S D C C

Small Devices C Compiler

Manuale d'Uso

Scritto da: Sandeep Dutta rel. 16/07/2001

Tradotto da: Andrea Contrucci rel 07/09/2002

INDICE

1 INTRODUZIONE

1.1 Note su SDCC

SDCC è un compilatore ANSI-C freeware, riconfigurabile, ottimizzato, scritto da Sandeep Dutta per i microprocessori a 8 bit. La versione attuale è configurata per i microprocessori MCS51 (8051, 8052, e derivati), le varianti Dallas DS80C390, i microprocessori basati sullo Zilog Z80.

Può essere riconfigurato per altri micro, e il supporto per i PIC, gli AVR e i 186 è in fase di sviluppo.

L'intero codice sorgente del compilatore è distribuito secondo GPL. SDCC utilizza ASXXXX & ASLINK, un assembler & linker freeware, riconfigurabili.

SDCC ha molte estensioni di linguaggio utilizzate per vari microprocessori, adattandosi effettivamente all'hardware.

Oltre le ottimizzazioni specifiche del micro da usare, SDCC effettua una serie di ottimizzazioni standard, quali:

- eliminazione delle sotto espressioni globali
- ottimizzazione dei loop
- propagazione e FOLDING delle costanti
- propagazione della copia
- eliminazione del codice "dead"
- tabelle di salto per istruzione Switch

SDCC usa uno schema di allocazione dei registri globali che dovrebbe essere facilmente implementabile per altre MCU a 8 bit.

L'ottimizzatore peep hole usa un meccanismo di sostituzione basato su regole che è indipendente dal processore in uso.

I tipi di dato supportati sono:

- char (8 bits, 1 byte),

- short e int (16 bits, 2 bytes),

- long (32 bit, 4 bytes)

- float (4 byte IEEE).

Il compilatore consente anche l'implementazione di codice assembler ovunque in una funzione. Inoltre, possono essere chiamate routine direttamente scritte in assembler.

SDCC prevede un'opzione (*--cyclomatic*) per riportare la complessità relativa di una funzione. Queste funzioni possono poi essere ulteriormente ottimizzate, oppure scritte in assembler, se necessario.

SDCC è distribuito con un source level debugger SDCDB, che attualmente utilizza **ucSIM**, un simulatore freeware per 8051 e derivati.

L'ultima versione è scaricabile da **http://sdcc.sourceforge.net/.**

1.2 Open Source

Tutti i programmi usati in questo compilatore sono *open source* e *freeware*; il codice sorgente di tutti i programmi accessori (asxxxx assembler/linker, pre-processor) è distribuito nel pacchetto complessivo. Questo manuale (in originale, ndt) è mantenuto utilizzando un word processor freeware (**LyX**).

Questo programma è *free software*; potete redistribuirlo e/o modificarlo entro i termini della licenza *GNU General Public License*, come pubblicato dalla *Free Software Foundation*. Questo programma è distribuito nella speranza che sia utile, ma SENZA ALCUNA GARANZIA; è esclusa anche la garanzia implicita di COMMERCIABILITÀ o ADATTABILITÀ PER UNO SCOPO PARTICOLARE. Vedere la licenza *GNU General Public License* per maggiori dettagli. Insieme a questo pacchetto, dovreste avere ricevuto copia di tale licenza; se così non fosse, scrivere alla *Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA*.

In altre parole, potete usare, condividere e migliorare questo programma. E' vietato impedire a chiunque altro di usare, condividere e migliorare quanto gli consegnerete.

1.3 Convenzioni Tipografiche

In questo manuale, seguiremo le seguenti convenzioni:
I comandi che devono essere immessi, sono evidenziati in **bold**; gli esempi di codice sorgente saranno in `courier new`; le note particoleri e i termini nuovi saranno in *corsivo.*

1.4 Compatibilità con le Versioni Precedenti

Questo manuale ha parecchie correzioni di errori rispetto alle precedenti versioni. Tuttavia, sono state anche introdotte alcune incompatibilità con le vecchie versioni. Non giusto per divertimento, ma per rendere il compilatore più stabile, efficiente e compatibile ANSI.

- `short` è ora equivalente ad `int` (16 bits), è usato per essere equivalente a `char` (8bits)
- la directory di default dove sono memorizzati i file include, le librerie e la documentazione è ora **/usr/local/share**
- i parametri di tipo `char` alle funzioni (vararg) sono convertiti in `int` a meno che non siano esplicitamente dichiarati (cast), p.e.:

```
char a=3;
printf ("%d %c\n", a, (char)a);
```

  invia `a` come un `int` e come un `char` rispettivamente.
- l'opzione *-regextend* è stata eliminata
- l'opzione *-noreparms* è stata eliminata

< *in sospeso: ulteriori incompatibilità?* >

1.5 Requisiti di Sistema

Di cosa avete bisogno prima di iniziare l'installazione di SDCC? Un computer, e il desiderio di programmare. Il metodo preferenziale di installazione è di compilare SDCC dal sorgente utilizzando il *gcc* e il *make* GNU. Per l'ambiente Windows, sono disponibili diverse distribuzioni binarie precompilate. Dovete avere un minimo di esperienza con i tools della linea di comando e nell'uso del compilatore.

1.6 Altre Risorse

La home page `http://sdcc.sourceforge.net/` è un ottimo sito dove trovare i set di distribuzione. Vi si trovano anche dei link alle mailing list che offrono aiuto o discutono su SDCC con altri utenti SDCC. Sono disponibili inoltre dei link ad altri siti riferentesi a SDCC. Questo manuale può essere trovato nella directory DOC del package sorgente come testo o html. Alcuni degli altri tool (simulatore e assembler) inclusi con SDCC contengono la propria documentazione e possono essere trovati nella distribuzione sorgente. Se volete l'ultima versione non ufficiale del software, il package completo del sorgente è disponibile direttamente da CVS anonima su `cvs.sdcc.sourceforge.net`.

1.7 Aspettative per il Futuro

Ci sono (e sempre ci saranno) alcune cose che devono essere fatte. Queste sono alcune di quelle che mi vengono in mente:

`sdcc -c --model-large -o large _atoi.c` (dove *large* può essere un diverso basename o directory)

```
char KernelFunction3(char p) at 0x340;
```

Se vi viene in mente qualcos'altro, inviatelo alla mailing list.

2 INSTALLAZIONE

2.1 Installazione su Linux/Unix

1. Scaricate il sorgente, che avrà un nome tipo **sdcc-2.x.x.tgz**
2. Utilizzate un terminale per riga di comando, p.e. **xterm**
3. Scompattate il file utilizzando un comando come **"tar -xzf sdcc-2.x.x.tgz"**; ciò creerà una subdirectory chiamata `sdcc` con tutti i sorgenti.
4. Cambiate la directory in quella di SDCC, p.e. scrivendo **"cd sdcc".**
5. Immettere **"./configure".** Ciò configurerà il package per la compilazione sul vostro sistema.
6. Immettere **"make"**. Tutti i sorgenti verranno compilati, dopo un po' di tempo.
7. Immettere **"make install"** nella root. Ciò copierà gli eseguibili, gli include, le librerie e i documenti nelle directory di installazione.

2.2 Installazione su Windows

<nota: è questa completa? Dove è la Borland?>

Per l'installazione sotto Windows, dovete innanzitutto scegliere tra un package binario precompilato, o installare il package sorgente insieme al Cygwin. La prima soluzione è più veloce da installare, mentre il package Cygwin include tutte le utilità open source usate per compilare il pacchetto sorgente completo del SDCC in ambiente Windows. Se non avete familiarità con l'ambiente a linea di comando di Unix, è meglio che leggiate la sezione contenente le informazioni per utenti Windows prima di procedere all'installazione.

2.2.1 Installazione sotto Windows utilizzando un package binario

Scaricare il package binario e scompattatelo utilizzando il vostro preferito programma di decompressione (WinZip, Gunzip, ecc.). Verrà creato un gruppo di sotto directory. Un esempio di struttura di directory dopo la scompattazione è:

`C:\usr\local\bin`	per gli eseguibili
`C:\usr\local\share\sdcc\include`	per i file include
`C:\usr\local\share\sdcc\lib`	per i file di libreria

Modificate la variabile di sistema **`PATH`** in modo che contenga il percorso della directory degli eseguibili. Per esempio, create un file `setsdcc.bat` contenente il comando: set `PATH=c:\usr\local\bin;%PATH%`

Quando compilate i programmi con **sdcc**, potreste aver bisogno di specificare il percorso delle directory delle librerie e degli include. Per esempio:

```
sdcc -I c:\usr\local\share\sdcc\include -L c:\usr\local\share\sdcc\lib\small
test.c
```

2.2.2 Installazione sotto Windows utilizzando Cygwin

Scaricate e installate il package Cygwin dal sito http://sources.redhat.com/cygwin/ . Attualmente, questo attiva il download di un piccolo programma di installazione che provvede automaticamente a scaricare e installare le parti selezionate del package (il pacchetto completo è un file di circa 80Mbyte).

Selezionate un terminale a linea di comando Unix/Bash dal menu del Cygwin.

Seguite le istruzioni di installazione precedentemente illustrate per l'istallazione sotto Unix/Linux (par. 2.1).

2.3 Test del compilatore SDCC

La prima cosa che dovete fare dopo l'installazione del compilatore SDCC è vedere se esso gira. Immettere "**sdcc --version**" al prompt; il programma deve eseguire e mostrare la versione. Se non parte, o appare un messaggio tipo che non viene trovato il programma **sdcc**, dovete ricontrollare la procedura di installazione. Verificate che il percorso della directory degli eseguibili del compilatore sia presente nella variabile di sistema `PATH`. Assicuratevi che il programma **sdcc** sia in tale directory, altrimenti qualcosa è andata storta nell'installazione (vedi 2.4).

Verificate che il compilatore lavori bene su un esempio di sorgente molto semplice. Immettete il seguente programma **`test.c`** utilizzando il vostro editor preferito:

```
int main(int t) {
    return t+3;
}
```

Compilatelo utilizzando il comando "**`sdcc -c test.c`**". Se tutto procede bene, il compilatore genererà due file: **`test.asm`** e **`test.rel`**. Se così è, congratulazioni! Avete appena compilato il vostro primo programma con SDCC. L'opzione **`-c`** utilizzata nell'esempio dice al compilatore di non effettuare il link del codice generato, giusto per semplificare le cose in questo stadio.

Il prossimo passo è quello di provare il linker. Immettere il comando "**`sdcc test.c`**". Se tutto va bene, il compilatore eseguirà il link con le librerie producendo un file **`test.ihx`**. Se fallisce (nessun file in uscita e il linker genera dei warning), probabilmente il problema è che SDCC non trova la directory `/usr/local/share/sdcc/lib` (vedi 2.4).

Il test finale è per assicurarsi che SDCC riesce ad usare gli header file standard e le librerie. Modificate il file **`test.c`** come segue:

```
#include <string.h>
main() {
  char str1[10];
  strcpy(str1, "testing");
}
```

Compilatelo utilizzando il comando "**`sdcc -c test.c`**". Ciò deve produrre un file **`test.ihx`** senza produrre alcun warning, quale ad esempio che non riesce a trovare il file **`string.h`** . Se così fosse, significa che SDCC non trova la directory `/usr/local/share/sdcc/include` (vedi 2.4).

2.4 Risoluzione dei problemi di installazione

2.4.1 SDCC non trova i file di libreria o gli header

L'installazione di default assume che i files suddetti siano situati in "`/usr/local/share/sdcc/lib`" e in "`/usr/local/share/sdcc/include`". In alternativa, è possibile specificare tali percorsi come opzioni del compilatore come segue: "`sdcc -L /usr/local/sdcc/lib/small -I /usr/local/sdcc/include test.c`".

2.4.2 SDCC non compila correttamente

Una cosa da provare è ricominciare da capo scompattando il package .tgz in una directory vuota. Configurate il tutto e scrivete:

```
make 2&>1 | tee make.log
```

Fatto questo, potete visualizzare il file **`make.log`** per individuare il problema. Oppure, una parte significativa del file log può essere allegata ad una e-mail da inviare alla mailing list per ricevere un aiuto.

2.4.3 Spiegazione di alcuni comandi

Il comando **`./configure`** è uno script che analizza il vostro sistema ed effettua alcune configurazioni per assicurarsi che il pacchetto sorgente sia compilato sul vostro sistema. Impiega alcuni minuti a terminare, ed effettua alcuni test per determinare quali opzioni del compilatore sono installate.

Il comando `make` esegue il tool GNU make, che compila automaticamente tutti i package sorgenti nei finali file eseguibili.

Il comando `make install` installa il compilatore, gli altri eseguibili e le librerie nelle opportune directory. Come detto, i percorsi sono:

`C:\usr\local\bin`	per gli eseguibili
`C:\usr\local\share\sdcc\include`	per i file include
`C:\usr\local\share\sdcc\lib`	per i file di libreria

2.5 Informazioni aggiuntive per Utenti Windows

Il metodo standard di installazione in un sistema Unix comporta la compilazione del package sorgente. Questo processo, facile sotto Unix, può essere piuttosto complesso sotto Windows. Il Cygwin è un package piuttosto grosso da scaricare, e il processo di compilazione è piuttosto lento. Una possibile alternativa è di installare un package precompilato.

Il pacchetto Cygwin consente agli utenti Windows di eseguire una interfaccia a riga di comando Unix, e implementa anche un file system tipo quello di Unix sopra Windows. Sono inclusi molti dei tool di sviluppo software GNU. Questo è buono se avete un po' di esperienza con la riga di comando e il file system di Unix, altrimenti potrete trovare più semplice installare il precompilato, che gira in ambiente standard Windows.

