Straighten Up and Fly Right

Aligning Your Data for Speed and Correctness

Written by Jonathan 'Wolf' Rentzsch (jon at redshed dot net)

Abstract

Data alignment is an important issue for all programmers who directly use memory. Data alignment affects how well your software performs, and even if your software runs at all. By understanding the causes behind alignment, we also can explain some of the "weird" behaviors of some processors.

Introduction

Straighten up and fly right
Straighten up and do right
Straighten up and fly right
Cool down papa, don't you blow your top!

If you program in C, C++, Objective C or Pascal, then you've been trained to think of memory as an array of bytes. Everything's size is measured in bytes. A `Ptr` is `typedefed` as a `char*`, a `Handle` as a `char**`.

However, your computer's processor does not read from and write to memory in only byte-sized chunks. Instead, it accesses it in two-, four-, eight- sixteen- or even 32-byte chunks. We'll call the size in which a processor accesses memory its **memory access granularity**.

The difference between how high level programmers think of memory and how modern processors actually work with memory raises interesting issues that are explored in this paper.

If you don't understand and address alignment issues in your software, the following scenarios, in increasing order of severity, are all possible:

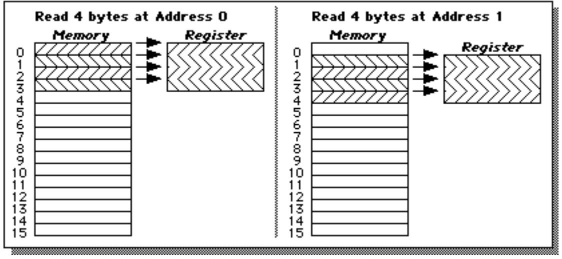
- Your software will run slower.
- Your application will lock-up.
- Your operating system will crash.
- Your software will silently fail, yielding incorrect results.

Alignment Fundamentals

To illustrate the principles behind alignment, let's examine a constant task, and how it's affected by a processor's memory access granularity. The task is simple: first we read four bytes from address 0 into the processor's register. Then we read four bytes from address 1 into the same register.

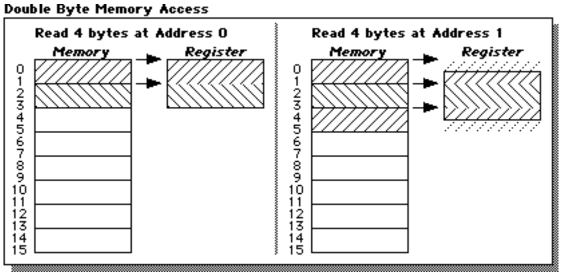
First let's examine what would happen on a processor with a one-byte memory access granularity:

Single Byte Memory Access



This fits in with the naïve programmer's model of how memory works: it takes the same four memory accesses to read from address 0 as it does from address 1. Now let's see what would happen on a processor with two-byte granularity, like the original 68000:

Double Byte Memory Access



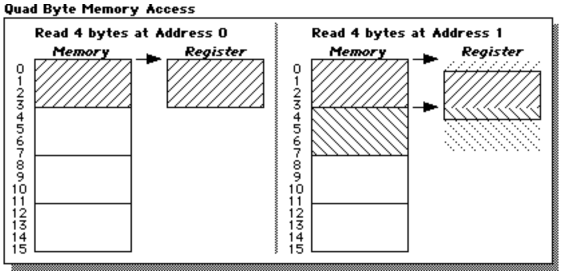
When reading from address 0, a processor with two-byte granularity takes half the number of memory accesses as a processor with one-byte granularity. Because each memory access entails a fixed amount overhead, minimizing the number of accesses can really help performance.

However, notice what happens when reading from address 1. Because the address doesn't fall evenly on the processor's memory access boundary, the processor has extra work to do. Such an address is known as an **unaligned address**.

Because address 1 is unaligned, a processor with two-byte granularity must perform an extra memory access, slowing down the operation.

