StormC2.0

COLLABORATORS			
	TITLE : StormC2.0		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		April 16, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

StormC2.0

1.1 StormC.guide

```
                Miglioramenti in StormC V2.0

  Software e Documentazione
© 1996 by HAAGE & PARTNER Computer GmbH

  Postfach 80
  61188 Rosbach
  Germany

  Tel: +49 6007/930050
  Fax: +49 6007/7543


Sommario


                Gestione progetti
                 Script di make in ARexx


                Il profiler
                 Più controllo con il profiler


                Conversioni
                 Portare codice dal SAS/C allo StormC
```

1.2 StormC.guide/STC_Sort

```
Gli script di make nella gestione dei progetti di StormC

Le regole riguardanti un "make" sono in sostanza molto semplici. Per
```

prima cosa, tutti i file componenti un progetto vengono controllati per verificare se è necessario ricompilarli.

Nel caso dei sorgenti C, ciò si riduce a confrontare le date dei file oggetto e di debug con quelle del sorgente e di ogni file header da esso incluso attraverso #include. Se almeno uno di questi ultimi è più recente dell'oggetto o del file di debug, il sorgente deve essere ricompilato.

Occorre ricompilare il sorgente anche se qualche header è stato cambiato da azioni esterne al compilatore. Questo è il caso, per esempio, di un header per la localizzazione generato da "CatComp".

Una volta determinati i file che devono essere ricompilati o ri-linkati, ciascuno di essi viene gestito inviando i comandi ARexx corrispondenti al compilatore StormC o al linker StormLink; tali comandi vengono quindi eseguiti in sequenza.

Gli script di make vengono usati quando si vogliano trattare file diversi dai consueti sorgenti C o Assembler.

La voce di menu "Scegli script di Make..." permette di indicare uno script ARexx per il progetto corrente oppure, se è selezionato il titolo di una sezione, per tutti i file in una determinata sezione. Attraverso questi script è possibile invocare "compilatori" esterni come, per esempio, CatComp per gestire i file di localizzazione in maniera automatica.

Questi script vengono invocati dal gestore di progetti tutte le volte che un file deve essere ricompilato. I file contenenti gli script devono avere estensione ".srx"; i file con questa estensione vengono posti nella sezione "ARexx" dal gestore di progetti.

La voce di menu "Rimuovi script di make" elimina lo script da una particolare voce o da tutti i componenti di una sezione.

Le regole che stabiliscono se un file dotato di script di make debba essere ricompilato o meno sono essenzialmente analoghe a quelle che abbiamo discusso per il caso dei sorgenti C.

In ogni caso, un file viene sempre ricompilato alla prima occasione dopo l'aggiunta di uno script di make a esso associato.

Come esempio di script di make, qui di seguito viene commentato lo script "catcomp.srx":

```
/*

Gli argomenti passato allo script sono il nome del file (ovvero il
path del componente del progetto) e il path "base" del progetto
stesso. Entrambi sono racchiusi fra virgolette per proteggere gli
eventuali spazi presenti nei path.

La lista degli argomenti del comando PARSE deve sempre essere
terminata da un punto, in modo che eventuali altri argomenti passati
da versioni future del compilatore vengono ignorati.
```

```
*/

PARSE ARG '"' filename '"' '"' projectname '"' .

/*

Il nome del file oggetto viene ottenuto dall'argomento filename. Non
necessariamente questo file "oggetto" deve essere frutto di un link,
e il nome del file non deve necessariamente terminare per ".o"; in
objectname mettiamo semplicemente il nome del file risultante. Nel
caso di CatComp, il risultato della compilazione è in realtà un file
header.

*/

objectname = LEFT(filename,LASTPOS('.cd',filename)-1)||".h"

/*

Tutto l'output viene inviato a una finestra di console.

*/

SAY ""
SAY "Catcomp Script c1996 HAAGE & PARTNER GmbH"
SAY "Compile "||filename||" to header "||objectname||"."

/*

Affinché il gestore di progetto possa determinare correttamente quali
file devono essere compilati di volta in volta, è necessario
informarlo della relazione che esiste fra il file "sorgente" e quello
"oggetto" corrispondente. In assenza di questa informazione, lo
script di make verrebbe invocato ad ogni compilazione.

È possibile indicare fino a un massimo di due file "oggetto", in
questo modo:

   OBJECTS filesorgente fileoggetto1 fileoggetto2

Con questo comando, il gestore di progetto manterrà l'associazione
fra i vari file e potrà evitare le ricompilazioni non necessarie.

Vedi anche lo script "StormC:rexx/phxass.src".

Il comando OBJECTS non deve essere usato se lo script di make si
limita a richiamare un assemblatore per i file nella sezione
"Sorgenti ASM" - in questo caso, i nomi dei file oggetto vengono
ricavati automaticamente.

*/

OBJECTS filename objectname

/*

A questo punto richiamiamo il programma vero e proprio. Eventuali
```

```
messaggi d'errore saranno visualizzati nella finestra di console.

*/

ADDRESS COMMAND "catcomp "||filename||" CFILE "||objectname

/*

Poiché CatComp crea un file header, è consigliabile inserirlo
nell'appropriata sezione del progetto. Il parametro QUIET evita
messaggi d'errore nel caso che l'header fosse già incluso nel
progetto.

*/

ADDFILE objectname QUIET

/* Fine dello script di make! */
```

