un errore, per modificare un valore, ecc.) se esiste una variabile, un'etichetta o una costante che **non** sia ancora stata **definita** in altri punti del programma con lo stesso **nome**.

```
.ifc       ndf pippo
.display   "pippo non definito"
.endc
```

Il **compilatore** assembla:

```
.display   "pippo non definito"
```

solo se **pippo** non risulta definito. In questo caso vedremo apparire sul monitor il messaggio **"pippo non definito"**.

PRIMO ESEMPIO

Per completare quanto appena spiegato, abbiamo scritto un programma, che abbiamo chiamato **SERIAL.ASM** (vedi il listato in fig.1), che effettua una elaborazione di **dati** e provvede a trasmetterli ad un dispositivo qualunque in **modalità seriale asincrona** tramite un **piedino** di una porta.

La **velocità** di trasmissione viene regolata tramite un'opportuna configurazione del **timer** del micro.

Infatti, a seconda delle necessità, è la **macro** che abbiamo chiamato **setbaud** che effettua la scelta e il settaggio della velocità di trasmissione **seriale asincrona** con la possibilità di scegliere tra queste quattro velocità: **9600 - 4800 - 2400 - 1200 baud**.

Il listato completo della macro **setbaud** è visibile in fig.2, mentre in fig.1 è visibile la parte del programma sorgente nella quale durante la compilazione viene inserita la macro.

Come potete notare dalla fig.2, si tratta di una macro (**setbaud**) parametrizzata (**m_baud**), pertanto durante la compilazione il valore (**t_baud**), definito nella riga del file sorgente che richiama questa macro, verrà "passato" e sostituito nella macro stessa utilizzando la Common Area.
Questo è ciò che succede anche se, come nel nostro caso, i nomi usati per definire i parametri nei file **SERIAL.ASM** e **SETBAUD.LMA** non sono gli stessi.

Nota: per rinfrescarvi la memoria sulla formazione

Fig.1 LISTATO del PROGRAMMA SERIAL.ASM

```
t_baud   .set       96
         .input     "SETBAUD.LMA"

main
         ldi        wdog,0feh          ;ricarica il Watchdog
         call       init_a             ;inizializzo le variabili
         call       init_p             ;inizializzo le porte
         setbaud    t_baud             ;macro config. velocita trasm/baud
loop     ldi        wdog,0feh          ;ricarica il Watchdog
         call       elabor             ;elaborazione dati
         call       trasmx             ;trasmissione seriale asincrona
         jp         loop
```

Fig.2 LISTATO del PROGRAMMA SETBAUD.LMA

```
   .macro    setbaud   m_baud
             .ifc      df set_tcr             ;
             warning   "set_tcr gia' definito"
             endc
             .ifc      df set_psc             ;
             .warning  "set_psc gia' definito"
             .endc
             .ifc      eq m_baud - 12         ; 1200 -------------+
   set_tcr   .set      140                    ;                   |
   set_psc   .set      2                      ;                   |
             .display   "1200 BAUD"           ;                   |
             .else                            ; altrimenti        |
             .ifc      eq m_baud - 24         ; 2400 -----------+ |
   set_tcr   .set      140                    ;                 | |
   set_psc   .set      1                      ;                 | |
             .display  "2400 BAUD"            ;                 | |
             .else                            ; altrimenti      | |
             .ifc      eq m_baud - 48         ; 4800 ---------+ | |
   set_tcr   .set      140                    ;               | | |
   set_psc   .set      0                      ;               | | |
             .display  "4800 BAUD"            ;               | | |
             .else                            ; altrimenti    | | |
             .ifc      eq m_baud - 96         ; 9600 -------+ | | |
   set_tcr   .set      70                     ;             | | | |
   set_psc   .set      0                      ;             | | | |
             .display  "9600 BAUD"            ;             | | | |
             .else                            ; altrimenti  | | | |
             .error    "ERRORE SELEZ.BAUD"    ; ERRORE      | | | |
             .mexit                           ;             | | | |
             .endc                            ;------------ + | | |
             .endc                            ;-------------- + | |
             .endc                            ;---------------- + |
             .endc                            ;------------------ +
             .endm                            ; fine macro
```

e l'utilizzo delle **macro**, vi consigliamo di rileggere l'articolo a loro dedicato, che abbiamo pubblicato sulla rivista **N.203**.

Vediamo dunque passo passo cosa succede quando lanciamo la compilazione del programma sorgente **SERIAL.ASM**.

Tralasciamo tutte le istruzioni iniziali, che al fine dell'argomento di questo articolo non interessano, e soffermiamoci sull'istruzione:

```
t_baud     .set      96
```

Come già sapete, quando il **compilatore** incontra la direttiva **.set** assegna un valore, che nel nostro caso è **96**, alla costante **t_baud**.
L'istruzione successiva:

```
           .input    "SETBAUD.LMA"
```

ci serve per definire **setbaud** come macro, in modo che il **compilatore**, quando incontra questo nome, inserisca il contenuto del file **SETBAUD.LMA**, cioè della macro per settare la velocità di trasmissione (vedi fig.2), all'interno del programma **SERIAL.ASM**.

Le tre istruzioni successive:

```
main       ldi       wdog,0feh
           call      init_a
           call      init_p
```

servono in esecuzione per caricare il Watchdog e inizializzare sia le variabili del programma sia le porte del micro coinvolte.
Quando il compilatore Assembler arriva a:

```
           setbaud   t_baud
```

riconosce che **setbaud** è una **macro** e pertanto la sostituisce con le istruzioni relative passando, come abbiamo già avuto modo di ricordarvi, il parametro **t_baud** alla macro (vedi fig.3).
A questo parametro assegnerà anche il valore definito con l'istruzione **.set** che abbiamo appena visto, cioè **96**.

Poiché infine questa macro è costituita a sua volta da direttive, le esegue ad una ad una.

Nota: a questo proposito vi ricordiamo che le direttive sono istruzioni che vengono eseguite durante la fase di Compilazione (vedi rivista **N.190**).

Aiutandoci con la fig.3, che riporta il file con estensione **.LIS** del nostro programma sorgente, vediamo ora come lavora il compilatore.
A partire dalla riga 155 incontriamo:

```
           .ifc      df set_tcr
           .warning  "set_tcr già definito"
           .endc
```

Questo gruppo di istruzioni equivale a: se la costante **set_tcr** è già definita, segnalami un **messaggio** di attenzione, ma prosegui ugualmente la compilazione generando il programma eseguibile, cioè il programma **SERIAL.HEX**.
La direttiva **.warning** infatti, si utilizza per visualizzare il messaggio di **errore non grave** racchiuso tra virgolette. Questo messaggio apparirà sul video durante la fase di **compilazione**, fase che comunque proseguirà per terminare normalmente.

Il compilatore perciò controlla che **set_tcr** non sia già stato definito all'interno del programma principale **SERIAL.ASM**.
Come potete controllare dal listato in fig.1, nel programma **SERIAL.ASM** non è stata inserita nessuna definizione di **set_tcr**, pertanto per il compilatore si attiverà la condizione "**non vero**" e quindi non eseguirà la direttiva **.warning**, ma proseguirà a **.endc** chiudendo così questa **.ifc**.

Per mostrarvi però cosa sarebbe successo nel caso **set_tcr** fosse stato definito, abbiamo provato ad inserire in **SERIAL.ASM** l'istruzione:

```
set_tcr    .set      30
```

Abbiamo quindi lanciato di nuovo la compilazione il cui esito è visibile in fig.4.
In questo caso compare a video il messaggio di **warning** con l'indicazione del file **SETBAUD.LMA** e del numero **6** che corrisponde alla riga di istruzione della macro che ha generato il messaggio.
Il numero **[157]** è invece il codice dell'errore dell'Assembler.

Nota: vi ricordiamo che l'estensione **.LIS** è propria del formato listato ottenuto durante la compilazione Assembler, come ampiamente spiegato nell'articolo relativo alle opzioni dell'Assembler per ST6 pubblicato sulla rivista **N.194**.

Notate comunque il messaggio *** **SUCCESS** *** che ci informa che il programma **SERIAL.ASM** è stato assemblato senza problemi.
Viene poi visualizzato il messaggio **One warning** per ricordare che esiste comunque un problema, anche se non grave.

Ma ora ritorniamo a dove eravamo rimasti e pro-

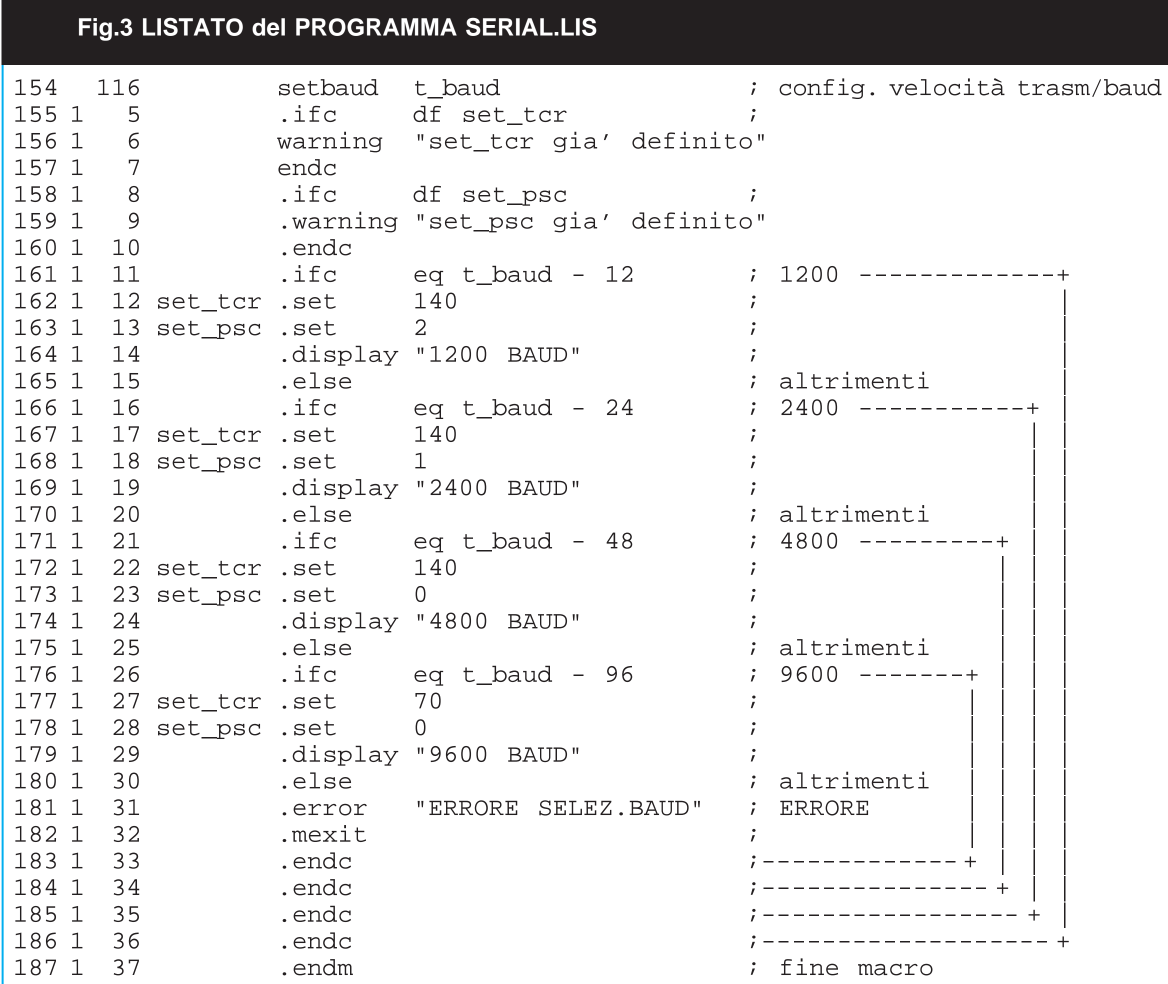


Fig.3 LISTATO del PROGRAMMA SERIAL.LIS

```
154   116         setbaud   t_baud                ; config. velocità trasm/baud
155 1   5         .ifc      df  set_tcr           ;
156 1   6         warning   "set_tcr gia' definito"
157 1   7         endc
158 1   8         .ifc      df  set_psc           ;
159 1   9         .warning "set_psc gia' definito"
160 1  10         .endc
161 1  11         .ifc      eq t_baud - 12        ; 1200 -------------+
162 1  12 set_tcr .set      140                   ;                   |
163 1  13 set_psc .set      2                     ;                   |
164 1  14         .display "1200 BAUD"           ;                   |
165 1  15         .else                           ; altrimenti        |
166 1  16         .ifc      eq t_baud - 24        ; 2400 -----------+ |
167 1  17 set_tcr .set      140                   ;                 | |
168 1  18 set_psc .set      1                     ;                 | |
169 1  19         .display "2400 BAUD"           ;                 | |
170 1  20         .else                           ; altrimenti      | |
171 1  21         .ifc      eq t_baud - 48        ; 4800 ---------+ | |
172 1  22 set_tcr .set      140                   ;               | | |
173 1  23 set_psc .set      0                     ;               | | |
174 1  24         .display "4800 BAUD"           ;               | | |
175 1  25         .else                           ; altrimenti    | | |
176 1  26         .ifc      eq t_baud - 96        ; 9600 -------+ | | |
177 1  27 set_tcr .set      70                    ;             | | | |
178 1  28 set_psc .set      0                     ;             | | | |
179 1  29         .display "9600 BAUD"           ;             | | | |
180 1  30         .else                           ; altrimenti  | | | |
181 1  31         .error    "ERRORE SELEZ.BAUD"   ; ERRORE      | | | |
182 1  32         .mexit                          ;             | | | |
183 1  33         .endc                           ;-------------+ | | |
184 1  34         .endc                           ;---------------+ | |
185 1  35         .endc                           ;-----------------+ |
186 1  36         .endc                           ;-------------------+
187 1  37         .endm                           ; fine macro
```

seguiamo con le successive istruzioni, visibili sempre in fig.3:

```
.ifc        df set_psc
.warning    "set_psc già definito"
.endc
```

Questo gruppo di istruzioni equivale a: se la costante **set_psc** è già definita segnalami un messaggio di attenzione, ma prosegui normalmente la compilazione generando comunque il programma eseguibile.

In questo caso, peraltro simile al precedente, è la costante **set_psc** ad essere controllata e poiché anche stavolta per il compilatore si attiverà la condizione “**non vero**”, non verrà eseguita la direttiva **.warning** e si proseguirà a **.endc** chiudendo così anche questa **.ifc**.

Apriamo un piccola parentesi per farvi notare che, contrariamente agli esempi proposti all’inizio dell’articolo, per **set_tcr** e **set_psc** non è stata u-

Fig.4 Messaggio di WARNING

```
C:\ST6\LX1208>ast6 -l -D -S -m serial
ST6 MACRO-ASSEMBLER version 4.00  - August 1992
Warning  SETBAUD.LMA 6: (157)  "set_tcr gia definito"
*** SUCCESS ***
Execution time: 0 second(s)
 One warning
```

Fig.5 Messaggio di compilazione riuscita

```
C:\ST6\LX1208>ast6 -l -D -S -m serial
ST6 MACRO-ASSEMBLER version 4.00  - August 1992
9600 BAUD
*** SUCCESS ***
Execution time: 1 second(s)
```

Fig.6 Messaggio di ERROR

```
C:\ST6\LX1208>ast6 -l -D -S -m serial
ST6 MACRO-ASSEMBLER version 4.00  - August 1992
Error  SETBAUD.LMA 32: (157)  "ERRORE SELEZ.BAUD"
Execution time: 0 second(s)
One error detected
No object created
```

tilizzata la direttiva **.else** per la gestione della condizione di “**non vero**”. Infatti, in questi due casi ci interessava solo che venisse evidenziata la condizione “**vero**” delle direttive **.ifc**.

Proseguiamo dunque con le istruzioni successive:

```
            .ifc        eq t_baud – 12
set_tcr     .set        140
set_psc     .set        2
            .display    “1200 BAUD”
            .else
            .ifc        eq t_baud – 24
set_tcr     .set        140
set_psc     .set        1
            .display    “2400 BAUD”
            .else
            .ifc        eq t_baud – 48
set_tcr     .set        140
set_psc     .set        0
            .display    “4800 BAUD”
            .else
            .ifc        eq t_baud – 96
set_tcr     .set        70
set_psc     .set        0
            .display    “9600 BAUD”
            .else
            .error      “Errore Selez. Baud”
            .mexit
            .endc
            .endc
            .endc
            .endc
```

Ci troviamo di fronte ad un esempio un po’ complesso di compilazione condizionata (**.ifc**) dove per condizione “**vero**” viene eseguita la direttiva **.display**, mentre per “**non vero**” viene posta una nuova condizione **.ifc**, che a sua volta ha una gestione per “**vero**” e rimanda a una nuova condizione di **.ifc** per “**non vero**” e così via.

Vediamo però passo passo cosa succede e analizziamo la prima sequenza:

```
            .ifc        eq t_baud – 12
set_tcr     .set        140
set_psc     .set        2
            .display    “1200 BAUD”
            .else
```

Il compilatore confronta il valore ricavato dalla espressione **t_baud – 12** con **zero** (condizione **eq**) e se risulta “**vero**” definisce la costante **set_tcr** e le associa il valore **140**, inoltre definisce la costante **set_psc** e le associa il valore **2**, infine esegue la direttiva **.display**.
Quest’ultima direttiva si utilizza essenzialmente per visualizzare dei messaggi a video durante la fase di Compilazione del programma.
Nel nostro caso se la condizione fosse vera, a video comparirebbe “**1200 BAUD**”, per segnalarci che il programma **SERIAL.ASM** utilizza una velocità di trasmissione di **1200 baud**.

È dunque ora necessario verificare qual è il risultato dell’espressione **t_baud – 12** e per farlo bisogna prima ricavare il valore di **t_baud**.
Se ricordate, la prima istruzione di **SERIAL.ASM** che abbiamo visto era:

```
t_baud      .set        96
```

che assegna a **t_baud** il valore **96**.
Pertanto l’espressione **t_baud – 12** dà come risultato:

96 –12 = 84.

