5

PROCESSOR DIFFERENCES

The differences between comparable process architectures, such as desktop RISC chips, can be dramatic, and when moving code between high-end desktop processors such as the IBM G5 and lower-end processors such as the Intel xScale, the amount of work required for a successful transition is often surprising.

Computer processor designs vary radically in their storage requirements (alignment of data and ordering of bytes), data sizes and formats, and, obviously, performance. This chapter covers some of the common issues you'll encounter when moving between processor architectures.

Note that when you're migrating from a high-performance system to a lower-end one, the feature set may be portable, but your chosen algorithms and data structures may not scale down as neatly. This issue is addressed in the discussion of scalability in Chapter 14.

Alignment

Most processors prefer (or even require) that memory accesses be *aligned*. This means that when the processor accesses a chunk of data *n* bytes in length, the chunk's beginning address must be some multiple of *n*. For example, a four-byte variable should be on a four-byte boundary (address is a multiple of four); a two-byte variable should be on a two-byte boundary (address is a multiple of two); and so on.

However, processors often have different requirements for memory accesses. For example, the Intel x86 architecture allows unaligned memory accesses but imposes a significant performance penalty on unaligned operations. A misaligned access on many RISC processors will result in a processor fault, causing either a crash or, if the fault is handled by a software trap, a *very* slow unaligned access (the access is handled entirely in software). And on the ARM line of embedded processors, a misaligned access will result in incorrect data, which is probably the least ideal outcome, since it can result in incorrect behavior that is silently accepted.

NOTE *Certain ARM implementations with memory management units will implement optional alignment checking, but this feature is not ubiquitous across the entire ARM family.*

For maximum portability, alignment should be forced to the highest granularity possible. Any tricks such as pointer manipulation should be avoided, because they might incur unexpected misaligned accesses. One of the more common memory-alignment errors occurs when accessing a memory buffer via an invalid pointer cast.

A `union` is a handy mechanism that will guarantee alignment between two different types. For example, Motorola's SIMD AltiVec instruction set requires 16-byte alignment when transferring data between the floating-point and vector (SIMD) units:

```
/* Based on code from:
http://developer.apple.com/hardware/ve/alignment.html */
/* NOTE: "vector" is a keyword specific to the
Altivec enabled GCC compilers */
vector float FillVectorFloat( float f1, float f2, float f3, float f4 )
{
   /* this union guarantees that the 'scalars' array will be
   aligned the same as the */
   /* 'vector float v' */
   union
   {
      float scalars[ vec_step( vector float ) ];
      vector float v;
   } buffer;

   /* copy four floating point values into array of scalars */
```

```
   buffer.scalars[0] = f1;
   buffer.scalars[1] = f2;
   buffer.scalars[2] = f3;
   buffer.scalars[3] = f4;

/* return vec float equivalent */
   return buffer.v;
}
```

MISALIGNED ACCESSES THROUGH POINTER CASTING

SAL has a WAVE file-parsing function, `_SAL_create_sample_from_wave()`, that could have easily taken the buffer and simply cast it to the appropriate structure:

```
typedef struct
{
    char         wh_riff[ 4 ];
    sal_u32_t    wh_size;
    char         wh_wave[ 4 ];
    char         wh_fmt[ 4 ];
    sal_u32_t    wh_chunk_header_size;
} _SAL_WaveHeader;

sal_error_e
SAL_create_sample_from_wav( SAL_Device *device,
                            SAL_Sample **pp_sample,
                            const void *kp_src,
                            int src_size )
{
_SAL_WaveHeader *pwh  = ( _SAL_WaveHeader * ) kp_src;
    .
    .
    .
    /* verify that this is a legit WAV file
       NOTE: wf_chunk_header_size might be a misaligned access! */
    if ( strncmp( pwh->wh_riff, "RIFF", 4 ) ||
         strncmp( pwh->wh_wave, "WAVE", 4 ) ||
         pwh->wh_chunk_header_size != 16 )
    {
        return SALERR_INVALIDPARAM;
    }
}
```

Depending on the alignment of `kp_src`, the comparison statement using `pwh->wh_chunk_header_size` may result in a misaligned access. While this won't happen in the vast majority of cases, since most buffers are allocated on paragraph or page boundaries, if you've written a naive buffer allocation/free system that works on byte boundaries, this could be a real problem.

