Capitolul 7
Interfaţa ASM - limbaje de nivel înalt

O parte importantă a programării în limbaj de asamblare o constituie legarea modulelor de program ASM cu module de program dezvoltate în limbajde de nivel înalt. În acest mod, se poate creşte eficienţa aplicaţiilor, prin dezvoltarea modulelor critice (care se utilizează foarte des) în ASM.

În acest capitol va fi exemplificată dezvoltarea de aplicaţii mixte în limbaj de asamblare şi în limbajul C. Alegerea limbajului C ca limbaj reprezentativ de nivel înalt este justificată de următoarele proprietăţi:

- răspândirea foarte mare a limbajului C, ca linbaj destinat atât programelor de sistem cât şi celor aplicative;
- existenţa unor medii de dezvoltare evoluate, total compatibile cu asambloarele 8086 (un exemplu semnificativ fiind familia de produse Borland);
- caracterul modular al limbajului C, prin care se permite compilarea separată a modulelor sursă;
- compatibilitatea totală a modulelor obiect (un modul obiect rezultat în urma compilării unui text sursă C are exact acelaşi format ca un modul obiect rezultat în urma asamblării unui text ASM).

Se va utiliza deci mediul de dezvoltare Borland (compilatorul bcc, editorul de legături tlink, asamblorul tasm, respectiv mediul integrat bc).

7.1 Transferul parametrilor. Simboli externi

În mod concret, interfaţa dintre module C şi module ASM va consta din apeluri de funcţii ASM din C şi invers, respectiv din accesul din C la date definite în ASM şi invers.

Compilatorul Borland C realizează transferul parametrilor prin stivă, în ordinea de la dreapta la stânga a listei de parametri. Descărcarea stivei este făcută de către modulul apelant. Întoarcerea de tipuri simple de date din funcţii se realizează prin registrul acumulator, eventual extins (deci prin AL, AX, sau DX:AX, corespunzător unui tip pe 1, 2 sau 4 octeţi). Tipurile reale (float, double, long double) sunt întoarse printr-o zonă specială a bibliotecii de virgulă mobilă sau în vârful stivei coprocesorului matematic. Pentru a nu complica interfaţa, vom considera că funcţiile C folosite în ASM nu întorc valori reale. Dacă este absolut necesar, se poate adopta soluţia ca funcţiile C să întoarcă pointeri la variabile reale statice.

Un alt caz complicat este transferul structurilor şi uniunilor, care este definit în C prin copierea bit cu bit a tuturor membrilor, atât la transfer spre funcţie cât şi la întoarcere din funcţie. Este şi aici de preferat soluţia mult mai eficientă de a transfera sau întoarce un pointer către structura sau uniunea respectivă.

Numele simbolilor externi în C (funcţii şi variabile externe) sunt generate implicit cu caracterul _ (subliniere) ca prim caracter. De exemplu, simbolul var este generat ca _var şi vizibil ca atare într-un modul ASM. Pentru variabile, numele este sinonim cu adresa unde este memorată variabila.

Deoarece, în C, vizibilitatea externă este permisă numai variabilelor definite la nivel exterior (în afara tuturor funcţiilor), acestea sunt implicit alocate static,

deci au adrese fixe de memorie. Dacă, de exemplu, var este un întreg definit în C, atunci el poate fi citit în ASM printr-o instrucţiune de tipul mov ax, _var.

Trebuie cunoscute dimensiunile fizice ale tipurilor de bază din C. Acestea sunt: char (1 octet), int (2 octeţi), short (2 octeţi), long (4 octeţi), float (4 octeţi), double (8 octeţi), long double (10 octeţi), pointeri (2 sau 4 octeţi, funcţie de modelul de memorie folosit).

Trebuie ţinut seama şi de faptul că operaţiile cu stiva sunt operaţii pe 16 biţi, deci în cazul în care se transmite un char la o funcţie, se pune pe stivă un cuvânt de 16 biţi, cu partea mai semnificativă 0. Este recomandabil ca, asemenea funcţiilor din biblioteca C standard, variabilele de tip char să fie transmise şi întoarse la şi de la funcţii ca variabile de tip unsigned int, ceea ce asigură protecţie la o eventuală extensie de semn la 16 biţi. De exemplu, variabila char '\x81' devine, prin extensie de semn, întregul 0xff81, dar variabila unsigned char '\x81' devine întregul 0x0081. Extensia de semn poate apare dacă tipul char este implicit signed (ceea ce se întîmplă la Borland C). Există opţiune de compilare care face ca tipul char să fie implicit unsigned, dar este bine să se scrie programe care să nu depindă de această opţiune.

Un alt fapt care trebuie avut în vedere este că C-ul este un limbaj de tip case-sensitive, deci contează dacă identificatorii sunt scrişi cu litere mici sau mari. Trebuie forţată la TASM opţiunea /ml sau /mx care generează simbolii ţinând cont de acest fapt.

Să considerăm următoarea secvenţă C:

```
include <stdio.h>
int n = 5;
char sir[] = "Un sir";
void main(void)
{
        printf("%s %d\n", sir, n);
}
```

Presupunând modelul de memorie small (adică toate adresele sunt pe 2 octeţi), imaginea stivei la intrarea în funcţia printf este cea din Figura 7.1.

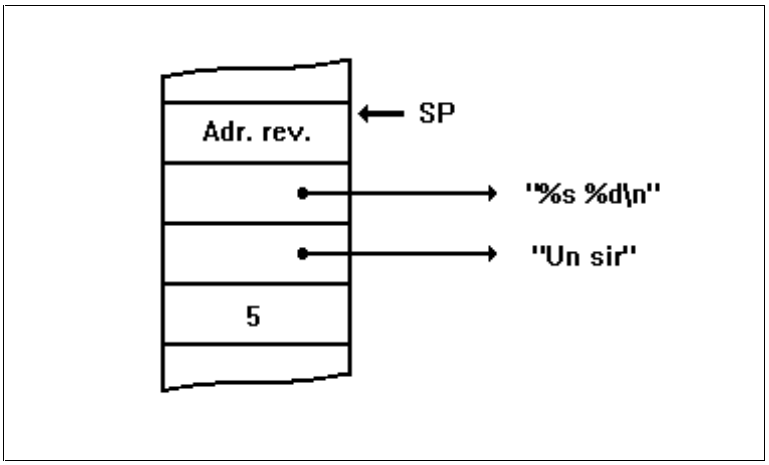

Figura 7.1 Imaginea stivei la un apel al funcţiei printf

Apelul lui printf din ASM se poate face cu secvenţa ASM de mai jos (echivalentă cu secvenţa C):

```
.model small
extrn   printf: near
.data
        n               dw      5
        sir             db      'Un sir', 0
        format          db      '%s %d\n', 0
.code
main    proc near
        push    n
        lea     ax, sir
        push    ax
        lea     ax, format
        push    ax
        call    near ptr printf
        add     sp, 6
        retn
main    endp
```

7.2 Dezvoltarea în mediu integrat sau prin linii de comandă

Sunt posibile două moduri de dezvoltare a aplicaţiei: în mediul integrat C sau prin linie de comandă.

În mediul integrat (lansat prin comanda bc) trebuie definit un proiect care să specifice explicit modulele în limbaj de asamblare. Definirea proiectului permite specificarea programului translator (Compiler sau Assembler).

Astfel, se poate lucra în regim de editare, compilare, depanare, etc., cu ambele tipuri de module (ASM şi C), mediul integrat recunoscând modulele scrise în ASM. Acest context este foarte util pentru depanare, deoarece toate facilităţile specifice mediilor integrate de dezvoltare pentru limbaje de nivel înalt (rulare pas cu pas, vizualizarea permanentă a unor variabile etc.), se aplică şi modulelor scrise în ASM.

O altă variantă de depanare o constituie Turbo Debugger-ul, care poate fi apelat direct sau din mediul integrat, acesta recunoscând, de asemenea, atât module ASM, cât şi module C.

Opţiunile necesare la compilarea programelor C (cea mai importantă fiind modelul de memorie), se stabilesc prin comanda **Options** a mediului integrat.

A doua posibilitate este compilarea, repectiv asamblarea separată a modulelor sursă, prin lansări ale compilatorului bcc şi ale asamblorului tasm de la consolă şi apoi legarea explicită a modulelor obiect, prin lansarea editorului de legături tlink. În acest caz, opţiunea de model de memorie pentrui modulele C se stabileşte în linia de apel a compilatorului (opţiunea -mx, unde x este un caracter ce identifică modelul). De asemenea trebuie precizată opţiunea -c (compile only), astfel încât compilatorul C să producă numai modul obiect. Tot în linia de comandă se precizează căile pentru fişierele header şi pentru biblioteci (opţiunile -I şi -L).