A questo punto è chiaro che l’istruzione diventa:

```
            .ifc        eq 84
```

e poiché l’oggetto della condizione, cioè **84**, non è uguale a **zero**, si attiva la condizione di “**non vero**”, e quindi le istruzioni:

```
set_tcr     .set        140
set_psc     .set        2
            .display    “1200 BAUD”
```

non vengono eseguite. Il compilatore passa dunque alle istruzioni poste dopo **.else**:

```
            .ifc        eq t_baud – 24
set_tcr     .set        140
set_psc     .set        1
            .display    “2400 BAUD”
            .else
```

e confronta nuovamente il valore ricavato dalla espressione **t_baud – 24** con **zero** e se “**vero**” definisce la costante **set_tcr** e le associa il valore **140**, definisce **set_psc** e le associa il valore **1**, infine esegue la direttiva **.display**.
Siccome però l’espressione **t_baud – 24** dà come risultato un valore **non** uguale a **0**:

96 – 24 = 72

anche in questo caso viene attivata la condizione di “**non vero**”. Il compilatore ignora dunque:

```
set_tcr     .set        140
set_psc     .set        1
            .display    “2400 BAUD”
```

e passa alle istruzioni successive a **.else**:

```
            .ifc        eq t_baud – 48
set_tcr     .set        140
set_psc     .set        0
            .display    "4800 BAUD"
            .else
```

Anche in questo caso il risultato dell'espressione **t_baud – 48** è un valore diverso da **zero**:

96 – 48 = 48

pertanto il compilatore ignora:

```
set_tcr     .set        140
set_psc     .set        0
            .display    "4800 BAUD"
```

e passa alle istruzioni dopo **.else**:

```
            .ifc        eq t_baud – 96
set_tcr     .set        70
set_psc     .set        0
            .display    "9600 BAUD"
            .else
```

In questo caso invece l'espressione **t_baud – 96**:

96 – 96 = 0

soddisfa la condizione per "**vero**" e perciò il compilatore esegue le istruzioni:

```
set_tcr     .set        70
set_psc     .set        0
            .display    "9600 BAUD"
```

definisce così la costante **set_tcr** e le associa il valore **70**, definisce **set_psc** e le associa il valore **0**, infine esegue la direttiva **.display** e a video comparirà la scritta "**9600 BAUD**".
A questo punto il compilatore ignora l'istruzione **.else** e quelle che seguono:

```
            .error      "Errore Selez. Baud"
            .mexit
```

e passa alla prima delle quattro **.endc** che chiude l'ultima **.ifc** vista, cioè:

```
            .ifc        eq t_baud – 96
```

Poi va alla seconda **.endc** che chiude:

```
            .ifc        eq t_baud – 48
```

Poi va alla terza **.endc** che chiude:

```
            .ifc        eq t_baud – 24
```

Poi va alla quarta **.endc** che chiude:

```
            .ifc        eq t_baud – 12
```

Per facilitarvi nella comprensione della sequenza logica delle istruzioni, alla destra del listato visibile in fig.3 abbiamo legato con dei trattini le condizioni **.ifc** alle rispettive **.endc**.
Si vede così abbastanza chiaramente che si tratta di una serie di **.ifc** racchiuse una dentro l'altra, dove la prima del listato è l'ultima ad essere "chiusa".
Si parla in questo caso di **.ifc** "**nested**" che tradotto vuol dire "nidificate".
Vi ricordiamo che è importantissimo "chiudere" sempre ogni **.ifc** con una **.endc**.
Il compilatore segnala infatti errore nel caso che siano state inserite un numero maggiore o minore di **.endc** rispetto alle **.ifc** inserite.
Segnala inoltre errore anche quando si inseriscono più **.else** rispetto alle **.ifc**.

Dopo l'ultima **.endc** il compilatore trova la direttiva **.endm** che gli segnala la fine della macro.

A questo punto prosegue con la compilazione delle rimanenti istruzioni del programma **SERIAL.ASM** e quando arriva alla routine che predispone il **timer** per gestire la velocità di trasmissione, carica nel registro **tcr** (Contatore del Timer) il valore corrispondente alla costante **set_tcr** (nel nostro esempio **70**) e nel **Prescaler** del registro **tscr** il valore corrispondente alla costante **set_psc** (nel nostro esempio **0**).
Questo permetterà di gestire i tempi strettamente legati alla velocità di trasmissione.
Vi ricordiamo che trattandosi di esempi, i valori **70** e **0** che abbiamo utilizzato sono indicativi, poiché quello che ci premeva farvi capire è il meccanismo con cui si ottengono questi valori.
A fine compilazione comparirà a video il messaggio visibile in fig.5.

Notate la dicitura "**9600 BAUD**" visibile prima della scritta *** **SUCCESS** *** che testimonia che è stata selezionata la velocità di **9600** baud per la trasmissione.

A questo punto vi starete chiedendo cosa succede se nel definire **t_baud**, anziché utilizzare uno dei valori numerici gestiti dalla macro **setbaud** (cioè 12 o 24 o 48 o 96) inseriamo un valore diverso, ad esempio 75.
Aiutandovi con il listato di fig.3 che abbiamo appena descritto provate a simulare il compilatore.

Tutte e quattro le espressioni che utilizzano **t_baud** danno un risultato diverso da zero; l'ultima dà addirittura un risultato negativo.
Ne consegue che verranno eseguite sempre le condizioni per "**non vero**" arrivando a:

```
.error     "Errore Selez. Baud"
.mexit
```

La direttiva **.error** viene utilizzata per fare apparire a video la segnalazione di errore seguita, dove ci sia, dalla frase inserita tra virgolette.
Quando il compilatore incontra questa direttiva, visualizza il messaggio a video e continua comunque la compilazione del programma, ma non genera nessun programma eseguibile (**.HEX**).
Questa direttiva si utilizza perciò per segnalare un caso di errore grave.
La direttiva **.mexit** che abbiamo inserito di seguito viene utilizzata per uscire forzatamente dalla compilazione di una macro senza dover arrivare alla sua fine naturale, cioè all'istruzione **.endm**.

Nella fig.6 potete vedere il messaggio che sarebbe apparso dopo la compilazione di **SERIAL.ASM** con **t_baud** non valido.
Viene infatti mostrato a video il messaggio di errore e la dicitura finale "**No object created**".

Torniamo ora all'esempio corretto dove **t_baud** vale **96** e la compilazione dà esito positivo.
Qualcuno potrebbe obiettare che sono state inserite molte istruzioni, con una conseguente perdita di spazio e tempo di esecuzione, per ottenere la configurazione di due costanti: **set_tcr** e **set_psc**.

Vorremmo però farvi osservare che se in futuro si presenterà la necessità di scrivere più di un programma che esegua una trasmissione e/o una ricezione seriale asincrona, ognuno a una diversa velocità di trasmissione tra le 4 proposte nella macro, sarà sufficiente definire in maniera corretta il valore di **t_baud** per avere già tutto predisposto.
Inoltre se siete dei corretti osservatori, avrete notato che la macro **setbaud** è composta esclusivamente da direttive dell'Assembler, e voi dovreste sapere che queste non occupano spazio di memoria, non vengono eseguite in fase di esecuzione del programma e non generano nessuna opcode.
A riprova di quanto detto abbiamo lanciato l'esecuzione del programma **SERIAL.HEX** tramite il simulatore **SimST626** (presentato sulla rivista **N.197**) e come visibile in fig.7, dopo l'istruzione:

```
        call        init_p
```

viene eseguita l'istruzione:

```
loop    ldi         wdog,FEh
```

e non vi è più traccia di:

```
        setbaud     t_baud
```

come invece riportato nel **SERIAL.ASM** di fig.2.

Se non disponete di un simulatore, per sapere se i dati sono stati correttamente inseriti nel registro **tcr** e nel registro **tscr** del Timer, vi dovete fidare di ciò che appare a video alla fine della compilazione e cioè di un messaggio simile a quello visibile in fig.5.
Esiste però un altro controllo che si può effettuare quando non si dispone di un simulatore.
È infatti sufficiente compilare il programma inserendo l'opzione **–S** per ottenere così anche il file con estensione **.SYM**.
Come già spiegato sulla rivista **N.194** relativamente alle opzioni del compilatore assembler, questo file contiene tutte le etichette e tutte le costanti utilizzate nel programma con a fianco il loro valore espresso in **esadecimale**.
Vediamo dunque, tramite la fig.8, il listato del file **SERIAL.SYM** e andiamo a verificare i valori di:

```
t_baud  :  EQU  00060H  C
```

dove appunto **60h** espresso in decimale è **96**.

```
set_psc :  EQU  00000H  C
```

dove il valore **00h** espresso in decimale è **0**.

```
set_tcr :  EQU  00046H  C
```

dove il valore **46h** espresso in decimale è **70**.

Fig.7 Esecuzione del file SERIAL.HEX

```
Assembler
  Ind.  Codice  Label   Mnemonico
  08AA  0DD8FE  main    ldi      wdog,FEh
  08AD  318A            call     init_a
> 08AF  418A            call     init_p
  08B1  0DD8FE  loop    ldi      wdog,FEh
  08B4  118A            call     elabor
  08B6  218A            call     trasmx
  08B8  A98A            jp       loop
```

SECONDO ESEMPIO

Per il secondo esempio abbiamo realizzato una **macro** che abbiamo chiamato **ritardo** e che abbiamo salvato nel file **RITARDO.LMA**.

Abbiamo quindi scritto un semplice programma che abbiamo chiamato **PROVA2.ASM** e in due diversi punti abbiamo utilizzato la **macro ritardo**.

Fig.8 LISTATO del file SERIAL.SYM

Per ottenere un file con estensione .SYM, bisogna compilare il programma sorgente inserendo l'opzione –S. In questo modo si ottiene l'elenco delle etichette definite in Program Space e delle costanti simboliche utilizzate nel programma sorgente. Come potete vedere in queste righe, accanto a ogni etichetta (definita con P) o costante (definita con C), è espresso l'indirizzo in valore esadecimale.

```
t_baud    :   EQU  00060H  C
ad_int    :   EQU  008a5H  P
init_a    :   EQU  008a3H  P
elabor    :   EQU  008a1H  P
init_p    :   EQU  008a4H  P
inizio    :   EQU  00880H  P
trasmx    :   EQU  008a2H  P
main      :   EQU  008aaH  P
loop      :   EQU  008b1H  P
nmi_int   :   EQU  008a9H  P
set_psc   :   EQU  00000H  C
set_tcr   :   EQU  00046H  C
tim_int   :   EQU  008a6H  P
A_int     :   EQU  008a8H  P
BC_int    :   EQU  008a7H  P
```

Fig.9 LISTATO del PROGRAMMA RITARDO.LMA

```
                .macro    ritardo time,?lop1
                .ifc      ndf freqz
                .error    "Frequenza quarzo non definita"
                .mexit
                .endc
                .ifc      gt time*freqz/6/13-256
                .error    "Tempo troppo lungo"
                .mexit
                .endc
                .ifc      le time*freqz/6/13-1
                .error    "Tempo troppo corto"
                .mexit
                .endc

                ldi       carmat,time*freqz/6/13-1
      lop1      dec       carmat
                jrnz      lop1
                .endm
```

Fig.10 LISTATO del PROGRAMMA PROVA2.ASM

```
carmat   .def       084h              ;Variabile per avere un ritardo
freqz    .set       8                 ;Segnala 8MHz di frequenza
         .input     "RITARDO.LMA"

main
         ldi        wdog,0feh         ;ricarica il Watchdog
         call       set_pin           ;configura le porte
         call       elab1             ;prima elaborazione
         call       delay1            ;esegui un ritardo
         call       elab2             ;seconda elaborazione
         call       delay2            ;esegui un ritardo
         jp         main              ;ripeti

delay1   ritardo    1200              ;Esegue un ritardo di 1200 us
         ret

delay2   ritardo    1500              ;Esegue un ritardo di 1500 us
         ret
```

In fig.9 potete vedere il listato della **macro** chiamata **ritardo**, mentre in fig.10 potete vedere il listato del programma **PROVA2.ASM** relativo alle sole istruzioni che ci interessano ai fini dell'esempio.

La macro riportata in fig.9 ci serve per generare un **ritardo variabile**, il cui valore andrà inserito all'interno del programma **PROVA2.ASM** in corrispondenza delle istruzioni che richiamano questa macro, cioè:

```
delay1    ritardo    1200

delay2    ritardo    1500
```

I valori numerici **1200** e **1500** sono i valori che verranno passati dal programma sorgente alla macro durante la compilazione e corrispondono al ritardo espresso in **microsecondi** che verrà generato.
Nelle istruzioni della macro è inoltre previsto un controllo sui valori numerici passati alla macro stessa, in modo che se il ritardo è minore di **10 microsecondi** o maggiore di **2496 microsecondi**, venga segnalato **errore**.

Una macro come quella da noi chiamata **ritardo** può risultare molto utile quando si devono inserire dei ritardi in determinati programmi, perché eviterà di dover calcolare di volta in volta il tempo dei **cicli** delle istruzioni.

Adesso vediamo cosa avviene quando **compiliamo** il programma **PROVA2.ASM**.

Seguendo il listato di fig.10 troviamo subito la prima istruzione:

```
carmat    .def    084h
```

dove la variabile **carmat** viene associata all'area di Data Space **084h**. Questa variabile è quella che verrà utilizzata dalla macro **ritardo**.

La seconda istruzione:

```
freqz    .set    8
```

definisce la costante **freqz** associandole il valore **8**.
Questo valore corrisponde alla frequenza di oscillazione del quarzo da **8 MHz** utilizzato per il clock.
E' ovvio che se si utilizzasse un quarzo da **4 MHz**, l'istruzione dovrebbe cambiare in:

```
freqz    .set    4
```

La terza istruzione che incontriamo riguarda la direttiva **.input**. Come abbiamo già avuto modo di ricordare con il 1° esempio, questa direttiva informa il **compilatore** che deve caricare la macro **ritardo** nel programma principale **PROVA2.ASM**, prelevandola dal file **RITARDO.LMA**.

Tralasciamo le istruzioni successive perché non strettamente inerenti all'argomento di questo articolo e andiamo direttamente a:

```
delay1    ritardo    1200
          ret
```

Questa sub-routine ha il compito di effettuare un ritardo di **1200 microsecondi**.

Il **compilatore** associa l'etichetta **delay1** alla istruzione **ritardo 1200** e, poiché ha riconosciuto che **ritardo** è una **macro**, inizia a compilare le istruzioni contenute nella stessa macro, che, come abbiamo già ricordato, si trovano in fig.9.

La prima istruzione di fig.9:

```
          .macro    ritardo time,?lop1
```

identifica la **macro ritardo** e informa il compilatore che in questa macro verrà passato il parametro **time** e che verrà utilizzata l'etichetta interna **?lop1**.

Ora il **compilatore** prende in esame il blocco di istruzioni:

```
.ifc      ndf freqz
.error    "frequenza quarzo non definita"
.mexit
.endc
```

che equivale a: se la **freqz** del quarzo **non** è stata definita, segnala a video un messaggio di errore con la scritta **"Frequenza quarzo non definita"**, quindi esci dalla macro senza generare il programma eseguibile (istruzione **.mexit**).
Poiché però nel programma sorgente **freqz** è stata definita **.set 8**, questo blocco di istruzioni viene totalmente ignorato.

Il compilatore passa quindi al successivo blocco di istruzioni:

```
.ifc      gt time*freqz/6/13–256
.error    "Tempo troppo lungo"
.mexit
.endc
```

che equivale a: se il risultato dell'espressione **time*freqz/6/13–256** è maggiore (**gt**) di **0** allora segnala a video il messaggio di errore **"Tempo troppo lungo"** ed esci dalla macro senza generare

nessun programma eseguibile.

Nota: vi ricordiamo che le **espressioni** sono state spiegate nella rivista **N.189**.

Il compilatore esegue automaticamente il calcolo di questa espressione, ma noi possiamo verificare, procedendo passo passo, se la condizione è **vera** o **non vera**.

Poiché **time** è il parametro della **macro ritardo** che viene passato nel programma **PROVA2.ASM**, quando si richiama la macro con l'istruzione:

```
delay1      ritardo      1200
```

noi sappiamo che **time** equivale a **1200**, quindi l'espressione **time*freqz/6/13–256** diventa:

1200*freqz/6/13–256

Poiché la costante **freqz** è stata definita associandola al valore **8** del quarzo, la nostra espressione diventa:

1200*8/6/13–256

Come prima operazione eseguiamo la moltiplicazione:

1200*8 = 9600

poi eseguiamo la prima divisione:

9600/6 = 1600

quindi la seconda divisione:

1600/13 = 123,0769

e infine, dopo aver scartato i **decimali**, eseguiamo l'ultima operazione con il solo numero intero:

123–256 = –133

Quindi l'istruzione:

```
.ifc      gt time*freqz/6/13–256
```

diventa in pratica:

```
.ifc      gt –133
```

Poiché il valore **–133** non è **maggiore** di **0**, la condizione posta da **.ifc** non viene soddisfatta e quindi il blocco di istruzioni viene ignorato e non viene segnalato **errore**.

Il **compilatore** passa poi al successivo blocco di istruzioni:

```
.ifc      le time*freqz/6/13–1
.error    “Tempo troppo corto”
.mexit
.endc
```

che equivale a: se il risultato dell'espressione **time*freqz/6/13–1** è minore o uguale (**le**) a **0**, segnala a video il messaggio di errore **“Tempo troppo corto”**, quindi esci dalla macro senza generare il programma eseguibile.

Il compilatore esegue automaticamente il calcolo di questa seconda espressione, ma noi possiamo verificare passo passo se questa condizione risulta **vera** o **non vera**.