2.5.1 Lavorare con Cygwin

SDCC è normalmente distribuito come file compresso `.tgz` , che è un formato simile a quello `.zip` in ambiente Windows. Cygwin include i tool di cui avete bisogno per estrarre i files sorgenti di SDCC. Per scompattarlo, seguite le istruzioni indicate nella sezione 2.5 per l'installazione in ambiente Unix/Linux. Prima di farlo, però, dovete imparare come lanciare la shell di Cygwin e alcuni comandi di base per spostare i file, cambiare directory, eseguire comandi e così via.

Il comando cambia directory è **`cd`**, il comando move (sposta) è **`mv`**. Per stampare la directory corrente, usare **`pwd`**; per creare un directory, **`mkdir`**.

Ci sono alcune differenze basilari tra i file system di Unix e di Windows che dovete apprendere: quando immettete il percorso di una directory, Unix e Cygwin usano le slash "/", mentre Windows usa le backslash "\". Sotto Unix, non esiste il concetto di lettere dei drive, come "c:"; tutto il file system appare come directory.

Se avete correttamente installato SDCC con il Cygwin, è possibile ottenere un eseguibile `.exe` utilizzando l'apposito makefile fornito per tale scopo. Per esempio, utilizzando il compilatore Borland a 32 bit, dovrete eseguire **`make -f Makefile.bcc`**. Una versione a riga di comando del compilatore Borland è scaricabile dal sito web della Inprise.

2.5.2 Eseguire SDCC da file precompilati

Se usate i file precompilati, è possibile specificare i percorsi delle librerie e degli header come opzioni del compilatore come segue: "`sdcc -L /usr/local/sdcc/lib/small -I /usr/local/sdcc/include test.c`".

Altrimenti, impostate la variabile di sistema **`PATH`** in modo che contenga il percorso della directory degli eseguibili. Per esempio, create un file `setsdcc.bat` contenente il comando: set `PATH=c:\usr\local\bin;%PATH%`.

2.6 SDCC su altre piattaforme

Nel caso di FreeBSD e altri Unix non GNU, assicuratevi che il make GNU sia installato quale make tool di default.

SDCC è stato scritto per poter girare sotto vari sistemi operativi e processori. Se nel vostro sistema potete eseguire GNU GCC/make, SDCC può essere regolarmente compilato ed eseguito.

2.7 Opzioni Avanzate di Installazione

Il comando **configure** ha molte opzioni. La più utilizzata è: `--prefix=<directory name>` , dove `<directory name>` è la destinazione finale degli eseguibili e librerie di SDCC (per default, `/usr/local`). Il processo di installazione creerà la seguente struttura di directory sotto la specificata `<directory name>`:

`bin/` - file eseguibili (aggiungere alla variabile ambiente PATH)
`bin/share/`
`bin/share/sdcc/include/` - include header files
`bin/share/sdcc/lib/`
`bin/share/sdcc/lib/small/` - Object file e librerie per modello small
`bin/share/sdcc/lib/large/` - Object file e librerie per modello large
`bin/share/sdcc/lib/ds390/` - Object file e librerie per Dallas DS80C390

Il comando `./configure --prefix=/usr/local` configurerà il compilatore perchè sia installato nella directory `/usr/local`

2.8 Componenti di SDCC

SDCC non è solo un compilatore, ma un insieme di tools provenienti da vari sviluppatori. Questi includono l'assembler, il linker, simulatori e altri componenti. Di seguito vedrete un elenco di alcuni componenti. Notare che l'assembler e il simulatore inclusi hanno una documentazione separata, che potrete trovare nel package sorgente nelle rispettive directory. Siccome SDCC cresce costantemente per inglobare il supporto di altri processori, vengono aggiunti altri package di altri sviluppatori e possono avere il proprio set di documentazione.

Potete dare un'occhiata ai file installati in `<installdir>`. Al momento della scrittura di questo manuale, potrete trovare i seguenti programmi:

In `<installdir>/bin`:
sdcc - Il compilatore C.
sdcpp - Il Preprocessore C.
asx8051 - L'assembler per i processori 8051 e derivati.
as-z80, as-gbz80 - L'assembler per lo Z80 e per il GameBoy Z80.
aslink - Il linker per i processori 8051 e derivati.
link-z80, link-gbz80 -Il linker per lo Z80 e per il GameBoy Z80.
s51 - Simulatore ucSim 8051.
sdcdb - Il source debugger.
packihx - Un tool per formattare i files Intel HEX.

In `<installdir>/share/sdcc/include` ci sono tutti gli include files.
In `<installdir>/share/sdcc/lib` ci sono i sorgenti delle librerie runtime, le sottodirectory *small*, *large* e *ds390* con i precompilati rilocabili.
In `<installdir>/share/sdcc/doc` ci sono i files della documentazione.

Procedendo con il supporto di altri processori, questa lista si espanderà includendo gli eseguibili per supportare tali processori.

Sdcc (compilatore C): E' il compilatore attuale; utilizza il preprocessore C e chiama poi l'assembler e il linker.

sdcpp (Preprocessore C): E' una versione modificata del preprocessore GNU. Il preprocessore inserisce i sorgenti da includere (*#include*), processa le direttive *#ifdef*, *#defines* e così via.

asx8051, as-z80, as-gbz80, aslink, link-z80, link-gbz80 (assembler e linker): Sono gli assembler e i linker, sviluppati da Alan Baldwin. John Hartman ha sviluppato la versione per gli 8051, e io (Sandeep) ho effettuato alcune modifiche e eliminazione di bug per far si che lavorassero opportunamente con SDCC.

s51 - Simulator: E' un simulatore freeware sviluppato da Daniel Drotos (drdani@mazsola.iit.uni-miskolc.hu). Questo simulatore è costruito per essere parte del processo di sviluppo. Per maggiori informazioni, visitate il sito di Daniel: http://mazsola.iit.uni-miskolc.hu/~drdani/embedded/s51

sdcdb - Source Level Debugger: E' il relativo source level debugger. L'attuale versione utilizza il simulatore S51, ma può essere facilmente modificato per utilizzare altri simulatori.

3 USO DI SDCC

3.1 Compilazione

3.1.1 Progetti a singolo sorgente

Per i progetti con un solo file sorgente 8051, il processo è molto semplice. Compilate il vostro programma con il comando

```
sdcc sourcefile.c
```

Il programma sorgente viene compilato, assemblato, linkato; i file in uscita sono:

`sourcefile.asm` - File sorgente in assembler creato dal compilatore
`sourcefile.lst` - Listing in assembler creato dall'assembler stesso
`sourcefile.rst` - File in assembler contenente informazioni sul linking
`sourcefile.sym` - Lista dei simboli del sorgente, creata dall'assembler
`sourcefile.rel` - File oggetto creato dall'assembler, inviato al linker
`sourcefile.map` - Mappa di memoria del modulo, creata dal linker
`sourcefile.ihx` - File finale in formato Intel Hex (potete selezionare il formato Motorola S19 con l'opzione *--out-fmt-s19*)
`sourcefile.cdb` - File opzionale (opzione *--debug*) contenente informazioni per il debug

3.1.2 Progetti a sorgenti multipli

SDCC può compilare solo UN sorgente alla volta. Come esempio, supponiamo che il progetto contenga i seguenti files:

`foo1.c` (contenente alcune funzioni)
`foo2.c` (contenente altre funzioni)
`foomain.c` (contenente alcune funzioni e la funzione *main()*)

I primi due files devono essere compilati separatamente tramite i comandi:

```
sdcc -c foo1.c
sdcc -c foo2.c
```

Quindi compilate il sorgente contenente la funzione *main()* e linkate i file mediante il comando:

```
sdcc foomain.c foo1.rel foo2.rel
```

In alternativa, il file `foomain.c` può essere compilato a parte come segue:

```
sdcc -c foomain.c
sdcc foomain.rel foo1.rel foo2.rel
```

Il file contenente la funzione *main()* deve essere il primo della lista specificata nella riga di comando, dal momento che il linker processa i files nell'ordine in cui sono scritti.

3.1.3 Progetti con librerie aggiuntive

Alcune routine riutilizzabili possono essere compilate in librerie (vedi la documentazione dell'assembler e del linker per spiegazioni su come creare un file `.lib`). Le librerie create in tal modo possono essere incluse nella riga di comando.
Assicuratevi di mettere l'opzione *-L <library-path>* per dire al linker dove trovare i files di libreria se non sono nella directory corrente.

In questo esempio, si suppone di avere il file sorgente `foomain.c` e un file di libreria `foolib.lib` nella directory `mylib`:

```
sdcc foomain.c foolib.lib -L mylib
```

Notare che `mylib` deve contenere il percorso completo.

Il modo più efficiente di usare le librerie è di mantenere moduli separati in sorgenti separati. Per un esempio, vedere la libreria standard `libsdcc.lib` nella directory `<installdir>/share/lib/small`.

3.2 Opzioni della Riga di Comando

3.2.1 Opzioni di Selezione del Processore

-mmcs51	Genera il codice per la famiglia MCS51. E' l'opzione di default.
-mds390	Genera il codice per il Dallas DS80C390
-mz80	Genera il codice per la famiglia Z80
-mavr	Genera il codice per la famiglia Atmel AVR (in sviluppo - incompleta)
-mpic14	Genera il codice per la famiglia PIC 14bit (in sviluppo - incompleta)
-mtlcs900h	Genera il codice per il Toshiba TLCS-900H (in sviluppo - incompleta)

3.2.2 Opzioni del Preprocessore C

-I<path> Percorso aggiuntivo dove il preprocessore cercherà i file `<*.h>` o `"*.h"`

-D<macro[=value]> Definizione di una macro sulla riga di comando da passare al preprocessore.

-M Dice al preprocessore di emettere delle regole (rule) per i make file, che descrivono le dipendenze di ciascun file oggetto. Per ogni sorgente, il preprocessore emette un comando il cui target è il nome del file oggetto del relativo sorgente, e le cui dipendenze sono tutte specificate nell' #include del sorgente. Tale comando può essere una linea singola o può continuare sulla riga successiva con un '\'. La lista dei comandi è emessa sullo standard output anzichè nel programma C preprocessato. L'opzione *-M* implica l'opzione *-E*.

-C Dice al preprocessore di non eliminare i commenti. Usata con l'opzione *-E*.

-MM Come *-M*, ma il risultato menziona solo i file header utente inclusi con `'#include "file"'`. Gli header di sistema inclusi con `'#include <file>'` vengono omessi.

-Aquestion(answer) Assegna la risposta *answer* alla domanda *question*, nel caso sia testata con la condizione del preprocessore `#if #question(answer)`. *-A-* disabilita le assegnazioni standard che descrivono normalmente la macchina target.

-Umacro Disattiva la macro *macro*. L'opzione *-U* è valutata dopo tutte le opzioni *-D*, ma prima di ogni opzione *-include* e *-imacros*.

-dM Dice al preprocessore di emettere solo una lista delle definizioni di macro che sono effettive alla fine del preprocesso. Usata con l'opzione *-E*.

-dD Dice al preprocessore di passare tutte le definizioni di macro all'uscita, nella sequenza in cui appaiono nell'uscita.

-dN Come *-dD*, eccetto che gli argomenti e i contenuti delle macro sono omessi. Solo i `#define name` sono inclusi nel file di uscita.

3.2.3 Opzioni del Linker

-L --lib-path<path_librerie> Questa opzione passa al linker il percorso di librerie addizionali. Il percorso deve essere assoluto (non relativo). Vedere la sezione 3.1.3 per ulteriori dettagli.

--xram-loc<Valore> L'indirizzo di partenza di default della RAM esterna è 0x0000. Il valore immesso può essere sia in formato esadecimale che decimale; per esempio: *-xram-loc 0x8000* oppure *-xram-loc 32768*.

--code-loc<Valore> L'indirizzo di partenza di default del segmento di codice è 0x0000. Notare che quando si usa questa opzione, la tabella dei vettori di interrupt viene riallocata all'indirizzo specificato. Il valore immesso può essere sia in formato esadecimale che decimale; per esempio: *-code-loc 0x8000* oppure *-code-loc 32768*.

--stack-loc<Valore> Definisce il valore iniziale dello stack pointer. Questo valore è 0x07 se si usa solo il banco 0 dei registri; se si usano anche altri banchi, lo stack pointer viene inizializzato alla locazione subito oltre il banco di registri usato. Il valore immesso può essere sia in formato esadecimale che decimale; per esempio: *--stack-loc 0x20* oppure *--stack-loc 32*. Se tutti e quattro i banchi di registri sono utilizzati, lo stack viene posto dopo il segmento dei dati (equivalente all'opzione *--stack-after-data*).

--stack-after-data Questa opzione fa sì che lo stack venga posto nella RAM interna dopo il segmento dei dati.

--data-loc<Valore> Definisce la locazione di partenza del segmento dati della RAM interna; il valore di default è 0x30. Il valore immesso può essere sia in formato esadecimale che decimale; per esempio: *--data-loc 0x20* oppure *--data-loc 32*.

--*idata-loc<Valore>* Definisce la locazione di partenza del segmento dati della RAM interna indirizzabile internamente; il valore di default è 0x80. Il valore immesso può essere sia in formato esadecimale che decimale; per esempio: --*idata-loc 0x88* oppure --*idata-loc 136*.

--*out-fmt-ihx* Definisce il formato del codice oggetto finale (uscita del linker) come Intel Hex. Questa è l'opzione di default.

--*out-fmt-s19* Definisce il formato del codice oggetto finale (uscita del linker) come Motorola S19.