Praticamente ogni script di make sarà una variante di quello appena visto. Un altro comando che può essere utile in alcuni casi è

```
   DEPENDENCIES file file1 file2 file3 ...
```

Questo comando dichiara che un componente del progetto dipende da ulteriori file; le date di questi ultimi verranno controllate per decidere se è necessario invocare lo script. Il componente del progetto viene sempre controllato, e non è necessario indicarlo usando questo comando. L'uso di DEPENDENCIES è indicato per i casi in cui lo script fa riferimento a file esterni (il compilatore StormC, per esempio, lo usa per dichiarare i file header inclusi da un sorgente con #include "abc.h"; notate che quelli inclusi con #include <abc.h> non vengono controllati).

L'impostazione di uno script di make viene ignorata per tutti i file della sezione "Sorgenti" (C); questi file vengono sempre gestiti direttamente dal compilatore StormC. Al contrario, la sezione "Sorgenti ASM" permette l'impostazione di script di make, ma in loro assenza gestisce i sorgenti assembler attraverso StormASM (che, a sua volta, invoca l'assemblatore PhxAss).

Argomenti per gli script di make

Gli script di make ricevono come argomenti il nome del file (ovvero path del componente del progetto) e il path "base" del progetto stesso. Entrambi sono racchiusi fra virgolette per proteggere gli eventuali spazi presenti nei path.

L'argomento successivo è un valore numerico che indica se i file oggetto devono essere scritti tutti in una sola directory:

0 indica che i file oggetto devono risiedere nella stessa directory in cui si trova il sorgente,

1 indica che i file oggetto devono risiedere in una directory specifica.

Il nome di questa directory (fra virgolette, come per gli altri path) viene passato come quarto argomento (indipendentemente dal valore dell'argomento precedente: il nome della directory è presente anche se essa non va usata, cioè il terzo argomento è 0).

La directory per i file oggetto è interessante solo per i programmi che generano effettivamente codice; script di make che generino sorgenti (come per "catcomp.src") scriveranno sempre i loro risultati nella stessa directory in cui si trova il file originale. Di conseguenza, soltanto gli script di make che abbiano a che fare con assemblatori o altri compilatori devono preoccuparsi di questi argomenti.

Gli script di make per i sorgenti assembler costituiscono in un certo senso un'eccezione. Essi ricevono infatti un terzo argomento addizionale: il nome del file oggetto che devono generare. Questo nome è già completo del path per la directory dei file oggetto, se necessario.

In ogni caso, la lista degli argomenti del comando PARSE deve essere terminata da un punto, in modo che eventuali altri argomenti passati da versioni future del compilatore vengono ignorati.

Un comando PARSE completo per script di make NON relativi a file assembler sarà dunque:

```
   PARSE ARG '"' filesrg '"' '"' progetto '"' usadirobj '"' dirobj '"' .
```

Mentre per i file assembler sarà:

```
   PARSE ARG '"' filesrg '"' '"' progetto '"' '"' fileobj  '"' usadirobj
'"' dirobj '"' .
```

Script di make predefiniti

La directory "StormC:Rexx" contiene un certo numero di script di make già definiti. Potete usarli per adattarli a diversi usi e situazioni.

Script per Assembler

Gli script di make per gli assemblatori non possono contenere comandi OBJECTS.

"phxass.srx"

Questo script traduce un file assembler usando l'assemblatore PhxAss. In realtà, questo script è superfluo in quanto la StormShell supporta direttamente il PhxAss, ma può essere utile per usare delle opzioni di assemblaggio diverse.

"oma.srx"

Questo script traduce un file assembler usando l'assemblatore OMA.

