libvaxdata: VAX Data Format Conversion Routines

By Lawrence M. Baker

Report Series XXXX–XXXX

U.S. Department of the Interior
U.S. Geological Survey

U.S. Department of the Interior
Gale A. Norton, Secretary

U.S. Geological Survey

P. Patrick Leahy, Acting Director

U.S. Geological Survey, Reston, Virginia 200x
Revised and reprinted: 200x

For product and ordering information:
World Wide Web: http://www.usgs.gov/pubprod
Telephone: 1-888-ASK-USGS

For more information on the USGS—the Federal source for science about the Earth,
its natural and living resources, natural hazards, and the environment:
World Wide Web: http://www.usgs.gov
Telephone: 1-888-ASK-USGS

Suggested citation:
Baker, L.M., 2005, libvaxdata: VAX Data Format Conversion Routines: U.S. Geological Survey Open-File Report 2005-xxx (*http://pubs.usgs.gov/2005/xxx*).

libvaxdata: VAX Data Format Conversion Routines

By Lawrence M. Baker

Introduction

libvaxdata provides a collection of routines for converting numeric data — integer and floating-point — to and from the formats used on a Digital Equipment Corporation (DEC, later Compaq Computer Corporation, now Hewlett-Packard Company) VAX 32-bit minicomputer (Brunner, 1991). Since the VAX numeric data formats are inherited from those used on a DEC PDP–11 16-bit minicomputer, these routines can be used to convert PDP–11 data as well. VAX numeric data formats are also the default data formats used on DEC Alpha 64-bit minicomputers running OpenVMS (Hewlett-Packard, 2005a, 2005b).

The libvaxdata routines are callable from Fortran or C. It is assumed that the caller uses two's-complement format for integer data and IEEE 754 format (ANSI/IEEE, 1985) for floating-point data. (It would be unusual to find a system that does not use these formats. Nevertheless, you may wish to consult the Fortran or C compiler documentation on your system to be sure.)

Some Fortran compilers support conversion of VAX numeric data on-the-fly when reading or writing unformatted files, either as a compiler option or a run-time I/O option (Hewlett-Packard, 2002, 2005b). This feature may be easier to use than the libvaxdata routines. Consult the Fortran compiler documentation on your system to determine if this alternative is available to you.

Description

The routines in libvaxdata are:

`from_vax_i2()`	16-bit integer byte swap
`from_vax_i4()`	32-bit integer byte reversal
`from_vax_r4()`	32-bit VAX F_floating to IEEE S_floating
`from_vax_d8()`	64-bit VAX D_floating to IEEE T_floating
`from_vax_g8()`	64-bit VAX G_floating to IEEE T_floating
`from_vax_h16()`	128-bit VAX H_floating to Alpha X_floating

`to_vax_i2()`	16-bit integer byte swap
`to_vax_i4()`	32-bit integer byte reversal
`to_vax_r4()`	32-bit IEEE S_floating to VAX F_floating
`to_vax_d8()`	64-bit IEEE T_floating to VAX D_floating
`to_vax_g8()`	64-bit IEEE T_floating to VAX G_floating
`to_vax_h16()`	128-bit Alpha X_floating to VAX H_floating

where *x*_floating is the nomenclature used on a DEC Alpha for its floating-point formats (Sites and Witek, 1995). S_floating is the IEEE 754 32-bit Single Format. T_floating is the IEEE 754 64-bit Double Format. X_floating is an IEEE 754-conforming 128-bit Double Extended Format.[1]

All calls take 3 arguments, an input array, an output array, and a conversion count:

C

```
Declaration  #include "convert_vax_data.h"
Prototype    void name( const void *in_array, void *out_array,
                        const int *count );
Usage        #define ARRAY_LEN n
             data_type in_array[ARRAY_LEN],
                       out_array[ARRAY_LEN];
             const int count = ARRAY_LEN;
             name( in_array, out_array, &count );
```

Fortran

```
Declaration  Subroutine NAME( in_array, out_array, count )
             Integer count
             data_type in_array(count), out_array(count)
Usage        Integer ARRAY_LEN
             Parameter ( ARRAY_LEN = n )
             data_type in_array(ARRAY_LEN),
            &          out_array(ARRAY_LEN)
             Call NAME( in_array, out_array, ARRAY_LEN )
```

where `name` (`NAME`) is the name of a libvaxdata routine, *n* (`count`) is the number of array elements to be converted, and `data_type` is an appropriate data type for the input (`in_array`) and output (`out_array`) data arrays. The `in_array` and `out_array` parameters may refer to the same array, since conversion is carried out element-by-element from `in_array` to `out_array`. The `in_array` and `out_array` parameters must not otherwise overlap.