Poiché abbiamo già visto che **time** vale **1200**, mentre a **freqz** si associa il valore **8**, l'espressione **time*freqz/6/13–1** diventa:

1200*8/6/13–1

Come prima operazione eseguiamo la moltiplicazione:

1200*8 = 9600

poi eseguiamo la prima divisione:

9600/6 = 1600

poi eseguiamo la seconda divisione:

1600/13 = 123,0769

e infine, dopo aver scartato i **decimali**, eseguiamo l'ultima operazione con il solo numero intero:

123–1 = 122

Quindi l'istruzione:

```
.ifc      le time*freqz/6/13–1
```

diventa in pratica:

```
.ifc      le 122
```

e poiché il valore **122** è **maggiore** di **0**, anche questo blocco di istruzioni verrà ignorato senza segnalare nessun **errore**, perché il valore di **1200 microsecondi** che vogliamo utilizzare come ritardo è un valore ammesso dalla macro.

A questo punto il compilatore passa a:

```
        ldi     carmat,time*freqz/6/13–1
```

e dopo aver fatto il calcolo, che darà come risultato sempre **122**:

```
        ldi     carmat,122
```

lo carica nella variabile **carmat** e lo trasforma in formato eseguibile.

Continuando la compilazione trova:

```
lop1    dec     carmat
        jrnz    lop1
        .endm
```

Con la direttiva **.endm**, il compilatore sa che la macro **ritardo** è finita e torna al programma sorgente **PROVA2.ASM** per proseguire la compilazione delle istruzioni, dove trova:

```
                ret
```

che serve per rientrare dalla **call delay1** (vedi fig.10). Quindi può proseguire con:

```
delay2          ritardo         1500
```

e riconoscendo la macro **ritardo**, ricompila nuovamente le istruzioni della macro sostituendo il **time 1200** con il nuovo tempo **1500**.

Quindi l'espressione:

```
        .ifc    gt time*freqz/6/13–256
```

viene semplificata in:

1500*8/6/13–256

il cui risultato è:

1500*8 = 12000
12000/6 = 2000
2000/13 = 153
153–256 = –103

L'istruzione diventa pertanto:

```
        .ifc    gt –103
```

Poiché il valore **–103** non è **maggiore** di **0**, questo blocco di istruzioni viene ignorato e non viene segnalato nessun **errore**.

Il **compilatore** passa poi al secondo blocco di istruzioni, dove l'espressione:

```
        .ifc    le time*freqz/6/13–1
```

viene semplificata in:

1500*8/6/13–1

il cui risultato è:

1500*8 = 12000
12000/6 = 2000
2000/13 = 153
153–1 = 152

L'istruzione diventa pertanto:

```
        .ifc    le 152
```

e poiché il valore **152** è **maggiore** di **0** anche questo blocco di istruzioni verrà ignorato senza segnalare nessun **errore**, perché il valore di **1500 microsecondi** che vogliamo utilizzare come ritardo è un valore ammesso dalla macro.

A questo punto il compilatore passa a:

```
        ldi     carmat,time*freqz/6/13–1
```

e dopo aver fatto il calcolo, che da come risultato sempre **152**:

```
        ldi     carmat,152
```

lo carica nella variabile **carmat** e lo trasforma in formato eseguibile.

Continuando la compilazione trova:

```
lop1    dec     carmat
        jrnz    lop1
        .endm
```

Con la direttiva **.endm** il compilatore sa che la macro **ritardo** è finita e torna al programma sorgente **PROVA2.ASM** per proseguire la compilazione delle istruzioni, dove trova:

```
        ret
```

che serve per rientrare dalla **call delay2** (vedi fig.10).

Ora proviamo a simulare il programma ottenuto con la compilazione, cioè **PROVA2.HEX**, e in fig.11 vediamo la parte relativa al nostro esempio.

Osservate le righe evidenziate in giallo che si rife-

riscono alla sub-routine **delay1** ricavata dalla macro **ritardo**:

```
delay1    ldi     carmat,7Ah
L0$       dec     carmat
          jrnz    L0$
          ret
```

Trasformando il valore esadecimale **7Ah** nel suo decimale corrispondente, otteniamo **122**, che, come desiderato, ci permetterà di ottenere un ritardo di **1200 microsecondi**.
Le righe evidenziate in verde si riferiscono invece alla sub-routine **delay2** ricavata sempre dalla macro **ritardo**:

```
delay2    ldi     carmat,98h
L1$       dec     carmat
          jrnz    L1$
          ret
```

Trasformando il valore esadecimale **98h** nel suo decimale corrispondente, otteniamo **152**, che, come desiderato, ci permetterà di ottenere un ritardo di **1500 microsecondi**.

Prima però di verificare se effettivamente si ottengono i ritardi voluti, apriamo una parentesi per ricordarvi che, quando il compilatore, come nel nostro caso, incontra nelle macro delle etichette o labels interne (**?lop1** in fig.9) le genera automaticamente nel file **.HEX** assegnandole un numero consecutivo. Ecco perché le istruzioni della macro:

```
lop1      dec     carmat
          jrnz    lop1
```

nella simulazione del programma sono diventate rispettivamente:

```
L0$       dec     carmat
          jrnz    L0$

L1$       dec     carmat
          jrnz    L1$
```

Chiudiamo la parentesi e andiamo a fare un piccolo controllo per verificare se effettivamente vengono ottenuti i ritardi voluti.
Sommiamo dunque i **cicli** delle istruzioni delle due sub-routine e moltiplichiamo il risultato per il tempo di un **ciclo macchina** che corrisponde a **1,625 microsecondi**.
Nell'articolo relativo al **software simulatore** per testare i micro **ST6** pubblicato sulla rivista **N.185**, abbiamo fornito l'elenco completo delle istruzioni Assembler indicando, tra le altre cose, il **numero dei cicli macchina**.

Fig.11 Esecuzione del file PROVA2.HEX

Assembler

Ind.	Codice	Label	Mnemonico	
08AA	0DD8FE	main	ldi	wdog,FEh
08AD	418A		call	set_pin
08AF	118A		call	elab1
08B1	918B		call	delay1
08B3	218A		call	elab2
08B5	018C		call	delay2
08B7	A98A		jp	main
08B9	0D847A	delay1	ldi	carmat,7Ah
08BC	FF84	L0$	dec	carmat
08BE	E8		jrnz	L0$
08BF	CD		ret	
08C0	0D8498	delay2	ldi	carmat,98h
08C3	FF84	L1$	dec	carmat
08C5	E8		jrnz	L1$
08C6	CD		ret	

Per un ritardo di **1200** abbiamo:

```
          call    delay1        1 x 4 cicli  =   4
delay1    ldi     carmat,7Ah    1 x 4 cicli  =   4
L0$       dec     carmat      122 x 4 cicli  = 488
          jrnz    L0$         122 x 2 cicli  = 244
          ret                   1 x 2 cicli  =   2
```

Sommando i **cicli macchina** di queste sub-routine otteniamo **742**.
Poiché un ciclo macchina corrisponde a **1,625 microsecondi** noi otteniamo un effettivo ritardo di:

742 x 1,625 = 1205,75 microsecondi

La differenza di **5,75 microsecondi** in più rispetto al ritardo impostato nel file sorgente non è un errore, ma, in questo caso, è dovuto al necessario arrotondamento operato sui decimali nell'espressione calcolata.

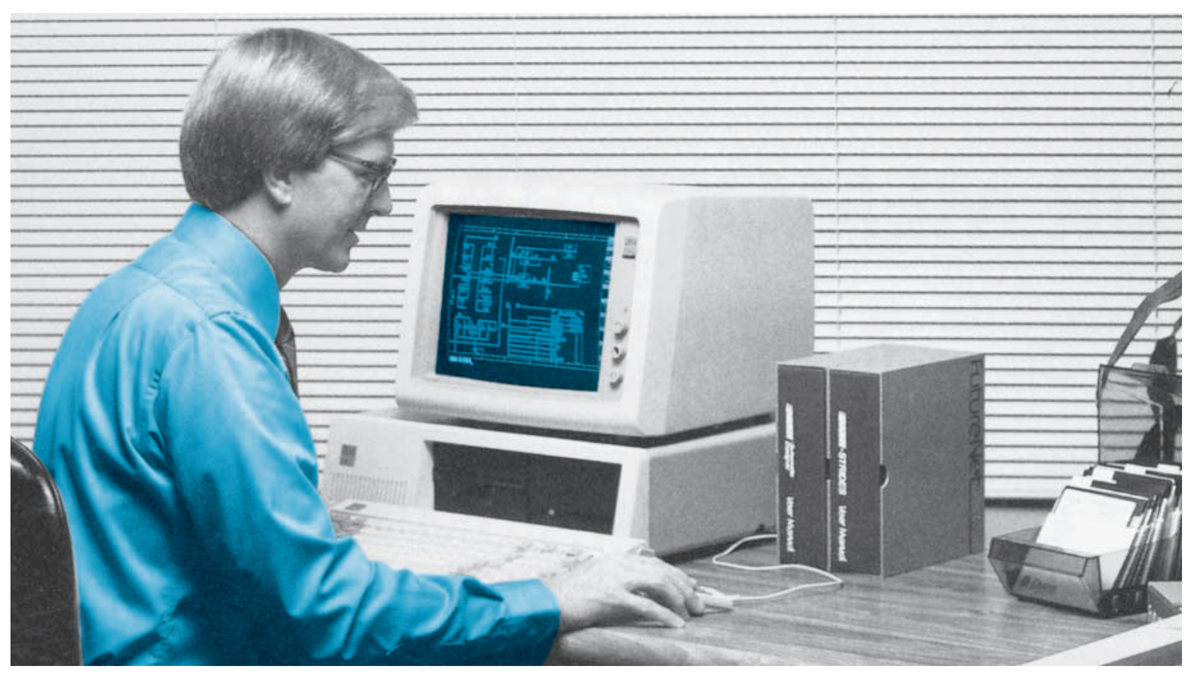
Per un ritardo di **1500** abbiamo:

```
          call    delay2        1 x 4 cicli  =   4
delay2    ldi     carmat,98h    1 x 4 cicli  =   4
L1$       dec     carmat      152 x 4 cicli  = 608
          jrnz    L1$         152 x 2 cicli  = 304
          ret                   1 x 2 cicli  =   2
```

Sommando i **cicli macchina** di queste sub-routine otteniamo **922**.
Poiché un ciclo macchina corrisponde a **1,625 microsecondi** noi otteniamo un effettivo ritardo di:

922 x 1,625 = 1498,25 microsecondi

La differenza di **1,75 microsecondi** in meno rispetto al ritardo impostato nel file sorgente non è un errore, ma, in questo caso, è dovuto al necessario arrotondamento operato sui decimali nell'espressione calcolata.



IL programma LINKER

Con l'articolo sul linker LST6 di cui ci occupiamo in queste pagine, proseguiamo l'esposizione dei diversi aspetti della programmazione dei microcontrollori della serie ST6. Non vi nascondiamo che l'argomento non è dei più semplici, ma con l'aiuto di qualche esempio, siamo certi che anche questa materia non avrà più segreti.

Fino ad oggi nella realizzazione di un programma in **Assembler** per i micro **ST6** ci siamo sempre posti l'obiettivo di scrivere un programma **sorgente**, cioè un file in formato **.ASM** dal quale ottenere un file in formato eseguibile **.HEX**.

Infatti, in tutti gli articoli pubblicati e nei diversi esempi di programmi che vi abbiamo fornito, abbiamo sempre pensato al programma come a una cosa unica, a sé stante, risultato della compilazione in Assembler di un unico file sorgente con tutt'al più l'inserimento, tramite la direttiva **.input**, di subroutine, macro o definizioni di variabili esterne, ma sempre in **formato sorgente**.

L'articolo di oggi si propone invece di illustrarvi un secondo **metodo** per la realizzazione dei vostri programmi, non necessariamente migliore dell'altro, ma sicuramente differente perché presuppone il conseguimento di un altro scopo.

Con il **linker**, termine che possiamo rendere in italiano con **programma di collegamento**, si può ottenere un programma finale eseguibile **.HEX** senza avere il corrispondente programma in formato sorgente, ma **collegando** programmi diversi assemblati in formato oggetto **.OBJ**.
Per semplicità possiamo dunque definire il **linker** come un programma che concatena moduli software al fine di realizzare un programma eseguibile completo.

Il primo passo per usare il **linker** è quello di disporre di una serie di programmi assemblati singolarmente in formato oggetto **.OBJ** utilizzando le opportune opzioni del programma compilatore **Ast6.**

Il secondo passo è quello di lanciare il programma **Lst6** di linkaggio dei file **.OBJ** con le opportune opzioni, in modo da ottenere il programma definitivo eseguibile in formato **.HEX**.

I PROGRAMMI in formato .HEX

Sulla base di quanto fin qui detto, qualcuno potrebbe domandarsi perché non usare il **linker** direttamente con i programmi in formato **.HEX**, invece di utilizzare dei programmi in formato **.OBJ**.

Quando si lancia la compilazione Assembler di un programma, ad esempio **PIPPO.ASM**, a compilazione conclusa, se non vi sono errori, si genera un programma in formato Intel eseguibile, nel nostro caso **PIPPO.HEX**.

Nel file in formato **.HEX**, le singole istruzioni del programma sorgente **.ASM**, sono tradotte in codice binario direttamente eseguibile e soprattutto vi è una **corrispondenza diretta** tra le locazioni di memoria, sia RAM che ROM, attribuite durante la stesura del programma sorgente e quelle ottenute dalla compilazione dell'eseguibile **.HEX**.

Infatti, all'interno di ogni programma, dopo la definizione dei registri e della variabili, viene posta la direttiva **.org 0800h** o **0880h** che serve a posizionare in maniera **assoluta** le istruzioni da quell'indirizzo di memoria ROM in poi.
La stessa cosa si ottiene alla fine con la direttiva **.org 0FF0h**, che posiziona le eventuali gestioni dei vettori di interrupt da quell'indirizzo di memoria assoluta in poi.

Comprenderete quindi che se si tentasse di "**unire**" tramite il linker parti di più programmi in formato **.HEX**, essendo ognuna di esse posizionata a un **indirizzo fisso** di **memoria**, si dovrebbe realizzare un programma ad **incastro**, in maniera che la routine che ci interessa inserire dopo le istruzioni del programma principale iniziasse esattamente ad una ben precisa locazione di memoria successiva a quella già occupata, in caso contrario si correrebbe il rischio di "sovrascrivere" porzioni di programma.
Unire moduli software diventerebbe così un lavoro estremamente difficile, se non impossibile.

A facilitare il nostro compito, ci viene in aiuto il formato **.OBJ**, che essendo **rilocabile** e **non eseguibile**, meglio si presta ad essere linkato.
Vediamo dunque cosa sono i programmi in formato **.OBJ** e come ottenerli.

per i microprocessori ST6

I PROGRAMMI in formato .OBJ

I programmi in formato oggetto **.OBJ** si ottengono utilizzando l'opzione **–O** quando si lancia la compilazione di un programma.
Ad esempio, se compiliamo il file sorgente **PIPPO.ASM** con le opzioni:

Ast6 –L –O PIPPO.ASM

otteniamo il programma **PIPPO.OBJ**.

Nota: ricordiamo ai lettori che le opzioni del compilatore Assembler e il loro utilizzo sono state ampiamente trattate nella rivista **N.194**.

Il programma generato in formato **.OBJ** ha due caratteristiche:

1 – non è direttamente **eseguibile**, pertanto non può essere simulato né caricato su un micro.

2 – le istruzioni contenute non sono in formato assoluto, bensì in formato "**rilocabile**".
In altre parole le istruzioni hanno un indirizzamento di memoria e di Program Counter **relativo** (e non assoluto come nel formato **.HEX**) e quindi possono essere "ricollocate" o, utilizzando un termine specifico, **rilocate**.

E' dunque utile chiarire cosa si intende per indirizzamento relativo e indirizzamento assoluto.

Pensate ad esempio alla numerazione delle pagine di un libro qualsiasi o di una rivista.
Ogni numero specifica la posizione della pagina rispetto alle altre, per cui il numero 10 specifica che quella pagina è la decima della rivista, il numero 128 specifica che quella pagina è la centoventottesima della rivista, e così via.
In questo caso si può parlare di **indirizzamento assoluto** e, a patto di non intervenire in maniera cruenta con tagli o strappi, questo indirizzamento **non cambierà** mai.

Se però decidete di raccogliere insieme gli articoli riguardanti un unico argomento, la numerazione delle pagine non sarà più consecutiva, cioè non avrà una progressione numerica, ma sarà **relativa** alla rivista dalla quale proveniva l'articolo.

Solo quando “concatenerete” uno all’altro gli articoli rinumerando le pagine, darete un nuovo indirizzamento assoluto alla vostra raccolta.

Chiusa questa parentesi, torniamo ai programmi **.OBJ** per precisare che non basta inserire l’opzione **–O** nella compilazione Assembler per ottenere questo formato.
Se provate a compilare un vostro programma inserendo questa opzione, vedrete che il compilatore vi segnalerà un certo numero di errori.

Proprio per le sue peculiarità, nei programmi sorgente bisogna inserire alcune precise direttive e modificarne o toglierne altre prima di generare il formato **.OBJ**.

Le direttive specifiche che servono per generare il formato **oggetto** e quindi anche per **linkare** i programmi **.OBJ**, sono:

.pp_on
.extern
.section
.window
.windowend
.global
.notransmit
.transmit

Nell’esempio che vi proponiamo di seguito cercheremo di chiarire in quale modo e perché vanno utilizzate queste direttive per ottenere un programma **.OBJ** senza errori.

I programmi PLEXER.ASM e PCONT.ASM

Per il nostro esempio abbiamo utilizzato un nostro datato, ma semplice programma dimostrativo che esegue un conteggio e lo visualizza su due display.
In fig.1 abbiamo riportato il listato del programma **CONTA.ASM** così come lo avevamo realizzato.

Dal programma **CONTA.ASM** abbiamo estratto le istruzioni che vedete evidenziate in azzurro in fig.1 e le abbiamo inserite in un nuovo programma che abbiamo chiamato **PLEXER.ASM**.
Questo programma ci mette a disposizione una serie di subroutine che gestiscono l’incremento o il decremento di un contatore e la visualizzazione a due cifre del risultato su 2 display in multiplexer.

Abbiamo quindi cancellato dal programma **CONTA.ASM** le istruzioni inserite in **PLEXER.ASM** e abbiamo salvato ciò che rimaneva con il nome **PCONT.ASM** per non confonderlo con l’originale.