Când se lucrează în regim de linie de comandă, trebuie ţinut seama de structura unui program C. Pe lângă modulele definite de utilizator, trebuiesc adăugate modulul de iniţializare c0x (se pune pe prima poziţie în lista de module obiect la tlink) şi bibliotecile C (aflate în subdirectorul \lib), care se specifică după numele fişierului executabil. Atît modulul de iniţializare cât şi

bibliotecile sunt specifice modelului de memorie folosit, fiind identificate prin ultima literă a numelui fişierului respectiv.

Bibliotecile care trebuie specificate depind de funcţiile utilizate în modulele C. Practic, se va lega totdeauna bibioteca cx.lib şi, eventual, mathx.lib, emu.lib (pentru emularea operaţiilor în virgulă mobilă), fp87.lib (dacă sistemul are coprocesor), graphics.lib (dacă s-au folosit funcţii grafice). În specificarea numelor fişierelor care conţin modulul de iniţializare şi bibliotecile respective, caracterul x depinde de modelul de memorie folosit.

Este utilă o vizualizare a conţinutului subdirectorului \lib din implementarea Borland C, pentru a identifica bibliotecile C. Este de asemenea utilă vizualizarea opţiunilor posibile de compilare, asamblare şi link-editare, ca şi sintaxa acestor comenzi (se poate obţine prin bcc, tasm sau tlink urmate imediat de <Enter>).

În cazul în care modulul de program principal este scris în limbaj de asamblare, trebuie ţinut seama de faptul că main este o funcţie ca oricare alta. De fapt, modulul de program principal este modulul precompilat c0, care, după ce face o serie de iniţializări, apelează procedura main. Din acesată cauză, dacă funcţia main este scrisă în ASM, ea trebuie declarată ca o procedură şi specificată ca simbol public:

```
public _main
 _main proc
        ...
        ret
 _main endp
end
```

Procedura _main trebuie încheiată prin instrucţiunea ret, ca orice procedură, iar modulul de program respectiv nu trebuie să conţină etichetă de start. Încheierea execuţiei se va face la return-ul din _main, controlul revenind modulului c0, care a apelat _main şi care este răspunzător de ieşirea în sistemul de operare. Nu este necesară o ieşire în DOS din procedura _main. De asemenea, nu este necesară nici iniţializarea registrelor DS şi ES, deoarece este făcută în modulul c0. Modulul c0 defineşte şi o stivă minimă (dacă modelul nu este tiny), care se poate mări, dacă este necesar, prin directiva ASM .stack.

În cazul în care fuţia cmain este scrisă în C, modulele ASM vor conţine practic proceduri şi definiţii de date, care trebuie declarate publice (cu ajutorul directivei ASM public). De asemenea, dacă modulele ASM apelează funţii C, acestea trebuie declarate în ASM ca simboli externi (cu ajutorul directivei ASM extrn).

7.3 Modele de memorie

Modelele de memorie corespund organizării segmentelor de date, cod şi stivă, specifice adresării de la procesoarele INTEL. Atât asamblorul TASM cât şi compilatorul Borland C recunosc aceleaşi modele de memorie, generând segmente cu acelaşi nume. La TASM, acest lucru corespunde folosirii directivelor de definire simplificată a segmentelor. Această compatibilitate totală simplifică foarte mult efortul de dezvoltare al aplicaţiilor mixte.

În ASM, modelul se specifică prin directiva .model, iar în C prin opţiunea Options/Compiler/Code generation.../Model în mediul integrat sau prin opţiunea -mx în linia de comandă, la lansarea compilatorului.

Modelele disponibile sunt tiny, small, compact, medium, large şi huge. Funcţie de aceste modele, se vor genera apeluri de funcţii implicit de tip far sau near şi pointeri la funcţii sau la date implicit de tip far sau near.

Implementările Borland C permit definirea explicită a pointerilor şi funcţiilor de tip far sau near, dar acest lucru nu este recomandabil. Este indicat să fie lăsat compilatorul să genereze implicit pointeri şi apeluri de funcţii, conform modelului de memorie definit. Dacă modulele C includ fişierele header predefinite, în care se găsesc prototipurile pentru funcţiile de bibliotecă folosite, apelurile de funcţii şi legarea cu modulele de bibliotecă se vor realiza în mod corect.

În modulele ASM se va folosi acelaşi model de memorie, dar trebuie ţinut cont de tipurile pointerilor şi funcţiilor definite

sau folosite în modulele C, deoarece, în ASM, accesul la date, funcţii, adrese de segment, offset-uri, etc. se face explicit. Se recomandă ca, în ASM, să se folosească explicit apeluri de tip near/far, definiţii de proceduri explicite near/far şi return explicit de tip near/far, deşi asamblorul generează corect apeluri şi definiţii de proceduri implicite, pe baza modelului ales. Totuşi, pentru a scoate în evidenţă tipurile respective, este bine să se folosească variante explicite în ASM.

Nu se recomandă în nici un caz ca, în aceeaşi aplicaţie, module diferite de program să aibe modele diferite de memorie.

Portabilitatea modulelor C la schimbarea modelului de memorie nu necesită practic nici un efort de întreţinere. Portabilitatea modulelor ASM faţă de modelele de memorie este însă o problemă dificilă. Apelurile de proceduri şi instrucţiunile de revenire pot fi gestionate corect de către asamblor. Restul operaţiilor trebuiesc gestionate explicit, eventual prin directive de asamblare condiţionată.

Să presupunem că transmitem unei proceduri un parametru de tip adresă de variabilă (de date) şi dorim să definim o structură de acces la stivă, care să nu depindă de modelul de memorie utilizat. Soluţia constă în folosirea directivelor de asamblare condiţionată şi a simbolului predefinit @MODEL (vezi 4.2).

```
        _TINY_        equ 1
        _SMALL_       equ 2
        _MEDIUM_      equ 3
        _COMPACT_     equ 4
        _LARGE_       equ 5
        _HUGE_        equ 6
        DATE_MARI     equ (@MODEL EQ COMPACT) OR (@MODEL EQ LARGE)
        COD_MARE      equ (@MODEL EQ MEDIUM) OR (@MODEL EQ LARGE)
        DATE_MICI     equ NOT (DATE_MARI)
        COD_MIC       equ NOT (COD_MARE)
        sablon struc
                _bp   dw     ?
                _ip   dw     ?
IF COD_MARE
                _cs   dw     ?
```

```
ENDIF
IF DATE_MICI
                adr_var         dw ?
ELSE
                adr_var         dd ?
ENDIF
        sablon ends

        portabila proc
                push    bp
                mov     bp, sp
        IF DATE_MARI
                les     bx, [bp].adr_var
        ELSE
                mov     bx, [bp].adr_var
        ENDIF
                pop     bp
                ret
        portabila endp
```

Şablonul de acces se modifică atât la schimbarea tipului datelor cât şi la cea a codului. Similar, accesul la adr_var presupune încărcarea offset-ului sau a perchii (segment:offset).

Segmentele definite care au importanţă din punct de vedere ASM sunt următoarele:

_TEXT sau SURSA_TEXT

Corespund directivei .code din ASM sau codului generat de compilatorul C. La modelele de cod mare (compact, large, huge) se generează numele SURSA_TEXT, unde SURSA este numele fisierului sursa C sau ASM. La ASM se poate folosi directiva .code nume, caz în care se va genera un segment cu numele nume_TEXT. La modelele de cod mic (tiny, small, compact), se generează segmente cu numele _TEXT, cu tipul de combinare PUBLIC, ceea ce va face ca toate segmentele _TEXT să se combine într-unul singur, rezultând astfel un unic segment de cod.

_DATA sau SURSA_DATA

Corespund directivei .data din ASM sau datelor din C in clasa extern şi static internal (deci cele cu alocare statică) iniţializate explicit. La modelul huge, compilatorul C generează numele SURSA_DATA, unde SURSA este numele fişierului sursă, iar la celelalte, numele _DATA. Tipul de combinare este PUBLIC, deci toate segmentele cu numele _DATA se vor combina într-unul singur.

_BSS

Corespunde directivei .data? din ASM sau datelor cu alocare statică neiniţializate explicit. Acestea vor fi iniţializate de modulul c0 cu valoarea 0. La modelul huge, atât datele neiniţializate cât şi cele iniţializate sunt definite în segmentele SURSA_DATA. Tipul de combinare este PUBLIC, deci toate segmentele cu numele _BSS se vor combina într-unul singur.

