Author: Steve Gorman
Title: Programming with the Intel architecture in the flat memory model

Abstract: As the Intel architecture moves off the desktop into a variety of other computing applications, developers will need to design their systems to take full advantage of the architecture's performance and extended addressing. Intel architecture's flat memory model provides for the fastest code execution and simplest system configuration. This paper will discuss:

- How to initialize the Intel386™ and Intel486™ processors to run in protected mode using the flat memory model.
- Issues, benefits and drawbacks associated with using the flat memory model.
- Tools available for programming in the flat memory model.

The flat memory model is Intel architecture's answer to "No more segmentation." This paper will cover how to initialize a system as well as the advantages, and tools available for running in the flat memory model. The paper will also compare flat memory model performance to real-mode and segmented protected mode operation.

What is Flat Memory Model

When running in the flat memory model the system designer essentially "Turns-Off" the segmentation feature of the 80386 architecture, by loading the segment registers with selectors for descriptors that have a base addresses of 0, privilege level set to 0 (full access rights), and by setting the limits to allow access the entire 32-bit address space. Once this is done there is no longer a need to change the segment registers. The 32-bit offsets used by the 80386 instructions, in protected mode, are sufficient to access the entire linear address space.

Why Would I Want to Use the Flat Memory Model

Here are some of the reasons:

- Extend the addressing capability beyond 1MB.
- Remove the 64KB barrier on segments.
- Eliminate segmentation.
- Improve the portability of applications to other architectures that support a flat linear address space.
- Performance over 16-bit real-mode or segmented protected mode.
- The tools available for developing flat memory.
- Future Applications

Extending the addressing capabilities to over 1MB of memory, can be done on any system using tricks. However, this typically hurts performance, adds to system cost, and is difficult to implement and maintain. This makes the 386 protected mode a good solution to the problem. To remove the 64KB segmentation barrier an 80386 architecture requires you to run the processor in 32-bit protected mode. For both of these reasons your only remaining question is segmented or flat memory models (see Protected Flat or Protected Segmented Model? section).

This leads to the third reason listed; eliminating segmentation. The desire to eliminate segmentation is likely someone thats had to work around the 64KB segmentation limit of the 8086 or is not familiar with a segmented architecture. Eliminating segmentation just for the sake of it, isn't a very good reason, but if it is to improve portability to other architectures that's a more reasonable one (see Protected Flat or Protected Segmented Model? section).

If moving to 32-bit protected mode because you believe it will boost your performance, requires you to take a close look at your application to determine if you will actuall get a performance boost (see Protected Mode Flat Memory Model over Real-Mode section). A performance boost this is not always a given, even when running the processor in 16-bit mode the application still has access to the 32-bit registers and instructions.

Tools are yet another interesting reason to choose segmented protected mode over flat protected mode. A number of tool vendors have appeared that make excellent tools for developing applications under the flat memory model, mainly due the popularity of DOS extenders. Many of these same tool vendors also support the development of a segmented memory model.

Protected Mode Flat Memory Model over Real-Mode

If your application needs access to greater than 1MB of memory or it manipulates data objects of greater than 64KB frequently, then protected mode is definitely the way to go. If you are looking at flat memory model over real-mode to get extra performance the answer isn't so obvious, you need to look at what your application is doing. If most of its time is spent dealing with 32-bit objects and with large data arrays then protected mode is likely to provide an advantage in performance over real mode, if your application doesn't have these 32-bit attributes then real-mode may prove to perform the best.

If you look closely at the instruction set you will see that instructions executed in real-mode are equal to or faster than the same instruction when executed under protected mode. A few years back the main advantage in performance that protected mode (specifically flat memory model) offered was due primarily in the difference in compiler technology. The compilers that supported protected mode programming were far more advanced at optimizing code for the 80386 processors instruction set than the existing real-mode compilers. It is important to note here that even in real-mode you still have full access to the 32-bit instructions. In recent years the real-mode compilers have come a long way in optimizing its use of the 80386 processors instruction set. This could potentially significantly reduce the performance advantage offered by the 32-bit specific compilers. All you may need to do is upgrade your compiler or turn on the compiler switch that indicates a 386 target system.

Protected Flat or Protected Segmented Model?