The marginally slower, but safer, solution is to copy the incoming data into a structure, which will be aligned correctly by the compiler:

```
sal_error_e
SAL_create_sample_from_wav( SAL_Device *device,
                            SAL_Sample **pp_sample,
                            const void *kp_src,
                            int src_size )
{
    _SAL_WaveHeader wh;
    .
    .
    .
    /* this still makes assumptions about padding, byte ordering, etc. */
    memcpy( &wh, kp_src, sizeof( wh ) );

    /* verify that this is a legit WAV file
       NOTE: wf_chunk_header_size will be aligned correctly */
if ( strncmp( wh.wh_riff, "RIFF", 4 ) ||
         strncmp( wh.wh_wave, "WAVE", 4 ) ||
         wh.wh_chunk_header_size != 16 )
    {
        return SALERR_INVALIDPARAM;
    }
}
```

However, raw copies don't handle byte ordering or padding issues, so often you need to parse the raw memory and transform it into the correct form, like so:

```
const sal_byte_t *kp_bytes = ( const sal_byte_t * ) kp_src;
    .
    .
    .
    /* read out wave header */
    memcpy( wh.wh_riff, kp_bytes, 4 );
    kp_bytes += 4;
    wh.wh_size = POSH_ReadU32FromLittle( kp_bytes );
kp_bytes += 4;
    memcpy( wh.wh_wave, kp_bytes, 4 );
    kp_bytes += 4;
    memcpy( wh.wh_fmt, kp_bytes, 4 );
    kp_bytes += 4;
    wh.wh_chunk_header_size = POSH_ReadU32FromLittle( kp_bytes );
    kp_bytes += 4;

    /* verify that this is a legit WAV file */
    if ( strncmp( wh.wh_riff, "RIFF", 4 ) ||
         strncmp( wh.wh_wave, "WAVE", 4 ) ||
         wh.wh_chunk_header_size != 16 )
    {
        return SALERR_INVALIDPARAM;
    }
```

Byte Ordering and Endianess

Multibyte data types such as integers may be represented in one of two forms: *little-endian* or *big-endian*, indicating the order bytes are represented within the data type. On a little-endian architecture such as the Intel x86, the least significant bytes are placed first (that is, at a lower address). A big-endian architecture, like the Motorola PowerPC, places the most significant bytes first.

There are also mixed-endian and bi-endian machines. For example, the PDP-11 stored 32-bit values as two big-endian shorts (most significant bytes at the lower address), but with the least significant short stored at the lower address (2-3-0-1 where 1 corresponds to the lowest address). Many modern CPUs and coprocessors (network processors, graphics processing units, and sound chips) support bi-endian operation, where they can operate in little-endian or big-endian mode. This ability helps both performance and portability. Unfortunately, rarely can an application control this feature; the operating system or device drivers usually control the endianess mode for specific hardware.

Big-Endian versus Little-Endian Values

Consider the following example:

```
union
{
   long l; /* assuming sizeof( long ) == 4 */
   unsigned char c[ 4 ];
} u;
u.l = 0x12345678;
printf( "c[ 0 ] = 0x%x\n", ( unsigned ) u.c[ 0 ] );
```

Here are the little-endian and big-endian values for this example:

Address	Little-Endian Value	Big-Endian Value
&c[0]	0x78	0x12
&c[1]	0x56	0x34
&c[2]	0x34	0x56
&c[3]	0x12	0x78

When run on a little-endian machine, you would expect the output to be as follows:

```
c[ 0 ] = 0x78
```

And on a big-endian CPU, you would expect to see this output:

```
c[ 0 ] = 0x12
```

This poses a significant problem: multibyte data cannot be shared directly between processors with different byte ordering. For example, if you were to write some multibyte data to a file and then read it back on an architecture of different endianess, the data would be garbled, like so:

```
void write_ulong( FILE *fp, unsigned long u )
{
   /* BAD!  Storing to disk in 'native' format of the current CPU */
   fwrite( &u, sizeof( u ), 1, fp );
}
unsigned long read_ulong( FILE *fp )
{
   unsigned long u;
   /* BAD!  Blithely assuming that the format on disk matches the
   processor's byte ordering! */
   fread( &u, sizeof( u ), 1, fp );
   return u;
}
```