_STACK

Corespunde stivei programului. Stiva este declarată cu o lungime minimă în c0, dar poate fi extinsă în ASM prin directiva .stack, sau în C prin declaraţia:

```
extern unsigned _stklen = ... ;
```

La modelele diferite de huge şi tiny, se generează un grup de segmente numit DGROUP, care cuprinde segmentele _DATA şi _BSS (între altele). La modelul tiny acest grup cuprinde şi segmentele _TEXT şi _STACK. La modelul huge, DGROUP nu cuprinde segmentele SURSA_DATA.

Compilatorul C generează următoarele segmente, funcţie de modelul de memorie folosit:

- **tiny**

Se va genera un unic segment de cod, cu numele _TEXT, un unic segment de date iniţializate _DATA şi un unic segment de date neiniţializate _BSS. Se generează un grup de segmente, numit DGROUP, care va include segmentele _TEXT, _DATA şi _BSS. Nu se defineşte segment de stivă. Acest model corespunde programelor de tip .COM din DOS. În ASM se vor folosi directivele .code pentru cod, .data pentru date iniţializate şi .data? pentru date neiniţializate. Registrul DS este iniţialiizat cu DGROUP. Toţi pointerii sunt de tip near. Apelurile de funcţii sunt implicit near. Modulul de inţializare este c0t.obj, iar bilbliotecile specifice sunt cs.lib şi maths.lib.

- **small**

Se va genera un unic segment de cod, cu numele _TEXT şi un grup de segmente, numit DGROUP, care va include segmentul de date iniţializate _DATA şi segmentul de date neiniţializate _BSS. Se defineşte explicit un segment de stivă. Toţi pointerii sunt implicit de tip near. Apelurile de funcţii sunt implicit de tip near. Registrul DS este iniţializat în modulul c0 cu DGROUP. Modulul de inţializare specific este c0s.obj, iar bilbliotecile specifice sunt cs.lib şi maths.lib.

- **compact**

Se va genera un unic segment de cod, cu numele _TEXT şi un grup de segmente de date, numit DGROUP care conţine segmentele _DATA (accesbil în ASM prin directiva .data) şi _BSS (accsesibil în ASM prin .data?). Pointerii la funcţii sunt implicit de tip near, iar pointerii la date sunt implicit de tip far. Apelurile de funcţii sunt implicit de tip near. Registrul DS este iniţializat în modulul c0 cu DGROUP. Modulul de inţializare specific este c0c.obj, iar bilbliotecile specifice sunt cc.lib şi mathc.lib.

- **medium**

Se vor genera mai multe segmente de cod, cu numele SURSA_TEXT, unde SURSA este numele fişierului sursă C sau ASM. În ASM, dacă directiva code are un parametru, se va genera un segment de cod cu numele respectiv. Se consideră că în cuprinsul segmentului de cod respectiv, este activă o directivă ASSUME CS: cu numele segmentului respectiv. Se generează grupul DGROUP care include segmentele _DATA şi _BSS (accesibile în ASM prin .data şi .data?). Pointerii la date sunt implicit de tip near, iar pointerii la funcţii sunt implicit de tip far. Apelurile de funcţii sunt implicit de tip far. Registru DS este iniţializat în modulul c0 cu DGROUP. Modulul de inţializare specific este c0m.obj, iar bilbliotecile specifice sunt cm.lib şi mathm.lib.

- **large**

Se vor genera mai multe segmente de cod, cu numele SURSA_TEXT, unde SURSA este numele fişierului sursă C sau ASM. În ASM, dacă directiva code are un parametru, se va genera un segment de cod cu numele respectiv. Se consideră că în cuprinsul segmentului de cod respectiv, este activă o directivă ASSUME CS: cu numele segmentului respectiv. Se generează grupul DGROUP care include segmentele _DATA şi _BSS (accesibile în ASM prin .data şi .data?). Toţi pointerii sunt implicit de tip far. Apelurile de funcţii sunt implicit de tip far. Registrul DS este iniţializat în modulul c0 cu DGROUP. Modulul de inţializare specific este c0l.obj, iar bilbliotecile specifice sunt cl.lib şi mathl.lib.

- **huge**

Se vor genera mai multe segmente de cod, cu numele SURSA_TEXT, unde SURSA este numele fişierului sursă C sau ASM. În ASM, dacă directiva code are un parametru, se va genera un segment de cod cu numele respectiv. Se vor genera mai multe segmente de date, cu numele SURSA_DATA, unde SURSA este numele fişierului sursă C sau ASM. Nu se generează segmente _BSS, toate datele fiind definite în segmentele SURSA_DATA. Se consideră că în cuprinsul modulului respectiv este activă o directivă ASSUME CS:SURSA_TEXT, DS:SURSA_DATA. Grupul de segmente DGROUP nu include segmentele SURSA_DATA. Asmblorul TASM genereaza însă un segment cu numele _DATA, corespunzător directivei .data, care se include în grupul DGROUP. Registrul DS este iniţializat în modulul c0 cu DGROUP iar la fiecare intrare într-o funcţie C (inclusiv main), registrul DS este iniţializat cu SURSA_DATA. Evident, la intrarea în funcţiile ASM, va trebui făcută o iniţializare cu DGROUP, salvând în prealabil registrul DS în stivă şi refăcându-l la ieşirea din funcţia ASM. Pentru accesul la segmentele de date definite în C se poate folosi operatorul SEG aplicat unei variabile extern, sau direct numele segmentului, în forma SURSA_DATA.

Toţi pointerii sunt implicit de tip far şi sunt normalizaţi, adică offsetul este redus la intervalul de valori 0...15. Astfel, corespondenţa segment:offset cu adresa fizică de memorie devine biunivocă. Incrementarea/decrementarea pointerilor se va reflecta acum atât în offset cît şi în adresă de segment, spre deosebire de modelele celelalte, la care incrementarea/decrementarea pointerilor de tip far acţionează numai asupra offsetului. Este posibilă deci definirea unor structuri de date de dimensiuni mai mari decât 64K, ceea ce la celelalte modele de memorie nu este posibil.

Apelurile de funcţii sunt implicit de tip far. Modulul de inţializare specific este c0h.obj, iar bilbliotecile specifice sunt ch.lib şi mathh.lib.

O problemă specială poate apare la modelele de memorie aşa-zise "de date mici" (small şi medium). Se pot defini date de orice fel, atât în clasa de alocare automatic (cu spaţiu rezervat în stivă, deci accesibile prin registrul de segment SS), cît şi în clasa de alocare static (cu spaţiu rezervat într-unul din segmentele ce compun grupul DGROUP, deci accesibile prin registrul de segment DS).

În aceste modele de memorie, pointerii către date sunt de tip near, deci ei memorează numai deplasamentele obiectelor în cadrul segmentelor respective (de date, pentru obiecte în clasa static, respectiv de stivă, pentru obiecte în clasa auto). Dacă se accesează obiecte (date) prin pointeri, nu se poate şti în

ce tip de segmente sunt definite aceste obiecte. Problema este rezolvată (în cazul modelelor small şi medium) prin generarea segmentelor de aşa manieră, încât adresele de început ale segmentului de stivă şi grupului DGROUP să coincidă. Practic, pe parcursul execuţiei, registrele DS şi SS vor avea aceeaşi valoare.

Să observăm că problema nu se pune în cazul modelului tiny (există un unic segment) şi nici în cazul modelelor large şi huge (toţi pointerii sunt de tip far). Deasemenea, problema nu se pune în cazul modelului "de cod mic şi date mari" compact, deoarece pointerii către funcţii vor conţine (indiferent de locul unde sunt definiţi) deplasamente în cadrul unicului segment de cod care se generează. Apelurile de funcţii prin pointeri se traduc în simple salturi indirecte intrasegment (instrucţiuni de tip call word ptr ...). Exemple de asemenea apeluri sunt:

```
call    word ptr [bp-4]     ;Apel indirect, pointer în clasa auto
call    word ptr _p         ;Apel indirect, pointer în clasa static
```

Dacă se fac indirectări multiple, acestea presupun definirea unor pointeri către pointeri către funcţii, adică a unor pointeri către date. Aceşti ponteri vor fi deci de tip far şi vor fi încărcaţi şi dereferenţiaţi totdeauna ca adrese complete (segment şi offset).

În concluzie, să reţinem că, la modelele small şi medium, registrele DS şi SS vor avea acelaşi conţinut la execuţie. Mediul integrat oferă opţiunea **Options/Compiler/Code Generation/Assume DS not equal SS**, care este recomandabil să fie plasată pe **Default for memory model**.

Pe lîngă segmetele descrise mai sus, modulul de iniţializare c0 mai defineşte o serie de segmente, folosite la iniţializare, la ieşirea în sistemul de operare, etc, dar care nu prezintă interes din punct de vedere ASM. Se poate vedea exact ce segmente se generează, studiind fişierul sursă c0.asm, furnizat în kit-ul implementării C (în directorul \examples\startup).

