Reading binary files from PCs on "other endian" machines

Written by Paul Bourke
March 1991

This document briefly describes the byte swapping required when a binary file created on a DOS/WIndows is to be read on a computer which has its bytes ordered the other way.

There are various datatypes which may be read, the simplest is characters where no byte swapping is required. The next simplest is an unsigned short integer represented by 2 bytes. If the two bytes are read sequentially then the integer value on a big endian machine is 256*byte1+byte2. If the integer was written with a little endian machine such as a DOS/WINDOWS computer then the integer is 256*byte2+byte1.

While this approach can be used for unsigned shorts, ints, and longs and can be easily modified for signed versions of the same, it is rather difficult for real numbers (floats and double precision numbers). Fortunately the standard IEEE numerical format is used almost exclusively now days so that the bytes making up the particular number can be swapped around appropriately in memory. This does assume that the size of the particular numerical type is the same length on both machines, the machine that wrote the file and the machine reading the file. The usual standards are short integers are 2 bytes, long integers are 4 bytes, floats are 4 bytes and doubles are 8 bytes.

In summary, to read 2 byte integers (signed or unsigned) one reads the 2 bytes as normal, eg: using fread(), and then swap the 2 bytes in memory. It turns out that for long integers, floats and doubles the requirements is to reverse the bytes as they appear in memory. See the source below for more details.

Source code

Some routines illustrating the methods required to do the byte swapping for various numerical types.

```
/*
   Read a short integer, swapping the bytes
*/
int ReadShortInt(FILE *fptr,short int n)
{
   unsigned char *cptr,tmp;

   if (fread(n,2,1,fptr) != 1)
      return(FALSE);
   cptr = (unsigned char *)n;
   tmp = cptr[0];
   cptr[0] = cptr[1];
   cptr[1] =tmp;

   return(TRUE);
}

/*
   Read an integer, swapping the bytes
*/
int ReadInt(FILE *fptr,int *n)
{
   unsigned char *cptr,tmp;

   if (fread(n,4,1,fptr) != 1)
      return(FALSE);
   cptr = (unsigned char *)n;
```

```
   tmp = cptr[0];
   cptr[0] = cptr[3];
   cptr[3] = tmp;
   tmp = cptr[1];
   cptr[1] = cptr[2];
   cptr[2] = tmp;

   return(TRUE);
}

/*
   Read a floating point number
   Assume IEEE format
*/
int ReadFloat(FILE *fptr,float *n)
{
   unsigned char *cptr,tmp;

   if (fread(n,4,1,fptr) != 1)
      return(FALSE);
   cptr = (unsigned char *)n;
   tmp = cptr[0];
   cptr[0] = cptr[3];
   cptr[3] =tmp;
   tmp = cptr[1];
   cptr[1] = cptr[2];
   cptr[2] = tmp;

   return(TRUE);
}

/*
   Read a double precision number
   Assume IEEE
*/
int ReadDouble(FILE *fptr,double *n)
{
   unsigned char *cptr,tmp;

   if (fread(n,8,1,fptr) != 1)
      return(FALSE);

   cptr = (unsigned char *)n;
   tmp = cptr[0];
   cptr[0] = cptr[7];
   cptr[7] = tmp;
   tmp = cptr[1];
   cptr[1] = cptr[6];
   cptr[6] = tmp;
   tmp = cptr[2];
   cptr[2] = cptr[5];
   cptr[5] =tmp;
   tmp = cptr[3];
   cptr[3] = cptr[4];
   cptr[4] = tmp;

   return(TRUE);
}
```

Macros

An alternative for all but doubles is to use these cute macros, then the swapping is done inline.

```
#define SWAP_2(x) ( (((x) & 0xff) << 8) | ((unsigned short)(x) >> 8) )
#define SWAP_4(x) ( ((x) << 24) | \
         (((x) << 8) & 0x00ff0000) | \
         (((x) >> 8) & 0x0000ff00) | \
         ((x) >> 24) )
#define FIX_SHORT(x) (*(unsigned short *)&(x) = SWAP_2(*(unsigned short *)&(x)))
#define FIX_INT(x)   (*(unsigned int *)&(x)   = SWAP_4(*(unsigned int *)&(x)))
#define FIX_FLOAT(x) FIX_INT(x)
```

Strategies for developers

There are three basic strategies for software developers when choosing how to create endian independent data files and associated software.