BYTE-ORDERING EXAMPLE: POWERPC VERSUS INTEL X86

Now let's consider an example to demonstrate the effects of byte ordering. If you were to execute the following on a PowerPC:

```
write_ulong( fp, 0x12345678 );
```

and then run it again on an Intel x86 like this:

```
unsigned long ul = read_ulong( fp );
```

you would be in for a surprise: the variable `ul` will contain 0x78563412 on the Intel processor. The reason is that the bytes on disk are in "PowerPC format" (stored as 0x12,0x34,0x56,0x78), which will be backward when read and stored into `ul` (0x12 in the lowest address, 0x34 in the next, and so on). This is probably one of the most common—if not *the* most common—bugs programmers encounter when migrating between platforms.

Standardized Storage Format

A solution to the problem of different byte ordering is to store data in a standardized byte order. Software running on processors that do not match this standardized format must then manually "swizzle" the bytes to convert from the canonical format to the processor's native format. Another option is to store data in the platform's native byte order and then mark what that order is in the file's header. Several file formats, such as TIFF, specify the endianess this way.

NOTE *Some file formats, such as the TIFF graphics format, don't have a fixed endianess. Instead, a program must inspect the TIFF header to determine its byte ordering.*

Now let's assume that a standardized storage format is big-endian. You could then write the code shown at the beginning of this section as follows:

```
void write_ulong( FILE *fp, unsigned long u )
{
   unsigned char c[ 4 ];
   c[ 0 ] = ( unsigned char ) ( u >> 24 );
   c[ 1 ] = ( unsigned char ) ( u >> 16 );
   c[ 2 ] = ( unsigned char ) ( u >> 8 );
   c[ 3 ] = ( unsigned char ) u;
   fwrite( c, sizeof( c ), 1, fp );
}
unsigned long read_ulong( FILE *fp )
{
   unsigned char c[ 4 ];
   unsigned long u = 0;

   fread( c, sizeof( c ), 1, fp );

   u |= ( ( unsigned long ) c[ 0 ] ) << 24;
   u |= ( ( unsigned long ) c[ 1 ] ) << 16;
   u |= ( ( unsigned long ) c[ 2 ] ) << 8;
   u |= ( ( unsigned long ) c[ 3 ] );
   return u;
}
```

This code example makes no assumption regarding the data's organization in memory; instead, it directly extracts the relevant values by shifting and masking. The only complaint with this code is that it exacts a performance toll even when the storage format matches the processor's native format.

To optimize such situations, you can detect byte ordering and perform manual construction/reconstruction only when necessary, as follows:

```
unsigned long read_ulong( FILE *fp )
{
   unsigned char c[ 4 ];
   unsigned long u;

   fread( c, sizeof( c ), 1, fp );

   /* this function is discussed next */
   if ( is_big_endian() )
   {
      /* this is fine, but only on big-endian systems */
      /* Obviously you'd move this conditional outside */
      /* this loop for performance */
```

```
      return * ( unsigned long * ) c;
   }

   u  = ( ( unsigned long ) c[ 0 ] ) << 24;
   u |= ( ( unsigned long ) c[ 1 ] ) << 16;
   u |= ( ( unsigned long ) c[ 2 ] ) << 8;
   u |= ( ( unsigned long ) c[ 3 ] );
   return u;
}
```

Now you simply write your `is_big_endian()` function, which you can base on the initial code fragment that illustrated the problem:

```
int
is_big_endian( void )
{
   union
   {
      unsigned long l;
      unsigned char c[ 4 ];
   } u;
   u.l = 0xFF000000;
   /* big-endian architectures will have the MSB at
   the lowest address */
   if ( u.c[ 0 ] == 0xFF )
      return 1;
   return 0;
}
```

NOTE *If you control your storage format, then you can avoid byte-ordering concerns by using a text format for data storage. This is discussed in more detail in Chapter 15.*

Fixed Network Byte Ordering

The TCP/IP network protocol specifies a big-endian network byte order, which means that parameters provided to the network layer (but not the actual data being transmitted) must be in big-endian format.

For example, a 32-bit IPv4 address and 16-bit port specification, such as the ones used in the `sockaddr` structure, must be in network order. This means that this code:

```
struct sockaddr_in svr;
/* UNPORTABLE: sin_port is expected to be in network byte order! */
svr.sin_port = PORT_NO;
```

will mysteriously fail on little-endian architectures, since `PORT_NO` is in the incorrect byte order.