Când există îndoieli asupra a ce se generează concret de către compilatorul C, se poate folosi opţiunea -S la compilare în linia de comandă, obţinându-se în acest fel un fişier sursă ASM, corespunzător modulului C respectiv sau se poate lansa Turbo Profiler-ul din mediu integrat, care, între alte informaţii, furnizează şi codul ASM generat.

7.4 O aplicaţie mixtă cu tipuri de date simple

Pentru a fixa ideile, considerăm un exemplu de program, compus dintr-un modul C şi un modul ASM, care va fi detaliat în trei cazuri de modele: compact, medium şi huge. Codul ASM pentru celelate modele se deduce simplu: la modelul small se tratează datele ca la medium şi codul ca la compact, iar la modelul large se tratează datele ca la compact şi codul ca la medium.

Modulul sursă C, presupus în fişierul test1c.c, este acelaşi în toate situaţiile, şi anume:

```
#include <stdio.h>
extern int f (int);
extern int (*pf) (int);
extern int *g (void);
extern int *gg (void);
extern int ext_i;
```

```
extern int ext_n;
int extc_i = 2;

int *ff (int *pi)
{
        *pi *= 2;
        return pi;
}
void main(void)
{
        int n;
        printf("\nIntroduceti un intreg: ");
        scanf("%d",&n);
        n = f(n);
        printf("Valoarea dubla este: %d\n",n);
        n = (*pf)(n);
        printf("Valoarea dubla este: %d\n",n);
        printf("Variabila externa initializata este: %d\n", ext_i);
        ext_n = 5;
        printf("Variabila externa neinitializata este: %d\n", *g());
        printf("Valoarea dubla este: %d\n", *gg());
}
```

Modulele ASM sunt presupuse în fişierele test1ca.asm, test1ma.asm şi, respectiv, test1ha.asm, corespunzătoare modelelor compact, medium şi huge.

În aplicaţia studiată se consideră următoarele entităţi de program definite în ASM:

- funcţia f, cu tipul int şi cu un parametru de tip int;
- pointerul la funcţie pf, iniţializat cu adresa funcţiei f;
- funcţia g, cu tipul int şi fără parametri;
- funcţia gg, cu tipul int * şi fără parametri;
- întregul ext_i, iniţializat cu valoarea 11;
- întregul ext_n, neiniţializat.

În modulul C se consideră următoarele entităţi:

- întregul extc_i, iniţializat cu valoarea 2;
- funcţia ff, cu tipul int * şi cu un parametru de tip int *.

Funcţia ff primeşte o adresă de întreg şi înmulţeşte cu 2 întregul de la adresa respectivă, întorcând adresa primită.

Funcţia f primeşte un întreg, afişează la consolă mesajul "Intregul este:" şi valoarea variabilei extc_i. Se întoarce programului apelant valoarea dublă a parametrului primit.

Funcţia g întoarce programului apelant adresa variabilei ext_n.

Funcţia gg apelează funcţia ff, cu parametrul adresa variabilei ext_n, întorcând programului apelant adresa returnată de de funcţia ff, adică adresa aceleiaşi variabile ext_n.

Programul principal, scris în C, citeşte de la consolă un întreg n şi apelează de două ori funcţia f, cu parametrul n, prima dată direct iar a doua oară prin pointerul pf. În ambele apeluri, valoarea întoarsă este depusă în aceeaşi variabilă n. Se afişează apoi variabila ext_i, se modifică explicit variabila ext_n

şi se apelează funcţiile g şi gg, afişând la întregii de la adresele întorse de aceste funcţii, adică variabila ext_n.

Afişările din modulul ASM sunt realizate prin macroinstrucţiunile putsi şi puti, care presupun datele adresabile prin registrul DS. Rezultă deci că aplicaţia va trebui să cuprindă şi modulul obiect io.obj (vezi Anexa B).

7.4.1 Dezvoltare în modelul de memorie compact

Modulul test1ca.asm, corespunzător modelului compact, este:

```
.model compact
include io.h
public   _f,  _pf,  _g, _gg, _ext_i, _ext_n
extrn   _ff: near,  _extc_i: word
sablon _f struc
        _bp     dw ?                        ; Şablon pentru accesul
        _ip     dw ?                        ; în stivă, în procedura
        n       dw ?                        ; de tip near f
sablon_f ends
.data
        _ext_i dw      11                   ; Întreg iniţializat
        _pf     dw      _f                  ; Pointer near iniţializat cu
                                            ; adresa (offset-ul) lui _f
.data?
        _ext_n dw      ?                    ; Întreg neiniţializat
.code
_f  proc near
        push   bp
        mov    bp, sp                       ; Poziţionare bp pentru acces
        mov    ax, [bp].n                   ; Preluare parametru
        add    ax, ax                       ; Dublare
        putsi  <cr,lf,'Intregul este: '>
        mov    bx, DGROUP:_extc_i           ; Segmentul este DGROUP
        puti   bx                           ; Tipăreşte întregul (bx)
        putsi  <cr, lf>
        pop    bp
        ret                                 ; (ax) = rezultat
_f  endp

_g  proc near
        mov    dx, DGROUP                   ; Segment
        lea    ax, DGROUP:_ext_n            ; Offset
        ret                                 ; Rezultat in (dx:ax)
_g endp

_gg proc near
        mov    ax, DGROUP                   ; Pregătire parametri apel
        push   ax                           ; Segment
        lea    ax,DGROUP:_ext_n
        push   ax                           ; Offset
        call   near ptr _ff                 ; Apel _ff
        add    sp, 4                        ; Descărcare stiva
        ret                                 ; Rezultat in (dx:ax)
_gg endp
end
```

Deoarece modelul de memorie este compact, toate apelurile de funcţii sunt de tip near, pointerul pf la funcţie este de tip near, iar toţi pointerii la date sunt de

tip far. Funcţiile f, g şi gg vehiculează adrese de 32 de biţi, transmise prin stivă sau întoarse prin perechea (DX:AX).

Programul va afişa la consolă textul:

```
Introduceti un intreg: 13
Intregul C este: 2
Valoarea dubla este: 26
Intregul C este: 2
Valoarea dubla este: 52
Variabila externa initializata este: 11
Variabila externa neinitializata este: 5
Valoarea dubla este: 10
```

Presupunând directorul mediului C ca fiind C:\BC, secvenţa de linii de comandă pentru dezvoltarea aplicaţiei este:

```
bcc -mc -c -Ic:\bc\include test1c.c
tasm test1ca.asm/ml
tlink c:\bc\lib\c0c test1c test1ca io,test1c,,c:\bc\lib\cc
```

7.4.2 Dezvoltare în modelul de memorie medium

Modulul test1ma.asm, corespunzător modelului de memorie medium este:

```
.model medium
include io.h
public   _f, _pf, _g, _gg, _ext_i, _ext_n
extrn   _ff: far, _extc_i :word
sablon_f struc
        _bp     dw ?                        ; Şablon pentru accesul
        _ip_cs dd ?                         ; în stivă, în procedura
        n       dw ?                        ; de tip far _f
sablon_f ends
.data
        _ext_i dw 11                        ; Întreg iniţializat
        _pf     dd _f                       ; Pointer far iniţializat cu
                                            ; adresa completă a lui _f
.data?
        _ext_n dw ?                         ; Întreg neiniţializat
.code
_f      proc far
        push    bp
        mov     bp, sp                      ;Poziţionare bp pentru acces
        mov     ax, [bp].n                  ;Preluare parametru
        add     ax, ax                      ; Dublare
        mov     bx, DGROUP:_extc_i          ; Segmentul este DGROUP
        putsi   <cr, lf,'Intregul C este: '>
        puti    bx                          ; Tipăreşte întregul (bx)
        putsi   <cr, lf>
        pop     bp
        retf                                ; (ax) = rezultat
_f      endp

_g      proc far
        lea     ax,DGROUP:_ext_n            ; Offset
        retf                                ; Rezultat in (ax)
_g endp

_gg     proc far
        lea     ax, DGROUP:_ext_n           ; Pregătire parametri apel
        push    ax                          ; Offset
```

```
                call    far ptr _ff                     ; Apel _ff
                add     sp, 2                           ; Descărcare stiva
                retf                                    ; Rezultat in (ax)
        _gg     endp
        end
```

Ceea ce se schimbă faţă de modelul anterior este faptul că apelurile şi pointerii la funcţii sunt de tip far, iar adresele şi pointerii la variabile sunt de tip near. Şablonul de acces în stivă se modifică, ţinând cont că un apel far salvează şi registrul CS în stivă. Pointerul pf se defineşte cu directiva dd (define doubleword). Adresa lui ext_n (offset) este întoarsă de funcţiile g şi gg prin AX.