LISTATO del programma CONTA.ASM

```
;* Programma per fare un conteggio *

        .title     "CONTA"
        .vers      "ST62E25"
        .w_on
        .romsize   4
        .input     "ST62X.DEF"

;VARIABILI usate da questo PROGRAMMA

lsb     .def       084h
msb     .def       085h
del1    .def       086h
del2    .def       087h
up_dw   .def       088h

        .org       0800h

inizio
        ldi        wdog,0ffh

        ldi        port_a,00000000b
        ldi        pdir_a,00001100b
        ldi        popt_a,00001100b

        ldi        port_b,00000000b
        ldi        pdir_b,11111111b
        ldi        popt_b,11111111b

        ldi        port_c,00000000b
        ldi        pdir_c,00000000b
        ldi        popt_c,00000000b

;***   Disabilita gli Interrupt
        ldi        adcr,0
        ldi        tscr,0
        ldi        ior,0

        reti

        jp         main

;***    GESTORI di INTERRUPT      ***

ad_int   reti
tim_int  reti
BC_int   reti
A_int    reti
nmi_int  reti
```

Fig.1 Dal programma CONTA.ASM, di cui vi forniamo il listato, abbiamo estratto le istruzioni evidenziate in azzurro e le abbiamo salvate nel file PLEXER.ASM (vedi fig.3). Le istruzioni rimaste sono state salvate nel file PCONT.ASM (vedi fig.2).

```
;***            SUBROUTINE              ***

;- multiplexa le 2 cifre sui display
mulplx
        ld          a,lsb
        addi        a,40h
        ld          x,a
        ld          a,(x)
        ldi         port_a,00001100b
        ld          port_b,a
        ldi         port_a,00000100b

        ld          a,msb
        addi        a,40h
        ld          x,a
        ld          a,(x)
        ldi         port_a,00001100b
        ld          port_b,a
        ldi         port_a,00001000b

        ret

;- incremento delle 2 cifre
;- con controlli
increm
        inc         lsb
        ld          a,lsb
        cpi         a,10
        jrnz        incr1
        ldi         lsb,0
        inc         msb
        ld          a,msb
        cpi         a,10
        jrnz        incr1
        ldi         msb,0
incr1
        ret

;- decremento delle 2 cifre
;- con controlli
decrem
        ld          a,lsb
        cpi         a,0
        jrnz        decr1
        ldi         lsb,9

        ld          a,msb
        cpi         a,0
        jrnz        decr2
        ldi         msb,9
        ret

decr2   dec         msb
        ret

decr1   dec         lsb
        ret
```

```
;***     PROGRAMMA PRINCIPALE      ***

main
        ldi         wdog,0feh

        ldi         lsb,0
        ldi         msb,0
        ldi         up_dw,1

        ldi         drw,digit.w

loop    ldi         del1,17
main1   ldi         del2,255
main2   ldi         wdog,0feh

        call        mulplx

        dec         del2
        jrz         main3
        jp          main2
main3   dec         del1
        jrz         main6

        jrs         0,port_a,main4
        ldi         up_dw,0
main4   jrs         1,port_a,main5
        ldi         up_dw,1
main5
        jp          main1
main6
        ld          a,up_dw
        cpi         a,0
        jrz         main7

        call        increm
        jp          loop

main7   call        decrem
        jp          loop

;*** tabella con i segmenti per far
; apparire sui display le cifre ***

        .block      64-$%64
digit   .byte       192,249,164,176,153
        .byte       146,130,248,128,144

;*** VETTORI DI INTERRUPTS ***

        .org        0ff0h
        jp          ad_int
        jp          tim_int
        jp          BC_int
        jp          A_int
        .org        0ffch
        jp          nmi_int
        jp          inizio

        .end
```

LISTATO del programma PCONT.ASM

```
;* Programma per fare un conteggio *

        .title    "PCONT"
        .vers     "ST62E25"
        .w_on
        .romsize  4
        .pp_on
        .input    "ST62X.DEF"

;VARIABILI usate da questo PROGRAMMA

del1    .def      084h
del2    .def      085h
up_dw   .def      086h
lsb     .def      087h
msb     .def      088h

        .extern  decrem,increm,mulplx
        .section  1

inizio
        ldi       wdog,0ffh

        ldi       port_a,00000000b
        ldi       pdir_a,00001100b
        ldi       popt_a,00001100b

        ldi       port_b,00000000b
        ldi       pdir_b,11111111b
        ldi       popt_b,11111111b

        ldi       port_c,00000000b
        ldi       pdir_c,00000000b
        ldi       popt_c,00000000b

;***  Disabilita gli Interrupt

        ldi       adcr,0
        ldi       tscr,0
        ldi       ior,0

        reti

        jp        main

;***    GESTORI di INTERRUPT       ***

ad_int   reti
tim_int  reti
BC_int   reti
A_int    reti
nmi_int  reti
```

Fig.2 Listato del programma PCONT.ASM. Per compilarlo in formato oggetto .OBJ abbiamo dovuto inserire le direttive evidenziate in giallo.

```
;***      PROGRAMMA PRINCIPALE     ***

main
        ldi       wdog,0feh
        ldi       lsb,0
        ldi       msb,0
        ldi       up_dw,1
        ldi       drw,digit.w

loop    ldi       del1,17
main1   ldi       del2,255
main2   ldi       wdog,0feh

        call      mulplx

        dec       del2
        jrz       main3
        jp        main2
main3   dec       del1
        jrz       main6

        jrs       0,port_a,main4
        ldi       up_dw,0
main4   jrs       1,port_a,main5
        ldi       up_dw,1
main5
        jp        main1
main6
        ld        a,up_dw
        cpi       a,0
        jrz       main7

        call      increm
        jp        loop

main7   call      decrem
        jp        loop

;*** tabella con i segmenti per far
; apparire sui display le cifre ***

        .window
digit   .byte     192,249,164,176,153
        .byte     146,130,248,128,144
        .windowend

;*** VETTORI DI INTERRUPTS ***

        .section  32
        jp        ad_int
        jp        tim_int
        jp        BC_int
        jp        A_int
        .block    4
        jp        nmi_int
        jp        inizio

        .end
```

LISTATO del programma PLEXER.ASM

```
;* Modulo per gestire un multiplexer
;* a due cifre

       .title    "PLEXER"
       .vers     "ST62E25"
       .w_on
       .romsize  4
       .pp_on
       .input    "ST62X.DEF"

;VARIABILI usate da questo PROGRAMMA

lsb    .def      084h
msb    .def      085h
       .section  1

;***         SUBROUTINE          ***

;- multiplexa le 2 cifre sui display
mulplx
       ld        a,lsb
       addi      a,40h
       ld        x,a
       ld        a,(x)
       ldi       port_a,00001100b
       ld        port_b,a
       ldi       port_a,00000100b

       ld        a,msb
       addi      a,40h
       ld        x,a
       ld        a,(x)
       ldi       port_a,00001100b
       ld        port_b,a
       ldi       port_a,00001000b

       ret
```

```
;- incremento delle 2 cifre
;- con controlli
increm
       inc       lsb
       ld        a,lsb
       cpi       a,10
       jrnz      incr1
       ldi       lsb,0
       inc       msb
       ld        a,msb
       cpi       a,10
       jrnz      incr1
       ldi       msb,0
incr1
       ret


;- decremento delle 2 cifre
;- con controlli
decrem
       ld        a,lsb
       cpi       a,0
       jrnz      decr1
       ldi       lsb,9

       ld        a,msb
       cpi       a,0
       jrnz      decr2
       ldi       msb,9
       ret

decr2  dec       msb
       ret

decr1  dec       lsb
       ret
```

Fig.3 Listato del programma PLEXER.ASM. Queste istruzioni sono state tratte dal programma CONTA.ASM (vedi in fig.1 le istruzioni evidenziate in azzurro), e per compilarle in formato .OBJ abbiamo inserito le direttive evidenziate in giallo.

A questo punto abbiamo due programmi, **PLEXER.ASM** e **PCONT.ASM**, che dobbiamo modificare e compilare separatamente per ottenere rispettivamente **PCONT.OBJ** e **PLEXER.OBJ**.
Vedremo così come, linkando questi programmi, si ottenga un terzo programma in formato **.HEX**.

Per generare in formato **.OBJ** il programma **PCONT**, abbiamo dovuto modificare il listato come visibile in fig.2. Per generare in formato **.OBJ** il programma **PLEXER**, abbiamo dovuto modificare il listato come visibile in fig.3.
In entrambe le figure abbiamo evidenziato in **giallo** le direttive inserite e ora analizzeremo nei dettagli queste modifiche via via che le incontreremo.

La direttiva .pp_on

Rispetto al programma originario, e cioè CONTA.ASM, nel programma **PCONT.ASM** dopo la direttiva **.romsize 4** abbiamo inserito la direttiva **.pp_on**, che abilita la paginazione della memoria del micro.

Normalmente questa direttiva va inserita quando si realizzano programmi per i microprocessori ST6 che dispongono di più di **4 kbyte** di memoria **Program Space** (**ROM**).

In questi modelli di micro infatti, esiste una memoria ROM che possiamo definire **primaria** di 4096

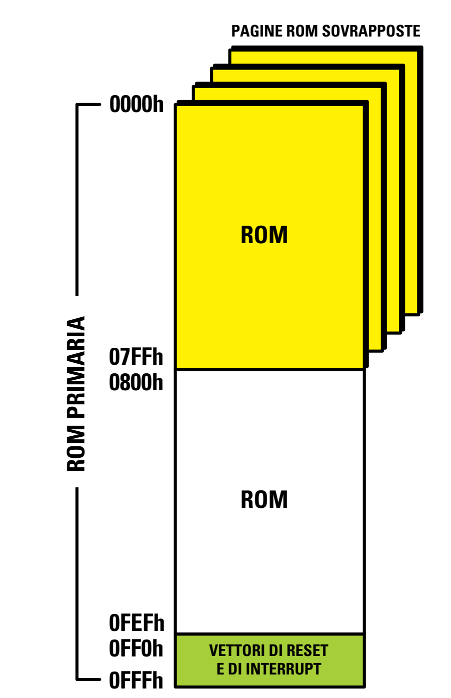


Fig.4 I micro con più di 4 kbytes di memoria Program Space hanno una ROM primaria suddivisa in tre pagine di grandezza diversa: da 0000h a 07FFh, da 0800h a 0FEFh e da 0FF0h a 0FFFh. Inoltre hanno un massimo di 30 pagine sovrapposte con locazione da 0000h a 07FFh.

bytes che va da locazione 0000h a 0FFFh e un massimo di 30 "pagine" sovrapposte di 2048 bytes di area ROM, tutte con locazione da 0000h a 07FFh, come visibile nel disegno di fig.4.

La stessa memoria **primaria** viene ulteriormente suddivisa in tre **pagine** di area ROM di grandezza diversa: la prima ha locazione 0000h – 07FFh, la seconda ha locazione 0800h – 0FEFh e la terza ha locazione 0FF0h – 0FFFh.

A ciascuna per comodità viene virtualmente associato un numero di pagina che va da 0 a 32 (vedi tabella di fig.5) e ogni pagina deve essere indirizzata tramite un'altra direttiva chiamata **.section**.
Utilizzando **.pp_on**, e quindi segnalando al compilatore che deve virtualmente suddividere la memoria in pagine, bisognerà utilizzare anche la direttiva **.section** che serve a indirizzare queste pagine.

Nel nostro esempio noi utilizziamo un micro ST62E25 che non supera i 4 kbyte di memoria (vedi **.vers** in fig.2), ma volendo generare un programma in formato **.OBJ** siamo obbligati ad inserire la direttiva **.section** e, di conseguenza, a definire anche **.pp_on**, altrimenti il compilatore segnalerà errore.

La direttiva .extern

Sempre rispetto al programma **CONTA**, la successiva istruzione che abbiamo inserito nel programma **PCONT** è la direttiva **.extern** con a fianco l'indicazione di tre etichette:

.extern decrem,increm,mulplx

La direttiva **.extern** va obbligatoriamente inserita ogniqualvolta si intende assemblare in formato **.OBJ** un programma contenente istruzioni che richiamano o saltano a labels di routine che non si trovano all'interno del programma stesso, ma sono inserite in altri programmi. In questo modo si avverte il compilatore di non segnalare errore quando non trova le routine richiamate.

Nel programma **PCONT** (vedi listato in fig.2) ci sono infatti tre routine chiamate con le istruzioni:

call mulplx
call increm
call decrem

che **non** vengono assolutamente **definite**, perché inserite nel programma **PLEXER** (vedi fig.3).

Inserendo la direttiva **.extern**, abbiamo avvertito il compilatore che le routine sono **esterne**, e che quindi non deve segnalare errore quando incontra le istruzioni che le richiamano.

Per avere una riprova di ciò, abbiamo provato a togliere l'istruzione:

.extern decrem,increm,mulplx

e abbiamo compilato **PCONT**.
In fig.6 è visibile la segnalazione di errore del compilatore, in cui queste tre etichette vengono indicate come "undefined symbol".

E' importante annotare che quando si utilizza questa direttiva per definire **labels** di **routine esterne** al programma, conviene sempre porla all'inizio così da rendere visibile già in fase di edit, che il programma contiene salti o richiami a routine esterne.

Possono essere definite come **.extern** solamente labels di Program Space e di Data Rom Windows.

Non possono essere definite come **.extern** i registri, le variabili di **Data space** o le costanti (**.def**, **.equ**, **.set**).

Pagina N°	INDIRIZZO VIRTUALE	INDIRIZZO REALE
0	0000 - 07FF	0000 - 07FF
1	0800 - 0FEF	0800 - 0FEF
2	1000 - 17FF	0000 - 07FF
3	1800 - 1FFF	0000 - 07FF
da 4 a 31	[n*800]-[(9n*80)+7FF]	0000 - 07FF
32	0FF0 - 0FFF	0FF0 - 0FFF

Fig.5 A ogni pagina di memoria, che ha un suo preciso indirizzo reale, viene associato per comodità un indirizzo virtuale rappresentato da un numero da 0 a 32.

Per finire, questa direttiva può essere inserita solamente nei programmi che verranno compilati con l'opzione **–O**. In caso contrario il compilatore segnalerà errore.

La direttiva .section

Confrontate ancora il programma **PCONT.ASM** di fig.2 all'altezza della label **inizio** con il programma **CONTA.ASM** di fig.1 alla stessa altezza.
Nel programma originale **CONTA.ASM** prima della label **inizio** avevamo inserito l'istruzione **.org 0800h**, mentre in **PCONT.ASM** l'abbiamo sostituita con **.section 1**.

Quando si assembla un programma in formato **.OBJ** si deve sostituire la direttiva **.org** con la direttiva **.section** seguita da un numero da **0** a **32**, altrimenti verrà segnalato errore.

In relazione a quanto detto precedentemente a proposito della direttiva **.pp_on**, che attiva la "paginazione" o, se preferite, la suddivisione in pagine della memoria ROM, inserendo nel programma la direttiva **.section** noi indichiamo al compilatore in quale "pagina" di memoria ROM deve inserire le istruzioni del programma da compilare in formato **.OBJ**.

Nel nostro caso, noi indichiamo al compilatore che le istruzioni del programma **PCONT** devono essere inserite nella Program Space di pagina **1**, e cioè nell'area ROM con locazione **0800h – 0FEFh** come visibile in fig.7.

All'interno dello stesso programma è possibile inserire più direttive **.section** per indirizzare pagine diverse ed inserire perciò le istruzioni in punti diversi di Program Space.
Poiché però vi sono alcune limitazioni sull'utilizzo delle istruzioni di salto da una pagina di memoria all'altra, bisogna fare attenzione alle caratteristiche di "salto" legate al numero di pagina utilizzato.

In fig.7 riportiamo la tabella illustrativa di queste limitazioni.

Nella colonna "**salto a...**", in corrispondenza delle righe di pagina **1** e di pagina **32** è indicato **tutte le pagine**, mentre nelle restanti è indicato **pagina 1**.
Questo significa che nelle pagine **1** e **32** di Program Space si possono inserire istruzioni di salto incondizionato (**jp, call**) a tutte le altre pagine di memoria, mentre nelle pagine **0** e da **2** a **31** si possono inserire solamente istruzioni di salto incondizionato alla pagina **1**.

```
C:\ST6\LX1208>ast6 -l -o PCONT.ASM
ST6 MACRO-ASSEMBLER version 4.00  - August 1992
Error  PCONT.ASM 94: (106) undefined symbol: decrem
Error  PCONT.ASM 91: (106) undefined symbol: increm
Error  PCONT.ASM 72: (106) undefined symbol: mulplx
Execution time: 0 second(s)
3 errors detected
No object created
```

Fig.6 Errore segnalato dal compilatore quando non si usa correttamente la direttiva .extern.

Pagina N°	INDIRIZZO VIRTUALE	INDIRIZZO REALE	SALTO A
0	0000 - 07FF	0000 - 07FF	Pagina 1
1	0800 - 0FEF	0800 - 0FEF	tutte le Pag.
2	1000 - 17FF	0000 - 07FF	Pagina 1
3	1800 - 1FFF	0000 - 07FF	Pagina 1
da 4 a 31	[n*800]-[(9n*80)+7FF]	0000 - 07FF	Pagina 1
32	0FF0 - 0FFF	0FF0 - 0FFF	tutte le Pag.

Fig.7 Esistono delle limitazioni sull'utilizzo dell'istruzione di salto da una pagina di memoria all'altra, per cui nelle pagine 0 e da 2 a 31 si possono inserire solo istruzioni di salto a pagina 1.

Facciamo un esempio. Compilando il programma:

```
.section 1
inizio   ................
         ................
         ................
         jp  letsta
rien1    ................
         ................
.section 2
letsta   ldi  a,23
         ................
         ................
         call  storx
         ................
.section 3
storx    addi  a,23
         ................
         ret
```

non verrà segnalato errore, perché le istruzioni sono formalmente corrette.
Quando però tenteremo di **linkare** questo programma, il **linker** segnalerà un errore simile a quello di fig.8, perché non sono state rispettate le condizioni. Infatti, da **pagina 1** con l'istruzione **jp letsta** si può passare alla **pagina 2**, ma poi l'istruzione **call storx** non può essere eseguita perché **storx** si trova nella **pagina 3**.