Finally, let's examine what would happen on a processor with four-byte memory access granularity, like the 68030 or 601:

Quad Byte Memory Access



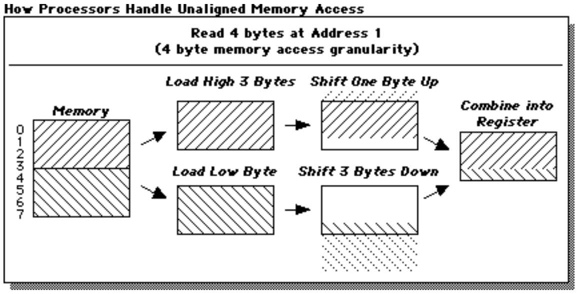
A processor with four-byte granularity can slurp up four bytes from an aligned address with one read. Also note that reading from an unaligned address doubles the access count.

Now that you understand the fundamentals behind aligned data access, we can explore some of the issues related to alignment.

Lazy Processors

A processor has to perform some tricks when instructed to access an unaligned address. Going back to our example of reading four bytes from address 1 on a processor with four-byte granularity, we can work out exactly what needs to be done:

How Processors Handle Unaligned Memory Access



The processor needs to read the first chunk of the unaligned address and shift out the "unwanted" bytes from the first chunk. Then it needs to read the second chunk of the unaligned address and shift out some of its information. Finally, the two are merged together for placement in the register. It's a lot of work.

Some processors just aren't willing to do all of that work for you.

The original 68000 was a processor with two-byte granularity and lacked the circuitry to cope with unaligned addresses. When presented with such an address, the processor would throw an exception. The Mac OS didn't take very kindly to this exception and would usually demand the user restart the machine. Ouch.

Later processors in the 680x0 series, such as the 68020, lifted this restriction and performed the necessary work for you. This explains why some software that works on the 68020 crashes on the 68000. It also explains why, to this day, some old Mac coders stuff uninitialized pointers with an odd addresses. On the original Mac, if the pointer was accessed without being reassigned to a valid address, the Mac would immediately drop into the debugger. Often you could then examine the calling chain stack and figure out where your mistake was.

All processors have a finite number of transistors to get work done. Adding unaligned address access support cuts into this "transistor budget". These transistors could otherwise be used to make other portions of the processor work faster, or add new functionality altogether.

An example of a processor that sacrifices unaligned address access support in the name of speed is MIPS. MIPS is a great example of a processor that does away with almost all frivolity in the name of getting real work done faster.

The PowerPC takes a hybrid approach. Every PowerPC processor to date has hardware support for unaligned 32-bit integer access. While you still pay a performance penalty for unaligned access, it tends to be small.

On the other hand, modern PowerPC processors lack hardware support for unaligned 64-bit floating-point access. When asked to load an unaligned floating-point number from memory, modern PowerPC processors will throw an exception and have the operating system perform the alignment chores *in software.* Performing alignment in software is *much* slower than performing it in hardware.

Speed

To illustrate the performance penalties of unaligned memory access, let's write some tests. The test is simple: we read, negate and write back the numbers in a ten-megabyte buffer. These tests have two variables:

1. The size, in bytes, in which we process the buffer.

   First we'll process the buffer one byte at a time. Then we'll move onto two-, four- and eight-bytes at a time.

2. The alignment of the buffer.

   We'll stagger the alignment of the buffer by incrementing the pointer to the buffer and running each test again.

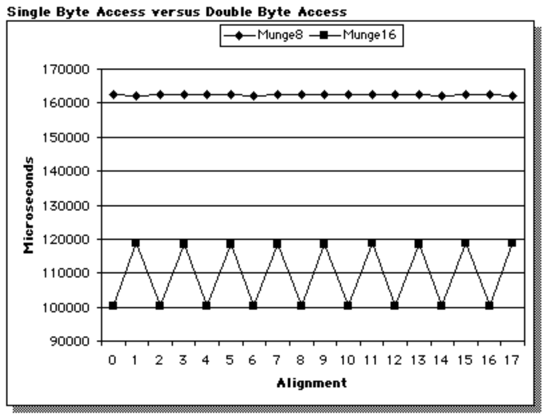
These tests were performed on a 400 MHz PowerBook G3. To help normalize performance fluctuations from interrupt processing, each test was run ten times, keeping the average of the runs. First up is the test that operates on a single byte at a time:

```
  void
Munge8(
  void    *data,
  UInt32  size )
{
  UInt8   *data8 = (UInt8*) data;
  UInt8   *data8End = data8 + size;

  while( data8 != data8End ) {
    *data8++ = -*data8;
  }
}
```