Secvenţa de linii de comandă este:

```
        bcc -mm -c -Ic:\bc\include test1c.c
        tasm test1ma.asm/ml
        tlink c:\bc\lib\c0m test1c test1ma io,test1m,,c:\bc\lib\cm
```

7.4.3 Dezvoltare în modelul de memorie huge

Modulul test1ha.asm, corespunzător modelului huge este:

```
.model huge
        include io.h
        public _f, _pf, _g, _gg, _ext_i, _ext_n
        extrn _ff:far, _extc_i:word

        sablon_f struc
                _bp     dw ?                    ; Şablon pentru accesul
                _ip_cs dd ?                     ; in stivă, în procedura
                n       dw ?                    ; de tip far _f
        sablon_f ends

.data
         _ext_i dw 11                           ; Întreg iniţializat
         _pf    dd _f                           ; Pointer far iniţializat cu
                                                ; adresa completă a lui _f
.data?
        _ext_n dw ?                             ; Întreg neiniţializat

 code
_f proc far
        push    bp
        mov     bp,sp                           ; Poziţionare bp pentru acces
        push    ds                              ; Salvare ds
        push    es                              ; Salvare es
        push    bx                              ; Salvare bx
        mov     ax,DGROUP
        mov     ds,ax                           ; Segment local de date

        mov     ax,[bp].n                       ; Preluare parametru
        add     ax,ax                           ; Dublare

        push    ax                              ; Salvare rezultat
        mov     ax,SEG _extc_i                  ; Seg. asociat lui _extc_i
        mov     es,ax
        mov     bx,es:_extc_i                   ; Preluare _extc_i
;
; Sirul din macroinstrucţiunea putsi se defineşte într-o
; directivă .data, deci în segmentul local, de aceea ds
; trebuie să indice DGROUP
```

```
;
        putsi   <cr,lf,'Intregul C este: '>
        puti    bx                          ;Tipăreşte întregul (bx)
        putsi   <cr,lf>
        pop     ax                          ; Rexultat

        pop     bx                          ; Refacere bx
        pop     es                          ; Refacere es
        pop     ds                          ; Refacere ds

        pop     bp
        retf                                ; (ax) = rezultat
_f endp

_g proc far
         mov    dx,DGROUP                   ; Segment
         lea    ax,DGROUP:_ext_n            ; Offset
         retf                               ; Rezultat in (dx:ax)
_g endp

_gg proc far
        mov     ax,DGROUP                   ; Pregătire parametri
        push    ax                          ; Segment local
        lea     ax,DGROUP:_ext_n
        push    ax                          ; Offset
        call    far ptr _ff                 ; Apel _ff
        add     sp,4                        ; Descărcare stiva
        retf                                ; Rezultat in (dx:ax)
_gg endp
end
```

Se observă că, în procedura f, unde se face acces la date, se poziţionează explicit registrul DS pe grupul de segment DGROUP, pentru a putea utiliza macroinstrucţiunile de afişare şi se încarcă explicit în ES adresa de segment a variabilei externe ext_c. Registrele DS şi ES se salvează la intrarea în funcţie şi se refac înainte de întoarcerea în programul apelant.

Secveţa de linii de comandă este:

```
bcc -mh -c -Ic:\bc\include test1c.c
tasm test1ha.asm/ml
tlink c:\bc\lib\c0h test1c test1ha io,test1h,,c:\bc\lib\ch
```

7.5 O aplicaţie mixtă cu tipuri de date structurate

Se consideră modulul sursă C, presupus în fişierul test2c.c:

```
        #include <string.h>
        struct tips { int n; char *sir; };
        void f (struct tips *);
        struct tips *g (struct tips *);
        struct tips * (*pg) (struct tips *) = g;

        struct tips a = { 100, "Iata un sir..." };

        struct tips *g(struct tips *x)
        {
                static int contor = 0;
                contor++;
                f(x);
                (x -> n) ++;
```

```
        if(contor % 2)
                strupr(x -> sir);
        else
                strlwr(x -> sir);
        f(x);
        return x;
}
```

Se consideră tipul de structură tips, care cuprinde un înntreg şi un pointer la un şir de caractere. Structura a este iniţializată cu valoarea 100, respectiv cu şirul "Iata un sir...".

Funcţia de tip void f, scrisă în ASM, primeşte o adresă de structură şi afişează la consolă cele două câmpuri (întreg şi şir

de caractere) cu ajutorul macroinstrucţiunilor piuti şi puts, definite în fişiewrul header io.h.

Funcţia g, scrisă în C, primeşte adresa x a unei structuri şi realizează următoarele operaţii:

- apelul funcţiei f, cu parametrul adresa x (care va afişa structura indicată de x);
- incrementarea câmpului numeric al structurii de la adresa x;
- modificarea succesivă a şirului de caractere din structura indicată de variabilax, din litere mici în litere mari şi succesiv; acest lucru este realizat prin incrementarea modulo 2 a unei varibile locale statice;
- apelul funcţiei f, cu parametrul adresa x (care va afişa structura indicată de x).

Programul principal, scris în ASM, realizează următoarele operaţii:

- apelează funcţia g în mod direct, cu parametrul adresa structurii a;
- pregăteşte în stivă adresa întoarsă de g (deci adresa structurii a), pentru un apel viitor;
- apelează funcţia f în mod direct, cu parametrul întors de apelul lui g (deci cu adresa structurii a);
- apelează indirect funcţia g, prin intermediul pointerului pg, cu parametrul adresa pregătită anterior în stivă;
- apelează funcţia f, cuu parametrul întors de apelul indirect al lui g (deci cu adresa structurii a).

7.5.1 Dezvoltare în modelul de memorie compact

Modulul test2ca.asm, corespunzător modelului de memorie compact, este:

```
.model      compact
include     io.h
public      _f, _main
extrn       _g: near, _a: far, _pg: word

tips    struc
        n       dw ?                    ; Şablon pentru
        sir     dd ?                    ; structura tips
tips    ends

sablon struc
        _bp             dw ?            ; Şablon pentru acces
         _ip            dw ?            ; în stivă
        _adr_struc      dd ?            ; Parametrul
sablon ends

.stack  1024
```

```
        .code

        _f      proc near
                push    bp
                mov     bp, sp
                push    ds                              ; Salvări
                push    di                              ; registre
                push    si                              ; folosite

        ;
        ; Încărcare adresă far din stivă
        ;
                les     di, dword ptr [bp]._adr_struc
        ;
        ; Acces la membrul n al structurii
        ;
                mov     ax, es:[di].n
        ;
        ; Tipărire n
        ;
                putsi   <cr, lf, 'Intregul este: '>
                puti    ax
                putsi   <cr, lf, 'Sirul este: '>
        ;
        ; Acces la membrul sir (pointer far) al structurii
        ;
                lds     si, dword ptr es:[di].sir
        ;
        ; Tipărire şir
        ;
                call    far ptr puts_proc
                pop     si                              ; Refaceri
                pop     di                              ; registre
                pop     ds                              ; folosite
                pop     bp
                ret
        _f endp

        _main proc near
                mov     ax,DGROUP                       ; Pregătire parametri
                push    ax                              ; Segmentul structurii a
                lea     ax,DGROUP:_a
                push    ax                              ; Offsetul structurii a
                call    near ptr _g                     ; Apel g
                add     sp,4                            ; Descărcare stivă
        ;
        ; g întoarce un pointer far în (dx:ax)
        ; Este pregătit acest pointer în stivă, pentru un apel viitor al lui g
        ; (altfel ar fi trebuit salvat într-o zonă de date)
        ;
                push    dx                              ; Segment
                push    ax                              ; Offset
        ;
        ; Pregătire parametru pentru apelul lui f
        ;
                push    dx                              ; Segment
                push    ax                              ; Offset
                call    near ptr _f                     ; Apel f
                add     sp, 4                           ; Descărcare stivă
```

```
;
; Acum se face apelul lui g, indirect, prin pointerul de tip near pg,
; cu parametrul de tip pointer far, pregătit anterior în stivă
;
        call    word ptr _pg
        add     sp, 4                   ; Descărcare stivă
;
; g întoarce un pointer far în (dx:ax)
; Se pregăteşte acest pointer în stivă, pentru apelul lui f
;
        push    dx                      ; Segment
        push    ax                      ; Offset
        call    near ptr _f             ; Apel f
        add     sp, 4                   ; Desccărcare stivă
        ret                             ; Terminare program
        _main endp
end
```

Pointerii la date sunt de tip far iar pointerul lka funcţie este de tip near. Se observă declararea simbolului extern a ca o etichetă de tip far, ceea ce este în concordanţă cu convenţia că numele unei variabile este sinonim cu adresa variabilei respective.

