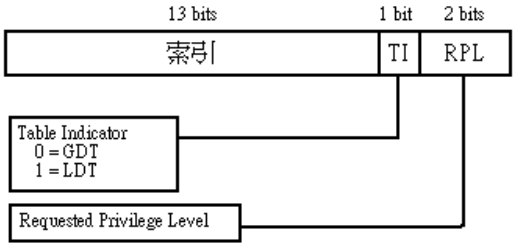
分段架構

Segment Selectors 和分段暫存器

在保護模式中，分段暫存器是存放 segment selector 的。共有六個分段暫存器：CS、DS、ES、FS、GS、和 SS，它們的用途和在實際模式中類似（例如，CS 指向程式碼的 segment，而 SS 指向堆疊的 segment）。Segment selector 由三個欄位組成：索引、Table Indicator（TI）、和 Requested Privilege Level（RPL）。索引是 segment 在 GDT（或 LDT）中的位置，而 TI 表示所要使用的 descriptor table（GDT 或 LDT），而 RPL 則是該 selector 的特權等級。如下所示：



Segment Selector

索引共有 13 bits，因此在一個 descriptor table 中，最多可以有 8192 個 segment。處理器把索引的值乘上 8（一個 segment descriptor 的大小），再加上 GDTR（或 LDTR，根據 TI 的設定）的位址，就得到 segment descriptor 的位址（GDTR 和 LDTR 分別是 GDT 和 LDT 的基底位址）。當 TI 為 0 時，會取用 GDT 中的 segment descriptor，而 TI 為 1 時，則會取用 LDT 中的 segment descriptor。RPL 有 2 bits，範圍可以由 0 至 3（0 的特權等級最高，而 3 最低）。關於 RPL 的一些較詳細的內容，會在保護機制一章中說明。
處理器不會使用 GDT 的第 0 個位置（稱為 null segment）。因此，可以把所有不會用到的分段暫存器設成 null segment（把索引和 TI 均設為 0），以表示目前沒有使用這個分段暫存器。如果試圖要存取 null segment，處理器會發出例外。此外，若把 CS 或 SS 設成 null segment，處理器也會發出 general-protection 例外。
在處理器內部，實際上分段暫存器除了 segment selector 的部份外（這部份稱為 Visible Part），還有一個 Hidden Part。這是為了避免處理器在存取邏輯記憶體時，還要到記憶體中讀取 descriptor table 所造成的額外負擔。當一個分段暫存器被指定一個 segment selector 時，處理器會自動讀入 descriptor table 中的一些資料，並把這些資料放到分段暫存器中。如此一來，處理器就不需要每次都去讀取 descriptor table 中的資料了。不過，如果系統上有多個處理器共用同一個 descriptor table 時，則作業系統要負責在 descriptor table 改變時，重新指定分段暫存器，否則暫存器中的資訊可能會是舊的，而導致錯誤的結果。

Segment Descriptor

Segement Descriptor 存放在 GDT 或 LDT 中，是描述一個 segment 的資料結構。每一個 segment descriptor 都是 8 bytes。下面是一個 segment descriptor 的格式：

Segment Descriptor

在 Segment Descriptor 中，基底位址（共 32 bits）被分為三個部份，而分段邊界（共 20 bits）則被分為兩個部份。分段邊界表示一個 segment 的大小，但是因為它只有 20 bits，所以用 G 來表示它的單位。當 G 為 0 時，大小是以 byte 為單位，即 segment 最大可以到