It takes an average of 162,445 microseconds to execute this function. Let's modify it to work on two bytes at a time instead of one byte at a time -- which will halve the number of memory accesses:

```
  void
Munge16(
  void    *data,
  UInt32  size )
{
  UInt16  *data16 = (UInt16*) data;
  UInt16  *data16End = data16 + (size >>
1); /* Divide size by 2. */
  UInt8   *data8 = (UInt8*) data16End;
  UInt8   *data8End = data8 + (size &
0x00000001); /* Strip upper 31 bits. */

  while( data16 != data16End ) {
    *data16++ = -*data16;
  }
  while( data8 != data8End ) {
    *data8++ = -*data8;
  }
}
```

This function takes 100,102 microseconds to process the same ten-megabyte buffer -- 38% faster than `Munge8`. However, that buffer was aligned. If the buffer is unaligned, the time required increases to 118,819 microseconds -- about a 19% speed penalty. The following chart illustrates the performance pattern of aligned memory accesses versus unaligned accesses:

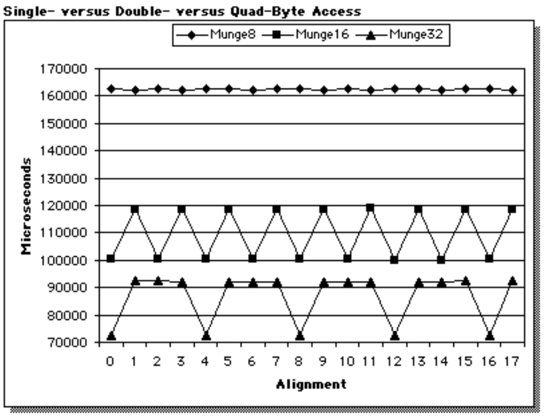


First thing we notice is that accessing memory one byte at a time is uniformly slow. The second item of interest is that when accessing memory two bytes at a time, whenever the address is not evenly divisible by two, that 19% speed penalty rears its ugly head.

Now let's up the ante and process the buffer four bytes at a time:

```
  void
Munge32(
  void    *data,
  UInt32  size )
{
  UInt32  *data32 = (UInt32*) data;
  UInt32  *data32End = data32 + (size >>
2); /* Divide size by 4. */
  UInt8   *data8 = (UInt8*) data32End;
  UInt8   *data8End = data8 + (size &
0x00000003); /* Strip upper 30 bits. */

  while( data32 != data32End ) {
    *data32++ = -*data32;
  }
  while( data8 != data8End ) {
    *data8++ = -*data8;
  }
}
```

This function processes an aligned buffer in 72,208 microseconds and an unaligned buffer in 92,453 microseconds, respectively. While both times are faster than processing the buffer two bytes at a time, the speed penalty for four-byte unaligned memory accesses rises to 28%:

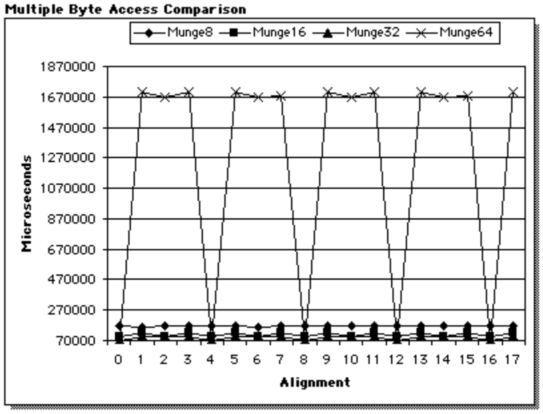


Now for the horror story: processing the buffer eight bytes at a time.