La giusta sequenza del nostro esempio è dunque la seguente:

```
.section 1
inizio   ................
         ................
         ................
         jp  letsta
rien1    call  storx
         ................
.section 3
letsta   ldi  a,23
         ................
         ................
         jp  rien1
.section 4
storx    addi  a,23
         ................
         ret
```

L'esempio appena riportato si riferiva a più **section** inserite in un unico programma, ma è evidente che si pone un problema analogo quando diverse **.section** sono inserite in più programmi che andranno concatenati con il **linker**.

Chiusa questa parentesi, torniamo al listato di **PCONT** (vedi fig.2) e soffermiamoci sull'istruzione **.section 32** e sulla successiva **.block 4**.

```
       reference to <increm> external
       reference to <mulplx> external
lst6: ** illegal jump inside program section #2, offset 0x0, file <PCONT.OBJ>
```

Fig.8 Il controllo sul rispetto delle condizioni necessarie all'esecuzione dell'istruzione di salto viene fatto dal programma linker Lst6. In questa figura è segnalato errore perché l'istruzione di salto da pagina 2 può essere eseguita solo verso pagina 1 (vedi fig.7), e non a pagina 3 come scritto nel programma a sinistra sopra questa figura.

Nelle stesse righe del programma originale **CONTA.ASM** vi erano le istruzioni **.org 0FF0h** e **.org 0FFCh**
Con **.section 32** si attiva la pagina di memoria relativa alla gestione dei vettori di reset e di interrupt.
La direttiva **.block 4** sostituisce **.org 0FFCh**, ma ha la stessa funzione di posizionare correttamente i vettori di **nmi** e di **reset**.

Le direttive .window e .windowend

Mettendo ancora una volta a confronto le righe del programma originale CONTA con quelle di **PCONT**, potete vedere che l'istruzione:

.block 64-$%64

è stata sostituita dalla direttiva **.window**, mentre dopo il secondo **.byte** è stata inserita la direttiva **.windowend**.

Come abbiamo avuto occasione di ripetere più volte (vedi soprattutto la rivista **N.190**), normalmente l'istruzione **.block 64-$%64** precede l'inserimento di dati in Program Space (**.byte, .ascii**, **.asciz**) che verranno caricati a blocchi di 64 bytes tramite la Data Rom Windows.

```
PROGRAM SECTIONS:

number  start   end     size
------  -----   ---     ----
1       0800    0F9F    00D3
32      0FF0    0FFF    0010


MODULE PCONT.OBJ:

section    type    start    size
-------    ----    -----    ----
1          P       0800     008A
32         P       0FF0     0010


MODULE PLEXER.OBJ:

section    type    start    size
-------    ----    -----    ----
1          P       088A     0049
```

Fig.9 Mappa della memoria risultante dal link ottenuto con PCONT.OBJ e PLEXER.OBJ. Non avendo inserito le direttive .window e .windowend i due programmi sono stati accodati.

Compito principale di **.block 64-$%64** è di "ottimizzare" l'utilizzo di Program Space.

Compilando il programma **PCONT** in formato **.OBJ** avremmo anche potuto lasciare l'istruzione **.block 64-$%64**, però i dati definiti con le due direttive **.byte** sarebbero stati allocati con allineamento al primo blocco di **64** byte di **Program Space** successivo all'ultima istruzione di **PCONT** e cioè **jp loop**.
Linkando i due programmi **PCONT.OBJ** e **PLEXER.OBJ**, il **linker** avrebbe "accodato" al programma **PCONT** le istruzioni del programma **PLEXER**, che quindi si sarebbero venute a trovare dietro a quest'area dati.
Avremmo pertanto avuto un programma finale **.HEX** non bene ottimizzato, sia come utilizzo di memoria Program Space sia come "leggibilità".

Per provarvi quanto detto, abbiamo linkato **PLEXER.OBJ** e **PCONT.OBJ** lasciando al suo interno l'istruzione **.block 64-$%64** e senza inserire la direttiva **.windowend**.

In fig.9 potete vedere la mappa della memoria del programma **.HEX** risultante.
Il programma **PCONT.OBJ** (vedi **Module**) inizia all'indirizzo di memoria **0800h** e termina all'indirizzo **0800h + 008Ah**, cioè a **088Ah**, mentre il programma **PLEXER.OBJ** inizia proprio da **088Ah** e termina a **088Ah + 0049h**, cioè a **08D3**.

Abbiamo poi simulato l'esecuzione del programma **.HEX** con un Simulatore Software e in fig.10 potete avere la riprova di quanto affermato poco sopra.

In alto è evidenziata l'ultima istruzione eseguibile di **PCONT** e cioè **jp loop** seguita da una serie di istruzioni **jrnz** che indirizzano sempre al byte successivo. Questo è il risultato dell'inserimento dell'istruzione **.block 64-$%64** che il compilatore traduce appunto in tanti salti di **1 byte** fino a quando non arriva ad un blocco di memoria divisibile esattamente per 64.
Infatti, quasi in fondo alla figura compare la label **digit** che identifica il punto di memoria esatto in cui sono stati inseriti i dati con i **.byte** e alla sua sinistra compare l'indirizzo di memoria relativo e cioè **0880h** che è appunto un indirizzo divisibile esattamente per **64**.

Spostate lo sguardo più sotto e nella riga evidenziata vedrete l'istruzione **mulplx ld a,lsb**, che è la prima istruzione del programma **PLEXER** e si trova effettivamente all'indirizzo di memoria **088Ah**.

Vediamo invece cosa succede inserendo **.window** al posto di **.block 64-$%64** e aggiungendo **.win-**

```
Assembler
Ind. Codice  Label   Mnemonico
0866 E983            jp      loop
0868 C18B    main7   call    decrem
086A E983            jp      loop
086C 00              jrnz    86Dh
086D 00              jrnz    86Eh
086E 00              jrnz    86Fh
086F 00              jrnz    870h
0870 00              jrnz    871h
0871 00              jrnz    872h
0872 00              jrnz    873h
0873 00              jrnz    874h
0874 00              jrnz    875h
0875 00              jrnz    876h
0876 00              jrnz    877h
0877 00              jrnz    878h
0878 00              jrnz    879h
0879 00              jrnz    87Ah
087A 00              jrnz    87Bh
087B 00              jrnz    87Ch
087C 00              jrnz    87Dh
087D 00              jrnz    87Eh
087E 00              jrnz    87Fh
087F 00              jrnz    digit
0880 C0      digit   jrnz    879h
0881 F9A4            jp      A4Fh
0883 B0              jrnz    87Ah
0884 9992            jp      929h
0886 82              jrnc    877h
0887 F8              jrnz    887h
0888 80              jrnz    879h
0889 90              jrnz    87Ch
088A 1F84    mulplx  ld      a,lsb
088C 5740            addi    a,40h
```

Fig.10 L'istruzione .block 64-$%64 è stata tradotta dal compilatore in salti di 1 byte fino ad un blocco di memoria divisibile per 64. Infatti i dati .byte, identificati dalla label digit, vengono inseriti all'indirizzo 0880h, che è divisibile per 64.

dowend. Ricompiliamo in Assembler il programma **PCONT** in formato **.OBJ** con il comando:

ast6 –L –O PCONT.ASM.

Abbiamo inserito anche l'opzione **–L** perché vogliamo generare anche **PCONT.LIS**.
Quando il compilatore incontra la direttiva **.window** prosegue fino a che non trova **.windowend** (che deve sempre essere inserita) e "memorizza" i dati (**.byte**, **.ascii**, ecc.) definiti tra questi estremi in una area rilocabile particolare definita come **Window section**.

In fig.11 abbiamo riprodotto la parte del file **PCONT.LIS** che riguarda queste direttive.
All'altezza della riga **119** potete notare la scritta **W00** che appunto rappresenta l'assegnazione alla Window section dei nostri 10 byte di data space identificati dalla label **digit**, visibili a destra nella stessa riga.
Notate inoltre che a fianco di **W00** c'è il numero **0000**: normalmente questo numero rappresenta la locazione di memoria in cui verrà memorizzata l'istruzione e in questo caso i nostri 10 bytes verranno "memorizzati" a partire dall'indirizzo **0** della **Window section**.

A questo punto possiamo linkare **PCONT.OBJ** e **PLEXER.OBJ** per ottenere l'eseguibile **.HEX** e in fig.12 riportiamo la mappa di memoria risultante.
Notate subito che rispetto alla mappa precedente (vedi fig.9) vi è una **Window section** che inizia a **08B5h** ed è lunga **000Ah** (cioè i 10 byte di **digit**).
La stessa Window section è poi richiamata più in basso, nel programma **PCONT.OBJ**, nella terza riga della seconda colonna (vedi type W).

```
 113 S01 0068 0100   S01 0068      91  main7   call    decrem
 file PCONT.lis  page 4

PCONT
 114 S01 006A E903   S01 006A      92          jp      loop
 115                               93
 116                               94  ;***********************************
 117                               95  ;tabella con i segmenti per far appa
 118                               96          .window
 119 W00 0000 C0     W00 0000      97  digit   .byte   192,249,164,176,153
 120 W00 0001 F9     W00 0001      97
 121 W00 0002 A4     W00 0002      97
```

Fig.11 Ricompilando il programma PCONT dopo aver inserito le direttive .window e .windowend, il compilatore "memorizza" le istruzioni racchiuse tra queste due direttive in un'area rilocabile definita window section: notate la scritta W00 all'altezza della riga 119. Il numero che segue (0000) rappresenta la locazione di memoria in cui vengono memorizzate le istruzioni racchiuse tra le direttive .window e .windowend.

```
WINDOW SECTIONS:

number   start    end      size
------   -----    ---      ----
0        08B5     08BE     000A


MODULE PCONT.OBJ:

section     type     start     size
-------     ----     -----     ----
1           P        0800      006C
32          P        0FF0      0010
0           W        08B5      000A


MODULE PLEXER.OBJ:

section     type     start     size
-------     ----     -----     ----
1           P        086C      0049
```

Fig.12 Mappa della memoria risultante dal link dei programmi PCONT.OBJ e PLEXER.OBJ dopo aver inserito le direttive .window e .windowend. Rispetto alla fig.9, c'è una window section lunga esattamente 10 byte (000Ah), il cui inizio non è più a 0000, ma a 08B5, perché il linker ha posizionato la window section in coda a tutte le istruzioni.

Notate però che l'indirizzo della Window section non è più 0000 come era in PCONT.LIS di fig.11 ma è diventato come già detto **08B5h**.
Il linker infatti ha unito in sequenza le istruzioni dei programmi **PCONT** e **PLEXER** e solo dopo, in coda a tutto, ha "rilocato" la Window section.

Per fare questo calcola innanzitutto la grandezza dell'area dati che si vuole inserire in Program Space tramite Window section (nel nostro esempio **digit** sono 10 bytes), poi si posiziona alla prima locazione di Program Space divisibile esattamente per 64 successiva all'ultima istruzione del programma finale .HEX.

Se la differenza fra questa locazione e quella relativa all'ultima istruzione del programma è maggiore della grandezza dell'area dati da inserire (**digit**), inserisce i dati prima di questa locazione (vedi fig.13), se invece è minore, li inserisce dopo (vedi fig.14).

Per concludere, con le direttive **.window** e **.windowend**, i dati da inserire in Program Space vengono automaticamente posizionati in coda a tutte le istruzioni, in un area già ottimizzata evitando così inutili sprechi di memoria e soprattutto predisponendoli ad essere caricati in maniera corretta nella Data Rom Window.

Nota: nella rivista **N.190** abbiamo spiegato il corretto utilizzo della Data Rom Window.

0800h
AREA ROM
per le istruzioni
di programma
08B5h
INDIRIZZO DI MEMORIA
DELL'ULTIMA ISTRUZIONE
11 BYTES
DIGIT
10 BYTES
08C0h
1° INDIRIZZO SUCCESSIVO
DIVISIBILE per 64 (40h)

Fig.13 Se la differenza tra la prima locazione di memoria ROM divisibile per 64 e la locazione dell'ultima istruzione è maggiore dell'area window section, i dati vengono inseriti prima di questa locazione.

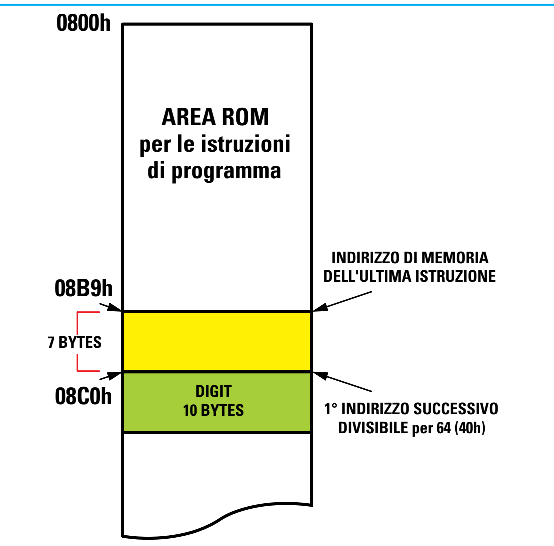


Fig.14 Se la differenza tra la prima locazione di memoria ROM divisibile per 64 e la locazione dell'ultima istruzione è minore dell'area window section, i dati vengono inseriti dopo questa locazione.

L'unica restrizione all'uso di queste direttive è che tra **.window** e **.windowend** si possono inserire un massimo di 64 byte di dati altrimenti il compilatore Assembler segnalerà questo errore:

Error current program section overflow (fatal)

In questo caso dovrete spezzare i vostri dati in blocchi di massimo 64 byte e utilizzare più volte le direttive **.window .windowend** come riportato nell'esempio che segue:

```
      .window
dig01 .byte        ...............
      .................
      .windowend
      .window
dig02 .ascii       ...............
      .byte        ...............
      .windowend
```

A questo punto abbiamo terminato l'analisi del programma **PCONT.ASM** e possiamo dedicarci al programma **PLEXER.ASM**.

Innanzitutto potete notare che davanti alle istruzioni che abbiamo estratto dal programma originale **CONTA.ASM** (vedi il listato in fig.3), sono state inserite le direttive necessarie al programma **PLEXER** per essere compilato:

```
.title     "plexer"
.vers      "st62e25"
.w_on
.romsize   4
```

Abbiamo quindi aggiunto la direttiva **.pp_on** (vedi riga evidenziata in giallo) che, come abbiamo già detto, attiva la "paginazione" e abbiamo ripetuto la definizione dei registri con la direttiva **.input** e la definizione delle variabili **lsb** e **msb**, che avevamo già definito in **PCONT.ASM**.

Questa ripetizione si è resa necessaria dal momento che queste variabili e alcuni registri vengono utilizzati nel programma e perciò se non li avessimo segnalati, il compilatore avrebbe dato errore e non avrebbe compilato il programma nel formato oggetto **.OBJ**.

Se state pensando che si potevano evitare queste istruzioni definendo i registri e le variabili come esterni con la direttiva **.extern**, siete in errore.

Come infatti abbiamo già detto, ma forse è utile ripetere, possono essere **definite** come **.extern** solamente **labels** di Program Space e di Data Rom Windows, mentre i registri, le variabili di **Data space** e le costanti (**.def**, **.equ**, **.set**) **non** possono essere **definite** con questa direttiva.

E' però importante farvi notare che le variabili **lsb** e **msb** sono state definite ad un indirizzo di memoria differente da quello che avevano nel programma **PCONT.ASM** (vedi di fig.2).
Torneremo più avanti su questo argomento.

In conclusione sottolineiamo che anche in questo programma è stata inserita la direttiva **.section 1**, che oramai conoscete.

A questo punto assembliamo in formato **.OBJ** i programmi **PCONT.ASM** e **PLEXER.ASM** digitando al prompt di DOS:

ast6 –L –O PCONTA.ASM
ast6 –L –O PLEXER.ASM

Otteniamo così **PCONTA.OBJ** e **PLEXER.OBJ**, che ora possiamo "unire" con il linker **lst6** per ottenere un programma eseguibile al quale diamo nome **XCONTA.HEX**.

Ottenere il formato .HEX con il linker lst6

Finora abbiamo sempre parlato di "unire" più programmi **.OBJ** per ottenere un programma eseguibile **.HEX**.

In realtà è meglio utilizzare il termine **collegare**, perché i programmi vengono collegati insieme e ogni indirizzo di memoria, che prima era relativo ad un singolo programma, diventa **indirizzo assoluto** nel programma finale **.HEX**.

A questo punto penserete che essendo i programmi correttamente compilati in formato **.OBJ**, linkandoli non incontreremo alcun ostacolo.

In realtà le cose non stanno proprio così, ma poiché non sarebbe utile anticipare i problemi, vediamo per ora come si lancia il linker Lst6 per collegare **PCONT.OBJ** e **PLEXER.OBJ** e ottenere **XCONTA.HEX**.
Al prompt di DOS digitiamo:

lst6 –S –I –T –V –M –O XCONTA PCONT PLEXER

Le scritte **–S –I –T –V –M** sono opzioni specifiche del linker Lst6 che verranno spiegate in maniera completa nel prossimo articolo.
Per non appesantire questo articolo, ci soffermiamo solo su **–O XCONTA PCONT PLEXER**.

Nota: attenzione a non confondere l'opzione **–O** del linker con l'opzione **–O** dell'Assembler.

L'opzione **–O** del linker seguita dal nome del programma finale, nel nostro caso **XCONTA**, serve ad indicare al linker come dovrà chiamare il programma eseguibile **.HEX**.

Come potete notare noi ci siamo limitati a scrivere **XCONTA**, perché l'estensione **.HEX** viene messa automaticamente dal programma **lst6**.

Se avessimo voluto ottenere un programma con una diversa estensione avremmo dovuto scrivere il nome per esteso: ad esempio **–O XCONTA.PGM**.

Dopo il nome dell'eseguibile, scriviamo in successione il nome dei programmi da concatenare, cioè **PCONT** e **PLEXER**, omettendo anche stavolta l'estensione **.OBJ**, perché assunta di default.