1MB。當 G 為 1 時，則 segment 的大小是以 4KB 為單位，即 segment 的大小最小是 4KB，最大是 4GB。在以 4KB 為單位時，位址最右邊的 12 bits 在測試 segment 邊界時會被忽略（例如，即使把分段邊界設為 0，在位址 0 到 4095 仍然是合法的）。如果程式試圖存取在 segment 邊界之外的資料，則會產生例外。這樣可以保護其它的 segment 不會被不正確的程式所影響。
P 是用來指示 segment 是否存在記憶體中。如果 P = 0，則表示 segment 目前不在記憶中；反之，若 P = 1，則表示 segment 在記憶體中。當把一個 P = 0 的 segment descriptor 載入到分段暫存器中的時候，處理器會發出一個 segment-not-present 的例外。記憶體管理程式可以利用這個特性，來進行虛擬記憶體管理。這提供了一個不使用分頁功能，也可以進行虛擬記憶體管理的方式。當 P = 0 時，segment descriptor 中的低字組（在上面標示為 0 的字組）可供系統程式自由使用，而高字組（標示為 4 的字組）的 bit 0 ~ bit 7 和 bit 16 ~ bit 31 也都可以供系統程式使用。作業系統可以利用這些空間來存放相關的資訊，例如分段在 swap file 中的位置等等。
Segment descriptor 的高字組中的第 20 個 bit 是可供系統程式自由使用的 bit，作業系統可以在這裡存放相關的資訊。第 21 個 bit 則保留，一定要設為 0。
D/B 在不同的狀況下，有不同的意義。當 segment 是一個可執行的程式碼的 segment，則這個旗標叫 D。D = 0 表示在這個 segment 中的程式內定使用 16-bit 的位址，而 D = 1 表示在這個 segment 中的程式內定使用 32-bit 的位址。若 segment 是一個堆疊或是資料的 segment ，則這個旗標叫 B。B = 0 表示這是一個 16-bit 的 segment，最大值為 FFFFH，而 B = 1 則表示這是一個 32-bit 的 segment，最大值為 FFFFFFFFH。

分段的型態

在 segment descriptor 中的 S 旗標，在 S = 0 時表示 segment 是一個系統 segment（如 LDT），而 S = 1 時則表示這是一個一般的程式 / 資料 segment。在 S = 1 時，型態的最左邊的 bit（即第 11 個 bit）為 0 表示這是一個資料 segment，否則表示這是一個程式 segment。資料 segment 存放程式所用的資料，而堆疊 segment 也算是一種資料 segment。資料 segment 的型態位元，由右至左分別稱為 A、W、和 E（分別是第 8、9、10 bit）。A 是 accessed，而 W 是 Write-enable，E 是 expand-direction。A 位元若設為 0，則在對這個 segment 進行任何存取動作之後，處理器會把 A 設為 1。這個功能可以用在虛擬記憶體管理中，判斷一個 segment 是否需要更新；也可以用來做 debug 的用途。W 位元若設為 1，才可以把資料寫入 segment 中。所以，不希望被意外變更的 segment，可以把它的 W 設為 0。不過，若把一個唯讀（W = 0）的 segment 載入 SS 中，會導致 General-protection 的例外。而 E 位元是 segment「擴展」的方向。一般的 segment（E = 0）是由下往上的，即其偏移量 offset 是由 0 至 segment 的邊界。但是有時候可能會需要由上往下的 segment，例如堆疊 segment 若由上往下會更有彈性（可以動態改變大小），這時就可以把 E 設為 1。E 設為 1 的時候，segment 的有效範圍會是由 segment 的邊界到 segment 的最大值（在 16-bit 的 segment 為 FFFFH，而 32-bit 的 segment 為 FFFFFFFH）。
在程式碼 segment 中，型態位元由右至左分別稱為 A（Accessed）、R（Read-enable）、和 C（Conforming）。這裡的 A 位元和資料 segment 的 A 位元完全相同。R 位元則是指出 segment 是否可讀取。程式碼 segment 可以是只能執行，但是不能讀取。若 R = 0，則不能讀取 segment 裡面的程式（但還是可以執行），若 R = 1，則可以讀取 segment 裡面的程式。C 位元為 1 時，則允許特權等級（CPL）較低（CPL 數字較大）的 segment（裡面的程式）直接執行這個 segment（以原來的 CPL）。若 C = 0，則不允許 CPL 較低的 segment 執行這個 segment（除非經由 call gate 或 task gate）。這些在保護機制一章中會有較細詳的說明。（註：CPL 較高的 segment 永遠不能直接執行 CPL 較低的 segment，即使 CPL 較低的 segment 把 C 設為 1 也不行。）
系統 segment（S = 0）則有很多種，種類同樣是由型態位元決定。例如，有些 segment 是存放 LDT 的，有些是存放 gate 的。系統 segment 的型態位元如下表所示：

bit 11	bit 10	bit 9	bit 8	描述
0	0	0	0	保留
0	0	0	1	16 bit TSS (Available)
0	0	1	0	LDT
0	0	1	1	16 bit TSS (Busy)
0	1	0	0	16 bit Call Gate
0	1	0	1	Task Gate
0	1	1	0	16 bit Interrupt Gate
0	1	1	1	16 bit Trap Gate
1	0	0	0	保留
1	0	0	1	32 bit TTS (Available)
1	0	1	0	保留
1	0	1	1	32 bit TSS (Busy)
1	1	0	0	32 bit Call Gate
1	1	0	1	保留
1	1	1	0	32 bit Interrupt Gate
1	1	1	1	32 bit Trap Gate