3.2.4 Opzioni per MCS51

--*model-large* Genera il codice per il modello di memoria Large; vedere par. 3.16 per dettagli. Se viene usata questa opzione su un sorgente di un progetto, anche tutti gli altri file sorgenti del progetto devono essere compilati con questa opzione. Inoltre, dato che tutte le routine di libreria standard sono compilate con il modello Small, devono anch'esse essere ricompilate.

--*model-small* Genera il codice per il modello di memoria Small; vedere par. 3.16 per dettagli. Questo è il modello di default.

3.2.5 Opzioni per DS80C390

--*model-flat24* Genera il codice per il modo di indirizzamento flat a 24bit. Questo è il solo modo che il compilatore genera attualmente ed è di default quando viene specificata l'opzione -*mds390*. Vedere il par. 3.17 per dettagli.

--*stack-10bit* Genera il codice per il modo con stack a 10bit del Dallas DS80C390. Questo è il solo modo che il compilatore supporta attualmente ed è di default quando viene specificata l'opzione -*mds390*. In questo modo, lo stack è posto nel primo Kbyte della XRAM interna, che è mappato in 0x400000. Notare che il supporto di tale modo è incompleto, poichè continua ad usare un solo byte come stack pointer. Questo significa che per ora verranno usati solo i primi 256 bytes del potenziale spazio di stack di 1K. Comunque, questo modo permette di recuperare tutti i 256 bytes di RAM scratchpad per l'utilizzo come segmenti DATA e IDATA. Il compilatore non genera alcun codice per porre il processore in modalità con stack a 10bit; è pertanto importante che il programma utente attivi tale modo operativo prima di chiamare qualsiasi funzione reentrant compilata in tale modalità. In teoria, ciò dovrebbe avvenire con l'opzione --*stack-auto*, ma non è stato testato. E' incompatibile con l'opzione --*xstack*. Ha inoltre senso solo se il processore è in modalità di indirizzamento contiguo a 24bit (vedi opzione --*model-flat24*).

3.2.6 Opzioni di Ottimizzazione

--*nogcse* Non viene effettuata l'eliminazione delle sub-espressioni globali; questa opzione può essere usata quando il compilatore occupa una larga parte dell'area stack o dati per la memorizzazione di temporanee del compilatore. Compare un messaggio di warning quando ciò avviene, e il compilatore indica il numero di byte extra che ha generato. E' consigliato di

NON utilizzare questa opzione; in sua vece è meglio utilizzare `#pragma NOGCSE` per effettuare l'eliminazione delle sub-espressioni globali solo per le funzioni che lo richiedono.

--*noinvariant* Non effettua l'ottimizzazione dei cicli invarianti, cosa da effettuare per le stesse ragioni viste per l'opzione precedente. Per dettagli su tale ottimizzazione, vedere 4.1.4 . E' consigliato di NON utilizzare questa opzione; in sua vece è meglio utilizzare `#pragma NOINVARIANT` per non effettuare l'ottimizzazione solo per le funzioni che lo richiedono.

--*noinduction* Non effettua l'ottimizzazione dei cicli indotti. E' consigliato di NON utilizzare questa opzione; in sua vece è meglio utilizzare `#pragma NOINDUCTION` per non effettuare l'ottimizzazione solo per le funzioni che lo richiedono.

--*nojtbound* Non effettua il controllo della condizione di "confine" (boundary) quando l'istruzione **switch** è usata mediante le tabelle di jump. Vedere 4.1.7 per dettagli. E' consigliato di NON utilizzare questa opzione; in sua vece è meglio utilizzare `#pragma NOJTBOUND` per non effettuare l'ottimizzazione solo per le funzioni che lo richiedono.

--*noloopreverse* Non effettua l'ottimizzazione inversa dei loop.

3.2.7 Altre Opzioni

-*c* --*compile-only* Compila e assembla il sorgente, ma non chiama il linker.

-*E* Esegue solamente il preprocessore C. Il risultato viene emesso nello standard output.

--*stack-auto* Tutte le funzioni del sorgente saranno compilate come reentrant, cioè i parametri e le variabili locali sono allocati nello stack. Vedere 3.5 per dettagli. Se viene attivata questa opzione, tutti i files sorgenti del progetto devono essere compilati con questa opzione.

--*xstack* Utilizza uno pseudo stack nei primi 256 bytes della RAM esterna per allocare le variabili e passare i parametri delle funzioni. Vedere 3.5 per dettagli.

--*callee-saves function1[,function2][,function3]....* Il compilatore, per default, usa una procedura di salvataggio del chiamante per il salvataggio dei registri tra le chiamate di funzioni, comunque ciò può provocare un inutile push e pop dei registri nello stack quando si chiamano piccole funzioni da funzioni più grandi. Questa opzione può essere usata per disattivare il salvataggio dei registri solo per le funzioni specificate nell'opzione. Il compilatore non salverà i registri chiamando quelle funzioni, quindi non viene generato codice all'ingresso e all'uscita dalle stesse. Ciò sostanzialmente riduce il codice e migliora il tempo di esecuzione del codice generato. In futuro, il compilatore stesso (mediante una analisi procedurale) sarà in grado di determinare lo schema più appropriato da usare per ciascuna funzione. NON utilizzare tale opzione per le funzioni di sistema, come ad esempio la *_muluint()*; se questa opzione è usata per una funzione di libreria, la libreria stessa deve essere ricompilata con tale opzione. Se il progetto consiste in più di un file sorgente, tutti i sorgenti devono essere compilati con la stessa stringa di opzione (cioè con lo stesso elenco di funzioni). Vedere anche `#pragma CALLEE-SAVES`.

--*debug* Quando è usata questa opzione, il compilatore genera le informazioni per il debug, che possono essere usate con SDCDB. Tali informazioni sono immagazzinate in un file `.cdb`. Per dettagli, vedere la documentazione di SDCDB.

--*regextend* Questa opzione è obsoleta e non è più supportata.

--*noregparms* Questa opzione è obsoleta e non è più supportata.

--*peep-file<filename>* Questa opzione può essere usata per aggiungere regole all'ottimizzatore peep-hole. Vedere 4.1.11 per dettagli.

-*S* Termina l'esecuzione al termine della compilazione; non esegue l'assembler. L'uscita è un file in assembler del sorgente specificato.

-*Wa_asmOption[,asmOption]...* Passa le opzioni *asmOption* all'assembler.

-*Wl_linkOption[,linkOption]...* Passa le opzioni *linkOption* al linker.

--*int-long-reent* Le librerie Intere (16bit) e Long (32bit) vengono compilate come reentrant. Per default, tali librerie sono compilate come non-reentrant.

--*cyclomatic* Questa opzione fa sì che il compilatore generi un messaggio di informazione per ogni funzione del sorgente. Il messaggio contiene importanti informazioni sulla funzione: il numero di nodi che il compilatore ha trovato nel controllo del flusso della funzione. Vedere 4.7 per dettagli.

--*float-reent* La libreria Floating Point viene compilata come reentrant.

--*nooverlay* Il compilatore non sovrappone i parametri né le variabili locali di alcuna funzione.

--*main-return* Questa opzione può essere usata quando il programma è chiamato da un programma di monitor. Il compilatore genera una **ret** prima di uscire dalla funzione *main()*. L'opzione di default è di fare un lock-up (non esce mai dal *main()*).

--*no-peep* Disabilita la ottimizzazione peep-hole.

--*peep-asm* Passa il codice assembler inline attraverso l'ottimizzatore peep-hole. Ciò può provocare cambiamenti inaspettati al codice assembler; è meglio analizzare le regole per il peep-hole definite nel file `<target>/peeph.def` prima di utilizzare questa opzione.

--*iram-size<Valore>* Impone al linker di controllare se l'utilizzo della RAM interna è entro i limiti specificati dal *Valore* dato.

--*nostdincl* Evita che il compilatore passi il percorso di default al preprocessore.

--*nostdlib* Evita che il compilatore passi il percorso di default al linker.

--*verbose* Mostra le vaie operazioni che il compilatore sta effettuando.

-*V* Mostra il comando che il compilatore sta eseguendo.

3.2.8 Opzioni di Dump Intermedio

Le opzioni seguenti sono disponibili per il debug e il retargetting del compilato. Sono fatte per visualizzare il codice intermedio (iCode) generato dal compilatore in modo leggibile dall'utente, nei vari stadi del processo di compilazione.

--*dumpraw* Questa opzione fa emettere al compilatore il dump intermedio in un file chiamato `<file_sorgente>.dumpraw` subito dopo che il codice è stato generato, per esempio, prima che venga effettuata una ottimizzazione. Non è detto che i blocchi siano in ordine di esecuzione.

--*dumpgcse* Crea un dump dell'iCode dopo ogni eliminazione di sotto espressioni, e lo pone in un file chiamato `<file_sorgente>.dumpgse`.

--*dumpdeadcode* Crea un dump dell'iCode dopo ogni eliminazione del "codice morto" (deadcode), e lo pone in un file chiamato `<file_sorgente>.dumpdeadcode`.

--*dumploop* Crea un dump dell'iCode dopo ogni ottimizzazione dei loop, e lo pone in un file chiamato `<file_sorgente>.dumploop`.

--*dumprange* Crea un dump dell'iCode dopo ogni analisi di range attivo, e lo pone in un file chiamato `<file_sorgente>.dumprange`.

--*dumlrange* Crea un dump del range di attività di ogni simbolo.

--*dumpregassign* Crea un dump dell'iCode dopo l'assegnazione dei registri, e lo pone in un file chiamato `<file_sorgente>.dumprassgn`.

--*dumplrange* Crea un dump del range di attività di ogni temporanea intermedia (iTemp).

--*dumpall* Crea tutti i dump elencati sopra.

3.3 Estensioni Linguaggio per i Tipi di Dati dei MCS51/DS390

Oltre ai tipi di dati ANSI, SDCC consente l'uso delle seguenti classi di memorizzazione specifici per MCS51.

3.3.1 **xdata**

Le variabili dichiarate **xdata** saranno poste nella RAM esterna (XRAM). Questa è la classe di memorizzazione di default per il modello Large.

```
xdata unsigned char xduc;
```

3.3.2 **data**

Le variabili dichiarate **data** saranno poste nella RAM interna. Questa è la classe di memorizzazione di default per il modello Small.

```
data int iramdata;
```

3.3.3 **idata**

Le variabili dichiarate **idata** saranno poste nella parte indirizzabile indirettamente della RAM interna.

```
idata int idi;
```

3.3.5 **bit**

Questo è sia un tipo di dati che una classe di memorizzazione. Le variabili dichiarate **bit** saranno poste nella parte indirizzabile al bit della RAM interna.

```
bit iFlag;
```

3.3.6 **sfr** / **sbit**

Questi sono sia tipi di dati che classi di memorizzazione. Sono usati per descrivere le variabili SFR (Special Function Register), sia in formato byte (**sfr**) che bit (**sbit**).

```
sfr at 0x80 P0;      /* special function register P0 at location 0x80 */
sbit at 0xd7 CY;     /* CY (Carry Flag) */
```

3.4 Puntatori

SDCC consente (mediante estensioni del linguaggio) ai puntatori di puntare esplicitamente a qualsiasi delle aree di memoria degli 8051. Oltre ai puntatori espliciti, il compilatore utilizza una classe di puntatori **_generic** che può essere usata per puntare a qualsiasi area di memoria.

Esempi di dichiarazioni di puntatori:

```
/* puntatore fisicamente in xram */
/* che punta a un oggetto in ram interna */
data unsigned char * xdata p;


/* puntatore fisicamente in area codice (rom) */
/* che punta a un oggetto in ram esterna */
xdata unsigned char * code p;


/* puntatore fisicamente in area codice (rom) */
/* che punta a un oggetto in area codice (rom) */
code unsigned char * code p;


/* puntatore generico fisicamente in ram esterna */
char * xdata p
```

Ciò dovrebbe aver reso l'idea.

Per compatibilità con le precedenti versioni del compilatore, le seguenti sintassi per le dichiarazioni di puntatori sono ancora supportate, ma spariranno nel prossimo futuro.

```
unsigned char _xdata *ucxdp; /* puntatore a un dato in xram */
unsigned char _data  *ucdp ; /* puntatore a un dato in ram interna */
unsigned char _code  *uccp ; /* puntatore a un dato in area code */
unsigned char _idata *uccp;  /* puntatore ai 128 bytes alti di iram */
```

Tutti i puntatori generici sono trattati come puntatori generici a 3 byte (4 byte per il DS390). Questi tipi di puntatori possono anche essere dichiarati esplicitamente:

```
unsigned char _generic *ucgp;
```

Il byte più significativo di un puntatore generico contiene informazioni sull'area del dato. L'assembler supporta routine che sono chiamate dovunque il dato sia memorizzato utilizzando puntatori generici. Ciò è utile per sviluppare routine di libreria riutilizzabili. Specificando esplicitamente il tipo di puntatore si ottiene un codice più efficiente. Puntatori dichiarati utilizzando contemporaneamente la nuova e la vecchia sintassi possono dare risultati imprevedibili.

3.5 Parametri e Variabili Globali

Le variabili automatiche (locali) e i parametri delle funzioni possono essere posti sia nello stack che nel data-space. L'azione di default del compilatore è di porre tali variabili nella RAM interna per il modello Small e nella RAM esterna per il modello Large. In pratica, ciò rende tali variabili di tipo **static**, quindi per default le funzioni sono non-reentrant.

Possono essere poste nello stack utilizzando sia l'opzione *--stack-auto* del compilatore che utilizzando la dichiarazione **reentrant** nella dichiarazione della funzione:

```
unsigned char foo(char i) reentrant
{
...
}
```

Dal momento che con l'8051 lo spazio per lo stack è limitato, le soluzioni appena illustrate vanno usate con parsimonia. Notare che la dichiarazione **reentrant** indica che i parametri e le variabili locali devono essere posti nello stack, ma non significa che il banco di registri della funzione sia indipendente.