E' invece molto **IMPORTANTE** l'ordine in cui vengono definiti i programmi da linkare, perché il linker seguirà quell'ordine per collegarli.

Nel nostro esempio i programmi sono due, ma potrebbero essere molti di più.

CONTROLLO delle CONDIZIONI

Lanciamo quindi il linker e, come già anticipato, a video compaiono le segnalazioni di **errore** visibili in fig.15. Dopo la visualizzazione della versione del Linker e la segnalazione del copyright c'è la scritta:

pass1 :

Il linker o, come più correttamente sarebbe giusto chiamarlo, il Linkage Editor, agisce infatti in due fasi o passi.
Il primo passo o **pass1** è quello di controllare che in tutti i programmi **.OBJ** da linkare ci siano le condizioni per poterli collegare segnalando eventuali errori.
Il secondo passo o **pass2** è quello specifico di collegare ogni singola istruzione e locazione di memoria dei vari programmi in modo da ottenere un unico programma eseguibile. E' in questa seconda fase che le locazioni di memoria dei singoli programmi vengono in un certo senso sistemate una in "coda" all'altra con la conseguente "rilocazione" o "rimappatura" degli indirizzi.

In tutti gli articoli sul linguaggio di programmazione Assembler usato dai microprocessori ST6, vi abbiamo sempre spiegato le procedure per scrivere i programmi unendo la teoria, della quale non si può fare a meno, alla pratica, con esempi che fossero semplici e immediati. Chi ha avuto la costanza di seguirci non ne è rimasto deluso e con questo articolo sul linker Lst6 potrà acquisire ulteriori elementi per programmare senza problemi.

Sotto **pass 1** leggiamo:

<PCONT.obj>: program section(s) size is 0x7C (124), window(s) size is 0xA (10)

Il linker calcola e segnala l'occupazione di Program Space (**124** bytes) e l'occupazione di window section (**10** bytes: ricordate la definizione di **digit** tra **.window** e **.windowend**) del programma **PCONT**.

Di seguito c'è:

<PLEXER.obj>: program section(s) size is 0x49 (73), window(s) size is 0x0 (0)

Il calcolo della memoria di Program Space (**73** bytes) e l'eventuale presenza di window section, avviene anche per il programma **PLEXER**.

Nelle tre righe seguenti leggiamo:

lst6 : ** undefined symbol <decrem> ; first referenced in file <PCONT.obj>
lst6 : ** undefined symbol <increm> ; first referenced in file <PCONT.obj>
lst6 : ** undefined symbol <mulplx> ; first referenced in file <PCONT.obj>
lst6 : <3> fatal error(s) occurred

Effettuando un controllo sulla possibilità di collegare **PCONT** e **PLEXER**, il linker rileva tre anomalie relative alle routine identificate dalle labels **decrem**, **increm** e **mulplx** e pertanto termina senza generare il programma eseguibile.
Segnala queste routine come indefinite (undefined symbol) e ci informa che il primo riferimento (first referenced) è nel programma PCONT.

La prima cosa che dobbiamo fare è controllare il programma **PCONT.ASM** dove però le tre labels sono state correttamente definite **esterne** con la direttiva **.extern decrem,increm,mulplx**.

A questo punto controlliamo anche il programma **PLEXER.ASM**, ma anche qui **decrem**, **increm** e **mulplx** sono definite e usate correttamente.
Dovrebbe perciò essere tutto a posto, ma nonostante ciò il linker le segnala come indefinite.

L'errore deriva dal fatto che nel programma **PLEXER** non è stata inserita la direttiva **.global**.

```
C:\ >LST6 -S -I -T -V -M -O XCONTA PCONT PLEXER

ST6 Linkage Editor  version 3.40
Copyright (C)  SGS-THOMSON Microelectronics  May 1995

pass1:
<PCONT.obj>: program section(s) size is 0x7C (124), window(s) size is 0xA (10)
<PLEXER.obj>: program section(s) size is 0x49 (73), window(s) size is 0x0 (0)
lst6: ** undefined symbol <decrem> ; first referenced in file  <PCONT.obj>
lst6: ** undefined symbol <increm> ; first referenced in file  <PCONT.obj>
lst6: ** undefined symbol <mulplx> ; first referenced in file  <PCONT.obj>
lst6: <3> fatal error(s) occured
```

Fig.15 Il programma linker agisce in due fasi o passi. Nella prima fase controlla se nei programmi .OBJ da linkare ci sono i presupposti per il loro collegamento. In questo caso non passa alla seconda fase perché rileva delle anomalie sull'uso delle labels decrem, increm e mulplx segnalandoci che il loro primo riferimento si trova in PCONT.OBJ.

La direttiva .global

Questa direttiva è assolutamente ininfluente in fase di compilazione in formato **.OBJ** e la prova è data dal fatto che il compilatore non ha segnalato nessun errore assemblando **PLEXER.ASM**.

Quando però si devono linkare programmi che contengono la direttiva **.extern** per segnalare l'utilizzo di routine esterne, nel programma che effettivamente contiene queste routine bisogna inserire la direttiva **.global** seguita dalla definizione delle labels di queste routine.

In questo modo segnaliamo al linker che queste routine sono richiamate in altri programmi e, in un certo senso, le rendiamo "disponibili".
E' importante ricordare che **.global** deve essere **obbligatoriamente** inserita prima della definizione delle routine che vogliamo rendere utilizzabili in altri programmi.

Il listato visibile in fig.3 va perciò modificato inserendo nel programma **PLEXER.ASM**, prima di **.section 1** l'istruzione:

.global decrem,increm,mulplx

Ovviamente il programma va ricompilato per generare **PLEXER.OBJ** e poi va rilanciato il linker.

RILOCAZIONE degli INDIRIZZI

Nella fig.16 abbiamo riportato la videata che appare dopo aver lanciato per la seconda volta il linker.

Questa volta sotto **pass1** non vengono segnalati errori, ma appare: **window #0 (10 bytes) mapped in program page #1, at offset 0xb5**.

Questa scritta ci informa che, grazie alle direttive **.window** e **.windowend** inserite in **PCONT**, il linker ha rilocato (mapped) all'indirizzo **0B5h** di Program page 1 un'area dati di **10** bytes.

Il linker passa quindi alla seconda fase e ne dà il resoconto sotto la scritta:

pass2 :

Il collegamento vero e proprio di **PCONT** e **PLEXER** è stato effettuato e segnala che in **PCONT** ha rilevato l'utilizzo delle tre routine esterne e che in **PLEXER** ha rilevato le stesse routine definite con **.global** e ha assegnato loro un indirizzo assoluto di memoria Program Space:

decrem	**89Eh**
increm	**889h**
mulplx	**86Ch**

```
ST6 Linkage Editor  version 3.40
Copyright (C)  SGS-THOMSON Microelectronics  May 1995

pass1:
<PCONT.obj>: program section(s) size is 0x7C (124), window(s) size is 0xA (10)
<PLEXER.obj>: program section(s) size is 0x49 (73), window(s) size is 0x0 (0)
window #0 (10 bytes) mapped in program page #1, at offset 0xb5
pass2:
    <PCONT.obj>
        reference to <decrem> external
        reference to <increm> external
        reference to <mulplx> external
    <PLEXER.obj>
        definition of <decrem> global program  89E(2206)
        definition of <increm> global program  889(2185)
        definition of <mulplx> global program  86C(2156)
program section(s) size is 0xCF (207)
<XCONTA.hex>: hexadecimal image
<XCONTA.dsd>: dsd file
<XCONTA.sym>: namelist
```

Fig.16 Il linker dà un resoconto scritto anche della 2° fase, che consiste nel collegare ogni singola istruzione e locazione di memoria così da ottenere un eseguibile .HEX.

Inoltre segnala che la grandezza del programma eseguibile sarà di **0CFh bytes** di Program Space e cioè di 207 bytes a partire da program section 1, e cioè dall'indirizzo di memoria **0800h** (vedi la tabella in fig.5). Infine segnala che ha generato:

XCONTA.hex
XCONTA.dsd
XCONTA.sym

Nota: non ci soffermiamo sulle peculiarità dei programmi con estensione **.dsd** e **.sym** ai quali abbiamo dedicato l'articolo apparso sulla rivista **N.194**.

Questa volta il linkaggio è andato a buon fine quindi non ci resta che effettuare una semplice prova di simulazione per verificare se **XCONTA.HEX** funziona correttamente.

Se vi ricordate, in entrambi i programmi **PCONT** e **PLEXER** avevamo definito le variabili **lsb** e **msb**, ma in locazioni di memoria diverse.
Poiché il linker non ha segnalato nessuna anomalia, siamo un po' curiosi di vedere cosa succede nella simulazione.
Carichiamo perciò il software simulatore, il cui uso è stato spiegato nelle riviste **N.184** e **N.185**, ed eseguiamo la simulazione istruzione per istruzione fino ad arrivare al punto visibile in fig.17, dove in **giallo** sono evidenziate le istruzioni che nel programma **PCONT** riguardavano le variabili **lsb** e **msb**, cioè:

```
ldi   lsb,00h
ldi   msb,00h
```

```
Assembler
Ind. Codice  Label     Mnemonico
082B 4D      tim_int   reti
082C 4D      BC_int    reti
082D 4D      A_int     reti
082E 4D      nmi_int   reti
082F 0DD8FE  main      ldi      wdog,FEh
0832 0D8700            ldi      lsb,00h
0835 0D8800            ldi      msb,00h
0838 0D8601            ldi      up_dw,01h
083B 0DC922            ldi      drw,22h
083E 0D8411  loop      ldi      lsb,11h
0841 0D85FF  main1     ldi      msb,FFh
0844 0DD8FE  main2     ldi      wdog,FEh
0847 C186              call     mulplx
0849 FF85              dec      del2
084B 14                jrz      main3
```

Fig.17 In giallo sono evidenziate le istruzioni ldi delle variabile lsb e msb del programma PCONT; in verde altre istruzioni che non rispettano il listato di PCONT. Vi facciamo notare (vedi colonna opcode) che le locazioni di memoria sono differenti.

Confrontando il loro **opcode** (vedi colonna **codice** in fig.17) con il listato di fig.2, si può notare che sono corrette. L'operazione **ldi** infatti, avviene esattamente nelle due locazioni di memoria definite in **PCONT**, cioè **087h** e **088h**.

Sempre in fig.17 abbiamo evidenziato in **verde** altre due istruzioni, cioè:

```
loop    ldi   lsb,11h
main1   ldi   msb,FFh
```

che sono invece sbagliate. Infatti, verificando il listato di **PCONT** dovevano essere:

```
loop    ldi   del1,11h
main1   ldi   del2,FFh
```

Verificando il loro **opcode**, possiamo vedere che l'operazione di **ldi** avviene nelle locazioni **084h** e **085h**, che corrispondono alle locazioni di **lsb** e **msb** definite nel programma **PLEXER**.

Il simulatore che, come sapete benissimo, utilizza il file con estensione **.dsd** per assegnare le etichette delle variabili e dei registri e rendere così leggibile il programma, quando ha decodificato le due ultime **opcode**, ha visualizzato le labels corrispondenti agli indirizzi **084h** e **085h**, che in questo file corrispondono alle etichette **del1** e **del2** del programma **PCONT**.

In fig.18 riportiamo il contenuto del file **XCONTA.DSD**, dove potete vedere che **lsb** e **msb** sono infatti definite 2 volte e in locazioni di memoria diverse.
Inoltre, **del1** e **del2** hanno la stessa locazione di memoria della seconda "serie" di **lsb - msb**.
Se però guardate più attentamente, vedrete che anche tutti i registri, l'accumulatore **a**, le porte logiche sono definite due volte, anche se in questo caso nella stessa locazione di memoria.
Questo sta a significare che nonostante il linker non abbia segnalato errore, c'è un problema.

Per poter assemblare in formato **.OBJ** sia **PCONT** che **PLEXER**, abbiamo dovuto inserire in entrambi i programmi le definizioni standard dei registri, dell'accumulatore, delle porte logiche e delle etichette utilizzate, perché altrimenti il compilatore avrebbe segnalato errore.
Quando però il linker ha unito i due **.OBJ**, ha come "sdoppiato" questi campi, generando una evidente confusione.

Per impedire che questo si verifichi ci vengono in aiuto due direttive: **.notransmit** e **.transmit**.

```
C:\XCONTA.DSD
lsb 87 00 R W          adcr D1 00 R W         W 83 00 R W
msb 88 00 R W          addr D0 00 R W         X 80 00 R W
pdir_a C4 00 R W       wdog D8 00 R W         Y 81 00 R W
pdir_b C5 00 R W       tscr D4 00 R W         adcr D1 00 R W
pdir_c C6 00 R W       up_dw 86 00 R W        addr D0 00 R W
psc D2 00 R W          lsb 84 00 R W          wdog D8 00 R W
tcr D3 00 R W          msb 85 00 R W          tscr D4 00 R W
ior C8 00 R W          pdir_a C4 00 R W
drw C9 00 R W          pdir_b C5 00 R W
popt_a CC 00 R W       pdir_c C6 00 R W
popt_b CD 00 R W       psc D2 00 R W
popt_c CE 00 R W       tcr D3 00 R W
port_a C0 00 R W       ior C8 00 R W
port_b C1 00 R W       drw C9 00 R W
port_c C2 00 R W       popt_a CC 00 R W
del1 84 00 R W         popt_b CD 00 R W
del2 85 00 R W         popt_c CE 00 R W
A FF 00 R W            port_a C0 00 R W
V 82 00 R W            port_b C1 00 R W
W 83 00 R W            port_c C2 00 R W
X 80 00 R W            A FF 00 R W
Y 81 00 R W            V 82 00 R W
```

Fig.18 Il programma XCONTA.DSD riferito alle variabili lsb e msb definite due volte in due differenti locazioni di memoria. Come potete notare, le istruzioni del1 e del2 hanno le stesse locazioni di memoria della seconda serie di variabili lsb e msb. Anche le definizioni dei registri, dell'accumulatore, delle etichette ecc., sono state sdoppiate provocando confusione. Per ovviare a ciò si utilizzano le direttive .notransmit e .transmit.

```
        .title  "PLEXER"
        .vers   "ST62E25"
        .w_on
        .romsize 4
        .pp_on

        .notransmit
        .input  "ST62X.DEF"

;VARIABILI usate da questo PROGRAMMA

lsb     .def    084h
msb     .def    085h
        .transmit

        .global decrem,increm,mulplx
        .section 1
```

Fig.19 Parte del listato del programma PLEXER.ASM in cui abbiamo inserito le direttive .notransmit e .transmit.

Le direttive .notransmit e .transmit

Come abbiamo già visto per la direttiva **.global**, anche le direttive **.notransmit** e **.transmit** non sono strettamente necessarie nella fase di compilazione in formato **.OBJ**, ma vanno assolutamente inserite quando i programmi da linkare contengono le definizioni delle stesse variabili, degli stessi registri, delle stesse etichette ecc.

In questi casi è sufficiente che in uno dei programmi venga inserita **.notransmit** prima delle definizioni delle variabili comuni, e **.transmit** immediatamente dopo l'ultima variabile comune.

In questo modo il **linker** utilizza le variabili, i registri ecc. di un solo programma e collega tutte le istruzioni relative a queste locazioni.

Nel nostro caso, abbiamo inserito le direttive nel programma **PLEXER.ASM** come riportato in fig.19, e poi abbiamo ricompilato il programma per avere **PLEXER.OBJ** e abbiamo rilanciato il linker.
In fig.20 riportiamo il file **XCONTA.DSD** corretto.

```
lsb 87 00 R W
msb 88 00 R W
pdir_a C4 00 R W
pdir_b C5 00 R W
pdir_c C6 00 R W
psc D2 00 R W
tcr D3 00 R W
ior C8 00 R W
drw C9 00 R W
popt_a CC 00 R W
popt_b CD 00 R W
popt_c CE 00 R W
port_a C0 00 R W
port_b C1 00 R W
port_c C2 00 R W
del1 84 00 R W
del2 85 00 R W
A FF 00 R W
V 82 00 R W
W 83 00 R W
X 80 00 R W
Y 81 00 R W
adcr D1 00 R W
addr D0 00 R W
wdog D8 00 R W
tscr D4 00 R W
up_dw 86 00 R W
```

Fig.20 Il file XCONTA.DSD ottenuto dopo aver inserito correttamente le direttive .notransmit e .transmit.

ULTIME CONSIDERAZIONI

Nell'esempio che vi abbiamo illustrato, abbiamo linkato due soli programmi, quindi è stato relativamente facile ricordare come scrivere le giuste istruzioni e la giusta sequenza per il linker.

Quando però i programmi diventano tanti e tante sono le routine da utilizzare, potrebbe risultare difficile gestire i programmi senza commettere nessun errore.
Per questo motivo vi suggeriamo un semplice metodo, utilizzato da molti programmatori, che vi consente di avere a disposizione anche il listato del programma eseguibile che si ottiene con il linker.
In questo modo potrete facilmente controllare l'opcode delle istruzioni e il loro indirizzamento nella memoria.

Prendiamo ancora una volta ad esempio i files **PCONT** e **PLEXER**.

Apriamo un editor qualsiasi e digitiamo:

ast6 –L –O PCONT
ast6 –L –O PLEXER

lst6 –S –I –T –V –M –O XCONTA PCONT PLEXER

quindi salviamo il file chiamandolo **XCONTA.BAT**.

A questo punto, ogni volta che dovremo compilare o linkare i due programmi, sarà sufficiente scrivere al prompt di DOS:

XCONTA

e automaticamente verranno lanciate in cascata prima le due compilazioni in formato **.OBJ** e poi il linker **lst6**.

Poiché nei comandi Assembler prima dell'opzione **–O** abbiamo inserito l'opzione **–L**, che genera anche il formato **.LIS** dei programmi, quando viene lanciato il linker, oltre a essere generato il programma eseguibile, nei files **.LIS** vengono sostituiti gli indirizzamenti **relativi** con gli indirizzamenti **assoluti** del programma finale.