In order to fix this, the BSD sockets and Winsock APIs provide helper functions that convert from host to network byte ordering and back:

```
uint32_t htonl( uint32_t hostlong );   /* host to network long */
uint16_t htons( uint16_t hostshort );  /* host to network short */
uint32_t ntohl( uint32_t netlong );    /* network to host long */
uint16_t ntohs( uint16_t netshort );   /* network to host short */
```

The portable version of the port assignment statement would then be:

```
struct sockaddr_in svr;
svr.sin_port = htons( PORT_NO ); /* convert from host to network ordering */
```

Byte ordering should not be a concern for most programs unless they are storing and/or loading binary data or directly extracting bytes by reference from larger multibyte values. As long as you convert to and from a predefined byte-ordering format for storage and avoid directly extracting bytes by reference from larger multibyte values, processor endianess should not be a major issue.

Signed Integer Representation

Many programmers assume that a signed integer is represented in two's complement form, since this is the most common representation on modern computer systems; however, the ANSI C and C++ specifications do not dictate the format of a signed integer. Some processors do use one's complement or even sign-magnitude format. If your code might run on those systems, you should not make assumptions about signed integer ranges and bit formats.

For example, instead of assuming that a 16-bit signed value has a minimum value of –32768, use the preprocessor constant `SHRT_MIN` defined in `<limits.h>`. Another common case is the assumption that ~0 == –1, which is not true on a one's complement machine, where –0 == ~0.

Size of Native Types

Processors have a *natural word size*, corresponding to their internal register size, which represents the optimal size of a variable. Originally, there was an expectation that C compilers would make the `int` type correspond to this word size, allowing a programmer to use `int` any time optimal performance was desired (assuming no other constraints on the range of the variable in question). This was true for many years; however, at some point, a critical mass of programs made the assumption that `sizeof(int)==4`.

The assumption about `int` size played havoc with compiler writers who needed backward compatibility but who were targeting 64-bit platforms.

POSH EXAMPLE: BYTE-ORDERING CAPABILITIES

POSH provides a host of byte-ordering assistance functions and macros. First, it has a slew of byte-swapping functions suitable for converting little-endian to big-endian and back:

```
extern posh_u16_t  POSH_SwapU16( posh_u16_t u );
extern posh_i16_t  POSH_SwapI16( posh_i16_t u );
extern posh_u32_t  POSH_SwapU32( posh_u32_t u );
extern posh_i32_t  POSH_SwapI32( posh_i32_t u );
```

In addition, it has serialization and deserialization functions that automatically convert from the native format to a user-specified destination format:

```
extern posh_u16_t *POSH_WriteU16ToLittle( void *dst, posh_u16_t value );
extern posh_i16_t *POSH_WriteI16ToLittle( void *dst, posh_i16_t value );
extern posh_u32_t *POSH_WriteU32ToLittle( void *dst, posh_u32_t value );
extern posh_i32_t *POSH_WriteI32ToLittle( void *dst, posh_i32_t value );

extern posh_u16_t *POSH_WriteU16ToBig( void *dst, posh_u16_t value );
extern posh_i16_t *POSH_WriteI16ToBig( void *dst, posh_i16_t value );
extern posh_u32_t *POSH_WriteU32ToBig( void *dst, posh_u32_t value );
extern posh_i32_t *POSH_WriteI32ToBig( void *dst, posh_i32_t value );

extern posh_u16_t  POSH_ReadU16FromLittle( const void *src );
extern posh_i16_t  POSH_ReadI16FromLittle( const void *src );
extern posh_u32_t  POSH_ReadU32FromLittle( const void *src );
extern posh_i32_t  POSH_ReadI32FromLittle( const void *src );

extern posh_u16_t  POSH_ReadU16FromBig( const void *src );
extern posh_i16_t  POSH_ReadI16FromBig( const void *src );
extern posh_u32_t  POSH_ReadU32FromBig( const void *src );
extern posh_i32_t  POSH_ReadI32FromBig( const void *src );
```