Integer Conversions

VAXes and Intel 80x86 systems (Intel, 2005) store integers in two's-complement format, ordering the bytes in memory from low-order (`l`) to high-order (`h`), called little-endian format:

```
Byte no.              3       2       1       0
                      |       |       |       |
16-bit integer                    hhhhhhhhllllllll
32-bit integer   hhhhhhhhnnnnnnnnmmmmmmmmllllllll
```

[1] The Alpha X_floating format is not necessarily compatible with another system's IEEE 754-conforming 128-bit floating-point format. In particular, it is *not* compatible with the IEEE 754-conforming 128-bit extended floating-point format implemented in software for IBM XL Fortran for AIX (International Business Machines, 2004). It *is* compatible with the IEEE 754-conforming 128-bit extended floating-point format defined for the Hewlett-Packard PA–RISC (Kane, 1995).

Apple Macintosh systems (Apple Computer, 2005) and most Unix systems (e.g., Sun [Sun Microsystems, 2005a], IBM [Silha, 2005], HP) also store integers in two's-complement format, but use the opposite (big-endian) byte ordering:

```
Byte no.                 0       1       2       3
                         |       |       |       |
16-bit integer        hhhhhhhhllllllll
32-bit integer        hhhhhhhhnnnnnnnnmmmmmmmmllllllll
```

A VAX-format integer is converted to big-endian format by reversing the byte order. No conversion is required when the caller uses little-endian byte order; the data are copied as-is (unless `in_array` and `out_array` are the same array, in which case the copy is skipped altogether).

Floating-Point Conversions

Intel 80x86 systems (Intel, 2005), Apple Macintosh systems (Apple Computer, 2004), and most Unix systems (Hewlett-Packard, 2002) implement the IEEE 754 floating-point arithmetic standard. VAX and IEEE formats are similar, after the bytes are rearranged. (VAX floating-point formats inherit the PDP–11 memory layout based on 16-bit words in little-endian byte order.)

The high-order bit is a sign bit (`s`). This is followed by a biased exponent (`e`), and a (usually) hidden-bit normalized mantissa (`m`). They differ in the number used to bias the exponent, the location of the implicit binary point for the mantissa, and the representation of exceptional numbers (e.g., ±infinity).

VAX floating-point formats: $(-1)^s$ _ $2^{(e-bias)}$ _ $0.1m$

```
Bit no.               31      23      15      7       0
                      |       |       |       |       |
F_floating            mmmmmmm_m1_mmmmmmseeeeeeeemm_m0_m     bias=128

D_floating            mmmmmmm_m1_mmmmmmseeeeeeeemm_m0_m     bias=128
                      mmmmmmm_m3_mmmmmmmmmmmmmm_m2_mmmmmmm

G_floating            mmmmmmm_m1_mmmmmmseeeeeeeeeee_m0_     bias=1024
                      mmmmmmm_m3_mmmmmmmmmmmmmm_m2_mmmmmmm

H_floating            mmmmmmm_m0_mmmmmmseeeeeeeeeeeeeee     bias=16384
                      mmmmmmm_m2_mmmmmmmmmmmmmm_m1_mmmmmmm
                      mmmmmmm_m4_mmmmmmmmmmmmmm_m3_mmmmmmm
                      mmmmmmm_m6_mmmmmmmmmmmmmm_m5_mmmmmmm
```

IEEE floating-point formats: $(-1)^s$ _ $2^{(e-bias)}$ _ $1.m$ (normalized)
$(-1)^s$ _ $2^{(1-bias)}$ _ $0.m$ (subnormal)

```
Bit no.               31      23      15      7       0
                      |       |       |       |       |
S_floating            seeeeeeeemm_m0_mmmmmmmm_m1_mmmmmm     bias=127
```

T_floating	`seeeeeeeeeee_m0_mmmmmmm_m1_mmmmm` `mmmmmm_m2_mmmmmmmmmmmmm_m3_mmmmm`	bias=1023