Avremo così a disposizione anche il listato definitivo di **XCONTA**, che potremo leggere in **PCONT.LIS** e **PLEXER.LIS**.

Sebbene questa parte vi possa essere sembrata alquanto complicata, non dovete sottovalutare il fatto che ottenere dei programmi collegando tra loro programmi già esistenti è una **pratica comune** ad altri linguaggi di programmazione.

Pertanto coloro che intendessero approfondire anche lo studio di **altri linguaggi** software, non potranno che trarre **vantaggio** dalla lettura degli articoli dedicati al **linker** per i microprocessori **ST6**.

I PROGRAMMI LST6 E AST6

Informiamo tutti i nostri lettori che è possibile scaricare il programma **linker lst6** unitamente alla **versione 4.50** dell'**Assembler ast6** dal nostro sito:

WWW.NUOVAELETTRONICA.IT

Entrambi i programmi sono **gratuiti**.

QUALCOSA in più sul TIMER

La funzione **timer** dei microprocessori ST6 e il suo corretto utilizzo nella programmazione con linguaggio Assembler, è uno degli argomenti che abbiamo trattato fin dalle nostre prime lezioni.

In quegli esempi però, non si è tenuto volutamente conto del fatto che alcuni tipi di micro, oltre i tre normali registri del timer, hanno un **piedino**, grazie al quale è possibile attivare alcune modalità di funzionamento particolari e molto interessanti.

A seconda del modello di microprocessore, questo piedino può avere una differente numerazione e può essere indicato con la scritta **TIM1** (vedi fig.1) oppure con la scritta **TIMER** (vedi fig.2).

Abbiamo quindi ritenuto opportuno scrivere una piccola appendice per spiegarvi brevemente i casi in cui va utilizzato questo piedino, che per comodità chiameremo semplicemente **TIMX** (vedi fig.3).

Con le sigle **TSCR**, **TCR** e **PSC** abbiamo invece chiamato i tre registri utilizzati dal timer.

In fig.3 è riportato lo schema a blocchi interno che fa capo al timer del microprocessore **ST6**.

Passiamo ora a spiegare a grandi linee come funziona il **timer** dei **micro ST6**.

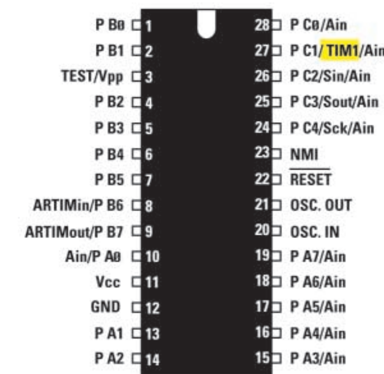
Il timer non è nient'altro che un **contatore** (**TCR**) che si **decrementa** di **1** ogni volta che un altro contatore, chiamato **prescaler** (**PSC**), decrementandosi a sua volta ad ogni impulso generato da un clock interno o esterno, arriva a **zero**.
Quando il **tcr** arriva a **zero**, setta alcuni valori all'interno del registro di controllo **TSCR**.

Ricaricando il contatore **TCR** e ripristinando il registro **TSCR**, è possibile far ripartire il timer tutte le volte necessarie al suo utilizzo.

E' possibile inoltre impostare un **divisore** sul **prescaler**, di modo che questo contatore si decrementi solamente ogni **1**, **2**, **4**, **8**, **16**, **32**, **64** o **128** impulsi di clock (interni o esterni).
Ma di questo abbiamo già parlato abbondantemente e spiegato con numerosi esempi negli articoli dedicati ai microprocessori ST6.

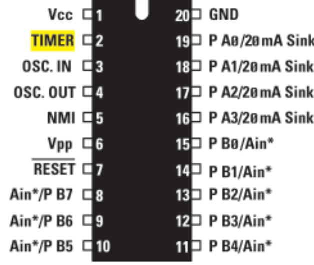
In questo articolo, parleremo solamente del **registro** di **controllo** (vedi fig.4) e cioè del:

TSCR Timer Status Control Register



ST 62T65C-ST 62E65C

Fig.1 I microprocessori serie ST62T65C e ST62E65C hanno un piedino dedicato all'attivazione del timer. Questo piedino è il 27 e viene solitamente chiamato TIM1.



ST 6210C-ST 6220C

Fig.2 I microprocessori serie ST6210C e ST6220C hanno un piedino dedicato all'attivazione del timer. Questo piedino è il 2 e viene solitamente chiamato TIMER.

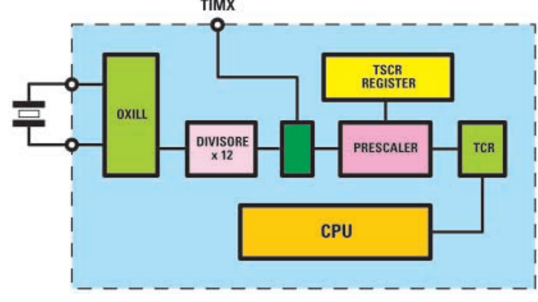


Fig.3 Al piedino del timer, che per comodità abbiamo chiamato nei nostri esempi TIMX, fanno capo tre registri chiamati TSCR - PRESCALER - TCR.

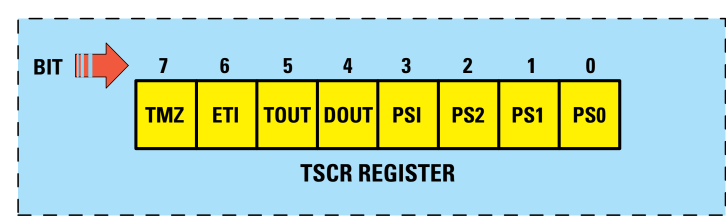


Fig.4 Il registro TSCR, abbreviazione di Timer Status Control Register, consente il controllo del Timer. In figura potete vedere il suo formato.

Con questo registro si effettua il controllo completo del timer. Il formato del registro è:

7	6	5	4	3	2	1	0
TMZ	ETI	TOUT	DOUT	PSI	PS2	PS1	PS0

Abbiamo contraddistinto i singoli bit con delle sigle e ora, bit per bit, ne diamo il significato, ma soprattutto vediamo come vanno gestiti questi bit nei nostri programmi per ottenere le diverse funzioni.

– bit 7 TMZ Timer Zero Bit

Questo bit viene gestito in maniera automatica dal timer, ma può essere anche settato o resettato da programma. Quando il registro contatore del timer, cioè il registro **tcr**, decrementandosi raggiunge lo **zero**, questo bit viene posto a **1**.
Per questo motivo, ogni volta che bisogna iniziare un nuovo conteggio del timer, questo bit va resettato a **0**, altrimenti il timer non riparte.

– bit 6 ETI Enable Timer Interrupt

Questo bit viene gestito attraverso il programma. Quando è posto a **0**, l'interrupt del timer **non** viene **abilitato**, quando invece è posto a **1**, la richiesta di interrupt del timer è **abilitata**. In questo caso, quando **TMZ**, cioè il bit **7**, diventa **1**, il programma salta al vettore relativo (definito nei nostri programmi-esempio con **tim_int** o **tad_int**).
Naturalmente dobbiamo avere correttamente abilitato il registro **ior** all'inizio del nostro programma.

I due bit che interessano direttamente il piedino **TIMX** del timer sono:

– bit 5 TOUT Timers Output Control
– bit 4 DOUT Data Output

Entrambi questi bit vengono gestiti dal programma e consentono di selezionare una delle **quattro** differenti **modalità** del timer.

Nota: in realtà le modalità sono **tre**, perché per le funzioni **Output Mode**, la modalità è la stessa e cambia solo il tipo di segnale in uscita, che può essere **0** o **1**.

Nella tabella successiva riportiamo le combinazioni possibili e le relative funzioni attivate.

TOUT	DOUT	TIMER PIN	MODALITÀ
0	0	Input	Event Counter
0	1	Input	Gated Mode
1	0	Output	Output Mode "0"
1	1	Output	Output Mode "1"

Vediamo ora insieme i singoli casi.

– Event Counter: TOUT = 0 DOUT = 0

In questa modalità il piedino **TIMX** del micro viene configurato in **input** e diventa l'input clock del registro prescaler **psc**.
Il **decremento** del registro prescaler **psc** avviene solo quando viene rilevato un **fronte** di **salita** (rising edge) su questo piedino.
Questa modalità viene ad esempio utilizzata per ottenere un **conta impulsi** o un **generico contatore** (vedi più avanti il primo programma-esempio).

– Gated Mode: TOUT = 0 DOUT = 1

In questa modalità il piedino **TIMX** viene configurato in **input** e il prescaler **psc** si decrementa sul clock del timer, cioè con la **frequenza interna divisa** per **12**.
Questo avviene solamente quando sul piedino **TIMX** viene rilevato uno stato logico "**high**", cioè **1**. Se lo stato è "**low**", cioè **0**, il registro prescaler **psc** non si decrementa mai ed il timer è bloccato fino al prossimo cambio di stato del piedino **TIMX.**
Questa modalità viene ad esempio utilizzata per ottenere un semplice **periodimetro** o per **misurare** la **durata** di un **impulso** (vedi più avanti il secondo peogramma-esempio).

– Output Mode "0": TOUT = 1 DOUT = 0

In questa modalità il piedino **TIMX** viene configurato in **output** e il prescaler **psc** si decrementa sul clock del timer, cioè con la **frequenza interna divisa** per **12**.
Quando il bit **7 TMZ** diventa **1**, sul piedino **TIMX** viene riportato il valore di **DOUT** e cioè **0**.

Questa modalità viene ad esempio utilizzata per ottenere un **generatore** di **frequenza**.

– Output Mode "1": TOUT = 1 DOUT = 1

Anche in questa modalità il piedino **TIMX** viene configurato in **output** e il prescaler **psc** si decrementa sul clock del timer, cioè con la **frequenza interna divisa** per **12**.
Quando il bit **7 TMZ** diventa **1**, sul piedino **TIMX** viene riportato il valore di **DOUT** e cioè **1**.
Come la precedente, anche questa modalità viene utilizzata per ottenere un **generatore** di **frequenza** (vedi più avanti il terzo programma-esempio).

– bit 3 PSI Prescaler Initialize Bit

Questo bit viene gestito attraverso il programma.
Quando è posto a **0**, il registro prescaler **psc** viene inizializzato al valore **7Fh** (in binario 01111111) e il suo conteggio viene bloccato.
In questa condizione quindi il timer non funziona.
Quando è posto a **1**, il conteggio (decremento) del prescaler è attivo ed il timer funziona.

– bit 2-1-0 PS2-1-0 Presc. Div. Factor

Questi bit vengono gestiti attraverso il programma.
La combinazione di questi tre bit permette di configurare il **fattore** di **divisione** del registro prescaler **psc** come si vede nella tabella proposta di seguito.

Grazie all'utilizzo corretto di questi tre bit, è possibile ottenere una vasta gamma di **temporizzazioni**.

PS2	PS1	PS0	DIVISORE
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Alla luce di quanto fin qui detto, è opportuno fare alcune piccole e semplici constatazioni.
Quando si intende utilizzare il **timer**, è evidente che se il micro ST6 scelto dispone del piedino **TIM1** o **TIMER**, questo diventa praticamente riservato e non deve essere assolutamente utilizzato per scopi diversi da quelli elencati sopra, come, ad esempio, per gestire un pulsante o spedire dati ad un display, perché il programma ed il relativo circuito potrebbero non funzionare correttamente.

Inoltre, occorre fare attenzione all'**Option Byte** dei **micro** della versione **C**.
E' infatti possibile che, in alcuni modelli, sia inserito un **bit** che permette di scegliere tra la modalità **Pull-Up** o la modalità **No Pull-Up** del piedino **TIMER**. Questo fatto potrebbe influenzare la gestione del timer.

Infatti, se, ad esempio, in un programma che utilizza un micro modello **ST6220C** e che prevede l'utilizzo del timer in **Gated Mode**, **non** settiamo a **1** il bit **D2** dell'**Option Byte** (vedi fig.5), bisognerà ricordarsi di farlo elettricamente collegando tra il pie-

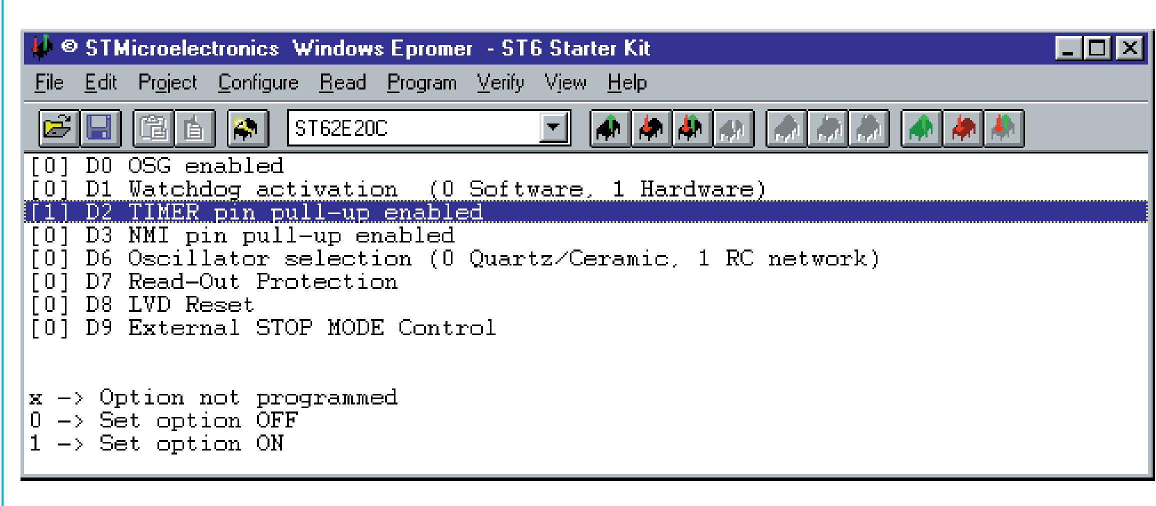


Fig.5 Quando utilizzate i micro della serie C che hanno il piedino del TIMER (vedi Tim1 e Timer nelle figg.1-2), ricordatevi di controllare se è possibile scegliere tra la modalità pull-up o no pull-up. Se così fosse, settate a 1 il piedino D2 dell'Option Byte oppure ricordatevi di collegare una resistenza esterna di pull-up da 10.000 ohm al piedino Timer.

dino **TIMX** e il positivo di alimentazione una resistenza esterna di pull-up da **10.000 ohm**, altrimenti il **timer** non **partirà mai**.
Non crediate di poterlo fare tramite il programma, perché, come abbiamo già detto, in questi tipi di micro il piedino **TIMER** è **riservato** e quindi non è configurabile tramite i tre normali registri delle porte (come, port_a, pdir_a e popt_a).

ESEMPI

Per spiegare quanto fin qui detto, abbiamo scritto tre semplici e completi programmi-esempio funzionanti:

containp.asm - durata.asm - ondaqua.asm

Di seguito spieghiamo con maggiori dettagli le sole istruzioni relative alla gestione del timer, ma voi trovate i sorgenti completi in questo cd-rom.

Per facilitare il riconoscimento delle istruzioni all'interno del programma, abbiamo inoltre provveduto a numerarle.

IL PROGRAMMA CONTAIMP.ASM

Per esemplificare la modalità "**Event Counter**" abbiamo scritto il programma **CONTAIMP.ASM**.

In questo programma viene utilizzato il piedino **TIMX** di un microcontrollore **ST6265** per "contare" gli impulsi. E' previsto il conteggio di un massimo di **9999 impulsi**, dopodiché si riparte.
E' inoltre prevista la visualizzazione in tempo reale di questo conteggio tramite la gestione di **4 display** numerici a **7 segmenti**.

Aprite con un normale editor, ad esempio Notepad, questo programma e posizionatevi alla riga da noi numerata **279**, dove trovate queste istruzioni:

```
ldi    tcr,1
ldi    tscr,01001000b
```

Con la prima istruzione abbiamo caricato il valore **1** nel contatore **tcr**.
Con la seconda istruzione carichiamo il valore binario **01001000** nel registro **tscr**.
Se adoperate la tabellina che abbiamo utilizzato ad inizio articolo, potete capire meglio perché abbiamo caricato questo numero nel registro **tscr**.

7	6	5	4	3	2	1	0
TMZ	ETI	TOUT	DOUT	PSI	PS2	PS1	PS0
0	1	0	0	1	0	0	0

I bit **5** e **4** denominati **TOUT** e **DOUT** sono a zero per attivare la modalità "**EVENT COUNTER**".

Il bit **6** denominato **ETI** è a **1** perché vogliamo gestire l'**interrupt** del **timer** quando il contatore **tcr** arriva a zero.

I bit **2-1-0** denominati **PS2-PS1-PS0** sono tutti a zero perché vogliamo che il divisore del **prescaler** sia uguale a **1**.
In questo modo, ogni fronte di salita rilevato sul piedino **TIMX**, va a decrementare subito il contatore del timer **tcr**.

Poiché noi abbiamo caricato il contatore **tcr** con il valore **1**, al primo impulso rilevato, il **tcr** va a zero e attiva l'interrupt (vedi **tad_int**).

Il bit **7** denominato **TMZ** viene posto a **0**, mentre il bit **3** denominato **PSI** viene posto a **1** e in questo modo il timer si attiva.

A questo punto posizionatevi alla riga da noi numerata **104**, dove trovate la routine:

tad_int

per la gestione dell'**interrupt** del timer.

Come abbiamo spiegato quando abbiamo parlato della modalità **Event Counter**, il programma attiva la routine **tad_int** quando il contatore **tcr** è a **zero**, cioè ad ogni impulso (fronte di salita) rilevato su **TIMX** e, tramite i due contatori che nel nostro programma abbiamo chiamato **unidec** e **cenmil** (vedi le istruzioni posizionate alle righe **48** e **49**), effettua un conteggio da **0** a **9999** impulsi visualizzandone il risultato su **4 display** numerici a **7 segmenti**.
Quando il conteggio degli impulsi arriva a **9999**, i contatori vengono azzerati e il conteggio riparte.