Le variabili locali possono essere assegnate a classi di memorizzazione e a indirizzi assoluti; per esempio:

```
unsigned char foo() {
    xdata unsigned char i;
    bit bvar;
    data at 0x31 unsiged char j;
    ...
}
```

Nell'esempio qui sopra, la variabile `i` viene allocata in RAM esterna, `bvar` nell'area indirizzabile al bit e `j` nella RAM interna. Quando compilato con l'opzione *--stack-auto* o quando una funzione è dichiarata reentrant, questo è possibile farlo solo per variabili **static**.

Non è possibile dichiarare classi di memorizzazione per i parametri delle funzioni (le classi verranno semplicemente ignorate); la loro allocazione è governata solo dal modello in uso, e dalle opzioni reentrant.

3.6 Overlaying

Per le funzioni non-reentrant, SDCC proverà a ridurre l'uso della memoria interna effettuando, se possibile, la sovrapposizione (overlay) dei parametri e delle variabili locali. Questi saranno allocati in un segmento sovrapponibile se la funzione non ne chiama altre, è di tipo non-reentrant e il modello in uso è Small. Se una classe di memorizzazione è esplicitamente dichiarata per una variabile, questa NON verrà sovrapposta.

Notare che il compilatore (non il linker) prende la decisione di sovrapporre i dati. Le funzioni chiamate da una routine di interruzione (ISR) devono essere precedute da `#pragma NOOVERLAY` se sono non-reentrant.

Inoltre, notare che il compilatore non effettua nessun processo sul codice assembler inline; quindi il compilatore potrebbe allocare erroneamente variabili locali e parametri di una funzione nel segmento sovrapponibile anche se il codice assembler chiama un'altra funzione C che può utilizzare l'area di overlay. In tal caso, usare `#pragma NOOVERLAY`.

I parametri e le variabili locali di funzioni contenenti moltiplicazioni o divisioni a 16 o 32 bit NON saranno sovrapponibili, poichè queste sono implementate usando funzioni esterne:

```
#pragma SAVE
#pragma NOOVERLAY
void set_error(unsigned char errcd)
{
    P3 = errcd;
}
#pragma RESTORE

void some_isr() interrupt 2 using 1
{
    ...
    set_error(10);
    ...
}
```

Nell'esempio qui sopra il parametro `errcd` della funzione `set_error` verrebbe assegnato al segmento sovrapponibile se non fosse presente la direttiva `#pragma NOOVERLAY`, cosa che causerebbe comportamenti imprevedibili durante l'esecuzione quando la funzione viene richiamata da una ISR (Interrupt Service Routine). La direttiva `#pragma NOOVERLAY` garantisce che i parametri e le variabili locali della funzione non siano in area sovrapponibile.

3.7 Interrupt Service Routines

SDCC consente la scrittura in C delle routine ISR (Interrupt Service Routine), mediante alcune istruzioni di estensione del linguaggio.

```
void timer_isr (void) interrupt 2 using 1
{
..
}
```

Il numero che segue l'istruzione *interrupt* è il numero di interrupt che eseguirà la routine. Il compilatore inserisce una `call` alla funzione ISR nella tabella dei vettori nella posizione indicata dal numero di cui sopra.

L'istruzione *using* dice al compilatore quale banco di registri (specifici dell' 8051) deve essere usato quando genera il codice della funzione ISR.

Notare che una o più funzioni sono chiamate dalla ISR, tali funzioni devono essere precedute da `pragma NOOVERLAY` se non sono reentrant.

Qui va fatta una nota speciale: le divisioni di `int` (16bit) e `long` (32bit), le operazioni di moltiplicazione e modulo sono implementate utilizzando routine di supporto esterne sviluppate in ANSI-C. Se una ISR necessita di utilizzare una di queste operazioni, le routine relative (vedi 3.14) devono essere ricompilate utilizzando l'opzione *--stack-auto*, e il vostro sorgente deve essere compilato usando l'opzione *--int-long-rent*.

Se avete un sorgente su più files, le ISR possono essere il qualsiasi di questi file, ma il loro prototipo DEVE essere presente o incluso nel file contenente la funzione `main()`.

I numeri degli interrupt e il corrispondente indirizzo per un 8051 standard sono descritti qui di seguito. SDCC aggiusta automaticamente la tavola dei vettori di interrupt al numero di interrupt più grande specificato.

Interrupt #	Descrizione	Vector Address
0	External 0	0x0003
1	Timer 0	0x000B
2	External 1	0x0013
3	Timer 1	0x001B
4	Serial	0x0023

Se la ISR è definita senza utilizzare un banco di registri oppure si è specificato il banco 0 (`using 0`), il compilatore salverà automaticamente i registri usati nello stack all'ingresso della routine, e li ripristinerà all'uscita; se la ISR chiama un'altra funzione, allora saranno salvati tutti i registri del banco. Questo schema può essere conveniente per piccole ISR che utilizzano pochi registri.

Se la ISR è definita utilizzando un banco di registri specifico, vengono salvati solo **A**, **B** e **DPTR**; se la ISR chiama un'altra funzione (che utilizza un altro banco di registri), allora saranno salvati nello stack tutti i registri del banco della funzione chiamata. Questo schema è raccomandato per ISR piuttosto grosse.

Chiamare altre funzioni da una ISR non è raccomandabile; cercate di evitarlo, se possibile. Vedere anche il modificatore *_naked* (par 3.9).

3.8 Funzioni Critiche

Per definire una funzione come critica, è presente uno speciale attributo: *critical*. In tal caso, SDCC genererà del codice per disattivare tutte le interruzioni prima di entrare in una funzione critica, e le riattiverà prima del ritorno al chiamante. Notare che concatenare (nesting) le funzioni critiche può provocare risultati imprevedibili.

```
int foo() critical
{
...
...
}
```

L'attributo *critical* può essere usato con altri attributi, quale ad esempio *reentrant*.

3.9 Funzioni Naked

Uno speciale attributo può essere associato alla dichiarazione di una funzione, definendola come *_naked* (nuda). Tale modificatore evita che il compilatore generi del codice di prologo ed epilogo per la funzione. Questo significa che l'utente è interamente responsabile di azioni quali salvare i registri che devono essere preservati, selezionare il banco dei registri desiderato, generare l'istruzione return al termine, ecc.

Praticamente significa che il contenuto della funzione deve essere scritto in assembler. Questo è particolarmente utile per le ISR in cui si deve evitare ingombranti (e non necessari) prologhi ed epiloghi.

Per esempio, comparate il codice generato da queste due funzioni:

```
data unsigned char counter;
void simpleInterrupt(void) interrupt 1
{
    counter++;
}

void nakedInterrupt(void) interrupt 2 _naked
{
    _asm
      inc _counter
      Reti      ; la reti DEVE essere esplicitamente
                ; inserita nelle funzioni _naked.
    _endasm;
}
```

Per un 8051, il codice generato per la *simpleInterrupt* diviene:

```
_simpleIterrupt:
    push    acc
    push    b
    push    dpl
    push    dph
    push    psw
    mov     psw,#0x00
    inc     _counter
    pop     psw
    pop     dph
    pop     dpl
    pop     b
    pop     acc
    reti
```

mentre *nakedInterrupt* diventa:

```
_nakedInterrupt:
    inc  _counter
    reti   ; la reti DEVE essere esplicitamente
           ; inserita nelle funzioni _naked.
```

Dato che non c'è niente che prevenga la scrittura di codice il C all'interno di una funzione *_naked*, è come se vi sparaste su un piede se lo fate, quindi si raccomanda di usare solo codice in assembler al loro interno.

3.10 Funzioni che utilizzano banchi di registri riservati

L'attributo *using* (che dice al compilatore di usare un banco di registri diverso dallo 0 di default) può essere usato solo per le funzioni di interrupt (vedi la nota in calce). In molti casi, ciò genera delle ISR più efficienti, dato che non devono salvare i registri nello stack.

L'attributo *using* non ha effetto sul codice generato per funzioni non di interrupt, ma occasionalmente può essere comunque utile[1].
(pending: non dovrebbe essere ancora stato implementato)

Una ISR che usa un banco di registri diverso dallo 0 suppone che possa trascurarne il contenuto, quindi non lo salva. Dato che negli 8051 e derivati gli interrupt a priorità più alta possono interrompere quelli a priorità più bassa, ciò significa che se viene attivata una ISR a priorità alta mentre viene processata una ISR più bassa che utilizza lo stesso banco di registri, possono accadere le cose più impensabili (e dannose).

Per prevenire ciò, non deve essere usato lo stesso banco di registri per routine a priorità diversa. E' facilmente implementabile utilizzando un banco per tutti gli interrupt a basso livello e un altro banco per quelli di alto livello (N.d.T.: attenzione che in alcuni 8051 derivati sono disponibili più livelli di priorità e riconfigurabili, nonchè molti più interrupt dell'originale Intel). Se modificate la priorità di un interrupt durante l'esecuzione del programma, dovete arrangiarvi: si suggerisce in tal caso di usare il banco di default per tutti e rinunciare a un po' di performance.

Le ISR sono più efficienti se non richiamano alcuna altra funzione. Se la ISR deve chiamare un'altra funzione, è più efficiente se tale funzione utilizza lo stesso banco della ISR chiamante; in ordine di efficienza, poi viene l'utilizzo del banco 0 nella funzione chiamata. E' molto poco efficiente se la ISR chiama una funzione che chiama un banco differente e non 0.

[1] Possibile eccezione: se una funzione è chiamata SOLO da funzioni ISR che utilizzano un particolare banco di registri, questa può essere dichiarata con lo stesso *using* della routine chiamante. Per esempio, se avete diverse ISR che usano il banco 1 e tutte loro chiamano la funzione *memcpy()*, può avere senso creare una versione specializzata di tale funzione utilizzando *using 1*, in modo da evitare che la ISR salvi il banco 0 al suo ingresso e lo ripristini all'uscita.

3.11 Indirizzamento Assoluto

I dati possono essere assegnati ad un indirizzo assoluto utilizzando la *at <address>* in aggiunta alla classe di memorizzazione:

```
xdata at 0x8000 unsigned char PORTA_8255 ;
```

In questo esempio, la variabile `PORTA_8255` verrà allocata all'indirizzo `0x8000` della RAM esterna. Questa possibilità è normalmente usata per accedere a dispositivi memory mapped connessi al microcontrollore. Il compilatore attualmente non riserva alcun spazio di memoria per variabili dichiarate in tal modo (sono implementate mediante delle `equ` in assembler). Perciò, è compito del programmatore aver cura che non ci siano sovrapposizioni con altre variabili dichiarate senza indirizzamento assoluto. I file listing dell'assembler (.lst) e i file generati dal linker (.rst e .map) sono l'ideale per verificare tali sovrapposizioni.

L'indirizzamento assoluto può essere definito per le variabili in tutte le classi di memorizzazione:

```
bit at 0x02 bvar;
```

In questo esempio, si alloca la variabile all'offset `0x02` nell'area indirizzabile al bit. Non c'è un reale vantaggio nell'assegnare un indirizzo assoluto a un variabile in tale maniera, a meno che non vogliate avere uno stretto controllo su tutte le variabili allocate.

3.12 Codice di Startup

All'inizio dell'area di memoria del programma (CODE area), il compilatore inserisce una chiamata alla routine C *_sdcc__external__startup()*. Questa routine è nelle librerie di runtime. Per default ritorna 0: se ritorna un valore diverso da 0, l'inizializzazione delle variabili static e global viene saltata e poi viene chiamata la funzione *main()*; altrimenti vengono inizializzate prima della chiamata. Potete aggiungere una routine *_sdcc__external__startup()* al vostro programma per scavalcare quella di default, nel caso dobbiate settare l'hardware o effettuare altre operazioni critiche prima dell'inizializzazione delle variabili static e global.

3.13 Codice Assembler Inline

SDCC consente l'uso di istruzioni assembler inline, con minime restrizioni sull'uso delle label.

Tutte le label utilizzate nell'assembler inline devono essere nel formato `nnnnn$`, dove *nnnnn* è un numero a 5 cifre inferiore a 100 (il che implica un limite massimo di 100 labels per ogni funzione). Si raccomanda di porre ogni istruzione assembler (incluse le label) in una riga separata (vedi esempi).

Quando si specifica l'opzione *--peep-asm*, il codice assembler inline passa attraverso l'ottimizzatore peephole. Ciò può provocare alcuni cambiamenti inaspettati al codice inline. Leggere attentamente le regole dell'ottimizzatore specificate nel file *SDCCpeeph.def* prima di usare tale opzione.

```
_asm
    mov     b,#10

00001$:
    djnz    b,00001$

_endasm ;
```

Il codice assembler inline può contenere qualsiasi istruzione valida per l'assembler, comprese le direttive assembler e i commenti. Il compilatore non effettua alcuna verifica al codice compreso tra le istruzioni `_asm ... _endasm`.

Il codice assembler inline non può fare riferimento a nessuna label del C, comunque può riferirsi alle label definite in assembler:

```
foo() {
  /* qui c'è del codice C */
  _asm
  ... ; qui c'è del codice assembler
  ljmp $0003
  _endasm;

  /* ulteriore codice C */

   clabel: /* l'assembler inline non può riferirsi a questa label
*/

  _asm
  $0003: ;label (può essere riferita solo da assembler inline)
  ... ; qui c'è del codice assembler
  _endasm ;

  /* ulteriore codice C */
}
```

In altre parole, l'assembler inline può riferirsi solo a label definite in assembler inline all'interno di una funzione. E, all'opposto, istruzioni C non possono accedere alle label definite in assembler inline.