"masm.srx"

Questo script traduce un file assembler usando l'assemblatore MASM.

Altri script

"catcomp.srx"

Questo script traduce un catalogo per la localizzazione invocando il programma CatComp.

"librarian.srx"

Anche il gestore di librerie StormLibrarian può essere controllato attraverso script di make. Un componente del progetto nella sezione "Gestore Librerie" può essere caricato direttamente nello StormLibrarian con un doppio click del mouse, oppure si può creare la libreria statica semplicemente con un doppio click mentre si tiene premuto il tasto Alt. Se però un progetto deve creare continuamente librerie condivise, è raccomandabile l'uso di uno script di make. La lista dei file oggetto viene creata all'interno dello StormLibrarian come usuale, quindi lo script di make invoca lo StormLibrarian, che non solo genera il file contenente la libreria statica, ma dichiara la libreria come un oggetto (usando OBJECTS) e stabilisce le dipendenze da tutti i file che la compongono (usando DEPENDENCIES). Dopo la prima compilazione, il gestore di progetti saprà che la libreria deve essere create ex-novo ogni volta che uno dei sorgenti C o assembler che la compongono è stato ricompilato.

Naturalmente, la libreria verrà ricreata anche se la lista dei file oggetto che la compongono è stata modificata usando lo StormLibrarian.

"fd2pragma.srx"

Questo script di make traduce un file FD in un file header contenente le direttive "#pragma amicall" necessarie per una libreria condivisa. In condizioni normali, questo script non dovrebbe essere necessario, poiché lo StormLink crea automaticamente un file header aggiornato ogni volta che produce una libreria condivisa.

1.3 StormC.guide/STC_Sections

IL PROFILER

Un profiler è uno strumento indispensabile per ottimizzare i programmi. Le ottimizzazioni fatte da un compilatore possono migliorare le prestazioni di un programma solo fino a un certo punto, ma un profiler può fornire al programmatore le informazioni necessarie a identificare le parti del programma che richiedono il maggiore tempo di esecuzione. Con queste informazioni, è possibile

riscrivere quelle parti usando algoritmi migliori, o almeno velocizzarle ottimizzando il codice manualmente.

Il profiler dello StormC è particolarmente potente; permette infatti di ottenere temporizzazioni molto precise e fornisce statistiche preziose sulle prestazioni del programma.

Come sempre, ci siamo mantenuti fedeli alla nostra massima: non c'è bisogno di compilare una versione speciale del programma per usare il profiler. Le normali informazioni di debug, contenute in un file esterno, sono sufficienti a questo scopo e ciò, in aggiunta alla capacità di avviare il profiler durante il debug, è una caratteristica unica fra i compilatori per Amiga.

Se volete usare il profiler, occorre che il progetto sia stato compilato con l'opzione "Debug ridotto" o "Debug completo". Selezionate anche "Usa il profiler" nella finestra "Esecuzione programma", e avviate il programma normalmente. È anche possibile effettuare simultaneamente il debug e il profiling, ma ciò può introdurre piccole deviazioni nei tempi misurati dal profiler.

Dopo l'avvio del programma, è possibile aprire la finestra del profiler.

L'icona nell'angolo superiore sinistro aggiorna i risultati del profiler, mentre la linea degli aiuti mostra il tempo di esecuzione complessivo. Questo valore indica il reale tempo di occupazione della CPU, e quindi non include il tempo durante il quale il programma è fermo in attesa di segnali, messaggi, completamento di I/O ecc., né il tempo usato da altri programmi in esecuzione contemporanea.

La seguente lista illustra il significato delle informazioni visualizzate:

1. Nome della funzione (Funzione)

Il nome delle funzioni membro è mostrato secondo la sintassi dell'operatore di scope (nome della classe::nome della funzione).

2. Tempo d'esecuzione relativo (Tempo)

Questo valore indica soltanto il tempo che il programma impiega all'interno della funzione o all'interno di funzioni del sistema operativo chiamate direttamente da essa. Il tempo impiegato in invocazioni di altre funzioni non è conteggiato.

Questo tempo è il migliore indicatore di quali funzioni richiedono la maggior parte del tempo di esecuzione. La somma di tutti i valori in questa colonna è tipicamente del 99%-100% (il punto percentuale mancante può essere perso nel codice di startup o nell'accumulo di piccole inesattezze).

3. Tempo d'esecuzione relativo cumulativo (Cumulativo)

In questa colonna è mostrato il tempo richiesto dalla funzione e da tutte le funzioni che essa invoca. Il valore relativo a main() sarà dunque del 99% nei casi normali.

