תוכניות דוגמא runswap1.c, swaps.asm, swap11.asm, swap12.asm

תוכניות הדוגמא המצוינות ממחישות את עיקרי ההבדלים בתכנות באסמבלי במודלים Small ו-Large. תתי התוכניות swap16, swap32 ו-swap_n מבצעות החלפה של ערכי משתנים (בגדלים שונים) של התוכנית הקוראת לפי פוינטרים שהוכנית הקוראית מעבירה.

ההבדלים הבאים לידי ביטוי בדוגמאות הללו הן:
- בעוד ששני הפוינטרים המועברים כפוינטרים במודל Small נמצאים בכתובות [BP+4] ו-[BP+6] בתוכניות במודל Large הם נמצאים בכתובות [BP+6] ו-[BP+10].

- בעוד שכאשר משתמשים בפוינטרים במודל Small מתיחסים להיסט כאילו הוא כל הכתובת, בתוכניות במודל Large יש לדאוג לטעון לאוגר סגמנט את מרכיב הסגמנט של הפוינטר בטרם נעשה בו שימוש.

התוכנית swap11.asm נכתבה בסגנון 8086 כאשר אוגר הסגמנט ES היה בבחינת אוגר הסגמנט הפנוי היחיד למטרות מהסוג של שימוש בפוינטר מלא. התוכנית swap12.asm מנצלת את העובדה שהחל מה-386 קיימים למטרות מסוג זה 2 אוגרי סגמנט נוספים: FS ו-GS.

```
/* runswap1.c - run swaps */

extern void swap16( int *x, int *y);
extern void swap32(long int *x, long int *y);
extern void swap_n( void *x, void *y, int n);

int main ()
{
 int x=10, y=99;
 long int u=333333, v=777777;
 char str1[10]="Hello", str2[10]="World!";

 printf("x = %d, y = %d\n", x, y);
 swap16(&x, &y);
 printf("x = %d, y = %d\n", x, y);

 printf("u = %ld, v = %ld\n", u, v);
 swap32(&u, &v);
 printf("u = %ld, v = %ld\n", u, v);

 printf("str1 = %s, y = %s\n", str1, str2);
 swap_n(str1, str2, 10);
 printf("str1 = %s, str2 = %s\n", str1, str2);

 return 0;

} /* main */
```

```
E:\>tcc  runswap1.c swaps.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
runswap1.c:
swaps.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   swaps.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  398k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4129872

E:\ACADEMIC\ASM>runswap1.exe
x = 10, y = 99
x = 99, y = 10
u = 333333, v = 777777
u = 777777, v = 333333
str1 = Hello, y = World!
str1 = World!, str2 = Hello

E:\>
```

```
;
;   swaps.asm - swap program model small
;
   .MODEL SMALL
   .CODE
   .386
;
;   extern void swap16( int *x, int *y);
;                       [BP+4]  [BP+6]
;
_swap16 PROC NEAR
   PUBLIC _swap16
    PUSH BP
    MOV BP,SP
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+4]
   MOV DI,[BP+6]
   MOV AX,[SI]
   XCHG  AX,[DI]
   MOV [SI],AX
;
    POP DI
    POP SI
    POP BP
    RET
_swap16 ENDP
;
;   extern void swap32( long int *x, long int *y);
;                       [BP+4]        [BP+6]
;
_swap32 PROC NEAR
   PUBLIC _swap32
    PUSH BP
    MOV BP,SP
    PUSH SI
    PUSH DI
;
   MOV SI,[BP+4]
   MOV DI,[BP+6]
   MOV EAX,[SI]
   XCHG  EAX,[DI]
   MOV [SI],EAX
;
    POP DI
    POP SI
    POP BP
    RET
_swap32 ENDP
```

```
;
; extern void swap_n( void *x, void *y, int n);
;                        [BP+4]    [BP+6]  [BP+8]
;
;
_swap_n PROC NEAR
  PUBLIC _swap_n
   PUSH BP
   MOV BP,SP
   PUSH SI
   PUSH DI
;
  MOV SI,[BP+4]
  MOV DI,[BP+6]
  MOV CX,[BP+8]
  JCXZ ToRet1
Aloop1:
  MOV AL,[SI]
  XCHG  AL,[DI]
  MOV [SI],AL
  INC SI
  INC DI
;
  LOOP Aloop1
;
ToRet1:
   POP DI
   POP SI
   POP BP
   RET
_swap_n ENDP
    END
```

```
/* runswap1.c - run swaps */

extern void swap16( int *x, int *y);
extern void swap32(long int *x, long int *y);
extern void swap_n( void *x, void *y, int n);

int main ()
{
 int x=10, y=99;
 long int u=333333, v=777777;
 char str1[10]="Hello", str2[10]="World!";

 printf("x = %d, y = %d\n", x, y);
 swap16(&x, &y);
 printf("x = %d, y = %d\n", x, y);