3.14 Supporto di int (16 bit) e long (32 bit)

Per le variabili **int** (16 bit) e **long** (32 bit), con e senza segno (**signed** e **unsigned**), le operazioni di moltiplicazione, divisione e modulo sono implementate mediante routine di supporto. Queste routine sono scritte in ANSI-C per facilitare il porting ad altre MCU, anche se sono utilizzate alcune ottimizzazioni in assembler per lo specifico modello in uso. I file seguenti contengono le routine, e possono essere trovati in *<installdir>/share/sdcc/lib*.

_mulsint.c - moltiplicazione a 16 bit con segno (chiama _muluint)
_muluint.c - moltiplicazione a 16 bit senza segno
_divsint.c - divisione a 16 bit con segno (chiama _divuint)
_divuint.c - divisione a 16 bit senza segno
_modsint.c - modulo a 16 bit con segno (chiama _moduint)
_moduint.c - modulo a 16 bit senza segno
_mulslong.c - moltiplicazione a 32 bit con segno (chiama _mululong)
_mululong.c - moltiplicazione a 32 bit senza segno
_divslong.c - divisione a 32 bit con segno (chiama _divulong)
_divulong.c - divisione a 32 bit senza segno
_modslong.c - modulo a 32 bit con segno (chiama _modulong)
_modulong.c - modulo a 32 bit senza segno

Siccome sono compilate come *non-reentrant*, le ISR non possono utilizzare tali operazioni. Se ciò è inevitabile, le routine di cui sopra devono essere ricompilate con l'opzione *--stack-auto,* dopo di che il sorgente deve essere compilato con l'opzione *--int-long-rent*.

3.15 Supporto per Floating Point

SDCC supporta i numeri floating point IEEE (singola precisione - 4 bytes). Le routine di supporto sono derivate dalla *floatlib.c* del gcc, e sono:

_fsadd.c - somma di numeri floating point
_fssub.c - sottrazione di numeri floating point
_fsdiv.c - divisione di numeri floating point
_fsmul.c - moltiplicazione di numeri floating point
_fs2uchar.c - converte floating point in unsigned char
_fs2char.c - converte floating point in signed char
_fs2uint.c - converte floating point in unsigned int
_fs2int.c - converte floating point in signed int
_fs2ulong.c - converte floating point in unsigned long
_fs2long.c - converte floating point in signed long
_uchar2fs.c - converte unsigned char in floating point
_char2fs.c - converte char in floating point
_uint2fs.c - converte unsigned int in floating point
_int2fs.c - converte int in floating point
_ulong2fs.c - converte unsigned long in floating point
_long2fs.c - converte long in floating point

Notare che se si usano tutte le routine contemporaneamente, il data space è insufficiente. Per un sicuro utilizzo delle routine floating point, si consiglia l'uso del modello Large.

3.16 Modelli di memoria per MCS51

SDCC consente due modelli per gli MCS51, Small e Large. Moduli compilati con modelli differenti non devono mai essere combinati insieme, altrimenti il risultato è imprevedibile.

Le routine di libreria in dotazione al compilatore sono compilate per entrambi i modelli, e sono contenute in directory separate, per facilitare le operazioni di link.

Quando si usa il modello large, tutte le variabili dichiarate senza classe di memorizzazione saranno allocate nella RAM esterna, inclusi i parametri e le variabili locali (per le funzioni *non-reentrant*).

Quando si usa il modello small, tutte le variabili dichiarate senza classe di memorizzazione saranno allocate nella RAM interna (scratch-pad).

Un uso giudizioso delle classi di memorizzazione specifiche del processore e di funzioni *reentrant* garantisce un codice più efficiente di quello ottenuto con il modello large. Quando il sorgente è compilato con il modello large, molte ottimizzazioni sono disattivate, pertanto si raccomanda di utilizzare il modello small a meno che non sia assolutamente inevitabile.

3.17 Modelli di memoria per DS390

L'unico modello supportato è il *Flat 24*; questo genera il codice per il modo di indirizzamento contiguo a 24 bit del processore Dallas DS80C390. In questo modo, possono essere indirizzati direttamente fino a 4MB di RAM esterna e di ROM (program space). Vedere i datasheet nel sito www.dalsemi.com (ora www.maxim-ic.com, dato che la Dallas è stata acquisita dalla Maxim n.d.t.) Per ulteriori informazioni su tale micro.

In precedenti versioni del compilatore, questa opzione era usata con il generatore di codice MCS51 (*-mmcs51*). Ora è presente un proprio generatore di codice, selezionato con l'opzione *-mds390*.

Notare che il compilatore non genera alcun codice per porre il processore in modalità di indirizzamento a 24 bit (anche se nella libreria ds390 la routine *tinibios* potrà farlo per voi). Se non utilizzate *tinibios*, il boot loader o un sorgente similare deve assicurare che il processore sia in modalità di indirizzamento a 24 bit prima di chiamare il codice di startup di SDCC.

Come per l'opzione *--model-large*, le variabili sono per default poste nel segmento **xdata**.

I segmenti possono essere posti ovunque nel campo di indirizzamento di 4MB utilizzando le opzioni *--*-loc*. Notare che se un segmento è allocato oltre 64kB, il flag *-r* deve essere passato al linker per generare la rilocazione del segmento, e deve essere specificato il formato di uscita Intel Hex. Il flag *-r* può essere passato al linker mediante l'opzione *-Wl-r*. Comunque, attualmente il linker non gestisce segmenti di codice maggiori di 64kB.

3.18 Defines create dal compilatore

Il compilatore genera le seguenti `#defines`:

`SDCC` - questo simbolo è sempre definito.

`SDCC_mcs51` oppure `SDCC_ds390` oppure `SDCC_z80`, ecc - dipende dal processore usato

`__mcs51` oppure`__ds390` oppure`__z80`, ecc - dipende dal processore usato

`SDCC_STACK_AUTO` - questo simbolo è definito quando si usa l'opzione *--stack-auto*

`SDCC_MODEL_SMALL` - quando si usa il modello *--model-small*.

`SDCC_MODEL_LARGE` - quando si usa il modello *--model-large*.

`SDCC_USE_XSTACK` - quando si usa l'opzione *--xstack*.

`SDCC_STACK_TENBIT` - quando si usa l'opzione *-mds390*

`SDCC_MODEL_FLAT24` - quando si usa l'opzione *-mds390*

4 Manuale Tecnico SDCC

4.1 Ottimizzazioni

SDCC effettua delle ottimizzazioni standard oltre a quelle specifiche del processore.

4.1.1 Eliminazione di sotto espressioni

Il compilatore effettua l'eliminazione di sotto espressioni locali e globali. Per esempio:

```
i = x + y + 1;
j = x + y;
```

Verranno convertite in:

```
iTemp = x + y ;
i = iTemp + 1 ;
j = iTemp
```

Alcune sotto espressioni non sono così ovvie come nel precedente esempio:

```
a->b[i].c = 10;
a->b[i].d = 11;
```

In questo caso, l'aritmetica degli indirizzi `a->b[i]` viene calcolata solo una volta; il codice equivalente in C sarà:

```
iTemp = a->b[i];
iTemp.c = 10;
iTemp.d = 11;
```

Il compilatore, inoltre, tenterà di immagazzinare tali variabili temporanee nei registri.

4.1.2 Eliminazione del Dead-Code

Il sorgente:

```
int global;
void f () {
  int i;
  i = 1;      /* memorizzazione inutile */
  global = 1; /* memorizzazione inutile */
  global = 2;
  return;
  global = 3; /* non raggiungibile */
}
```

verrà cambiato in:

```
int global;
void f ()
{
  global = 2;
  return;
}
```

4.1.3 Copy-Propagation

Il sorgente:

```
int f() {
  int i, j;
  i = 10;
  j = i;
  return j;
}
```

verrà cambiato in:

```
int f() {
  int i, j;
  i = 10;
  j = 10;
  return 10;
}
```

Nota: le memorizzazioni inutili create da questa ottimizzazione saranno eliminate dalla eliminazione del dead-code (par 4.1.2).

4.1.4 Ottimizzazione dei loop

SDCC effettua due tipi di ottimizzazione dei loop: aggiustamento delle variabili invarianti nei loop (loop invariant), e la riduzione di impatto delle variabili indotte (strenght reduction). Oltre alla seconda ottimizzazione, l'ottimizzatore marca le variabili indotte in modo che si tenti di allocarle nei registri interni per tutta la durata del loop.

A causa di questa scelta, questa ottimizzazione causa un incremento nell'uso dei registri che può provocare un indesiderato spostamento di variabili temporanee nello stack o nel data space.

Il compilatore emette un messaggio di warning quando è costretto a allocare le variabili temporanee in tal modo. Se tale spostamento non è permesso, l'ottimizzazione del loop può essere disattivata per l'intero sorgente (con l'opzione *--noinduction*) o solo per una data funzione (usando `#pragma NOINDUCTION`).

Loop Invariant:

```
for (i = 0 ; i < 100 ; i ++)
  f += k + l;
```

diventa:

```
itemp = k + l;
for (i = 0; i < 100; i++)
  f += itemp;
```

Come detto in precedenza, alcune variabili invarianti non sono così evidenti, tutti i calcoli di indirizzi static sono spostati fuori dal loop.

Strength Reduction:

```
for (i=0;i < 100; i++)
  ar[i*5] = i*3;
```

diventa:

```
itemp1 = 0;
itemp2 = 0;
for (i=0;i< 100;i++) {
  ar[itemp1] = itemp2;
  itemp1 += 5;
  itemp2 += 3;
}
```

La più impegnativa moltiplicazione viene sostituita da un più semplice addizione.

4.1.5 Loop Reversing

Questa ottimizzazione serve a ridurre un appesantimento del controllo dei termini del loop a ogni iterazione. Alcuni semplici loop possono essere invertiti e implementati usando una istruzione assembler **djnz** (decrement and jump if not zero). SDCC analizza secondo i seguenti criteri per determinare se un loop è reversibile (nota: alcuni compilatori più sofisticati usano una analisi dipendente dai dati per tale decisione, mentre SDCC usa una analisi più semplice).

- Il ciclo for è nel seguente formato:
```
for (<sym> = <espressione> ; <sym> [< | <=] <espressione>;
  [<sym>++ | <sym> += 1])
    <corpo del ciclo>
```

- Il `<corpo del ciclo>` non contiene *continue* o *break*

- Tutti i *goto* sono interni al loop

- Non ci sono chiamate di funzioni nel corpo

- Alla variabile di controllo del loop non viene assegnato alcun valore dentro il loop

- La variabile di controllo del loop non è implicata in alcuna operazione aritmetica nel corpo del loop

- Non ci sono *switch* nel loop

Notare che l'istruzione **djnz** può essere usata solo per valori a 8 bit, pertanto è vantaggioso dichiarare la variabile di controllo come **char**. Ovviamente, ciò non è sempre possibile.

4.1.6 Semplificazioni Algebriche

SDCC effettua parecchie semplificazioni algebriche; quelle che seguono sono solo una piccola parte di queste.

```
i = j + 0 ;  /* modificata in */  i = j;
i /= 2 ;     /* modificata in */  i >>= 1;
i = j - j ;  /* modificata in */  i = 0;
i = j / 1 ;  /* modificata in */  i = j;
```

Notare che le espressioni indicate qui sopra generalmente sono introdotte dall'espansione delle macro o come risultato della copy propagation (vedi 4.1.3).

4.1.7 Istruzioni *switch*

SDCC cambia le istruzioni *switch* in tabelle di jump quando le seguenti condizioni sono verificate:

- Le label dei *case* sono in sequenza numerica, anche se non è necessario che siano in ordine numerico, né che il primo numero sia 1 o 0.

```
switch(i) {
  case 4:...
  case 5:...
  case 3:...
  case 6:...
  }
```

```
switch (i) {
  case 1: ...
  case 2: ...
  case 3: ...
  case 4: ...
  }
```

  Entrambi questi esempi possono essere implementati con una tabella di jump.

- Le label dei *case* sono almeno tre, dal momento che sono necessarie due istruzioni condizionali per gestire le condizioni di confine.

- Le label dei *case* sono meno di 84, dal momento che ogni label occupa 3 bytes e le tabelle di jump possono essere al massimo lunghe 256 bytes.

Gli *switch* che hanno vuoti nella sequenza numerica o che hanno più di 84 label dei *case* possono essere divisi in più di uno *switch*, in modo da generare codice più efficiente:

```
switch (i) {
case 1: ...
case 2: ...
case 3: ...
case 4: ...
case 9: ...
case 10: ...
case 11: ...
case 12: ...
}
```

Se il sorgente non ottimizzabile qui sopra viene diviso in due switch come segue:

```
switch (i) {
case 1: ...
case 2: ...
case 3: ...
case 4: ...
}
switch (i) {
case 9:  ...
case 10: ...
case 11: ...
case 12: ...
}
```

Ora, entrambi possono essere implementati mediante tabelle di jump.

4.1.8 Operazioni di shift a bit

Gli shift al bit sono operazioni molto frequenti nella programmazione embedded. SDCC tenta di implementare tali operazioni nel modo più efficiente possibile:

```
unsigned char i;
...
i>>= 4;
...
```

Genera il seguente codice:

```
mov a,_i
swap a
anl a,#0x0f
mov _i,a
```

In generale, SDCC non crea un loop se il numero di shift da effettuare è noto. Un altro esempio:

```
unsigned int i;
...
i >>= 9;
...
```

genererà:

```
mov a,(_i + 1)
mov (_i + 1), #0x00
clr c
rrc a
mov _i,a
```

Notate che SDCC memorizza i numeri nel formato little-endian (cioè il byte meno significativo per primo).