- Decide that the file format will be one particular endian. In this case software running on machines of the same endian does nothing special, software running on other machines byte swap everything on reading and writing. This is common for file formats and software designed with an implicit endian assumption which get ported at a future date to other machines.
- Store in the file the endian-ness of the file. The software writes the binary file in the natural endian of the underlying hardware but pays attention to the endian-ness when reading binary files. Both endian files need to be handled, the software has knowledge of its own endian-ness so it can do the right thing.
- The poorer cousin of the last approaches is not to store the endian-ness and for software to always write in its natural endian. This leads to two possible file types and the user is expected to know which endian a file is and chooses the appropriate one when specifying which file to read. This is obviously the least attractive approach.

Reading FORTRAN unformatted binary files in C/C++

Or....FORTRAN Weirdness, what were they thinking?

Written by Paul Bourke
April 2003

Problem

Ever wanted to read binary files written by a FORTRAN program with a C/C++ program? Not such an unusual or unreasonable request but FORTRAN does some strange things ..... consider the following FORTRAN code, where "a" is a 3D array of 4 byte floating point values.

```
open(60,file=filename,status='unknown',form='unformatted')
write(60) nx,ny,nz
do k = 1,nz
  do j = 1,ny
   write(60) (a(i,j,k),i=1,nx)
  enddo
enddo
close(60)
```

What you will end up with is not a file that is (4 * nx) * ny * nz + 12 bytes long as it would be for the equivalent in most (if not all) other languages! Instead it will be nz * ny * (4 * nx + 8) + 20 bytes long. Why?

Reason

Each time the FORTRAN write is issued a "record" is written, the record consists of a 4 byte header, then the data, then a trailer that matches the header. The 4 byte header and trailer consist of the number of bytes that will be written in the data section. So the following

```
write(60) nx,ny,nz
```

gets written on the disk as follows where nx,ny,nz are each 4 bytes, the other numbers below are 2 byte integers written in decimal

```
0 12 nx ny nz 0 12
```

The total length written is 20 bytes. Similarly, the line

```
write(60) (a(i,j,k),i=1,nx)
```

gets written as follows assuming nx is 1024 and "a" is real*4

```
10 0 a(1,j,k) a(2,j,k) .... a(1024,j,k) 10 0
```

The total length is 4104 bytes. Fortunately, once this is understood, it is a trivial to read the correct things in C/C++.

A consequence that is a bit shocking for many programmers is that the file created with the above code gives a file that is about 1/3 the size than one created with this code.

```
open(60,file=filename,status='unknown',form='unformatted')
write(60) nx,ny,nz
do k = 1,nz
  do j = 1,ny
    do i = 1,nx
      write(60) a(i,j,k)
    enddo
  enddo
enddo
close(60)
```

In this case each element of a is written in one record and consumes 12 bytes for a total file size of nx * ny * nz * 12 + 20.

Note

- This doesn't affect FORTRAN programs that might read these files, that is because the FORTRAN "read" commands know how to handle these unformatted files.
- The discussion here does not address the transfer of binary files between machines with a different endian. In that case after a short, int, float, double is read the bytes must be rearranged. Fortunately this is relatively straightforward with these macros.

```
#define SWAP_2(x) ( (((x) & 0xff) << 8) | ((unsigned short)(x) >> 8) )
#define SWAP_4(x) ( ((x) << 24) | (((x) << 8) & 0x00ff0000) | \
         (((x) >> 8) & 0x0000ff00) | ((x) >> 24) )
#define FIX_SHORT(x) (*(unsigned short *)&(x) = SWAP_2(*(unsigned short *)&(x)))
#define FIX_LONG(x) (*(unsigned *)&(x) = SWAP_4(*(unsigned *)&(x)))
#define FIX_FLOAT(x) FIX_LONG(x)
```

- It appears that the endianness of the 4 byte header and trailer reflect the endianness of the machine doing the writing. Of course if you know the format of the data being written then you can simply skip over the header/trailer bytes, but if you need to decode the file or do error checking then knowledge of the endian of the machine where the file was written and the endian of the machine where the file is being read is necessary.
- And lastly, the above does not address the possibility (fairly rare these days) that the files may be transferred between two machines with different internal representations of floating point numbers. If that is the case then you're really in trouble and should probably revert to transferring the data in a readable ASCII format.
- Update (Jan 2008): It would appear that on 64 bit machines the 2 header elements are each written as 4 bytes instead of 2 bytes each.
- If the file is not already in existence then writing files in FORTRAN to avoid the above, one can use the access='stream' option. This option was introduced reasonably recently explicitly to overcome this issue.