What flat memory model offers over the segmented model is primarily, simplicity (eliminates segmentation), improved portability to other flat 32-bit architectures, and possibly tools. Some might argue that you also get greater performance, but I believe this to be negligible. The elimination of segmentation is the main argument used to justify that performance is improved. It is true that using far pointers, which would be required from time to time in the segmented model, is slower than using near pointers (no segment register needed), but it seems that when people are comparing the segmented to flat memory models they associated segmentation with the 8086's segmentation. With the 8086 each segment was limited to only 64KB and if an application wanted to work with objects greater than 64KB the programming became much more complex. With protected mode this limitation is removed, allowing segments to be up to 4GB (or the full address space) each. With this limitation remove each individual application worries little about segmentation which is typically only needed when interfacing with the operating system. Assuming that interfacing to the operating system is done a very small percentage of the time the performance difference between the two should be very little.

Simplicity is good for the obvious reason, but the negative is all of the protection lost. The protection feature aids significantly in the debugging of the application and in the ability to produce a more secure and stable system. Even if all of the protection features are not used, just having segmentation around is a significant debugging advantage.

Portability is one of the better reasons to use the flat memory model. Most of the 32-bit architectures available use a flat memory model. Therefore porting your memory manager, task managers, I/O systems, and other software is easier to do when using the flat memory model.

Tools are yet another good reason you may want to use the flat memory model, especially if you are embedding DOS and BIOS into your target system. There are a number of compilers and tools for developing applications for the Intel architecture. When developing an application to run under the DOS environment flat memory model development is enable by what is called a DOS extender, which is produced by a number of different vendors. If DOS is not part of your system, then most of the compilers and linkers/locators that support the flat memory model can also support the segmentation memory model.

TOOLS

There are an abundant amount of development tools available for developing applications for the Intel architecture. Of these tools, the tools designed for developing DOS based applications are by far the most popular. For real-mode based development the Microsoft and Borland tools have been the most popular, recently even these vendors have added support for 32-bit protected mode development. They have also greatly improved the optimization for 80386 based processors for both real and protected mode DOS and Windows based development. Other vendors like Watcom and Metaware have been producing compilers for the 32-bit Intel processor for a greater period of time and have created quite a following in the 32-bit environment. These compiler vendors primarily target DOS extender and 32-bit Windows application development while other vendors provide tools that use these compiler technologies to target the development to the embedded environment. Vendors like Pharlap, Systems & Software Inc., Paradigm, and Concurrent Sciences provide the Linkers/Locators, and other tools that enable the use of these compilers in the embedded world. Nearly all of the tool vendors provide compilers and other tools for the Intel processors.

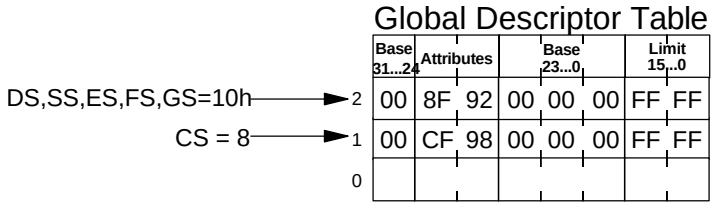
If DOS is expected to be part of the system, the use of a DOS extender is a easy way to get beyond the 1MB and 64KB segment barrier of DOS and real mode. These DOS extenders are available from a number of different vendors and are typically shipped with the various compilers that support the development of DOS extender applications. A good technical reference on DOS extenders and how they work is *DOS and Windows Protected Mode: Programming with DOS Extenders in C* , by Al Williams.

Initializing the Processor to Run Flat Memory Model

To use the flat memory model the processor must run in protected mode. To switch the processor into protected mode:

- First a global descriptor table (GDT) must be created. This table is created by the tools used to build the software (typically called a builder or locator) at run-time, or from within a source file. The 32-bit protected mode compilers and builder / locator tools provide support for generating flat memory model code.
- Next point the CPU register GDTR to the created GDT.
- Set the protect enable (PE) bit in the machine status word (MSW or CR0).
- Execute a short jump, to flush the processors prefetch queue.
- Initialize the data (DS, ES, FS, and GS) and stack (SS) segment registers with a selector for the data segment.
- Initialize the ESP register to point to the top of the stack
- Execute a far jump (one that will load the CS register) to code that will initialize the rest of the hardware like, serial ports, interrupt descriptor table, timers, interrupt controllers, etc...

Because this is the simplest protected mode configuration, the global descriptor table requires only a few entries. Again, many of the tools have controls that will create the descriptor table for you for the flat memory model. The following is a sample of what the GDT table could look like:



This table could be more complex, but that is entirely up to the system developer. Typically you would find aliases in the GDT for the GDT itself and the IDT, but they are not required.