4.1.9 Rotazione di bit

Un caso speciale di operazioni di shift al bit sono le rotazioni di bit. SDCC riconosce le seguenti espressioni come una rotazione di un bit a sinistra:

```
unsigned char i;
...
i = ((i << 1) | (i >> 7));
...
```

genera il codice:

```
mov a,_i
rl a
mov _i,a
```

SDCC utilizza il riconoscimento del pattern nell'albero della sintassi per determinare questo tipo di operazioni. Variazioni di questo caso saranno comunque riconosciute come rotazioni di bit:

```
i = ((i >> 7) | (i << 1)); /* rotazione a sinistra */
```

4.1.10 Bit più significativo

E' frequentemente richiesto di ottenere il bit più significativo di un tipo intero (**long**, **int**, **short** o **char**). SDCC riconosce le seguenti espressioni per produrre codice ottimizzato:

```
unsigned int gint;
foo () {
  unsigned char hob;
  ...
  hob = (gint >> 15) & 1;
  ..
}
```

produce il seguente codice:

```
            61  ;  hob.c 7
000A E5*01  62  mov a, (_gint + 1)
000C 33     63  rlc a
000D E4     64  clr a
000E 13     65  rrc a
000F F5*02  66  mov _foo_hob_1_1, a
```

Variazioni di questo caso non saranno riconosciute. Questa è una espressione standard C, quindi raccomando vivamente di usare l'esempio sopra per ottenere il bit più significativo, ed è portabile. Ovviamente, viene riconosciuto se dovesse essere incorporato in espressioni più complesse quale:

```
xyz = gint + ((gint >> 15) & 1);
```

4.1.11 Ottimizzatore Peep-hole

Il compilatore utilizza un meccanismo di ricerca e sostituzione dei pattern (modelli) regolamentato per l'ottimizzatore peep-hole, ed è ispirato all'ottimizzatore di Christopher W. Fraser (cwfraser@microsoft.com).

Il compilatore utilizza un set di regole di default; ulteriori regolo possono essere aggiunte mediante l'opzione *--peep-file <filename>*. Il linguaggio di tali regole è illustrato mediante degli esempi:

```
replace {
  mov %1,a
  mov a,%1
} by {
  mov %1,a
}
```

La regola qui sopra cambierà la sequenza assembler:

```
  mov r1,a
  mov a,r1
```

in

```
  mov r1,a
```

Nota: tutte le occorrenze di %1 (variabile modello) devono denotare la stessa stringa. Quindi, con la regola appena mostrata, la sequenza assembler di seguito rimarrà invariata:

```
  mov r1,a
  mov a,r2
```

Altri casi speciali di ottimizzazione possono essere aggiunti dall'utente mediante l'opzione *--peep-file <filename>*. Per esempio, alcune varianti di 8051 permettono solo istruzioni `ajmp` e `acall`. Le seguenti regole cambiano tutte le `ljmp` e `lcall` in `ajmp` e `acall` rispettivamente:

```
replace { lcall %1 } by { acall %1 }
replace { ljmp %1 } by { ajmp %1 }
```

Anche il codice assembler inline viene passato attraverso l'ottimizzatore peep-hole, così l'ottimizzatore può essere utilizzato come un espansore di macro a livello assembler. Le regole stesse sono dipendenti dalla MCU, mentre l'infrastruttura del linguaggio delle regole è indipendente dalla MCU.

La sintassi per le regole è:

```
rule := replace [ restart ] {
          <sequenza assembly> '\n'
        } by { '\n'
           <sequenza assembly> '\n'
        } [if <NomeFunzione>] '\n'
```

dove `<sequenza assembly>` è una sequenza di istruzione che, incluse le label, devono essere su linee separate.

L'ottimizzatore applicherà le regole una ad una, partendo dalla prima in ordine di scrittura, finchè non ce ne saranno più.

Se viene specificata l'opzione *'restart'*, l'ottimizzatore ricomincerà la scansione delle regole a partire dalla prima; questa opzione peggiora le performance di compilazione, ed è fatta per essere usata in situazioni dove una trasformazione deve riattivare la regola stessa. Un buon esempio di ciò è la seguente regola:

```
replace restart {
  pop %1
  push %1
} by {
  ; nop
}
```

Notare che il modello di rimpiazzo non può essere vuoto, ma può però essere un commento. Nell'esempio qui sotto, senza l'opzione *'restart'*, solo la coppia più interna di pop e push sarebbe eliminata:

```
  pop ar1
  pop ar2
  push ar2
  push ar1
```

risulterebbe infatti:

```
  pop ar1
  ; nop
  push ar1
```

Con l'opzione *restart*, la regola sarà applicata un'altra volta al codice risultante, quindi tutte le coppie pop-push vengono eliminate, ottenendo:

```
  ; nop
  ; nop
```

Una funzione condizionale può essere aggiunta al comando. Tale possibilità è un po' complicata, facciamo quindi un esempio:

```
replace {
 ljmp %5
 %2:
} by {
 sjmp %5
 %2:
} if labelInRange
```

L'ottimizzatore effettua una ricerca del nome *labelInRange* nella tabella dei nomi di funzione definita nella funzione *callFuncByName* nel file sorgente *SDCCpeeph.c*. Se trova una corrispondenza, la funzione cercata viene chiamata.

Notare che può non essere alcun parametro per queste funzioni; in questo caso, l'uso di %5 è essenziale, dato che la *labelInRange* si aspetta di trovare la label in questa particolare variabile (la tabella contenente le variabili obbligatorie viene passata come parametro).
Se volete creare altre funzioni tipo queste, date un'occhiata approfondita alla *labelInRange* e al meccanismo di chiamata nel sorgente *SDCCpeeph.c*. In questo file, troverete anche le regole di base che sono incorporate con il compilatore; potete aggiungere le vostre regole nel set di default se non volete specificarle nell'opzione *--peep-file*.

4.2 Pragma

SDCC supporta le seguenti direttive **#pragma**; queste direttive sono applicabili solo a livello di funzione:

- `SAVE` - memorizza tutte le opzioni correnti
- `RESTORE` - ripristina le opzioni salvate dall'ultimo `SAVE`. Notare che `SAVE` e `RESTORE` non possono essere nidificate. SDCC utilizza lo stesso buffer per salvare le opzioni tutte le volte che si chiama `SAVE`.
- `NOGCSE` - arresta l'eliminazione delle sub-espressioni
- `NOINDUCTION` - arresta l'ottimizzazione loop induction
- `NOJTBOUND` - non genera il codice per il check dei limiti quando il comando *switch* viene trasformato in tabelle di jump
- `NOOVERLAY` - il compilatore non sovrapporrà i parametri e le variabili locali della funzione
- `NOLOOPREVERSE` - arresta l'ottimizzazione loop inverso
- `EXCLUDE NONE | {acc[,b[,dpl[,dph]]]` - disabilita la generazione di coppie push/pop nelle funzioni ISR (mediante l'istruzione *interrupt*). La direttiva deve essere messa subito prima la definizione della ISR e sarà valida per TUTTE le funzioni ISR successive. Per attivare il normale salvataggio dei registri per le ISR, usare `#pragma EXCLUDE NONE`.
- `CALLEE-SAVES function1[,function2[,function3...]]` - il compilatore usa per default la convenzione di salvare i registri della funzione chiamante, tuttavia ciò può provocare un inutile push e pop dei registri quando si chiama una piccola funzione da una più grande. Questa direttiva può essere usata per disattivare il salvataggio dei registri per le funzioni elencate; il compilatore genererà del codice all'ingresso e all'uscita di tali funzioni per salvare e ripristinare i registri da loro usati. Ciò riduce sostanzialmente la dimensione del codice e migliora le performance in run-time. In futuro il compilatore, mediante analisi interprocedurale, sarà in grado di determinare lo schema più appropriato per ogni chiamata di funzione. Se è specificata l'opzione *--callee-saves*, la lista delle funzioni elencate nella direttiva viene posta in coda a quelle dell'opzione.

Le direttive `#pragma` sono fatte per disattivare certe ottimizzazioni che possono causare un eccesso di uso di stack o memoria dati per contenere le variabili temporanee. Questo è abbastanza frequente nelle funzioni grandi.

Le direttive `#pragma` dovrebbero essere usate come nel seguente esempio, per controllare opzioni e ottimizzazioni per una data funzione. Le direttive devono essere poste prima e/o dopo le funzioni; ponendole dentro il corpo di una funzione, si possono avere risultati inattesi.

```
#pragma SAVE   /* salva i settaggi correnti */
#pragma NOGCSE /* disabilita l'eliminazione delle sub-espressioni
*/
```

```
#pragma NOINDUCTION /* disabilita le ottimizzazioni indotte */

int foo ()
{
    ...
    /* funzione di grosse dimensioni */
    ...
}
#pragma RESTORE /* riattiva le ottimizzazioni */
```

Il compilatore emetterà un messaggio di warning quando avverrà un eccesso di allocazione di memoria. E' fortemente raccomandato l'uso delle direttive `SAVE` e `RESTORE` quando si cambiano le opzioni di ottimizzazione di una funzione.

4.3 Routines di Libreria

NOTA: questa sezione è piuttosto incompleta.

Qui di seguito vengono elencate le routine di libreria fornite con il compilatore.

stdio.h	Questa libreria contiene le seguenti funzioni:
printf	effettua una scrittura formattata nell'output standard
sprintf	effettua una scrittura formattata in una stringa
printf_small	versione semplificata di printf

Queste routine sono state sviluppate da Martijn van Balen
(balen@natlab.research.philips.com)

Ogni specificatore di formato è così composto: `%[flags][width][b|B|l|L]type`

I *flags* sono i seguenti:

- **-** allinea a sinistra l'argomento convertito nella larghezza di campo specificata.
- **+** numero preceduto dal segno + o -
- **spazio** numero preceduto da uno spazio se positivo, segno - se negativo

`width`

Specifica il numero minimo di caratteri per il formato di numeri o stringhe

- Per i numeri, se necessario vengono aggiunti spazi a sinistra. Se *width* inizia con uno 0, verranno usati degli zeri anzichè gli spazi.
- Per le stringhe, vengono aggiunti spazi a sinistra (o a destra, se specificato il flag '-') se necessario.

`b|B`

Argomento byte (usato con d, u, o, x, X)

`l|L`

Argomento long (usato con d, u, o, x, X)

`type`

d	numero decimale
u	numero decimale senza segno (unsigned)
o	numero ottale senza segno
x	numero esadecimale senza segno (0-9, a-f)
X	numero esadecimale senza segno (0-9, A-F)
c	carattere
s	stringa (puntatore generico)
p	puntatore generico (**I**:data/idata, **C**:code, **X**:xdata, **P**:paged)
f	floating point (deve ancora essere implementato)

La *printf_small* è una versione molto semplificata della *printf*. Supporta solo i seguenti formati:

formato	tipo uscita	tipo argomento
%d	decimale	short/int
%ld	decimale	long
%hd	decimale	char
%x	esadecimale	short/int
%lx	esadecimale	long
%hx	esadecimale	char
%o	ottale	short/int
%lo	ottale	long
%ho	ottale	char
%c	carattere	char
%s	carattere	_generic pointer

Questa routine utilizza lo stack molto intensamente (dovrebbe essere usato il parametro *--stack-after-data* quando si usa tale funzione), inoltre occupa circa 1kB di area programma. Si aspetta anche che sia presente una funzione esterna chiamata *putchar(char)* (questo può essere modificato). Quando si usa il formato *%s*, la stringa (puntatore) deve essere convertito (casted) in un puntatore generico a carattere:

```
printf_small("my str %s, my int %d\n",(char _generic *)mystr,myint);
```

stdarg.h

Contiene le definizioni delle seguenti macro da usare con liste di parametri variabili (notare che una funzione può avere una lista di parametri variabili solo se è *'reentrant'*):

```
va_list, va_start, va_arg, va_end
```

setjmp.h

Contiene le definizioni per le routine ANSI *setjmp()* e *longjmp()*. Notare che in questo caso, le routine devono essere usate da funzioni che utilizzano lo stesso banco di registri, altrimenti i risultati possono essere imprevedibili. Il buffer per il salto richiede 3 bytes (lo stack pointer e 2 bytes per l'indirizzo di ritorno) e può essere in qualsiasi area di memoria.

stdlib.h

Contiene le seguenti funzioni:

atoi(), atol().

string.h

Contiene le seguenti funzioni:

strcpy(), strncpy(), strcat(), strncat(), strcmp(), strncmp(), strchr(), strrchr(), strspn(), strcspn(), strpbrk(), strstr(), strlen(), strtok(), memcpy(), memcmp(), memset()

ctype.h

Contiene le seguenti funzioni:

iscntrl(), isdigit(), isgraph(), islower(), isupper(), isprint(), ispunct(), isspace(), isxdigit(), isalnum(), isalpha().

malloc.h

Queste routine sono state sviluppate da Dmitry S. Obukhov (dso@usa.net). Allocano memoria nella RAM esterna. Segue una descrizione di come usarle:

```
 // Esempio:
#define DYNAMIC_MEMORY_SIZE 0x2000
.....
unsigned char xdata dynamic_memory_pool[DYNAMIC_MEMORY_SIZE];
unsigned char xdata * current_buffer;
.....
void main(void)
{
  ...
  init_dynamic_memory(dynamic_memory_pool,DYNAMIC_MEMORY_SIZE);

  // Ora è possibile usare la malloc().
  ...
  current_buffer = malloc(0x100);
  ...
}
```

serial.h

Queste routine sono state sviluppate da Dmitry S. Obukhov (dso@usa.net). Sono routine lanciate da interrupt, usano un buffer circolare di 256 bytes, vogliono la presenza di RAM esterna. Consultare la documentazione in *SDCCDIR/sdcc51lib/serial.c*. Notare che questo header DEVE essere incluso nel file contenente la funzione *main()*.

ser.h

Sono routine seriali alternative sviluppate da Wolfgang Esslinger (wolfgang@WiredMinds.com); sono più compatte e veloci. Consultare la documentazione in *SDCCDIR/sdcc51lib/ser.c*.

ser_ir.h

Sono routine seriali alternative sviluppate da Josef Wolf (jw@raven.inka.de), e non utilizzano la RAM esterna

reg51.h

Contiene le definizioni standard dei registri dell'8051

float.h

Contiene routine floating point, quali:*min(), max()* e altre.

