



x86 Addressing Concepts

May 3, 2004

by
David Moore

This whitepaper briefly explains x86 addressing complexities, such as segmentation and memory models.

Segmentation

Segmentation is a type of memory addressing used in x86 processors, in which addresses are broken into two parts: a *segment* and an *offset*. The first x86 processors did this in order to create a 20-bit address from a 16-bit segment and a 16-bit offset. Later processors introduced a new processor mode called *protected mode* that still used segmented addressing, but now there was not a direct correlation between the segment and the address. In protected mode, the segment is an index into a table, called a *descriptor table*. Each entry in the descriptor table describes a segment. Among other things, it specifies the base address and size. The offset of an address is added to the base address to give the linear address in memory where the routine or variable begins.

In 16-bit modes, segments are limited to 64K bytes in size, so this limits the size of arrays and other data. It also limits how much code can be referenced by the cs register at one time. This 64K byte limitation requires larger applications to be split into multiple code and or data segments. This is the root of the complexity that x86 segmentation causes. Note that in 32-bit segmented systems, a segment can be up to 4GB in size, so code and data can be meaningfully organized into segments, if desired, instead of being arbitrarily organized just to avoid overflow. Segmentation adds the notions of near and far, and memory models. These are the topics of this section.

Memory Addresses

x86 processors, starting with the 386, support these addressing modes:

- (1) 16-bit real mode
- (2) 16-bit protected mode
- (3) 32-bit flat protected mode
- (4) 32-bit segmented protected mode

smx currently supports the first three of these modes.

In the first mode, memory addresses consist of 16-bit *offsets* and 16-bit *segments*. Addressing range is 1 MB. A variation on this mode offered by VAutomation and supported by Paradigm C++ is “extended mode,” which gives an address range of 16 MB, by left-shifting the address 8

bits instead of the usual 4 when constructing the address, to form a 24-bit address instead of a 20-bit address. Segments in extended mode are still limited to 64KB.

In the second mode, addresses consist of 16-bit offsets and 16-bit *selectors*. Addressing range is 4GB — but segments are still limited to 64KB.

In the third mode, addresses consist of 32-bit offsets from 0 called *linear addresses*. Addressing range is 4GB. This addressing mode is usually referred to as *flat mode* or *flat model*, since the code is not segmented. That is, flat mode applications do not change segment registers. (Those using flat mode should read the 32-bit version of the *smx* Target Guide instead of this manual.)

In the fourth mode, addresses consist of 32-bit offsets and 16-bit segments. Addressing range is 4GB. Code and data can be organized into as many or few segments as desired, of any size. This mode gives the advantages of segment protection without any constraints on segment sizes.

Near and Far

near is a C keyword that indicates that the associated object may be accessed with an address relative to the segment in *ds*. (This is a flat 16-bit pointer in 16-bit modes or a flat 32-bit pointer in 32-bit protected mode.) Most C compilers put near data objects into *DGROUP* (discussed below). A near data object is accessible without changing *ds*. When applied to code, near means that it is possible to branch to the code without changing *cs*.

far is a C keyword that indicates that the associated object must be accessed with an address of the form *segment:offset*. (This is a 16:16 pointer in 16-bit modes or a 16:32 pointer in 32-bit protected mode.) When applied to data, this means that a segment value must be loaded into *ds* or *es* to access the data. When applied to code, it means that a branch to the code requires changing *cs*.

Data and code can be explicitly made near or far using the keyword *near* or *far*. Otherwise, the compiler's memory models dictate this.

Memory Models

Memory models are defined by x86 compilers to provide rules that specify whether particular data or code is *near* or *far*, when this is not explicitly specified using the **near** or **far** keywords in the code. Memory models serve as a set of default rules to save the programmer from having to specify these keywords in every declaration in the code.

The goal for efficiency is to use the smallest memory model possible. Near pointers require less memory to store than far pointers, and the use of near pointers results in faster code. However, the total sizes of code and data determine the minimum memory model that can be used, since segments are limited to 64K bytes. For example, a small model program can have at most 64KB of data and 64KB of code.

16-bit segmented versions of *smx* support the following memory models defined by the C compilers:

- | | | |
|--------------|-----------|-----------|
| (1) small: | near code | near data |
| (2) compact: | near code | far data |
| (3) medium: | far code | near data |
| (4) large: | far code | far data |

Another way to look at this chart is that near implies one segment and far implies multiple segments. So, for example, compact model has one code segment and multiple data segments.

In addition to these memory models, *smx* supports huge files (those that have their own DGROUP). This is discussed in the *smx* User's Guide, in section x86 Issues for *smx*/ huge files.

32-bit flat protected mode has only one memory model that is equivalent to small model.

32-bit segmented protected mode is not supported by *smx*. The few segmented 32-bit compilers that exist typically support models similar to those listed above for 16-bit modes, but may provide full support (e.g. run-time library, etc.) only for compact model.

What is DGROUP?

DGROUP is a construct used by many 16-bit x86 compilers. Unfortunately, it may not be well explained in your compiler's manuals. For that reason, we'll offer a brief explanation here, since understanding DGROUP is important.

DGROUP is often referred to as the *default data segment*, but it is not a segment.

DGROUP is a *group* of segments consisting of the *_DATA* segment for initialized variables, the *_BSS* segment for uninitialized variables, and other segments, as determined by the compiler and your application. In essence, a group is a super-segment that is treated like an ordinary segment by the linker. For 16-bit systems, a group is limited to 64K bytes, just like an ordinary segment.

The importance of DGROUP is that, for all memory models, except Borland huge model, the *ds* register points to the start of DGROUP. Hence, to access a variable in DGROUP, it is necessary only to supply an offset. All near memory is in DGROUP. Far variables, by contrast, are normally accessed by first loading a segment address or selector into *es*, then supplying an offset. Usually, this process is repeated for every far variable access — even if successive accesses are to variables in the same segment! *Clearly, near variable accesses are much more efficient.* (Note: Microsoft *based variables* get around the foregoing problem by loading the segment register only once for successive accesses to variables in the same far segment.)

In older versions of Microsoft C and Borland C, variables not labeled *near* were, by default, *far* in *large data* models (*compact*, *medium*, and *huge*). For newer versions of Microsoft Visual C++ and Borland C++, the reverse is true: Variables not labeled *far* are, by default, *near* in large data models. There are some exceptions to this rule, which are discussed in the section x86 Issues for *smx*/ DGROUP overflow.

In large data models, *ds* does not always point to DGROUP — sometimes it must be used for other purposes, such as string instructions. However, *ds* is guaranteed to point to DGROUP before and after function calls, and it points to DGROUP most other times, as well.