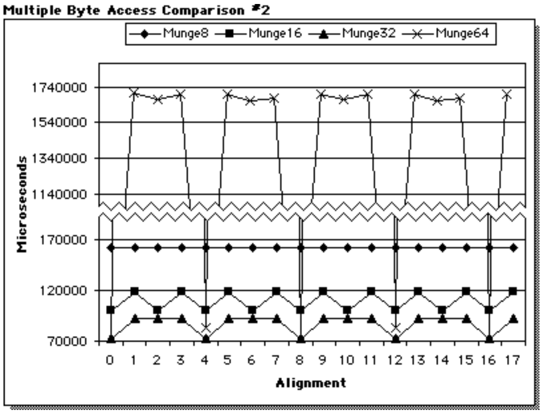
```
  void
Munge64(
  void    *data,
  UInt32  size )
{
  double  *data64 = (double*) data;
  double  *data64End = data64 + (size >>
3); /* Divide size by 8. */
  UInt8   *data8 = (UInt8*) data64End;
  UInt8   *data8End = data8 + (size &
0x00000007); /* Strip upper 29 bits. */

  while( data64 != data64End ) {
    *data64++ = -*data64;
  }
  while( data8 != data8End ) {
    *data8++ = -*data8;
  }
}
```

Munge64 processes an aligned buffer in 68,780 microseconds -- about 5% faster than processing the buffer four bytes at a time. However, processing an unaligned buffer takes an amazing 1,700,243 microseconds -- two orders of magnitude slower than aligned access, an outstanding 2,372% performance penalty!

What happened? Because the G3 lacks hardware support for unaligned floating-point access, the processor throws an exception *for each unaligned access.* The operating system catches this exception and performs the alignment in software. Here's a chart illustrating the penalty, and when it occurs:



The penalties for one-, two- and four-byte unaligned access are dwarfed by the horrendous unaligned eight-byte penalty. Maybe this chart, removing the tremendous gulf between the two numbers, will be clearer:



Atomicity

The 68K and PowerPC, like most modern processors, offer atomic instructions. These special instructions are crucial for synchronizing two or more concurrent tasks. As the name implies, atomic instructions must be *indivisible* -- that's why they're so handy for synchronization: they can't be preempted.

It turns out that in order for atomic instructions to perform correctly, the addresses you pass them must be at least four-byte aligned. This is because of a subtle interaction between atomic instructions and virtual memory.

If an address is unaligned, it requires at least two memory accesses. But what happens if the desired data spans two pages of virtual memory? This could lead to a situation where the first page is resident while the last page is not. Upon access, in the middle of the instruction, a page fault would be generated, executing the virtual memory management swap-in code, destroying the atomicity of the instruction. To keep things simple and correct, both the 68K and PowerPC require that atomically manipulated addresses always be at least four-byte aligned.

Unfortunately, the PowerPC does not throw an exception when atomically storing to an unaligned address. Instead, the store simply always fails. This is bad because most atomic functions are written to retry upon a failed store, under the assumption they were preempted. These two circumstances combine to where your program will go into an infinite loop if you attempt to atomically store to an unaligned address. Oops.

Altivec/Velocity Engine

Altivec is all about speed. Unaligned memory access slows down the processor and costs precious transistors. Thus, the Altivec engineers took a page from the MIPS playbook and simply don't support unaligned memory access. Because Altivec works with sixteen byte chunks at a time, all addresses passed to Altivec must be sixteen-byte aligned. What's scary is what happens if your address is not aligned.

Altivec won't throw an exception to warn you about the unaligned address. Instead, Altivec simply ignores the lower four bits of the address and charges ahead, operating *on the wrong address.* This means your program may silently corrupt memory or return incorrect results if you don't explicitly make sure all your data is aligned.

There is an advantage to Altivec's bit-stripping ways. Because you don't need to explicitly truncate (align-down) an address, this behavior can save you an instruction or two when handing addresses to the processor.

This is not to say Altivec can't process unaligned memory. You can find detailed instructions how to do so on the *Altivec Programming Environments Manual.* It requires more work, but because memory is so slow compared to the processor, the overhead for such shenanigans is surprisingly low.