La consolă se va tipări:

```
Intregul este: 100
Sirul este: Iata un sir...
Intregul este: 101
Sirul este: IATA UN SIR...
Intregul este: 101
Sirul este: IATA UN SIR...
Intregul este: 101
Sirul este: IATA UN SIR...
Intregul este: 102
Sirul este: iata un sir...
Intregul este: 102
Sirul este: iata un sir...
```

Secvenţa de linii de comandă este:

```
bcc -mc -c -Ic:\bc\include test2c.c
tasm test2ca.asm/ml
tlink c:\bc\lib\c0c test2ca test2c io,test2c,,c:\bc\lib\cc
```

7.5.2 Dezvoltare în modelul de memorie medium

Modulul test2ma.asm, corespunzător modelului medium este:

```
.model          medium
include         io.h
public          _f, _main
extrn           _g:far, _a:near, _pg:dword
tips struc
        n       dw ?                    ; Şablon pentru
        sir     dw ?                    ; structura tips
tips ends

sablon struc
        _bp             dw ?            ; Şablon pentru acces
        _ip_cs          dd ?            ; în stivă
        _adr_struc      dw ?            ; Parametrul
sablon ends
```

```
        .stack  1024
        .code

_f proc far
        push    bp
        mov     bp, sp
        push    di                              ; Registre
        push    si                              ; folosite
;
        mov     di,word ptr [bp]._adr_struc
;
        mov     ax,DGROUP:[di].n
;
        putsi   <cr,lf,'Intregul este: '>
        puti    ax
        putsi   <cr,lf,'Sirul este: '>
;
        mov     si,word ptr DGROUP:[di].sir
;
        call    far ptr puts_proc
        pop     si                              ; Refaceri
        pop     di                              ; registre
        pop     bp
        retf
 _f endp

 _main proc far
        lea     ax,DGROUP:_a                    ; Pregătire parametru
        push    ax                              ; Offsetul structurii a
        call    far ptr _g                      ; Apel g
        add     sp,2                            ; Descărcare stivă
;
; g întoarce un pointer near în (ax)
;
        push     ax                             ;Offset
;
; Pregătire parametru pentru apelul lui f
;
        push    ax                              ; Offset
        call    far ptr _f                      ; Apel f
        add     sp,2                            ; Descărcare stivă
;
        call    dword ptr _pg
        add     sp, 2                           ; Descărcare stivă
;
; g întoarce un pointer near în (ax)
;
        push    ax                              ; Offset
        call    far ptr _f                      ; Apel f
        add     sp, 2                           ; Descărcare stivă
        retf                                    ; Terminare program
_main endp
end
```

Diferenţele faţă de cazul precedent rezultă din faptul că funcţiile şi pointerii la funcţii sunt de tip far, iar adresele variabileleor şi pointerii la date sunt de tip near.

Simbolul extern **a** (de tip structură) se declară în ASM ca o etichetă de tip near.

Secvenţa de linii de comandă este:

```
bcc -mm -c -Ic:\bc\include test2c.c
tasm test2ma.asm/ml
tlink c:\bc\lib\c0m test2ma test2c io,test2m,,c:\bc\lib\cm
```

7.5.3 Dezvoltare în modelul de memorie huge

Modulul test2ha.asm, corespunzător modelului huge este:

```
        .model          huge
        public          _f, _main
        extrn           _g:far, _a: far, _pg:dword
        include         io.h

        tips struc
                n       dw ?                    ; Şablon pentru
                sir     dd ?                    ; structura tips
        tips ends

        sablon struc
                _bp             dw ?            ; Şablon pentru acces
                _ip_cs          dd ?            ; în stivă
                _adr_struc      dd ?            ; Parametru
        sablon ends

.stack  1024
.code

_f proc far
        push    bp
        mov     bp, sp
        push    ds                              ; Salvări
        push    di                              ; registre
        push    si                              ; folosite
;
        mov     ax, DGROUP
        mov     ds, ax
;
        les     di,dword ptr [bp+6]
        mov     ax,es:[di].n
        putsi   <cr,lf,'Intregul este: '>
        puti    ax
        putsi   <cr,lf,'Sirul este: '>
        lds     si,dword ptr es:[di].sir
        call    far ptr puts_proc
;
        pop     si                              ; Refaceri
        pop     di                              ; registre
        pop     ds                              ; folosite
        pop     bp
        retf
_f endp

_main proc far
        push    ds
        push    es
        mov     ax,DGROUP                       ; Comutare pe segmentul
```

```
        mov     ds,ax                           ; propriu de date
;
        mov     ax, SEG _a                      ; Pregătire parametru
        mov     es, ax                          ; Segmentul structurii a
        push    es
        lea     ax, es:_a                       ; Offsetul structurii a
        push    ax
        call    far ptr _g                      ; Apel g
        add     sp,4                            ; Descărcare stivă
        ;
        ;       g întoarce un pointer far în (dx:ax)
        ;       Este pregătit acest pointer în stivă, pentru
        ;       un apel viitor al lui g (altfel ar fi trebuit salvat
        ;       într-o zonă de date)
        ;
        push    dx                              ; Segment
        push    ax                              ; Offset
        ;
        ;       Pregătire parametru pentru apelul lui f
        ;
        push    dx                              ; Segment
        push    ax                              ; Offset
        call    far ptr _f                      ; Apel f
        add     sp, 4                           ; Descărcare stivă
        ;
        ; Acum se face apelul lui g, indirect, prin pointerul
        ; de tip near pg (care e în alt segment de date), cu
        ; parametrul de tip pointer far, pregătit anterior în stivă
        ;
        mov     ax, SEG _pg
        mov     es,ax
        call    dword ptr es:_pg
        add     sp,4
        ;
        ; g întoarce un pointer far în (dx:ax)
        ; Se pregăteşte acest pointer în stivă, pentru apelul lui f
        ;
        push    dx
        push    ax
        call    far ptr _f
        add     sp,4

        pop     es
        pop     ds
        retf
_main endp
end
```

Diferenţa faţă de celelalte cazuri este că datele externe nu mai sunt în mod necesar definite în acelaşi segment (grup) cu datele din modulul ASM. Adresarea structurii a şi a pointerului pg trebuie făcută în consecinţă, folosind operatorul SEG pentru luarea segmentelor în care aceşti simboli sunt definiţi.

Secvenţa de linii de comandă este:

```
bcc -mh -c -Ic:\bc\include test2c.c
tasm test2ha.asm/ml
tlink c:\bc\lib\c0h test2ha test2c io,test2h,,c:\bc\lib\ch
```

7.6 Dezvoltarea într-un limbaj mixt (C şi ASM)

Mediul de dezvoltarea Borland C oferă posibilitatea inserării de cod ASM în programe sursă C. Fiecare instrucţiune ASM trebuie precedată de cuvântul cheie asm sau să fie inclusă într-un bloc de tip ASM, de forma:

```
asm {
        ; Instrucţiuni ASM
}
```

Acest mode de dezvoltare este foarte comod, deoarece vizibilitatea obiectelor definite în C se extinde şi asupra blocurilor ASM. Astfel, putem folosi nume de variabile, nume de parametri formali ai funcţiilor etc., fără a mai scrie caracterul _ în faţa numelor respective. Se evită astfel şi secvenţele de intrare şi revenire din funcţii, care sunt realizate automat de compilatorul C.

O altă facilitate este accesul direct din C asupra registrelor procesorului, realizat **prin variabilele C predefinite _AX, _BX,** _CX etc., care pot fi utilizate, de exemplu, în instrucţiuni de atribuire.

Există restricţii de utilizare asupra etichetelor. Mai precis, etichetele trebuie definite ca etichete C, deci în afara blocurilor ASM.

Ca şi la dezvoltarea în module separate, programatorul trebuie să gestioneze explicit modelele de memorie, în sensul definirii precise a adreselor far sayu near. Pentru aceasta, se utilizează modelele de memorie C şi/sau cuvintele cheie near şi far ale implementării Borland C.

Un modul dezvoltat în această manieră este scris de fapt într-un limbaj mixt (C şi ASM). De altfel, fişierele sursă de acest tip se creează de obicei cu extensia .CAS (de la C şi ASM).