4. Tempo d’esecuzione assoluto (Totale)
5. Tempo d’esecuzione massimo (Massimo)
6. Tempo d’esecuzione minimo (Minimo)

Questi tre valori descrivono l’andamento delle invocazioni di ogni funzione. Il miglioramento di prestazioni ottenibile per una certa funzione varia in funzione del fatto che ogni invocazione richieda più o meno la stessa quantità di tempo (massimo e minimo simili), oppure che alcune invocazioni richiedano notevolmente più tempo delle altre (grande differenza fra i due valori). In quest’ultimo caso, può essere più efficiente ottimizzare questi casi particolari.

7. Numero di invocazioni (Chiamate)

A volte una funzione copre una grande quantità del tempo totale d’esecuzione semplicemente perché è chiamata molto spesso, anche se ogni invocazione è in realtà molto veloce. Ottimizzare funzioni simili è spesso estremamente arduo; può essere utile allora dichiararle che funzioni inline ("__inline" in C, "inline" in C++).

Nella parte superiore della finestra sono presenti alcuni controlli:

La linea superiore (linea d’aiuto) mostra brevi descrizioni dei vari pulsanti.

Immediatamente sotto, a sinistra, si trovano tre pulsanti. Il primo, che abbiamo già incontrato, aggiorna i risultati del profiler.

Il secondo pulsante vi consente di salvare i dati visualizzati come testo ASCII; dopo averlo premuto apparirà un file requester in cui potrete indicare il nome del file da salvare.

Il terzo pulsante invia gli stessi dati alla stampante (usando il dispositivo "PRT:")

Nella parte destra di questa linea si trova un gadget ciclico che vi permette di stabilire l’ordine di visualizzazione delle funzioni. La prima voce, "Tempo", ordina la lista in base ai valori nella seconda colonna, mentre "Tempo cumulativo" la ordina in base alla terza colonna, "Alfabetico" segue l’ordine alfabetico dei nomi delle funzioni e "Numero chiamate" ordina la lista in base all’ultima colonna.

La riga successiva mostra un gadget stringa che vi permette di indicare un modello per i nomi delle funzioni (seguendo la sintassi di AmigaDOS). Il profiler visualizzerà soltanto le funzioni il cui nome corrisponde al modello indicato; ciò può essere utile per riportare a dimensioni ragionevoli liste di funzioni eccessivamente lunghe. Questa caratteristica può anche essere usata per visualizzare solo le funzioni membro di una certa classe, inserendo in questo campo il nome della classe seguito da "#?".

Alla destra di questo campo si trova un gadget di tipo numerico che vi consente di indicare un valore minimo (in percentuale) al di sotto del quale le funzioni non verranno visualizzate. In effetti, funzioni che occupano solo il 5% o il 10% del tempo di esecuzione sono

difficili da ottimizzare, e anche raddoppiando la loro velocità si otterrebbero risulati modesti (del 2.5 o 5% nel nostro caso).

In aggiunta a queste restrizioni esplicite, il profiler mostra soltanto funzioni che siano state chiamate almeno una volta durante l'esecuzione del programma.

Un doppio click sul nome di una funzione vi porterà direttamente all'interno dell'editor con il cursore pronto sul sorgente relativo.

La finestra del profiler è anche aperta e aggiornata automaticamente nel momento in cui il programma termina. Chiudendo la finestra di controllo del debugger, anche la finestra del profiler viene chiusa, e i risultati ottenuti vengono persi. Se vi interessa accedere in seguito a queste informazioni, accertatevi di averne salvata o stampata una copia prima di chiudere la finestra.

Informazioni tecniche sul profiler

Due codici operativi della serie LINE-A, $A123 e $A124, sono usati per marcare le chiamate di funzione.

Queste due istruzioni sono inutilizzate in tutti i processori della famiglia Motorola 68000, e causano un'eccezione; questa eccezione viene usata per aggiornare le statistiche sui tempi d'esecuzione e il numero di chiamate.

L'uso delle eccezioni ha come conseguenza una diminuzione relativa della velocità della CPU: in altri termini, il programma impiegherà più tempo che in condizioni normali. La differenza può anche essere sensibile se il programma invoca moltissime funzioni brevi. Il profiler è comunque più veloce e più accurato di molti altri prodotti analoghi esistenti per AmigaOS. Questa tecnica ha anche il vantaggio di non richiedere una compilazione speciale per poter effettuare il profiling.