Programming for the Flat memory model

If porting code from an architecture that is based on a 32-bit flat memory model, your biggest concern is that of porting to any new architecture, but if moving from an 8086 or 16-bit environment there is more than just the architecture you need to be concerned with. If most of the existing code is written in a high level language, your major concern will be related to the size difference between data types. For example, with most 16-bit "C" compilers the default size of an "int" is 16-bit, where as for a 32-bit "C" compiler an "int" will most likely be 32-bits. This problem can be minimized by using good programming practices, i.e. create user definable types (for example use typedef's in "C") for size dependent variables. This increase in the size of an "int" can also have an impact on the overall memory requirements and system performance. If data objects go from 16 to 32-bits, then obviously cause the memory requirements to grow. This may be unnecessary and can be avoided by declaring variables to be only as large as needed and not relying on the default size of a high level language.

Another 16 verses 32-bit concern is stack operations. Since for the 32-bit environment items are put on to the stack aligned to 32-bits and for the 16-bit environment the stack would be aligned to 16-bits. The main source of problems with the change in stack size is for assembly language source files that must retrieve parameters from the stack. This is probably the biggest headache when moving from 8086 environment to the 32-bit protected mode environment. Essentially all assembly language functions that retrieve parameters must be modified, i.e. MOV AX, [BP-2] must change to MOV AX, [BP-4]. This will also have an effect on performance when dealing with 16-bit bus versions of the 386 processor. If you are dealing with a large amount of assembly language functions and find a lot of these types of issues existing in your code, you may want to consider looking at 16-bit protected mode. 16-bit protected mode offers greater than 1MB of addressabiltiy and still has the 64KB segment limits, but does not offer a flat memory model (16-bit protected mode is outside the scope of this paper).

Performance

To maximize your performance you should take a closer look at is how your program can be changed to take full advantage of the extended addressing and the enhanced instruction set. One obvious performance enhancement is in the 32-bit instructions, but remember real-mode applications also have access to these 32-bit instruction. A more significant improvement is in the new 32-bit addressing mode. This new addressing mode allows more CPU registers to be used as index addressing registers and adds scaled index addressing. The 32-bit addressing mode is available in real-mode, but is difficult to implement and most real-mode based compiler do not do a good job using this capability. The programmer should also look at how they can take advantage of the greatly increase segment size and memory space.

When using 16-bit bus versions of the 386 processor, like the Intel386™ SX, CX, or EX processors, performance can be significantly impacted when using the processor in 32-bit mode. On a 16-bit bus system when using 32-bit data items the processor will perform a two fetch operation from memory instead of one for 16-bit data items, this is obviously slower. This impact can be quite significant and could void the performance gains of using a 32-bit processor. So, in the case of 16-bit systems, care should be taken to declare 32-bit data items only when needed. It is also important to note that most protected mode compilers do not take the 16-bit bus versions of the processor into consideration. This can be best seen when moving from a 16-bit real mode compiler to a 32-bit protected mode compiler, the default size of an "int" changes from 16-bits to 32-bits.

Next on the list of performance issues is interrupt response time. The interrupt process for protected mode systems has more overhead associated with it than real-mode based systems, causing a slower interrupt response time. The hope is that the 32-bit capabilities of protected mode make up the difference.

If using a DOS extender to take advantage of the 32-bit capabilities of the 80386 processor, you should consider the performance issues pointed out above with the additional overhead of interfacing to a real-mode operating system from a protected mode application. When making operating systems call to DOS or BIOS the protected mode DOS extender must switch the processor back into real-mode to make the function call. Once the call is finished the processor must then be switched back into protected mode. This has a considerably greater amount of overhead as compared to making the system call from a real-mode DOS program, but since many programs make operating system calls a small percentage of the time this many not be a significant performance hit.

Summary

As you can see, the 80386 processor offers the system developer a wide range of capabilities and choices. The flat memory model is just one of these. The flat memory model does offer the easiest development solution, but not necessarily the best. With the great popularity of the Intel architecture there are plenty of tools that are available to help you develop the system you desire. Along with the vast number of tools is also an abundance of reference materials on the Intel architecture. For more information on the Intel architecture, I recommend the *I48 i486(TM) Microprocessors Programmer's Reference Manual* and *80386 Systems Software Writer's Guide* by Intel. For more information on how DOS extenders work, I recommend *DOS and Windows Protected Mode: Programming with DOS Extenders in C* by Al Williams.