Tutte le routine di libreria sono state compilate usando *--model-small*, e sono tutte *non-reentrant*. Se volete usarle con il modello large o le volete far diventare *reentrant*, devono essere ricompilate con le opportune opzioni.

4.4 Interfacciamento con le Routine Assembler

4.4.1 Registri Globali usati per passare i parametri

Il compilatore utilizza sempre i registri globali *DPL*, *DPH*, *B* e *ACC* per passare il primo parametro a una routine. Dal secondo in poi, i parametri sono posti nello stack (per le routine *reentrant* o se è specificata l'opzione *--stack-auto*) oppure nella RAM interna o esterna (dipende dal modello di memoria utilizzato)

4.4.2 Routine Assembler non-reentrant

Nell'esempio che segue, la funzione cfunc() chiama una routine assembler asm_func(), passando due parametri:

```
extern int asm_func(unsigned char, unsigned char);

int c_func (unsigned char i, unsigned char j)
{
    return asm_func(i,j);
}

int main()
{
    return c_func(10,9);
}
```

La routine assembler è:

```
.globl _asm_func_PARM_2
.globl _asm_func
.area OSEG
_asm_func_PARM_2:
.ds      1
.area CSEG
_asm_func:
 mov a, dpl
 add a, _asm_func_PARM_2

 mov dpl, a
 mov dpl, #0x00
 ret
```

Notare che il valore di ritorno è posto in:

dpl per valori di un byte
dph+dpl per valori di due byte
b+dph+dpl per valori a tre bytes (puntatori generici)
Acc+b+dph+dpl per valori a quattro bytes

La convenzione per i nomi è:

```
_<nome_funzione>_PARM_<n>
```

dove *n* è il numero del parametro a partire dal primo (1) a sinistra. Il primo parametro è passato in:

dpl	per valori di un byte
dptr	per valori di due byte
b+dptr	per valori di tre byte
acc+b+dptr	per valori di quattro byte

Assemblare le routine in linguaggio macchina mediante il comando

```
asx8051 -losg asmfunc.asm
```

Poi compilare e linkare la routine assembler con il sorgente C mediante:

```
sdcc cfunc.c asmfunc.rel
```

4.4.3 Routine Assembler reentrant

In questo caso, i parametri successivi al primo sono passati nello stack, dove vi sono messi (push) da destra a sinistra. Pertanto, il secondo parametro sarà in cima allo stack.

```
extern int asm_func(unsigned char, unsigned char);

int c_func (unsigned char i, unsigned char j) reentrant
{
    return asm_func(i,j);
}

int main()
{
    return c_func(10,9);
}
```

La routine assembler è:

```
.globl _asm_func
_asm_func:
    push  _bp
    mov  _bp,sp
    mov  r2,dpl
    mov  a,_bp
    clr  c
    add  a,#0xfd
    mov  r0,a
    add  a,#0xfc
    mov  r1,a
    mov  a,@r0
    add  a,r2
    mov  dpl,a
    mov  dph,#0x00
    mov  sp,_bp
    pop  _bp
    ret
```

La procedura di compilazione e linking è la stessa; *_bp* è lo stack frame pointer ed è usato per calcolare l'offset nello stack dei parametri e variabili locali.

4.5 Stack Esterno

Lo stack esterno è posto all'inizio del segmento di RAM esterna, ed è ampio 256 bytes.

Quando si usa l'opzione *--xstack* in compilazione, i parametri e le variabili locali di tutte le funzioni *reentrant* sono poste in quest'area. Questa opzione è utilizzata per programmi con grande uso dello stack.

Quando si usa l'opzione *--stack-auto*, tutti i parametri e le variabili locali sono posti nello stack esterno (notare che le librerie di supporto devono essere ricompilate con la medesima opzione).

Il compilatore utilizza la porta P2 del processore 8051 per emettere il byte più significativo dell'indirizzo del segmento di RAM esterna, quindi, quando si usa lo stack esterno, NON si può usare questa porta per connettere dispositivi I/O.

4.6 Conformità allo standard ANSI

Deviazioni dallo standard ANSI:

- Le funzioni non sono sempre *reentrant*
- Nelle strutture non possono essere assegnati valori direttamente, non possono essere passate come parametri di funzioni o assegnate con altre, e non possono essere un valore di ritorno da una funzione:

```
  struct s { ... };
  struct s s1, s2;
  foo()
  {
    ...
     s1 = s2 ; /* non è consentito in SDCC, ma valido in ANSI C
*/
    ...
  }
  struct s foo1(struct s parms) /* non è consentito in SDCC, */
                               /* ma valido in ANSI C */
  {
    struct s rets;
    ...
    return rets;  /* non è consentito in SDCC, */
                 /* ma valido in ANSI C */
 }
```

- Non sono supportati i **longlong** (interi a 64 bit)
- Non sono supportati i floating point in doppia precisione **double**
- Al momento, non sono supportate *setjmp* né *longjmp*
- La dichiarazione di funzioni nel vecchio stile K&R non è supportata:

```
  foo(i,j) /* questo vecchio stile di dichiarazione delle */
  int i,j; /* funzioni è valido in ANSI C ma non in SDCC */
  {
      ...
  }
```

- Le funzioni dichiarate come pointer ne devono avere un riferimento esplicito nella chiamata:

```
  int (*foo)();
  ...

  (*foo)(); /* deve essere chiamata in questa maniera, ma */
            /* ANSI C permette le chiamate come 'foo()'   */
```

4.7 Cyclomatic Complexity

La complessità ciclomatica (cyclomatic complexity) di una funzione è definita come il numero di percorsi indipendenti che il programma può assumere durante l'esecuzione della funzione. Questo numero è importante, poichè definisce il numero di casi di test disponibili per rendere valida una funzione.

Lo standard industriale per tale numero è 10; se il valore trovato da SDCC è maggiore di 10, dovreste pensare a una semplificazione della logica della funzione.

Notare che il livello di complessità non è proporzionale al numero di linee di codice componenti il corpo di una funzione: funzioni grandi possono avere complessità bassa, come piccole funzioni possono avere una elevata complessità.

SDCC usa la seguente formula per calcolare la complessità:

complessità = (numero di angoli nel grafico del controllo di flusso) - (numero di nodi nel grafico del controllo di flusso) + 2;

Avendo detto che lo standard è 10, dovete fare attenzione che talvolta è inevitabile avere complessità maggiori. Per esempio, se avete una *switch* con più di 10 label *case*, ciascuno dei *case* aggiunge 1 al livello di complessità.

Il livello di complessità non è una misura assoluta della complessità di un algoritmo, ma comunque fornisce una buona base di partenza per quelle funzioni che potrebbero essere meglio ottimizzate.

5 CONSIGLI

Vengono ora illustrate alcune linee guida che possono aiutare il compilatore a generare codice più efficiente. Alcuni consigli sono mirati al nostro compilatore, altri sono una buona pratica di programmazione.

- Utilizzate il più piccolo tipo di dato necessario per contenere il vostro valore. Se si sa in partenza che un valore sarà sempre inferiore a 256, utilizzate un **char** anziché uno **short** o un **int**.

- Utilizzate il tipo **unsigned** se sapete in partenza che il valore non diventerà mai negativo. Questo è particolarmente utile se dovete effettuare divisioni o moltiplicazioni.

- NON effettuare mai salti all'interno di un loop.

- Dichiarare le variabili come locali quando possibile, specialmente le variabili di controllo dei loop (induzione).

- Dal momento che il compilatore non effettua alcuna implicita promozione ad intero, il programmatore deve usare un esplicito cast quando ciò è necessario.

- Ridurre la dimensione delle operazioni di divisione, moltiplicazione e modulo può ridurre sostanzialmente le dimensioni del codice. Per esempio:

```
foobar(unsigned int p1, unsigned char ch)
{
    unsigned char ch1 = p1 % ch ;
    ....
}
```

  Per l'operazione di modulo, la variabile *ch* viene prima di tutto promossa a **unsigned int**, poi viene effettuata l'operazione (questo provoca una chiamata alla routine di supporto *_muduint()*), e il risultato viene poi trasformato in **int**. Se il programma viene modificato:

```
foobar(unsigned int p1, unsigned char ch)
{
    unsigned char ch1 = (unsigned char)p1 % ch ;
    ....
}
```

  ciò riduce sostanzialmente il codice generato (future versioni del compilatore saranno abbastanza furbe da individuare tali opportunità di ottimizzazione).

5.1 Note sulla gestione della memoria degli MCS51

La famiglia degli 8051 ha un minimo di 128 bytes di memoria interna così strutturata:

Bytes 00-1F - 32 bytes per memorizzare 4 banchi di registri da R0 a R7
Bytes 20-2F - 16 bytes per memorizzare 128 variabili a bit
Bytes 30-7F - 60 bytes per uso generale

Normalmente, il compilatore utilizza solo il primo banco di registri, ma è possibile specificare che altri banchi di registri siano usati dalle routine di interrupt. Per default, il compilatore pone lo stack dopo l'ultimo banco di registri usato; p.e. se si usano i primi due banchi, lo stack inizierà dalla locazione interna 0x10. Questo implica che, come usiamo lo stack, questo userà i restanti banchi di registri, poi i 16 bytes usati per l'indirizzamento al bit (128 bit), e dopo i 60 bytes general purpose.

Generalmente, il compilatore usa i 60 bytes alti per memorizzare i dati "vicini" ("near"); si può anche dichiarare che alcune variabili locali siano memorizzati in quest'area.

Se si usa una delle 128 variabili a bit o si utilizzano indirizzi dell'area dei 60 byte, bisogna porre attenzione che lo stack non cresca troppo tanto da sovrascrivere tali aree. Non c'è alcun controllo in esecuzione per evitare tutto questo.

La quantità di stack che sarà usata dipende dall'uso che verrà fatto dello stack interno per salvare i registri prima della chiamata delle funzioni (*--stack-auto* dichiara di porre i parametri e le variabili locali nello stack) e dal livellodi nidificazione delle stesse.

Se notate che lo stack sta sovrascrivendo i vostri dati, potete agire così:

- L'opzione *--xstack* dichiara di usare lo stack esterno in XRAM per salvare i registri (e anche i parametri e le variabili locali se si specifica anche *--stack-auto*). Tuttavia ciò produce un codice più grosso ed è più lento in esecuzione.

- L'opzione *--stack-loc* permette di specificare l'inizio dello stack; per esempio, per far iniziare lo stack sicuramente dopo altri dati. Tuttavia ciò può provocare lo spreco di memoria inutilizzata dai banchi dei registri vuoti, e se la quantità di dati "near" aumenta, si può intaccare il fondo dello stack.

- L'opzione *--stack-after-data* è simile alla precedente, ma pone automaticamente lo stack al termine delle variabili "near". Di nuovo, ciò può provocare lo spreco di memoria inutilizzata dai banchi dei registri vuoti.

- L'opzione *--data-loc* permette di specificare l'indirizzo di partenza dell'area "near". Può essere usata per spostare tale area in avanti, abbastanza da permettere allo stack di crescere liberamente. Funziona solamente se non sono usate variabili al bit, in modo che tale spazio sia usato dallo stack.

Concludendo, se trovate che lo stack sovrascrive le vostre aree dati, l'approccio che utilizza al meglio la memoria interna è quello di porre i dati dopo l'ultimo banco di registri usati o, se usate variabili al bit, dopo l'ultima variabile al bit utilizzando la *--data-loc*; per esempio, se sono usati due banchi di registri e nessuna variabile al bit, potete usare le opzioni *--data-loc 16* e la *--stack-after-data.*

Se utilizzate le variabili al bit, un'altra possibilità è di tentare di comprimere l'area dati nella zona dei banchi di registri inutilizzati, e di far iniziare lo stack dopo l'ultima variabile al bit.

6 ADATTAMENTO AD ALTRE MCU

Le operazioni di adattamento del compilatore ad altre MCU sono troppe per essere descritte in questo manuale. Quella che segue è una breve descrizione di ciascuno dei sette passi del compilatore e le loro dipendenze dalla MCU.

- Analisi sintattica del sorgente e costruzione dell'albero sintattico.
  Questa fase è largamente indipendente dalla MCU (eccetto per le estensioni di linguaggio). In questa fase viene effettuato anche il controllo semantico, come anche alcune ottimizzazioni iniziali, quali la risistemazione delle label e la verifica della corrispondenza dei pattern (come le sequenze di rotazione dei bit, ecc.).

- Generazione di codice intermedio per le fasi successive.
  Questa fase è completamente indipendente dalla MCU. Tale generazione del codice suppone che la macchina abbia un numero illimitato di registri, e li identifica con il nome *iTemp*. Il compilatore può emettere una versione leggibile di tale codice intermedio mediante l'uso dell'opzione *--dumpraw*.