IL PROGRAMMA DURATA.ASM

Per esemplificare la modalità "**Gated Mode**" abbiamo scritto il programma **DURATA.ASM.**

In questo programma viene utilizzato il piedino **TIMX** di un microcontrollore **ST6265** per calcolare il tempo totale di accensione di una caldaia oppure di un innaffiatoio o anche di una pompa ecc., nell'arco delle **24 ore**. E' inoltre prevista la sua visualizzazione su **4 display** a **7 segmenti**.

Il tempo è calcolato in **secondi-minuti-ore** e sul display è possibile visualizzare a scelta i minuti ed i secondi oppure le ore ed i minuti, se si modificano alcune righe del programma sorgente.

Quando la caldaia (o l'innaffiatoio ecc.) si spegne, mette a **0** lo stato logico del piedino **TIMX** e il conteggio del tempo si ferma, quando si riaccende met-

te a **1** lo stato logico di **TIMX** e il conteggio riparte. Il tempo massimo del conteggio è di 24 ore.

Aprite ora con un editor, ad esempio Notepad, questo programma e posizionatevi alla riga da noi numerata **79**, dove trovate le istruzioni:

```
ldi    tcr,16
ldi    tscr,01011111b
```

Anche in questo caso adoperate la tabellina che abbiamo utilizzato ad inizio articolo, per capire meglio perché abbiamo caricato questo numero nel registro **tscr**.

7	6	5	4	3	2	1	0
TMZ	ETI	TOUT	DOUT	PSI	PS2	PS1	PS0
0	1	0	1	1	1	1	1

Notate anzitutto che il bit **5** denominato **TOUT** è a **0**, mentre il bit **4** denominato **DOUT** è a **1**.

Questa "combinazione" serve per attivare la modalità "**GATED MODE**".

I bit **2-1-0** denominati **PS2 PS1 PS0** sono tutti a **1**, quindi il divisore del prescaler è settato a **128**.
Avendo utilizzato un quarzo da **2,4576 MHz** e avendo caricato il contatore **tcr** con il valore **16** e settato il prescaler a **128**, abbiamo ottenuto una base tempi di **10 millisecondi**.

Il bit **6** denominato **ETI** è settato a **1** per attivare l'interrupt e quindi, ogni **10 millisecondi**, il programma attiva la routine **tad_int** (vedi istruzione alla riga **104**), dove gestisce sia il calcolo del tempo di utilizzo che la sua visualizzazione sui display.

Nota: nel nostro programma le istruzioni sono scritte in modo da visualizzare i **secondi** e i **minuti** di utilizzo. Infatti, alla riga **139** trovate l'istruzione:

```
ld    a,secondi
```

mentre alla riga **145** trovate l'istruzione:

```
ld    a,minuti
```

Se invece dei minuti e dei secondi, volete visualizzare i **minuti** e le **ore** dovete cambiare la riga **139** con l'istruzione:

```
ld    a,minuti
```

e la riga **145** con l'istruzione:

```
ld    a,ore
```

IL PROGRAMMA ONDAQUA.ASM

Per esemplificare la modalità "**Output Mode**" abbiamo scritto il programma **ONDAQUA.ASM**.

Con questo programma generiamo un'onda quadra della frequenza di **100 Hz** sul piedino **TIMX**.

Anche in questo caso abbiamo previsto di utilizzare un quarzo esterno che oscilli alla frequenza di **2,4576 MHz** e abbiamo opportunamente settato il contatore **tcr** e il registro di configurazione **tscr**.

Aprite dunque con un editor questo programma e posizionatevi alla riga da noi numerata **127** dove trovate le istruzioni:

```
ldi    tcr,16
ldi    tscr,01111110b
```

Adoperate anche in questo caso la tabellina che abbiamo utilizzato ad inizio articolo, per capire meglio perché abbiamo caricato questo numero nel registro **tscr**.

7	6	5	4	3	2	1	0
TMZ	ETI	TOUT	DOUT	PSI	PS2	PS1	PS0
0	1	1	1	1	1	1	0

Il bit **5** denominato **TOUT** è settato a **1** per attivare la modalità "**Output Mode**".

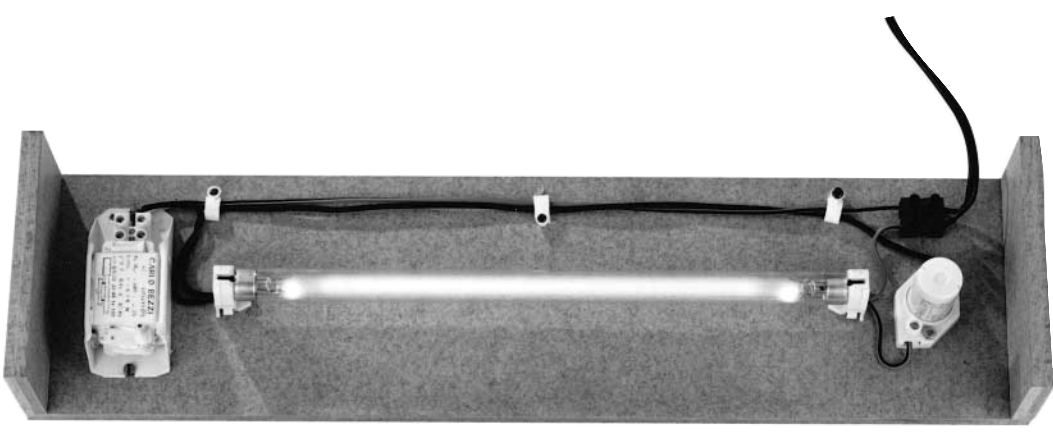
I bit **2-1-0** denominati **PS2 PS1 PS0** sono settati a **1-1-0** in modo da configurare a **64** il divisore del prescaler.

Con il quarzo a **2,4576 MHz**, il contatore **tcr** caricato con **16** e il divisore del prescaler a **64**, abbiamo ottenuto una base tempi di **5 millisecondi**.

Settando a **1** il bit **6** denominato **ETI** abbiamo attivato la richiesta di **interrupt** del **timer** e quindi ogni **5 millisecondi** il programma salterà alla routine **tad_int** visibile dalla riga **138** alla **148**.

Il bit **4** denominato **DOUT** è stato configurato a **1**. Questo significa che la prima volta che si attiva l'interrupt, nel piedino **TIMX**, che fino ad allora conteneva uno stato logico **0** (in virtù delle istruzioni che si trovano dalla riga **119** alla riga **121**), verrà "mosso" il valore del bit **DOUT** e cioè **1**.

Nelle attivazioni successive dell'interrupt del timer (vedi **tad_int** dalla riga **138** alla riga **148**), il bit **DOUT** verrà configurato alternativamente a **0** e poi a **1** e così via, in modo da generare una forma di onda quadra a **100 Hz**.



LAMPADA per CANCELLARE EPROM

Recatevi presso un qualsiasi negozio di materiale elettrico e provate a richiedere delle lampade ultraviolette che lavorino sui **2.300-2.700 Angstrom**, e come è accaduto a noi, vi verranno proposte delle comuni lampade Argon o delle lampade per abbronzarsi, che sono inadatte ai nostri scopi perché non emettono raggi ultravioletti sulla lunghezza d'onda richiesta.
Dopo aver spiegato che le lampade ci servivano per **cancellare** delle **EPROM**, ci è stato suggerito di rivolgerci presso i negozi di **computer**, dove abbiamo fatto un'amara scoperta.
La maggior parte di queste rivendite è **sprovvista** di queste lampade o, nel migliore dei casi, i pochi modelli di cui dispongono hanno prezzi esagerati.
Constatato che l'articolo sui **microprocessori** della serie **ST62** (vedi rivista **N.172/173**) ha incontrato un successo superiore alle nostre attese, tutti i lettori che hanno realizzato il **programmatore LX.1170** ci hanno tempestato di richieste chiedendoci una **lampada** per cancellare gli **ST62E**.
In passato avevamo disponibile una **lampada** per **cancellare** le memorie Eprom dei microprocessori, che si poteva direttamente collegare ai **220 volt** della rete, ma poiché è andata fuori produzione e quindi risulta anche per noi introvabile, ci siamo dati da fare per cercarne un'altra che la potesse sostituire.
La lampada che abbiamo trovato è di forma cilindrica, ha un diametro di **1,5 cm** e misura in lunghezza **30 cm**; per funzionare ha bisogno di due **zoccoli**, di un **reattore** e di uno **starter**, e poiché anche questi componenti non sono facilmente reperibili, abbiamo pensato di presentarla in Kit ai nostri lettori.
Inizialmente avevamo contattato qualche Industria per corredarla di un appropriato mobile, ma dopo aver conosciuto il prezzo per realizzarlo, abbiamo abbandonato questa idea, perché con solo quattro pezzi di legno ed un po' d'iniziativa, chiunque potrà realizzare un mobile adatto, risparmiando notevolmente.
Noi, per realizzare il contenitore che vedete nelle foto, abbiamo chiesto ad un falegname nostro amico due tavolette delle dimensioni di **47 x 12 cm**, che abbiamo adoperato come piano base e piano superiore, e due piccoli righelli alti **5,5 cm**, che abbiamo inchiodato ed incollato ai due lati delle tavolette.
A chi non piace tenere il contenitore grezzo, basterà che si procuri un pennello ed un poco di vernice **opaca** per legno, oppure una bomboletta di vernice spray, e con una spesa irrisoria avrà un elegante **cancella - Eprom**.
Con delle viti per legno abbiamo fissato sul piano superiore i due **zoccoli** per la lampada, lo **zoccolo** per lo **starter** ed il **reattore**.
Con del filo isolato di plastica abbiamo eseguito un semplice **impianto elettrico** (vedi fig.1) e per fissare le estremità del cordone di rete per i **220 volt** abbiamo usato due mammuth.
Sapendo che questa lampada può cancellare qualsiasi Eprom in circa **17-20 minuti**, le abbiamo collegato, dopo averlo programmato sui **20 minuti**, il **timer LX.1181**, che trovate pubblicato sulla rivista **N.174**.
Considerando la lunghezza della lampada, potrete cancellare contemporaneamente circa **10-12** memorie.
Collocate i microprocessori che volete cancellare sopra un cartoncino o sopra un sottile ritaglio di lamierino o compensato, quindi infilateli sotto la lampada in modo che la loro **finestra** risulti distante all'incirca **2 cm** dal bulbo in vetro.
Come abbiamo già avuto modo di accennare in precedenti articoli, prima di inserire il microprocessore sotto la lampada assicuratevi che la sua **finestra** sia pulita. Se così non fosse, sgrassatela con un batuffolo di cotone imbevuto di alcol o di acetone.

Non è conveniente fissare per molti minuti la luce **viola** emessa dalla lampada **accesa**, perché irritante per gli **occhi**.
Sul numero **172/173** l'articolista ha scritto che questa luce nuoce gravemente agli occhi, ma ha un poco esagerato, forse perché non avendola mai personalmente utilizzata, si è lasciato influenzare dalle affermazioni del tecnico.
In realtà l'occhio non viene **danneggiato**, come si potrebbe erroneamente dedurre da quanto scritto, ma si **irrita** provocando una condizione analoga (anche se molto inferiore) a quella prodotta quando si guarda l'**arco** di una saldatura elettrica senza la protezione degli occhiali.
Quindi se guardate per **5-6 minuti** di seguito i raggi ultravioletti emessi dalla lampada, vi sembrerà di avere negli occhi dei fastidiosi ed irritanti **granelli di sabbia**.
Per far scomparire il dolore, basterà fare qualche impacco con una soluzione di **acido borico**.

Per premunirvi contro questo fastidioso inconveniente, coprite, per il tempo che la lampada rimane accesa, la parte frontale del contenitore con un pezzo di cartoncino o con un panno, in modo da impedire alla luce di raggiungere gli occhi.

COSTO DI REALIZZAZIONE LX.1183

Il kit LX.1183 composto da una lampada lunga 30 cm della potenza di 8 Watt, completa di Reattore, Starter Mammuth, Zoccoli mignon e di un cordone di rete completo di spina .... € 25,30

NOTA: Questa è la lampada che utilizziamo nel nostro laboratorio per cancellare le nostre Eprom e Microprocessori, perché con altre di potenza minore non sempre riuscivamo a cancellarle.

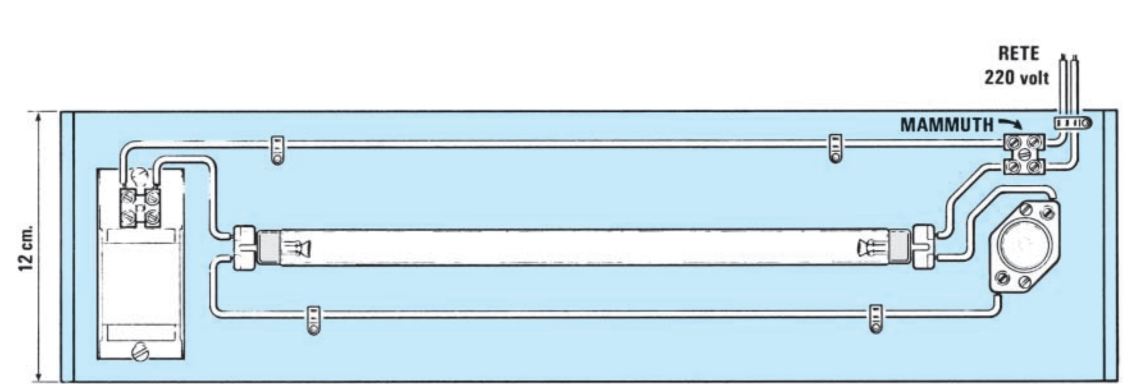


Fig.1 Su una piccola tavoletta in legno delle dimensioni di 12 x 47 cm abbiamo fissato a sinistra il reattore e a destra lo starter. Assieme alla lampada forniamo anche i due portalampada miniatura che fisserete con viti in legno sulla tavoletta alla distanza richiesta. Nel disegno si nota chiaramente come dovete effettuare il semplice schema elettrico.

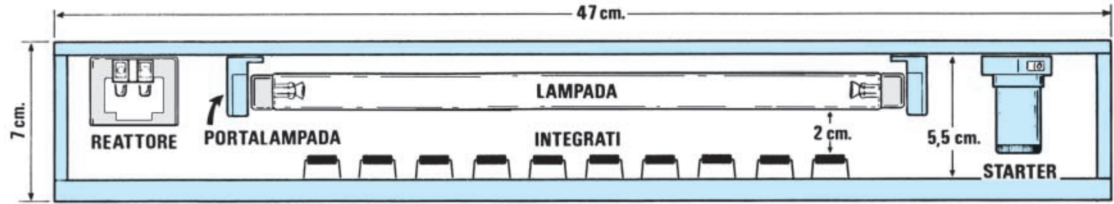


Fig.2 Sui due lati della tavoletta in legno abbiamo applicato due righelli alti circa 5,5 cm in modo da ottenere una distanza tra il bulbo della lampada ed il corpo superiore degli integrati (Eprom o Microprocessori cancellabili) che non risulti minore di 2 cm. Questa distanza non è critica, ma se superate i 4 cm dovrete aumentare i tempi d'esposizione.

INDICE ANALITICO TEORIA

A

A/D converter

C

Carry flag
Cicli macchina

D

Direttiva .ascii
Direttiva .asciz
Direttiva .block
Direttiva .byte
Direttiva .def
Direttiva .equ
Direttiva .extern
Direttiva .ifc
Direttiva .ifc - approfondimenti
Direttiva .macro
Direttiva .pp_on
Direttiva .set
Direttiva .w_on
Direttiva .window
Direttiva .windowend

E

Espressioni

F

Formato .hex
Formato .obj
Formato delle istruzioni
Funzione SPI

I

Interrupt
Istruzioni

L

Linker

M

Memoria EEprom
Memoria RAM e RAM aggiuntiva

O

Opcode delle istruzioni
Option Byte della serie C
Option Byte della serie C - funzioni attivabili
Opzioni del compilatore Assembler

P

Porte
Porte - gestione ottimale

R

Registri
Reset

T

Timer
Timer - registri
Tipi di registri

V

Variabili

W

Watchdog
Watchdog - approfondimenti

Z

Z flag

INDICE ANALITICO KIT

LX.1170 Programmatore per gli ST62/10-15-20-25

LX.1170 Consigli per migliorare il programmatore LX.1170

LX.1171 Scheda test per provare gli ST6

LX.1183 Lampada per cancellare le Eprom

LX.1202 Bus per testare i micro ST62/10-15-20-25

LX.1203 Stadio di alimentazione per BUS LX.1202

LX.1204 Scheda test con Display per provare gli ST6

LX.1205 Scheda test con Relè per provare gli ST6

LX.1206 Scheda per pilotare 4 diodi triac con un ST6

LX.1207 Scheda per pilotare un display numerico LCD con un ST6

LX.1208/N Scheda per pilotare un display alfanumerico LCD con un ST6

LX.1325 Programmatore per micro ST62/60-65

LX.1329 Bus per testare i micro ST62/60-65

LX.1380 Scheda test per la funzione SPI

LX.1381 Scheda test per la funzione SPI

LX.1382 Scheda test per la funzione SPI

LX.1430 Interfaccia per LX.1170 per gli ST6 serie C

INDICE ANALITICO SOFTWARE

Software simulatore DSE622 – sigla **DF.622**

Correzione degli errori con il simulatore DSE622

Se i programmi in DOS per ST6 non girano sotto Windows 95

Nuovo software simulatore per micro ST62/10-15-20-25 – sigla **ST622.1** e **ST622.2**

Nuovo software simulatore per micro ST62/60-65 – sigla **ST626.1** e **ST626.2**

Programma per LX.1170 – sigla **DF.1170.3**

Programma per LX.1325 e test per ST62/60 – sigla **DF.1325**

Programma per LX.1430 e per programmare gli ST6/C – sigla **DF.ST6-C**

Programmi per il TIMER

Programmi per la SPI – sigla **DF.1380**

Programmi test per LX.1202 – sigla **DF.1202.3**

Programmi test per LX.1207 – sigla **DF.1207.3**

Programmi test per LX.1208/N – sigla **DF.1208.3**