 printf("u = %ld, v = %ld\n", u, v);
 swap32(&u, &v);
 printf("u = %ld, v = %ld\n", u, v);

 printf("str1 = %s, y = %s\n", str1, str2);
 swap_n(str1, str2, 10);
 printf("str1 = %s, str2 = %s\n", str1, str2);

 return 0;

} /* main */
```

```
E:\>tcc -ml runswap1.c swap11.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
runswap1.c:
swap11.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   swap11.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  397k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4129616

E:\ACADEMIC\ASM>runswap1.exe
x = 10, y = 99
x = 99, y = 10
u = 333333, v = 777777
u = 777777, v = 333333
str1 = Hello, y = World!
str1 = World!, str2 = Hello

E:\>
```

```
;
;  swap12.asm - swap program model large
;
  .MODEL LARGE
  .CODE
  .386
;
;  extern void swap16( int *x, int *y);
;                      [BP+6]   [BP+10]
;
_swap16 PROC FAR
  PUBLIC _swap16
   PUSH BP
   MOV BP,SP
   PUSH GS
   PUSH FS
   PUSH SI
   PUSH DI
;
  MOV SI,[BP+6]
  MOV GS,[BP+8]
  MOV DI,[BP+10]
  MOV FS,[BP+12]
;
  MOV AX,GS:[SI]
  XCHG  AX,FS:[DI]
  MOV GS:[SI],AX
;
   POP DI
   POP SI
   POP FS
   POP GS
   POP BP
   RET
_swap16 ENDP
;
;  extern void swap32( long int *x, long int *y);
;                      [BP+6]        [BP+10]
;
_swap32 PROC FAR
  PUBLIC _swap32
   PUSH BP
   MOV BP,SP
   PUSH GS
   PUSH FS
   PUSH SI
   PUSH DI
;
  MOV SI,[BP+6]
  MOV GS,[BP+8]
  MOV DI,[BP+10]
  MOV FS,[BP+12]
  MOV EAX,GS:[SI]
  XCHG  EAX,FS:[DI]
  MOV GS:[SI],EAX
;
   POP DI
   POP SI
   POP FS
   POP GS
   POP BP
   RET
_swap32 ENDP
```

```
;
; extern void swap_n( void *x, void *y, int n);
;                       [BP+6]    [BP+10]  [BP+14]
;
;
_swap_n PROC FAR
  PUBLIC _swap_n
   PUSH BP
   MOV BP,SP
   PUSH GS
   PUSH FS
   PUSH SI
   PUSH DI
;
  MOV SI,[BP+6]
  MOV GS,[BP+8]
  MOV DI,[BP+10]
  MOV FS,[BP+12]
  MOV CX,[BP+14]
;
  JCXZ ToRet1
Aloop1:
  MOV AL,GS:[SI]
  XCHG  AL,FS:[DI]
  MOV GS:[SI],AL
  INC SI
  INC DI
;
  LOOP Aloop1
;
ToRet1:
   POP DI
   POP SI
   POP FS
   POP GS
   POP BP
   RET
_swap_n ENDP
    END
```

```
;
;  swap11.asm - swap program model large
;
  .MODEL LARGE
  .CODE
  .386
;
;  extern void swap16( int *x, int *y);
;                      [BP+6]  [BP+10]
;
_swap16 PROC FAR
  PUBLIC _swap16
   PUSH BP
   MOV BP,SP
   PUSH ES
   PUSH SI
   PUSH DI
;
  MOV SI,[BP+6]
  MOV DI,[BP+10]
;
  MOV ES,[BP+8]
  MOV AX,ES:[SI]
  MOV ES,[BP+12]
  XCHG  AX,ES:[DI]
  MOV ES,[BP+8]
  MOV ES:[SI],AX
;
   POP DI
   POP SI
   POP ES
   POP BP
   RET
_swap16 ENDP
;
;  extern void swap32( long int *x, long int *y);
;                      [BP+6]        [BP+10]
;
_swap32 PROC FAR
  PUBLIC _swap32
   PUSH BP
   MOV BP,SP
   PUSH ES
   PUSH SI
   PUSH DI
;
  MOV SI,[BP+6]
  MOV DI,[BP+10]
;
  MOV ES,[BP+8]
  MOV EAX,ES:[SI]
  MOV ES,[BP+12]
  XCHG EAX,ES:[DI]
  MOV ES,[BP+8]
  MOV ES:[SI],EAX
;
   POP DI
   POP SI
   POP ES
   POP BP
   RET
_swap32 ENDP
```

```
;
; extern void swap_n( void *x, void *y, int n);
;                      [BP+6]    [BP+10]  [BP+14]
;
;
_swap_n PROC FAR
  PUBLIC _swap_n
   PUSH BP
   MOV BP,SP
   PUSH ES
   PUSH SI
   PUSH DI
;
  MOV SI,[BP+6]
  MOV DI,[BP+10]
  MOV CX,[BP+14]
;
  JCXZ ToRet1
Aloop1:
  MOV ES,[BP+8]
  MOV AL,ES:[SI]
  MOV ES,[BP+12]
  XCHG  AL,ES:[DI]
  MOV ES,[BP+8]
  MOV ES:[SI],AL
  INC SI
  INC DI
;
  LOOP Aloop1
;
ToRet1:
   POP DI
   POP SI
   POP ES
   POP BP
   RET
_swap_n ENDP
    END
```