On top of these are macros that convert a value to native format. These macros are redefined depending on the byte order of the current platform:

```
#if defined POSH_LITTLE_ENDIAN

#  define POSH_LittleU16(x) (x)
#  define POSH_LittleU32(x) (x)
#  define POSH_LittleI16(x) (x)
#  define POSH_LittleI32(x) (x)
#  if defined POSH_64BIT_INTEGER
#    define POSH_LittleU64(x) (x)
#    define POSH_LittleI64(x) (x)
#  endif /* defined POSH_64BIT_INTEGER */

#  define POSH_BigU16(x) POSH_SwapU16(x)
#  define POSH_BigU32(x) POSH_SwapU32(x)
#  define POSH_BigI16(x) POSH_SwapI16(x)
#  define POSH_BigI32(x) POSH_SwapI32(x)
```

```
#  if defined POSH_64BIT_INTEGER
#    define POSH_BigU64(x) POSH_SwapU64(x)
#    define POSH_BigI64(x) POSH_SwapI64(x)
#  endif /* defined POSH_64BIT_INTEGER */

#else

#  define POSH_BigU16(x) (x)
#  define POSH_BigU32(x) (x)
#  define POSH_BigI16(x) (x)
#  define POSH_BigI32(x) (x)

#  if defined POSH_64BIT_INTEGER
#    define POSH_BigU64(x) (x)
#    define POSH_BigI64(x) (x)
#  endif /* POSH_64BIT_INTEGER */

#  define POSH_LittleU16(x) POSH_SwapU16(x)
#  define POSH_LittleU32(x) POSH_SwapU32(x)
#  define POSH_LittleI16(x) POSH_SwapI16(x)
#  define POSH_LittleI32(x) POSH_SwapI32(x)

#  if defined POSH_64BIT_INTEGER
#    define POSH_LittleU64(x) POSH_SwapU64(x)
#    define POSH_LittleI64(x) POSH_SwapI64(x)
#  endif /* POSH_64BIT_INTEGER */

#endif
```

With these macros, an application can trivially convert to and from any byte ordering without needing to explicitly detect the current platform's endianess. The previous function to read an unsigned long value then becomes:

```
unsigned long read_ulong( FILE *fp )
{
   unsigned char c[ 4 ];
   unsigned long u;

   fread( c, sizeof( c ), 1, fp );
   return POSH_ReadU32FromBig( c );
}
or
unsigned long read_ulong( FILE *fp )
{
   unsigned long u;

   fread( u, sizeof( u ), 1, fp );
   return POSH_BigU32( u );
}
```

As a result, numerous models were introduced for 64-bit architectures, with varying emphasis on interoperability with 32-bit platforms versus ideal performance for 64-bit platforms. These models have names like LP64, ILP64, LLP64, ILP32, and LP32, which indicate the size of the core C data types, as shown in Table 5-1. L corresponds to a `long`, P corresponds to pointer size, I corresponds to `int`, and LL corresponds to a `long long`. (Other models exist as well; these are just a few of the more common ones.)

NOTE `long long` *is a type specific to a few compilers, notably GCC. Other compilers, such as Microsoft Visual C++, use an* `_int64` *type instead.*

Table 5-1: Some Programming Models

Type	LP64	ILP64	LLP64	ILP32	LP32
char	8	8	8	8	8
short	16	16	16	16	16
int	32	64	32	32	16
long	64	64	32	32	32
long long			64		
pointer	64	64	64	32	32

Most programmers are familiar with the traditional 32-bit programming model, ILP32, where integers, longs, and pointers are 32 bits in size. LP32, originally used by the Win16 C API, is an even simpler specification designed around the idiosyncrasies of the Intel 8086 family, which had 16-bit integer registers but 20-bit (8086) or 24-bit (80286) addressing. (And, even more idiosyncratic, the 8086 and 80286 processors used a segmented addressing architecture.)

Since the ILP32 model lacks 64-bit types, it is inappropriate for 64-bit CPUs, which have an address space beyond the 4 GB limit of 32-bit systems. For 64-bit CPUs, you need 64-bit pointers, which all the other models have. All that remains then is to decide what is more important:

- Maintaining the assumption that `sizeof(int)==sizeof(long)==sizeof(void *)`
- Maintaining the assumption that `sizeof(int)==`*`machine word size`*
- Maintaining the assumption that `sizeof(int)==4`

Since the first two assumptions are mutually exclusive on 64-bit architectures, confusion ensues (thus the proliferation of models).