La gestione della ricorsione è limitata: il tempo massimo e minimo di esecuzione saranno solitamente non affidabili, e il tempo totale di esecuzione (e, di conseguenza, i valori in percentuale) potrebbe essere errato. Un caso semplice di ricorsione diretta (in cui f() chiama f()) produrrà dei valori affidabili, ma la ricorsione mutua (in cui f() chiama g() e g() chiama f()) causerà l'accumulo del tempo totale su una delle due funzioni.

Chiamate non ricorsive all'interno di queste funzioni saranno comunque corrette.

L'effetto di chiamate a longjmp() è in generale impredicibile, ma nella maggior parte dei casi causerà soltanto lievi imprecisioni nelle statistiche della funzione invocata.

Teoreticamente, non tutte le funzioni possono essere misurate in questo modo: il profiler può accedere soltanto a quelle il cui codice inizia con un'istruzione LINK o MOVEM. In quasi tutti i casi, però, almeno una delle due è presente nel codice della funzione, anche dopo le ottimizzazioni più spinte, e fortunatamente le poche funzioni che

non fanno uso di queste istruzioni sono per necessità così piccole (nessuna variabile locale né registri all'infuori di D0, D1, A0 e A1) che la loro ottimizzazione sarebbe comunque pressoché impossibile.

Generalmente, le funzioni inline non possono essere misurate.

1.4 StormC.guide/STC_Project

Portare codice dal SAS/C allo StormC

Ci siamo fatti un punto d'onore di dotare il compilatore dello StormC di molte proprietà importanti del compilatore SAS/C, incluso il supporto a varie keyword e #pragma specifiche del SAS/C. Ciò nonostante, a seconda del vostro stile di programmazione, possono sorgere piccoli o grandi problemi quando si trasferisce codice dal SAS/C allo StormC.

Per prima cosa, dovreste tener presente che lo StormC è un compilatore ANSI-C e C++, mentre il SAS/C è un compilatore C e ANSI-C (il precompilatore C++ non si è mai prestato a un uso serio), e quindi prevede molti vecchi costrutti che lo StormC non accetta. Ciò può causare diversi problemi nella migrazione del codice, a meno che non siate abituati a usare il SAS/C nel modo ANSI strict (usando l'opzione ANSI del SAS/C).

Impostazioni di progetto

Innanzitutto, assicuratevi che il progetto che costruirete sui vostri sorgenti SAS/C sia impostato per la compilazione in ANSI-C.

Abilitate quanti più warning possibile, e modificate il vostro programma finché esso non produce più warning.

Anche nel caso di progetti in ANSI-C puro, un successivo passaggio al C++ è raccomandabile, e porterà diversi vantaggi: saranno richiesti i prototipi per tutte le funzioni, e le conversioni implicite da void * ad altri tipi puntatore non saranno più considerate legali.

Sebbene questa trasformazione possa richiedere una certa quantità di lavoro noioso (in particolare, l'incompatibilità dei puntatori void richiederà cambiamenti a molte chiamate a malloc() e ad AllocMem()), il risultato sarà una maggiore fiducia nella correttezza del programma.

I nomi lunghi dei simboli in C++ danno ulteriore sicurezza durante la fase di link: se la definizione di una funzione è in qualche modo inconsistente con il suo prototipo dichiarato altrove, il linker lo segnalerà interrompendo la generazione del codice e riportando un errore di "simbolo non definito".

Il passaggio al C++ vi darà anche l'occasione di estendere il vostro programma con i più moderni concetti sull'orientamento agli oggetti, nonché molte altre piccole migliorie del C++ (come la possibilità di dichiarare variabili in qualunque punto all'interno di un blocco di

istruzioni).

Sintassi

Alcune keyword del SAS/C non sono riconosciute dallo StormC, altre sono supportate bene, ma lo StormC le accetta solo nella loro sintassi "ufficiale" secondo lo standard ANSI.

Lo StormC accetta union anonime, ma non struct implicite. Strutture equivalenti non sono considerate identiche; se avete fatto uso di questa caratteristica del SAS/C, sarà necessario inserire dei cast nel vostro codice.

Se questa caratteristica è importante nel vostro progetto, potreste voler convertire il vostro programma in C++: le struct equivalenti non sono altro che un particolare aspetto dell'ereditarietà del C++ sotto mentite spoglie.