Pentru exemplificare, să considerăm o funcţie de căutare liniară într-un tablou de întregi. În limbajul C, am scrie această funcţie în forma:

```
int cauta (int *a, size_t n, int x)
{
        int i;
        for (i = 0; i < n; i++)
                if (x == a[i])
                        return i;               /* S-a găsit întregul x */
        return -1;                              /* Nu s-a găsit întregul x */
}
```

Dorim avum să scriem aceeaşi funcţie în "limbajul CAS". Presupunem pentru început un model "de date mici". Implementarea funcţiei este următoarea:

```
int cauta (int near *a, size_t n, int x)
{
        asm {
                push    si                      ; Salvări
                push    cx                      ; registre
                push    dx

                mov     si, a                   ; Adresa near a tabloului
                mov     cx, n                   ; Număr elemente
                jcxz    not_gasit               ; Test n == 0
                mov     dx, x                   ; Element căutat
                sub     si, 2                   ; Pornim cu o poziţie mai la stânga
        }
```

```
        reluare:
                asm {
                        add     si, 2                   ; Element curent din tablou
                        cmp     [si], dx                        ; Comparăm cu cel căutat
                        loopnz  reluare                 ; Nu este, se reia bucla
                        jnz     not_gasit               ; Am ieşit din buclă
                        mov     ax, n                   ; Dacă ZF a fost 1, calculăm
                        sub     ax, cx                  ; indicele elementului găsit
                        dec     ax                      ; ca fiind n - CX - 1
                        jmp     final                   ; Salt la ieşire
                }
        not_gasit:
                asm     mov     ax, -1                  ; Nu s-a găsit elementul x
        final:
                asm {
                        pop     dx                      ; Refaceri
                        pop     cx                      ; registre
                        pop     si
                }
                return  _AX;                            ; Instrucţiune C
        }
```

Se observă tehnica de scriere a buclei de căutare (prefixul LOOPNZ) şi calculul indicelui elementului găsit, ca diferenţă dintre dimensiunea n a tabloului şi valoarea incrementată a contorului CX la ieşirea din buclă. Revenirea în funcţia apelantă se face cu instrucţiunea C return _AX (ne amintim convenţia de întoarcere a datelor simple).

Un punct important este încărcarea adresei tabloului. Dacă adresa este near, o încărcăm cu o instrucţiune MOV (în stivă se găseşte deplasamentul în cadrul segmentului curent de date). În cazul unui model de "date mari", în stivă se găseşte un pointer far (o variabilă de tip DoubleWord), caz în care încărcarea s-ar face cu o instrucţiune:

```
        les     si, a                   ; Adresa far a tabloului
```

Compararea elementului curent cu cel căutat s-ar scrie:

```
        cmp     es:[si], dx             ; Comparăm cu cel căutat
```

În fine, ar mai trebui salvat şi restaurat registrul ES, iar prototipul funcţiei ar trebui scris în forma:

```
        int cauta (int near *a, size_t n, int x);
```

Modificările de mai sus transformă transformă funcţia într-una adecvată modelelor "de date mari".

Un program de test pentru funcţia de mai sus s-ar scrie în genul:

```
#include <stdio.h>
int x [ ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, };
#define NREL(a) (sizeof(a)/sizeof(a[0]))
void main (void)  {
        int i, val = 6;
        i = cauta (x, NREL(x), val);
        if (i > 0)
                printf("Elementul %d se afla pe pozitia %d din tablou\n", val, i);
        else
                printf("Elementul %d nu se afla in tablou\n", val);
}
```

Trebuie totuşi observat că această tehnică mixtă de dezvoltare are şi dezavantaje. Un modul CAS este destul de greu de întreţinut iar transportul său în alt mediu de dezvoltare decât Borland ar putea crea probleme de compatibilitate. Tehnica mixtă merită folosită atunci când avem de implementat secvenţe relativ scurte de program, pentru care nu se justifică un modul (fişier sursă) separat.

7.7 Performanţe în ASM şi C

În acest subcapitol vom face o analiză comparativă a unui algoritm implementat în C şi ASM. Algoritmul ales este un algoritm de sortare internă, anume sortarea prin partiţionare şi interschimbare, numită şi sortare rapidă (**quicksort**).

Algoritmul este recursiv şi se poate implementa foarte comod atât în C cât şi în ASM. O implementare polimorfică a acestui algoritm este oferită şi de biblioteca standard a limbajului C.

7.7.1 Implementarea în C

Varianta C considerată va fi de asemenea polimorfică (capabilă să sorteze tablouri de orice fel). Prototipul funcţiei C este:

```
void qsort (void *tab, size_t n, size_t size, PFCMP cmp);
```

Semnificaţia parametrilor este:

- tab - adresa de început a tabloului;
- n - numărul de elemente al tabloului;
- size - dimansiunea în octeţi a unui element;
- cmp - pointer la o funcţie externă de comparaţie, definit prin:

```
typedef int (*PFCMP) (void *a, void *b);
```

Funcţia de comparaţie (scrisă de utilizator) primeşte adresele a două elemente din tablou şi întoarce o valoare negativă, zero sau pozitivă, după cum elementul de la prima adresă este mai mare, egal sau mai mic decât elementul de la a doua adresă. Sensul noţiunilor mai mare, egal sau mai mic este complet abstract, fiind definit de yutilizator chiar prin funcţia de comparaţie.

Pentru transferul datelor se vor utiliza funcţiile ajutătoare swap şi copy, listate mai jos.

```
typedef unsigned char BYTE;

void swap(void *a, size_t size, int i, int j)
{
	/*
		Schimba a[i] cu a[j]. Elementele tabloului a[] au size octeti.
	*/

	register BYTE *b = ((BYTE *)a) + i * size;
	register BYTE *c = ((BYTE *)a) + j * size;
	register BYTE temp;
	while(size--) {
		temp = *b;
		*b++ = *c;
		*c++ = temp;
	}
}
```

```
void copy(void *a, void *b, size_t size)
{
        /*
                Copiaza obiectul de la adresa b in cel de la adresa a.
                Obiectele au size octeti.
        */
        memcpy(a, b, size);
}
```

Algoritmul de sortare rapidă este recursiv. Funcţia care implementează algoritmul recursiv primeşte ca date de intrare o parte a tabloului care trebuie sortat, prin adresa de început şi doi indici notaţi left şi right. Iniţial, funcţia se apelează cu indicii 0 şi n-1.

Se alege un element arbitrar al tabloului v, numit pivot, notîndu-l cu mark (variante uzuale sînt v[left] sau v[(left+right)/2]). Se partiţonează tabloul în raport cu mark, în sensul că toate elementele aflate la stînga lui mark să fie mai mici sau egale decît acesta, iar toate elementele aflate la dreapta lui mark să fie mai mari sau egale decît acesta.

În acest moment, pivotul se află pe poziţia sa finală, iar tabloul este partiţionat în două subtablouri. Se apelează acum aceeaşi funcţie, cu indicii left şi k-1, respectiv k+1 şi right, unde k este indicele pivotului în urma partiţionării. În felul acesta se sortează cele două subtablouri. Dacă left >= right algoritmul se opreşte.

Algoritmul se poate îmbunătăţi în felul următor. Să pornim cu doi indici i şi j, iniţializaţi cu left şi, respectiv, cu right. Cît timp v[i] < mark, incrementăm i, apoi, cît timp v[j] > mark, decrementăm j. Dacă acum i <= j, interschimbăm v[i] cu v[j], actualizînd similar indicii i şi j.Tot acest proces continuă pînă cînd i > j. Acum se apelează recursiv funcţia, cu indicii left şi j, respectiv i şi right (dacă left < j, respectiv dacă i < right).

În implementarea polimorfică, deoarece la compilare nu se cunoaşte dimensiunea unei înregistrări, se alocă spaţiu dinamic pentru înregistrarea mark, copiindu-se elementul din mijlocul tabloului şi eliberînd spaţiul la ieşirea din funcţie.

O îmbunătăţire posibilă a metodei este recurgerea la o metodă directă de sortare, dacă lungimea tabloului este inferioară unei

limite prestabilite, evitînd astfel apelul recursiv. De exemplu, la un tablou de 1000 de înregistrări, cînd dimensiunea partiţiei ajunge 2, se vor face 500 de apeluri ale funcţiei, pentru a sorta de fiecare dată un tablou de 2 elemente. Este mult mai eficient dacă acest lucru se face direct. În implementarea de faţă, s-a ales limita 2, caz în care cele două înregistrări se compară direct. Pentru eficienţă, variabilele intens folosite în etapa de partiţionare (i şi j) sînt declarate în clasa register. Implementarea este următoarea:

```
void quick_sort(BYTE *v, size_t size, int left, int right, PFCMP cmp)
{
        register int i, j;
        BYTE *mark;

        i = left; j = right;
        switch(j - i) {
                case 0: return;
```

```
                        case 1: if(cmp(v + i*size, v + j*size) > 0)
                                        swap(v, size, i, j);
                                return;
                        default: break;
                }
                mark = (BYTE *) malloc(size);
                copy(mark, v + ((left + right)/2)*size, size);
                do {
                        while(cmp(v + i*size, mark) < 0)
                                i++;
                        while(cmp(v + j*size, mark) > 0)
                                j--;
                        if(i <= j)
                                swap(v, size, i++, j--);
                } while (i <= j);
                if(left < j)
                        quick_sort(v, size, left, j, cmp);
                if(i < right)
                        quick_sort(v, size, i, right, cmp);
                free(mark);
        }

        void Quick_C (void *v, size_t n, size_t size, PFCMP cmp)
        {
                quick_sort(v, size, 0, (int) n-1, cmp);
        }
```

7.7.2 Implementarea în ASM

Să considerăm acum implementarea algoritmului de sortare rapidă în limbaj de asmblare, într-o versiune specializată (pentru tablouri de întregi). Toate adresele se consideră de tip near, fiind offset-uri în cadrul segmentului adresat prin registrul DS.

