SMPTE 268M (DPX) Byte Order Issues

Bob Friesenhahn <bfriesen@GraphicsMagick.org>
GraphicsMagick Group
August 22, 2006

Background

Traditionally, the DPX file format has been assumed (but not clearly stated) to always store its data within an array of contiguous 32-bit words. The DPX file header is clever in that it stores a magic number in a 32-bit word as part of the file header so that by inspecting the bytes in the header word, the reading program can tell if the 32-bit words needs to be byte swapped from a different "endian" orientation. The common endian schemes are "big endian" (SPARC, Motorola 68K, MIPS, Alpha, and PowerPC) or "little endian" (Intel x86, Itanium, VAX), although other variations do exist. In the big endian scheme, bytes are read from most significant to least significant ("left to right"), while the little endian scheme reads bytes from least significant to most significant ("right to left"). The big endian scheme is easier to understand and parse, and has been adopted by the IETF (Internet standards organization) as the standard convention for networking use. Regardless, both variations are quite popular, particularly due to the popularity of the Intel 'x86 CPU family.

Recently, some vendors have apparently introduced DPX formats which are based on an array of 8-bit or 16-bit storage elements rather than an array of 32-bit storage elements. While it is quite common to do this for other file formats (TIFF is a good example), this approach differs from the traditional interpretation of the format and in fact can be demonstrated to break the DPX format's inherent solution to the endian problem.

This short paper provides an short overview of byte-ordering on big endian CPUs and little endian CPUs, as well as taking note of particular problems encountered when varying from the 32-bit storage element.

Byte Order

The following summarizes the behavior of a four octet (byte) sequence comprised of bytes 0, 1, 2, 3 when read into different size storage elements on big endian and little endian CPUs. The byte designations are used to show where the individual byte positions are marshalled to. When multiple bytes comprise a word, the most significant byte is listed first:

Values Read	Big Endian				Little Endian			
Four 8-bit values	0	1	2	3	0	1	2	3
Two 16-bit values	0,1	2,3			1,0	3,2		
One 32-bit value	0,1,2,3				3,2,1,0			

If we take our 32-bit value (0,1,2,3 for big endian, and 3,2,1,0 for little endian) and extract the most significant 16 bits and the least significant 16 bits, on big endian we end up with two 16-bit words containing 0,1 and 2,3, while little endian produces 3,2 and 1,0. If we were to convert the two little endian 16-bit words (1,0 and 3,2) to big endian, we would end up with 2,3 and 0,1. **Please note that this has reversed the order of the 16-bit samples**! If we take the 32-bit little endian version and convert it to big endian, and then extract the two 16-bit words, we would end up with 0,1 and 2,3 (as expected). Writing as bytes and then reading as a 32-bit word, also reorders the bytes. This is why DPX's notion of storing all values within 32-bit words is not compatible with direct 8-bit and 16-bit value storage.

Demonstration Program

A small test program has been written (in the C++ language) to demonstrate behavior on big and little endian architectures. This program writes four bytes to a file (human readable ASCI values '0', '1', '2', '3') and then proceeds to read it into an array of four bytes, and array of 16-bit shorts, and one 32-bit integer. The values of the byte in each position are displayed. The output of this program on a big endian CPU is:

```
Four 8-bit values (file order):
  value[0] = 0
  value[1] = 1
  value[2] = 2
  value[3] = 3
Four bytes read as two 16-bit values (MSB to LSB):
  value[0] = 0,1 (12337)
  value[1] = 2,3 (12851)
Four bytes read as one 32-bit value (MSB to LSB):
  value = 0,1,2,3 (808530483)
  16-bit subvalues (most and least significant):
   value[0] = 0,1 (12337)
   value[1] = 2,3 (12851)
```

and on a little endian CPU the output is

```
Four 8-bit values (file order):
  value[0] = 0
  value[1] = 1
  value[2] = 2
  value[3] = 3
Four bytes read as two 16-bit values (MSB to LSB):
  value[0] = 1,0 (12592)
  value[1] = 3,2 (13106)
Four bytes read as one 32-bit value (MSB to LSB):
  value = 3,2,1,0 (858927408)
  16-bit subvalues (most and least significant):
   value[0] = 3,2 (13106)
   value[1] = 1,0 (12592)
```