Regrettably, the ANSI standard does not take a position on this issue, leaving it up to each compiler writer (and compiler user) to deal with this on a case-by-case basis. Sun, SGI, and Compaq/DEC use the LP64 model for their Unix variants, whereas Microsoft uses the LLP64 (or, more accurately, P64) model for 64-bit Windows support.

Microsoft was concerned primarily with a clean, easy, and safe migration to Win64. To ensure this, the Microsoft developers wanted to avoid, as much as possible, breaking assumptions in 32-bit code while still gaining 64-bit

pointers. The LLP64 model provides this by creating 64-bit integers only by using the `_int64` or `long long` types. Structures that do not contain pointers retain the exact same size between ILP32 and LLP64, an important consideration for backward-compatibility.

This puts you, the ostensibly portable programmer, in a predicament: you must decide whether to use the C native types (`short`, `int`, and `long`) or a set of sized types like those provided by C99 (`inttypes.h`), as shown in Table 5-2.

Table 5-2: C99 Sized Types

Type	Description
`int8_t`	Signed 8-bit integer
`uint8_t`	Unsigned 8-bit integer
`int16_t`	Signed 16-bit integer
`uint16_t`	Unsigned 16-bit integer
`int32_t`	Signed 32-bit integer
`uint32_t`	Unsigned 32-bit integer
`int64_t`	Signed 64-bit integer
`uint64_t`	Unsigned 64-bit integer

As a rule, if you absolutely must enforce a particular size—for example, when creating a rigidly formatted structure definition or when you require a guaranteed range—use the sized types. If you do not require a specific range, such as when you need an indexing variable that will reach only into the thousands, the C native integer type should allow the compiler to make the right choice for you, but, unfortunately, this is not always the case. Some platforms err on the side of compatibility and provide 32-bit integers when the architecture is natively 64-bit.

A program that requests a particular size variable, such as a 32-bit integer, by using C99's `uint32_t` type, may find itself suffering from very poor performance when migrating to a lower-end platform that does not support the operations on those sizes natively. For example, the 8086 processor is a 16-bit processor, so 32-bit integer operations often required a function call. Be careful to specify exact sizes only when you truly need them, such as when you have range or packing concerns.

Address Space

One of the major signposts for the advancement of computer architectures has been *address space*, or the total amount of memory a computer system can easily access.

Early computers could access only the tiniest amount of memory due to limitations with both the size of a pointer and the available hardware.

POSH EXAMPLE: SIZED TYPES

POSH supplies analogs to the C99 (`inttypes.h`) definitions, as follows:

`posh_byte_t`	Unsigned 8-bit quantity
`posh_i8_t`	Signed 8-bit integer
`posh_u8_t`	Unsigned 8-bit integer
`posh_i16_t`	Signed 16-bit integer
`posh_u16_t`	Unsigned 16-bit integer
`posh_i32_t`	Signed 32-bit integer
`posh_u32_t`	Unsigned 32-bit integer
`posh_i64_t`	Signed 64-bit integer
`posh_u64_t`	Unsigned 64-bit integer

As a general, but inaccurate, rule, a computer system may access no more memory than the size of a pointer will allow; that is, addressable bytes are 2-to-the-pointer bits in size. However, there are many exceptions to this, such as systems where pointers are larger than the actual address space. The Motorola 68000 could address only 16 MB, even though it had 32-bit pointer registers, and the Intel 8086 could address only 64 KB easily (with a single pointer access) but up to 1 MB in total using its segmented memory architecture. Today, we're seeing machines with 64-bit pointers; however, even those can access a much smaller range of memory, sometimes as low as 40 bits. Older computer systems used paged, windowed, or banked memory access to reach more memory than was addressable natively.

Programs that work with large arrays or structures need to be aware of any potential limitations as they migrate to lower-end platforms. This is often a surprising gotcha that programmers don't expect. For example, something as innocuous as this:

```
static unsigned char buffer[ 0x20000 ];
```

suddenly stops building when targeting a lower-end system with, say, 16-bit pointers.

Summary

Along with operating system differences, the most fundamental component of a platform is the choice of processor. Processors can differ radically in performance, features, and implementations issues, and this is one of the most common areas during portable software development. This chapter covers the majority of key issues related to architectural differences between processors.