Algoritmul este acelaşi cu cel prezentat în 7.7.1, implementarea ASM urmărind exact programul C. Variabilele din descrierea în C a algoritmului sunt menţinute în registrele procesorului şi în stivă.

Parametrii se transmit prin stivă, conform şablonului definit în program iar stiva este descărcată de programul apelant. Această modalitate de apel permite ca funcţia ASM să poată fi apelată din C, dacă modelul cu care se lucrează este small.

```
        .286
        .model small
        .code
        public  _qasm
        ;
        ;       qasm(int v[], int left, int right)
        ;
        sablon          struc
                _bp_ip          dw 2 dup(?)
                v               dw ?
                left            dw ?
                right           dw ?
        sablon ends

_qasm proc near
        push    bp
        mov     bp,sp
```

```
        pusha
        ;
        ;       Asignarea variabilelor
        ;
        ;       i = si, j = di, v = bx, mark = dx
        ;       left, right = în stivă
        ;
        mov     bx, [bp].v                      ; v
        mov     si, [bp].left                   ; i = left
        mov     di, [bp].right                  ; j = right
        mov     ax, di                          ; compară
        sub     ax, si                          ; right - left
        cmp     ax, 1                           ; cu 1
        jng     et00
        jmp     et1                             ; mai mare: se executa
                                                ; rutina normal
et00:
        je      et0                             ; right = left + 1
        jmp     gata                            ; left = right: iesire
et0:
        ;
        ; Sunt 2 elemente
        ; Se compară şi eventual se schimbă
        ;
        shl     si, 1                           ; Întregii sunt pe
        shl     di, 1                           ; doi octeţi
        mov     ax, [bx][si]                    ; v[left]
        cmp     ax, [bx][di]                    ; v[right]
        jnle    aici
        jmp     gata                            ; Sunt O.K.
aici:
        xchg ax, [bx][di]                       ; Dacă nu, se
        mov [bx][si], ax                        ; schimbă
        jmp gata                                ; şi gata
et1:
        ;
        ; Sunt mai mult de două elemente
        ;
        mov     ax, si                          ; Calcul
        add     ax, di                          ; (left + right)/2
        shr     ax, 1
        push    bx
        shl     ax, 1                           ; Adresa se adună
        add     bx, ax                          ; la v
        mov     dx, [bx]                        ; mark = v[(left+right)/2]
        pop     bx
_do:                                            ; Ciclu do
while_i:                                        ; Ciclu while după
                                                ; v[i] < mark
        shl     si, 1                           ; Compară v[i] cu
        cmp     [bx][si], dx ; mark
        jge     end_i                           ; Sfârşit ciclu dacă
                                                ; v[i] >= mark
        shr     si, 1
        inc     si                              ; Dacă nu, se face i++
        jmp     while_i                         ; şi se reia
end_i:
        shr     si, 1
```

```
while_j:                                ; Ciclu while după
                                        ; v[j] > mark
        shl     di, 1
        cmp     [bx][di], dx            ; Compară v[j] cu mark
        jle     end_j                   ; Sfârşit ciclu dacă
                                        ; v[j] <= mark
        shr     di, 1
        dec     di                      ; Dacă nu, se face j--
        jmp     while_j                 ; şi se reia
end_j:
        shr     di, 1
;
        cmp     si, di                  ; Compară i cu j
        jg      _while                  ; Salt dacă i > j
;
        shl     di, 1                   ; Întregii sunt
        shl     si, 1                   ; pe doi octeţi
        mov     ax, [bx][si]            ; Schimbă v[i]
        xchg    ax, [bx][di]            ; cu
        mov     [bx][si], ax            ; v[j]
        shr     si, 1
        shr     di, 1
        inc     si                      ; i++
        dec     di                      ; j--
_while:
        cmp     si, di                  ; Compară i cu j
        jle     _do                     ; Dacă i <= j se continuă
                                        ; ciclul _do
        cmp     [bp].left, di           ; Compară left cu j
        jge     not_rec                 ; Salt daca j <= left
                                        ; j > left: se apelează
                                        ; _qasm pentru prima partiţie
;
;       qasm(v, left, j)
;
        push    di                      ; j
        push    [bp].left               ; left
        push    bx                      ; v
        call    _qasm
        add     sp, 6
not_rec:
        cmp     si, [bp].right          ; Compară i cu right
        jge     gata                    ; Salt dacă i >= right
                                        ; i < right: se apelează
                                        ; _qasm pentru a doua partiţie
;
; qasm(v, i, right)
;
        push    [bp].right              ; right
        push    si                      ; i
        push    bx                      ; v
        call    _qasm
        add     sp,6
gata:
        popa
        pop     bp
        ret
_qasm endp
end
```

```
;
;       Interfaţa cu limbajul C se realizează prin funcţia Quick_a,
;       care are un prototip asemănător cu funcţia Quick_C.
;
;       void Quick_a(void *v, size_t n, size_t size, PFCMP cmp);
;       {
;               qasm(v, 0, (int) n-1);
;       }
;
```

7.7.3 Compararea performanţelor

Evaluarea performanţelor unui algoritm de sortare internă constă în estimarea sau măsurarea numărului de operaţii de comparaţie între elemente şi a numărului de operaţii de copiere sau interschimbare a două elemente. Aceste evaluări se efectuează în trei situaţii distincte de tablouri: tablou aleator, tablou deja sortat (cazul cel mai favorabil) şi tablou sortat invers (cazul cel mai defavorabil).

În cazul de faţă au fost considerate două dimensiuni de tablouri, de 1000 şi respectiv 10000 de întregi, măsurându-se şi timpul de execuţie.

Pentru algoritmul de sortare rapidă considerat, s-au testat 3 variante: prima (Quick_C) este implementarea polimorfică prezentată în 7.7.1, a doua (Quick_a) este implementarea în ASM din 7.7.3 iar a treia variantă (qsort) este funcţia polimorfică din biblioteca standard C.

Rezultatele sînt date în Tabelele 7.2, 7.3 şi 7.4.

	Dimensiune = 1000				***Dimensiune = 10000***			
Alg.	**Timp**	**Comp**	**Move**	**Copy**	**Timp**	**Comp**	**Move**	**Copy**
Quick_C	0.3	12901	2443	608	4.3	187234	31508	6219
Quick_a	0.0	12901	2443	0	0.8	187234	31508	0
Qsort	0.2	13321	-	-	2.9	172655	-	-

Tabelul 7.2 Performanţe în cazul unui tablou aleator

	Dimensiune = 1000				***Dimensiune = 10000***			
Alg.	**Timp**	**Comp**	**Move**	**Copy**	**Timp**	**Comp**	**Move**	**Copy**
Quick_C	0.1	8963	488	488	2.3	121821	4095	4095
Quick_a	0.0	8963	488	0	0.4	121821	4095	0
Qsort	0.1	10819	-	-	2.2	145262	-	-

Tabelul 7.3: Performanţe în cazul unui tablou sortat

	Dimensiune = 1000				*Dimensiune = 10000*			
Alg.	**Timp**	**Comp**	**Move**	**Copy**	**Timp**	**Comp**	**Move**	**Copy**
Quick_C	0.2	8972	988	489	2.4	121832	9094	4095
Quick_a	0.0	8972	988	0	0.4	121832	9094	0
Qsort	0.1	10818	-	-	2.3	145260	-	-

Tabelul 7.4: Performanţe în cazul unui tablou sortat invers

Din tabelele de mai sus rezultă că eficienţa tuturor celor trei implementări este ridicată. Atunci cînd timpul de rulare este însă esenţial, trebuie folosită implementarea specializată în limbaj de asamblare. Implementarea Quick_a asigură un timp de rulare de **3.5 ori mai scurt** în cazul unui fişier aleator şi de **5.5 ori mai scurt** în cazul unui fişier sortat sau sortat invers. Toate implementările au utilizat modelul de memorie small.