- Poi c'è il grosso delle ottimizzazioni. E' indipendente dalla MCU e può essere suddivisa in altre sotto fasi:

  - Scomposizione del codice intermedio (iCode) in blocchi elementari
  - Analisi del flusso di controllo e dati dei blocchi elementari
  - Eliminazione delle sotto espressioni locali e poi quelle globali
  - Eliminazione del codice inutile (dead code)
  - Ottimizzazione dei loop
  - Se l'ottimizzazione dei loop effettua cambiamenti, effettua di nuovo le eliminazioni delle sotto espressioni

- Determinazione del range di vita
  Si intendono le variabili iTemp definite dal compilatore che sono sopravvissute alle ottimizzazioni. Tale analisi è essenziale alla allocazione dei registri, dato che questi calcoli definiranno quali variabili iTemp saranno assegnate ai registri e per quanto tempo.

- Allocazione dei registri
  Questo processo avviene in due fasi:
  La prima è chiamata impacchettamento dei registri (per la mancanza di un temine migliore). Per ridurre la "pressione" sui registri, vengono utilizzate molte espressioni dipendenti dalla MCU.

  La seconda parte, un po' più indipendente dalla MCU, alloca i registri alle rimanenti variabili iTemp in vita. Parecchio codice dipendente dalla MCU generato da questa fase è dovuto all'esiguo numero di registri indice dell' MCS51.

- Generazione del codice
  Sfortunatamente, questa fase è interamente dipendente dalla MCU, e solo una piccolissima (se non nessuna) parte del codice può essere riutilizzata da una diversa MCU. Tuttavia, può essere riutilizzato lo schema per allocare un operando assembler armonizzato per ogni operando iCode.

- Ottimizzazione peep-hole
  Come già detto, tale ottimizzazione è basata su regole, che possono essere riprogrammate per altre MCU.

7 SDCDB - Source Level Debugger

SDCC è distribuito con un debugger del sorgente. Il debugger utilizza una interfaccia a riga di comando; la lista dei comandi è stata mantenuta il più possibile simile a quella del *gdb* (GNU Debugger). La configurazione e il processo sono parte della installazione standard del compilatore, che installa anche il debugger nella directory specificata.

Lo SDCDB consente di eseguire il debug sia a livello di sorgente C che di sorgente Assembler.

7.1 Compilare per il Debug

L'opzione di debug deve essere specificata per tutti i file di cui si vuole vengano generate le informazioni per il debug. Il compilatore genera un file **.cdb** per ognuno dei file specificati. Il linker aggiorna il file **.cdb** con le informazioni sugli indirizzi; questo è il file che sarà poi utilizzato dal debugger.

7.2 Come Funziona il Debugger

Quando si specifica l'opzione *--debug*, il compilatore genera delle informazioni aggiuntive sui simboli, alcune delle quali sono inserite nel sorgente assembler, altre nel file **.cdb**, che verrà poi completato dal linker con gli indirizzi dei simboli. Utilizza il Simulatore Daniel's S51 per eseguire il programma, la cui esecuzione è controllata dal debugger. Quando si esegue un comando per il debugger, questo viene tradotto in un appropriato comando per il simulatore.

7.3 Avvio del Debugger

Il debugger viene lanciato mediante la seguente riga di comando (supponiamo che il file da debuggare abbia nome *foo*):

```
sdcdb foo
```

Il debugger quindi cercherà i seguenti files:

`foo.c` - file sorgente;

`foo.cdb` - file con le informazioni sui simboli;

`foo.ihx` - file oggetto in formato intel hex.

7.4 Opzioni della Riga di Comando

--directory=<directory_file_sorgente> Questa opzione può essere usata per specificare la o le directory in cui effettuare la ricerca dei files sorgenti, **.cdb** e **.ihx**. Gli elementi della lista di directory devono essere separati da '**:**'.
Per esempio, se i file sorgenti possono essere nelle directory /home/src1 e /home/src2, in tal caso si dovrà specificare `--directory=/home/src1:/home/src2`. Notare che non vanno messi gli spazi.

-cd <directory> Passa alla directory specificata.

-fullname Utilizzata dai front end GUI (Graphic User Interface)

-cpu <tipo-cpu> L'argomento è passato al simulatore; fare riferimento al relativo manuale per i dettagli.

-X <frequenza-di-clock> L'argomento è passato al simulatore; fare riferimento al relativo manuale per i dettagli.

-s <serial-port-file> L'argomento è passato al simulatore; fare riferimento al relativo manuale per i dettagli.

-S <serial in,out> L'argomento è passato al simulatore; fare riferimento al relativo manuale per i dettagli.

7.5 Comandi del Debugger

Come accennato in precedenza, l'interfaccia dei comandi del debugger è stata deliberatamente mantenuta il più simile possibile al debugger GNU **gdb**. Questo permette un più facile integrazione con le interfacce grafiche esistenti (come *ddd*, *xxgdb*, o *xemacs*) per il debugger GNU (se avete linux o unix, n.d.t.).

7.5.1 `break [linea | file:linea | funzione | file:funzione]`

Imposta il breakpoint alle specificate linee o funzioni:

sdcdb>break 100
sdcdb>break foo.c:100
sdcdb>break funcfoo
sdcdb>break foo.c:funcfoo

7.5.2 `clear [line | file:line | function | file:function]`

Cancella il breakpoint alle specificate linee o funzioni:

sdcdb>clear 100
sdcdb>clear foo.c:100
sdcdb>clear funcfoo
sdcdb>clear foo.c:funcfoo

7.5.3 `continue`

Continua nel debug del programma dopo un breakpoint.

7.5.4 `finish`

Continua nel debug del programma fino alla fine della funzione corrente.

7.5.5 `delete [n]`

Elimina il breakpoint numero **n**. Se utilizzato senza numero, elimina tutti i breakpoint definiti dall'utente.

7.5.6 `info [break | stack | frame | registers]`

info break - elenca tutti i breakpoints
info stack - mostra lo stack delle chiamate di funzioni
info frame - mostra le informazioni sul frame attualmente in esecuzione.
info registers - mostra il contenuto di tutti i registri.

7.5.7 `step`

Esegui il programma fino alla prossima riga di sorgente.

7.5.8 `next`

Esegui il programma fino alla prossima chiamata di subroutine.

7.5.9 `run`

Inizia l'esecuzione del programma.

7.5.10 `ptype variabile`

Stampa le informazioni di tipo della *variabile*.

7.5.11 `print variabile`

Stampa il valore della *variabile*.

7.5.12 `file nome_file`

Carica il file specificato. Questo è un metodo alternativo per caricare il file per il debug.

7.5.13 `frame`

Stampa le informazioni sul frame corrente.

7.5.14 `set srcmode`

Alterna tra sorgente in C e sorgente in Assembler.

7.5.15 `! comando-simulatore`

Invia la stringa che segue '**!**' al simulatore, e ne mostra la relativa risposta. Notare che il debugger non interpreta il comando inviato al simulatore, quindi se un comando come 'go' è inviato al simulatore, il debugger può perdere il controllo dell'esecuzione e può mostrare valori errati.

7.5.16 `quit`

"Watch me now. I am going Down. My name is Bobby Brown"
(Questa frase l'ho riportata pari pari come appare nell'originale |-), n.d.t.)

7.6 Interfacciamento con XEmacs

Vengono forniti due files (in emacs lisp) per l'interfacciamento con XEmacs: *sdcdb.el* e *sdcdbsrc.el*. Questi files possono essere trovati nella ditectory *<installdir>/bin* dopo il completamento dell'installazione. Devono essere caricati in Xemacs per far sì che l'interfaccia funzioni. Ciò può essere fatto allo start-up di Xemacs inserendo nel vostro file '.xemacs' (trovabile nella vostra directory HOME) quanto segue:

```
(load-file sdcdbsrc.el)
```

.xemacs è un file lisp, quindi le parentesi () che circondano il comando sono necessarie. Il file può essere caricato anche dinamicamente mentre Xemacs è in esecuzione: impostate la variabile di sistema `'EMACSLOADPATH'` alla directory di installazione *installdir>/bin*, poi immettete il comando *ESC-x load-file sdcdbsrc*. Per far partire l'interfaccia, immettete il seguente comando:

```
ESC-x sdcdbsrc
```

Vi verrà richiesto di immettere il nome del file da debuggare.

Le opzioni della riga di comando che sono passate direttamente al simulatore sono limitate ai valori di default contenuti nel file *sdcdbsrc.el*. Le variabili sono elencate qui di seguito, e i valori possono essere modificati:

```
sdcdbsrc-cpu-type '51

sdcdbsrc-frequency '11059200

sdcdbsrc-serial nil
```

Quella che segue è una mappatura dei tasti per l'interfaccia del debugger:

```
;; Current Listing ::
;;Tasto    Associazione          Commento
;;-----    ------------          --------
;;
;; n       sdcdb-next-from-src   Prossimo comando SDCDB
;; b       sdcdb-back-from-src   Comando precedente SDCDB
;; c       sdcdb-cont-from-src   Comando continua SDCDB
;; s       sdcdb-step-from-src   Comando step SDCDB
;; ?       sdcdb-whatis-c-sexp   Comando  ptype  per  i  dati  nel
buffer
;; x       sdcdbsrc-delete       Senza argomenti, cancella tutti i
;;                               breakpoints, o quelli specificati
;;                               in delete arg (C-u arg x)
;; m       sdcdbsrc-frame        Senza argomenti, mostra il frame
;;                               corrente, o quello specificato
;; !       sdcdbsrc-goto-sdcdb   Vai al buffer di output di SDCDB
;; p       sdcdb-print-c-sexp    Stampa i dati nel buffer
;; g       sdcdbsrc-goto-sdcdb   Vai al buffer di output di SDCDB
;; t       sdcdbsrc-mode         Alterna  il  modo  Sdcdbsrc  (lo
spenge)
;; C-c C-f sdcdb-finish-from-src Comando di fine SDCDB
;; C-x SPC sdcdb-break           Imposta il break per la linea
;; ESC t   sdcdbsrc-mode         Alterna il modo Sdcdbsrc
;; ESC m   sdcdbsrc-srcmode           Alterna il modo listing
;;
```

8 ALTRI PROCESSORI

8.1 Port per Z80 e gbz80

SDCC può compilare sorgenti sia per lo Zilog Z80 che per il Nintendo Gameboy Z80 (gbz80). Il porting è incompleto: il supporto per il tipo **long** è incompleto (non sono implementati mul, div e mod), e manca il supporto sia per i **float** che per i campi di bit. Escludendo queste mancanze, il codice generato è corretto.

Come sempre, i sorgenti di esempio sono di riferimento (vedi *z80/ralloc.c* e *z80/gen.c*). La gestione dello stack è simile a quella generata dal compilatore IAR per Z80. Il registro IX è utilizzato come base pointer, HL come registro temporaneo, BC e DE sono disponibili per memorizzare le variabili. Il registro IY è attualmente inutilizzato. Un effetto collaterale nell'uso di IX come base pointer è che lo stack delle funzioni è limitato a 127 bytes (ciò verrà rimediato in una successiva versione).

9 SUPPORTO

SDCC è cresciuto, divenendo un progetto piuttosto grande. Il compilatore da solo (senza preprocessore, assembler e linker) è composto da circa 40.000 linee di codice (eliminate le righe vuote). La natura Open Source di tale progetto è una base per la sua continua crescita e supporto. Potete usufruire del supporto di tantissimi sviluppatori e utilizzatori finali.

SDCC è perfetto? No, ed è per questo che abbiamo bisogno del vostro aiuto. Gli sviluppatori sono orgogliosi di eliminare i bug segnalati. Potete aiutarci anche solo segnalando eventuali bug o consigliando altri utenti di SDCC. Ci sono molti modi per contribuire, e noi vi incoraggiamo a prendere parte nel rendere SDCC un grande software.

9.1 Segnalazione dei Bugs

Inviate una e-mail alla mailing list all'indirizzo user-sdcc@sdcc.sourceforge.net o devel-sdcc@sdcc.sourceforge.net. I bug segnalati saranno corretti quanto prima. Quando segnalate un bug, è molto utile allegare un piccolo programma di test che riproduce il problema. Se riuscite a isolare il problema guardando il codice assembler generato, questo è ancora più utile. Compilare il sorgente con l'opzione *--dumpall* può essere utile nell'individuazione di problemi di ottimizzazione.

10 RICONOSCIMENTI

Sandeep Dutta (sandeep.dutta@usa.net)
SDCC, compilatore, generatore di codice per MCS51, debugger, porting per AVR

Alan Baldwin (baldwin@shop-pdp.kent.edu)
Versione iniziale di ASXXXX e ASLINK

John Hartman (jhartman@compuserve.com)
Porting di ASXXXX e ASLINK per MCS51

Dmitry S. Obukhov (dso@usa.net)
Routines malloc e i/o seriale

Daniel Drotos (drdani@mazsola.iit.uni-miskolc.hu)
Per il suo simulatore freeware

Malini Dutta (malini_dutta@hotmail.com)
Moglie di Sandeep, per la pazienza e il supporto

Sconosciuto
Per il preprocessore C GNU

Michael Hope
Porting per Z80 e gbz80, sviluppo per 186

Kevin Vigor
Porting per il Dallas DS80C390

Johan Knol
Molte correzioni e migliorie, librerie DS390/TINI

Scott Datallo
Porting per PIC

Andrea Contrucci (andrea@poloitalia.com)
Per la presente traduzione in italiano.

Grazie inoltre a tutti gli altri sviluppatori volontari che hanno contribuito con codice, test, creazione delle pagine-web, set di distribuzione, ecc. Voi sapete chi siete :-)

Questo documento è stato inizialmente redatto da Sandeep Dutta