address. For example, given the pointer 2F84:0532, we convert that to the absolute address 2FD72, which we then normalize to 2FD7:0002. Here are a few more pointers with their normalized equivalents:

```
0000:0123    0012:0003
0040:0056    0045:0006
500D:9407    594D:0007
7418:D03F    811B:000F
```

Now you know that huge pointers are always kept normalized. Why is this important? Because it means that for any given memory address, there is only one possible huge address—segment:offset pair—for it. And that means that the == and != operators return correct answers for any huge pointers.

In addition to that, the >, >=, <, and <= operators are all used on the full 32-bit value for huge pointers. Normalization guarantees that the results there will be correct also.

Finally, because of normalization, the offset in a huge pointer automatically wraps around every 16 values, but—unlike far pointers—the segment is adjusted as well. For example, if you were to increment 811B:000F, the result would be 811C:0000; likewise, if you decrement 811C:0000, you get 811B:000F. It is this aspect of huge pointers that allows you to manipulate data structures greater than 64K in size.

There is a price for using huge pointers: additional overhead. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers.

Turbo C's Six Memory Models

Avoiding overhead—except when you want it—is just what Turbo C allows you to do. There are six different memory models you can choose from: tiny, small, medium, compact, large, and huge. Your program requirements determine which one you pick. Here's a brief summary of each:

Tiny: As you might guess, this is the smallest of the memory models. All four segment registers (CS, DS, SS, ES) are set to the same address, so you have a total of 64K for all of your code, data, and arrays. Near pointers are always used. Use this when memory is at an absolute premium. Tiny model programs can be converted to .COM format by linking with the /t option.

Small: The code and data segments are different and don't overlap, so you have 64K of code and 64K of static data. The stack and extra segments start at the same address as the

179

data segment. Near pointers are always used. This is a good size for average applications.

Medium: Far pointers are used for code, but not for data. As a result, static data is limited to 64K, but code can occupy up to 1 Mb. This is best for large programs that don't keep much data in memory.

Compact: The inverse of medium: Far pointers are used for data, but not for code. Code is then limited to 64K, while data has a 1-Mb range. This choice is best if your code is small but you need to address a lot of data.

Large: Far pointers are used for both code and data, giving both a 1-Mb range. It is needed only for very large applications.

Huge: Far pointers are used for both code and data. Turbo C normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing static data to occupy more than 64K.

The following illustrations (Figures 12.2 through 12.7) show how memory in the 8086 is apportioned for the six Turbo C memory models.