從上表可以可以看出，除了 LDT 和 Task Gate 之外（這兩者沒有 16 bit 和 32 bit 之分），bit 11 為 1 的均為 32 bit，而 bit 11 為 0 者為 16 bit，且其它 bit 完全相同。

Segment Descriptor Table

系統中有兩種 Descriptor Table - GDT（Global Descriptor Table）和 LDT（Local Descriptor Table），每個 Descriptor Table 可以存放 8192 個 Segment Descriptors。系統中一定要有一個 GDT，而 LDT 就可以有很多個，也可以不要 LDT。在系統中，GDT 可以被所有的程式和工作（task）所使用，而一個程式可能會有自己的 LDT。
GDT 的基底位址存放在 GDTR 中，是一個線性位址，也就是說，GDT 不算是一個 segment。在 GDTR 中除了存放 GDT 的基底位址外，也存放 GDT 的邊界。GDT 的邊界是以 byte 為單位的，但是因為一個 segment descriptor 總是 8 bytes 長，所以 GDT 的邊界應該設成 8N-1 的形式。
LDT 則和 GDT 不同。LDT 存放在一個系統 segment 中，所以對使用者而言，只需要以一個 segment selector 就可以表示出一個 LDT 的位址和邊界了（當然，這個 segment selector 一定要指向 GDT 中的 segment descriptor）。不過，為了效率的因素，處理器在載入 LDTR 的時候，會自動載入它的線性基底位址和大小、屬性等等資訊。在多工環境下，每個工作可以有自己的 LDT，所以在工作切換時，會自動載入正確的值到 LDTR 中。
在將 GDTR 存放到記憶體中的時候（使用 SGDT 指令），會在記憶體中存放一個 48-bit pseudo-descriptor。因為這個 pseudo-descriptor 是由一個 16 bit 的邊界值和 32 bit 的基底位址所組成的，因此，在存放 GDTR 時，要注意對齊的問題。要避免在某些狀況下（CPL = 3 且 EFLAGS 中的 AC = 1 時）發生 alignment check fault，最好是把 pseudo-descriptor 放在「奇數位址」，即除以 4 會餘 2 的位址。這樣，16 bit 的邊界會對齊在 2 的倍數，而 32 bit 的基底位址也會對齊在 4 的倍數，就不會發生 alignment check fault 了。在存放 IDTR 時，也要和存放 GDTR 時一樣。而存放 LDTR 或工作暫存器（task register）時，則要對齊在 4 的倍數上。

Segment 的規劃

Segment 的彈性很大，可以應用在各種環境中。例如，在簡單的單工作業系統或是嵌入式系統中，可以把所有的 Segment（程式 segment、資料 segment、堆疊 segment 等等）的基底位址都定在 0000000H，大小則定為 4GB。而在比較複雜的系統上，可以把程式 segment 和資料 segment 分開（不重疊），甚至有多個不同的 segment。例如，在多工作業系統中，可以把一個（segment descriptor 在 GDT 中的）segment 分配給一個 process，在該 process 中再以 LDT 來分配區域性的 segment（如程式 segment、資料 segment 等等）。即使是在單工作業系統中，區分適當的 segment 也會有些好處。例如，segment 的保護機制（參考「保護機制」的「邊界和型態檢查」），可以加強系統的強固性，也可以幫助程式設計師找出程式中的錯誤。
有些系統的 segment（例如：各種 gate、TSS、LDT 等）也是很重要的。把這些系統的 segment 和一般的 segment 分開，也可以避免程式意外破壞了這些系統 segment 的內容。當然，這樣就必須把修改這些系統 segment 內容獨立成為作業系統的 API，可能會稍微影響效率（經由作業系統提供的 API 來改變這些資料，總是比直接修改慢一點）。