Structure Alignment

Examine the following structure:

```
typedef struct {
          char      a;
          long      b;
          char      c;
}         Struct;
```

What is the size of this structure in bytes? Many programmers will answer "6 bytes". It makes sense: one byte for `a`, four bytes for `b` and another byte for `c`. 1 + 4 + 1 equals 6. Here's how it would layout in memory:

Field Type	Field Name	Field Size	Field End
`char`	`a`	1	1
`long`	`b`	4	5
`char`	`c`	1	6
Total Size in Bytes:			6

However, if you were to ask CodeWarrior to `sizeof( Struct )`, chances are the answer you'd get back is 8, not 6. The reason dates back to the original 68000. Remember the 68000 would throw an exception upon encountering an odd address. If you were to read from or write to field `b`, you'd attempt to access an odd address. If a debugger weren't installed, the Mac OS would throw up a System Error dialog box with one button: Restart. Yikes!

So, instead of laying out your fields just the way you wrote them, the compiler **padded** the structure so that `b` and `c` would reside at even addresses:

Field Type	Field Name	Field Size	Field End
`char`	`a`	1	1
Padding		1	2
`long`	`b`	4	6
`char`	`c`	1	7
Padding		1	8
Total Size in Bytes:			8

Padding is the act of adding otherwise unused space to a structure to make fields line up in a desired way. Now, when the 68020 came out with built-in hardware support for unaligned memory access, this padding was unnecessary. However, it didn't hurt anything, and even helped a little in performance.

Nowadays, on PowerPC machines, two-byte alignment is nice, but four-byte or eight-byte is better. But, in a noble quest to maintain backwards compatibility with in-memory binary structures, CodeWarrior still pads structures the way its ancestors did. You can inform CodeWarrior it's okay to break with the past by using a `pragma` to turn on PowerPC-style alignment:

```
#pragma options align=power
typedef struct {
          char      a;
          long      b;
          char      c;
}         StructPPC;
#pragma options align=reset
```

This code creates a structure with a layout like this:

Field Type	**Field Name**	**Field Size**	**Field End**
`char`	`a`	1	1
padding		3	4
`long`	`b`	4	8
`char`	`c`	1	9
padding		3	12
Total Size in Bytes:			12

Incidentally, Project Builder / gcc doesn't require you explicitly turn on PowerPC-style alignment -- it operates that way by default.

Alignment and the Mac OS

The Mac OS was written in an era where memory was very limited. To minimize memory waste, many of the Mac OS Toolbox structures are one- or two-byte aligned, making atomic access much more difficult, if not impossible.

IBM's POWER -- the PowerPC's architectural ancestor -- required the stack always be sixteen-byte aligned. For some reason, Apple reduced this alignment requirement to eight-byte alignment, but IBM still requires it for their PowerPC-based operating systems.

Finally, beginning with 68040, the Mac OS allocates pointers and handles to sixteen-byte boundaries. Originally this was done so `BlockMoveData()` could take advantage of the 68040's new MOVE16 instruction, which efficiently copies data sixteen bytes at a time. When the PowerMacs eventually came along, they found a land where all dynamically allocated memory meshed perfectly with its alignment requirements.

Summary

If you don't understand and explicitly code for data alignment:

- Your software may hit performance-killing unaligned memory access exceptions, which invoke *very* expensive alignment exception handlers.
- Your application may attempt to atomically store to an unaligned address, causing your application to lock-up.
- Your application may attempt to access an odd address on an old Mac, resulting in a system crash.
- Your application may attempt to pass an unaligned address to Altivec, resulting in Altivec reading from and / or writing to the wrong part of memory, silently corrupting data or yielding incorrect results.

Credits

I'd like to thank Alex Rosenberg for reading over a draft of this paper and informing me that pointers and handles allocated by the Mac Toolbox have been sixteen-byte aligned since the 68040 rode into town with its MOVE16 instruction.

Thanks to Matt Slot for his public domain FastTimes library. FastTimes provides a simple interface to the complex heuristics required to discover and use the best timing service on a given Macintosh.

Kudos to Ian Ollmann for his submissions to various mailing lists and for reviewing this paper.

Finally, thanks to Duane Hayes for running my tests on his 604-equipped PowerMac 7500. I had a 601, 603e, G3 and G4 in my personal collection, and Duane filled my processor testing gap with his bevy of processor daughtercards.