Il controllo dei tipi è molto più stretto nello StormC. Ciò è vero particolarmente nel caso del qualificatore const applicato ai parametri di una funzione. Per esempio:

```
typedef int (*ftype)(const int *);

int f(int *);

ftype p = f; // Errore!
```

Per prevenire simili errori, dovrete inserire i cast appropriati oppure (e questa soluzione è preferibile) scrivere le dichiarazioni corrette per le vostre funzioni. Tenete presente che il qualificatore const è un aiuto importante per assicurare la correttezza del vostro programma.

I commenti fino a fine riga del C++ ("//") vengono accettati anche nel modo ANSI-C, ma i commenti C annidati non lo sono. In ogni caso, potete abilitare un warning che vi segnali questi casi pericolosi.

I nomi di variabili non possono contenere caratteri accentati, né il segno del dollaro.

Keyword

In generale, è meglio evitare l'uso di keyword fuori standard, almeno per programmi che un giorno o l'altro potreste voler portare su un altro sistema operativo o un altro compilatore.

StormC fa un maggior uso della direttiva #pragma prevista dall'ANSI-C per adattare il software alle particolari caratteristiche dell'AmigaOS (per esempio, #pragma chip e #pragma fast).

Quando una keyword potrebbe non esistere su altri compilatori, ma non è comunque assolutamente necessaria, è preferibile far ricorso a delle macro:

```
#define INLINE __inline

#define REG(x) register __##x

#define CHIP __chip
```

Queste macro possono poi essere facilmente adattate a compilatori diversi.

Anche alcune keyword opzionali non riconosciute dallo StormC possono essere definite come macro:

```
#define __asm

#define __stdarg
```

Ecco una lista delle keyword tipiche del SAS/C e di come esse vengono interpretate dallo StormC:

__aligned

Non è supportata. Non c'è alcun modo semplice di sostituire questa keyword, ma fortunatamente il suo uso è raro.

__chip

Questa keyword forza l'allocazione di un dato in un hunk di memori chip nel file oggetto. Notate che questa dichiarazione, come tutti gli altri specificatori di classe d'allocazione e altri qualificatori, deve precedere il tipo nella dichiarazione:

```
__chip UWORD NormalImage[] = { 0x0000, .... }; // corretto

UWORD __chip NormalImage[] = { 0x0000, .... }; // errato
```

La seconda forma non viene accettata e non è consistente con la sintassi ANSI-C.

Nello StormC è preferibile l'uso di "#pragma chip" e "#pragma fast". Tenete presente che "__chip" modifica soltanto la classe d'allocazione di una singola dichiarazione, mentre "#pragma chip" modifica tutte le dichiarazioni fino a un successivo "#pragma fast".

__far e __near

Non sono supportate. Non c'è alcun modo semplice di sostituire queste keyword, ma fortunatamente il loro uso è raro.

__interrupt

Questa keyword è supportata esattamente come nel SAS/C.

__asm, __regargs, __stdarg

Non sono supportate né necessarie. Se volete che una funzione riceva i suoi parametri nei registri, dichiaratela con la keyword ANSI "register", o modificate le singole dichiarazioni dei parametri con

"register" o ancora con una specifica esatta del registro (per esempio, "register __a0"). In caso contrario, gli argomenti saranno passati nello stack.

__saveds

Viene gestita similmente alla "__saveds" del SAS/C. Questa keyword non ha alcun effetto quando è in uno il modello per i dati "far"; nel modello relativo ad A4 salva A4 sullo stack e ne carica il valore dal simbolo "__LinkerDB"; nel modello relativo ad A6 fa la stessa cosa per A6.

Non usate "__saveds" a cuor leggero; questa dichiarazione dovrebbe essere riservata esclusivamente a funzioni che saranno chiamate dall'esterno del vostro programma, per esempio un dispatcher per una classe BOOPSI.

Nella versione corrente del compilatore, è preferibile usare soltanto il modello "dati far" quando si creano librerie condivise. Tenete presente che il modello "dati near" sottrae un ulteriore registro all'ottimizzatore, lasciandogli soltanto i registri da A0 ad A3. Ciò può facilmente annullare il vantaggio del modello near se non fate grande uso di variabili globali.

__inline

Come per gli altri casi, questa keyword è accettata come uno specificatore di funzione.

Ciò vuol dire che il suo uso nella definizione di una funzione deve corrispondere a quanto dichiarato nel prototipo.

Se una funzione "__inline" deve essere chiamata da più moduli, la sua intera definizione (e non soltanto il prototipo) dovrebbe essere spostata in un file header.

__stackext

Non è supportata. Il controllo dello stack e la sua estensione automatica non sono disponibili nella versione corrente del compilatore.