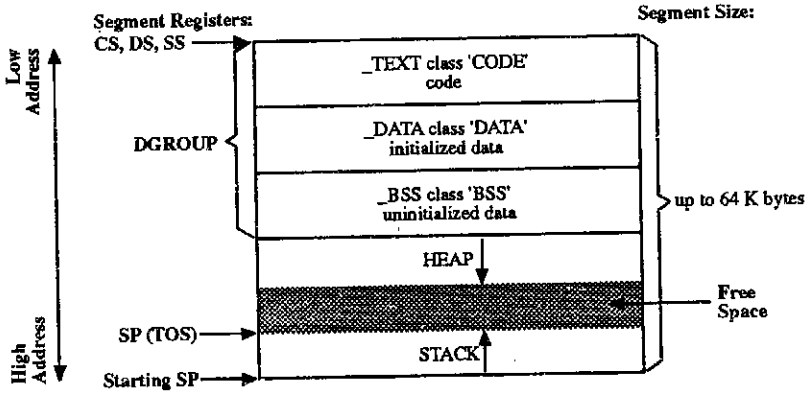

Figure 12.2: Tiny Model Memory Segmentation

180

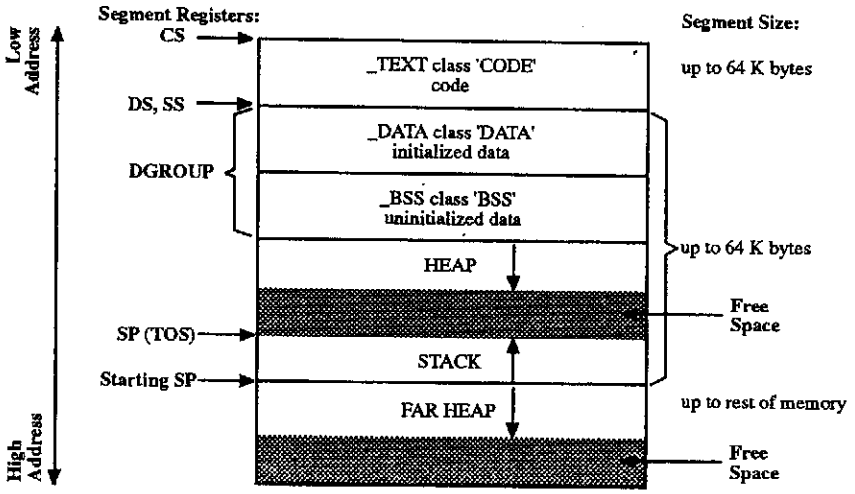

Figure 12.3: Small Model Memory Segmentation

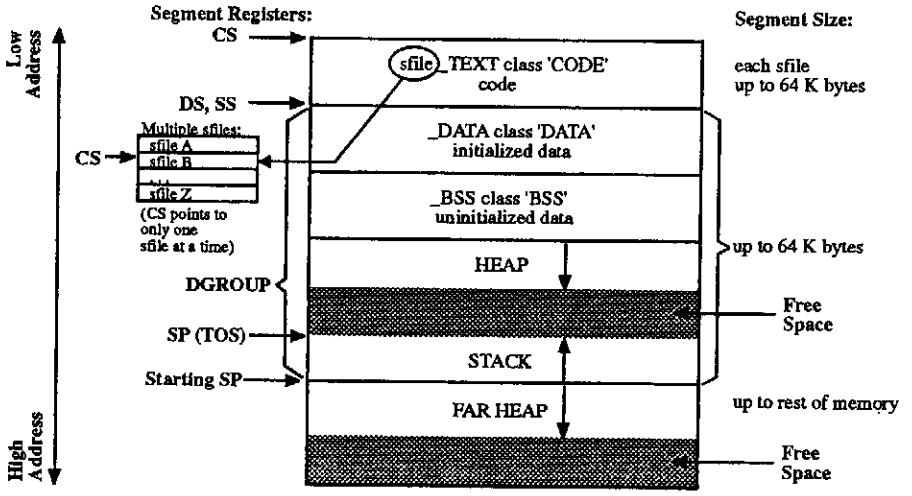

Figure 12.4: Medium Model Memory Segmentation

181

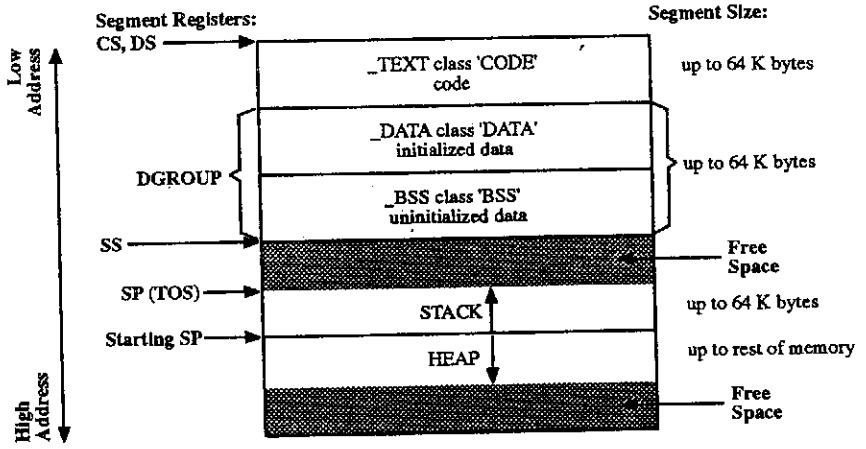

Figure 12.5: Compact Model Memory Segmentation

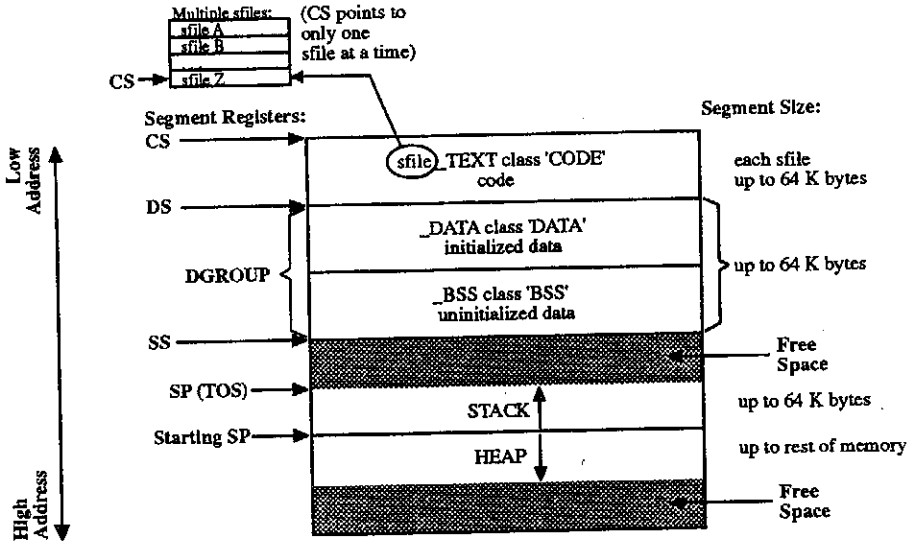

Figure 12.6: Large Model Memory Segmentation

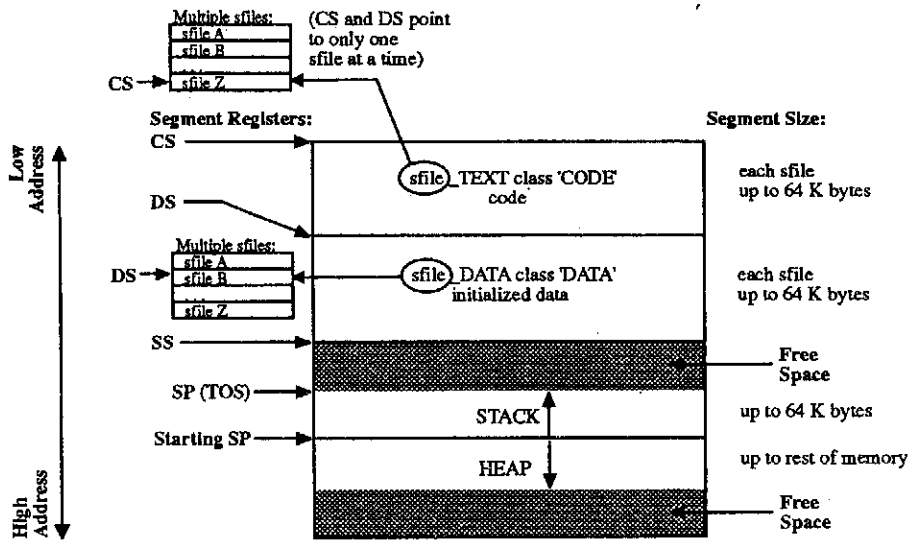

Figure 12.7: Huge Model Memory Segmentation

Table 12.1 summarizes the different models and how they compare to one another. The models are often grouped according to whether their code or data models are *small* (64K) or *large* (1 Mb); these groups correspond to the rows and columns in Table 12.1. So, for example, the models tiny, small, and compact are known as *small code models* because, by default, code pointers are near; likewise, compact, large, and huge are known as *large data models* because, by default, data pointers are far. Note that this is also true for the huge model—the default data pointers are far, not huge. If you want huge data pointers, you must declare them explicitly as huge.

183