The text of the program follows:

```
// Demonstration program to illustrate the implications of reading a
// sequence of four bytes on little endian and big endian CPUs.  This
// program provides different results on little endian CPUs than it
// does on big endian CPUs.  Note that the input data is carefully
// selected to produce human-readable ASCII values ('0', '1', '2',
// '3') in each byte position and when values are dumped, the value of
// each octet is dumped in order from most significant byte to the
// least significant byte as a list.  A value displayed in parenthesis
// is the numeric value of the complete field.
//
// Written by Bob Friesenhahn <bfriesen@simple.dallas.tx.us> and
// placed in the public domain.

#include <string>
#include <iostream>
#include <inttypes.h>
#include <fstream>
```

```
using namespace std;

uint8_t get_uint8_value(const uint32_t value, uint32_t shift)
{
  return ((value >> shift) & 0xFF);
}

uint16_t get_uint16_value(const uint32_t value, uint32_t shift)
{
  return ((value >> shift) & 0xFFFF);
}

void dump_32bits(ostream& stream, const uint32_t& value)
{
  stream
    << get_uint8_value(value, 24)
    << ","
    << get_uint8_value(value, 16)
    << ","
    << get_uint8_value(value, 8)
    << ","
    << get_uint8_value(value, 0);
}

void dump_16bits(ostream& stream, const uint16_t& value)
{
  stream
    << get_uint8_value(value, 8)
    << ","
    << get_uint8_value(value, 0);
}

int main(int /*argc*/,char **/*argv*/)
{
  // Create test file containing four ordered bytes
  const char *testfile = "endiantest.dat";
  ofstream outfile;
  outfile.open(testfile, ios::out | ios::trunc);
  outfile << '0' << '1' << '2' << '3' << endl;
  outfile.close();

  // Test reading four bytes as an array of 8-bit values
  ifstream infile;
  infile.open(testfile);
  char char_values[5];
  infile.read(char_values,4);
  char_values[4]=0;
  infile.close();

  cout << "Four 8-bit values (file order): " << endl;
  for (uint32_t i = 0 ; i < 4; i++)
    {
```

```
    cout << "  value[" << i << "] = " << char_values[i] << endl;
  }

// Test reading four bytes as an array of 16-bit values
uint16_t short_values[2];
infile.open(testfile);
infile.read(reinterpret_cast<char *>(&short_values),4);
infile.close();

cout << "Four bytes read as two 16-bit values (MSB to LSB):" << endl;
for (uint32_t i = 0 ; i < 2; i++)
  {
    cout << "  value[" << i << "] = ";
    dump_16bits( cout, short_values[i] );
    cout << " (" << short_values[i] << ")" << endl;
  }

// Test reading four bytes as one 32-bit value
infile.open(testfile);
uint32_t int_value;
infile.read(reinterpret_cast<char *>(&int_value),sizeof(int_value));
infile.close();

cout << "Four bytes read as one 32-bit value (MSB to LSB): " << endl;
cout << "  value = ";
dump_32bits( cout, int_value );
cout << " (" << int_value << ")" <<endl;

// Dump out 16-bit values read as part of 32-bit word
uint16_t short_values_word[2];
short_values_word[0]=get_uint16_value(int_value, 16);
short_values_word[1]=get_uint16_value(int_value, 0);
cout <<  "  16-bit subvalues (most and least significant):" << endl;
for (uint32_t i = 0 ; i < 2; i++)
  {
    cout << "   value[" << i << "] = ";
    dump_16bits( cout, short_values_word[i] );
    cout << " (" << short_values_word[i] << ")" << endl;
  }

(void) ::remove(testfile);
return 0;
}
```