X_floating	`seeeeeeeeeeeeeeemmmmmmm_m0_mmmmm`
	`mmmmmm_m1_mmmmmmmmmmmmm_m2_mmmmm`
	`mmmmmm_m3_mmmmmmmmmmmmm_m4_mmmmm`
	`mmmmmm_m5_mmmmmmmmmmmmm_m6_mmmmm`
		bias=16383

VAX format to IEEE format Conversions

After rearranging the bytes, a VAX floating-point number is converted to IEEE floating-point format by subtracting *(1+VAX_bias–IEEE_bias)* from the exponent field to (1) adjust from VAX *0.1m* hidden-bit normalization to IEEE *1.m* hidden-bit normalization and (2) adjust the bias from VAX format to IEEE format. True zero (*s=e=m=0*) and dirty zero (*s=e=0, m≠0*) are special cases which must be recognized and handled separately.

Numbers whose absolute value is too small to represent in the normalized IEEE format illustrated above are converted to subnormal format (*e=0, m≠0*). Numbers whose absolute value is too small to represent in subnormal format are set to zero (silent underflow).

Overflow during the conversion is not possible; the largest floating-point number in each VAX format is smaller than the largest floating-point number in the corresponding IEEE floating-point format.

If the mantissa of the VAX floating-point number is too large for the corresponding IEEE floating-point format, bits are simply discarded from the right. Thus, the remaining fractional part is chopped, not rounded to the lowest-order bit. This can only occur when the conversion requires IEEE subnormal format.

A VAX floating-point reserved operand (*s=1, e=0, m=any*) causes a `SIGFPE` exception to be raised. The converted result is set to zero.

IEEE format to VAX format Conversions

Conversely, an IEEE floating-point number is converted to VAX floating-point format by adding *(1+VAX_bias–IEEE_bias)* to the exponent field. +zero (*s=e=m=0*), –zero (*s=1, e=m=0*), ±infinity (*s=any, e=all-1's, m=0*), and *NaN*s (*s=any, e=all-1's, m≠0*) are special cases which must be recognized and handled separately. Infinities and *NaN*s cause a `SIGFPE` exception to be raised. The result returned has the largest VAX exponent (*e=all-1's*) and zero mantissa (*m=0*) with the same sign as the original.

Numbers whose absolute value is too small to represent in the normalized VAX format illustrated above are set to zero (silent underflow). (VAX floating-point formats do not support subnormal numbers.) Numbers whose absolute value exceeds the largest representable VAX-format number cause a `SIGFPE` exception to be raised (overflow). (VAX floating-point formats do not have reserved bit patterns for infinities or *NaN*s.) The result returned has the largest VAX exponent and mantissa (*e=m=all-1's*) with the same sign as the original.

The bytes are then rearranged to the VAX 16-bit word floating-point fomat.

Examples

The following C function, `from_vax_rhdr()`, converts the floating-point data header from a data file written on a VAX:

```
/* VAX Data Conversion Routines */

#include "convert_vax_data.h"

#ifndef FORTRAN_LINKAGE
#define FORTRAN_LINKAGE
#endif

/************************************************************ from_vax_rhdr() */

void FORTRAN_LINKAGE from_vax_rhdr( const void *inbuf, void *outbuf ) {

   register const float *in;            /* Microsoft C: up to 2 register vars */
   register float *out;                 /* Microsoft C: up to 2 register vars */
   int n;
   float in_null, out_null;


   in  = (const float *) inbuf;
   out = (float *) outbuf;

   in_null = in[1];
   n = 1;
   from_vax_r4( &in_null, &out_null, &n );

   n = 38;                                                /*   1..38   binary */
   from_vax_r4( in, out, &n );
   in += n;
   out += n;

   *out = ( *in == in_null ) ? out_null : *in ;           /*    39     ASCII  */
   in++;
   out++;

   n = 89;                                                /*  40..128  binary */
   from_vax_r4( in, out, &n );

}
```

The equivalent Fortran subroutine, `FROM_VAX_RHDR`, is:

```
****************************************************************** FROM_VAX_RHDR
*
      Subroutine FROM_VAX_RHDR( inbuf, outbuf )
*
      Real inbuf[128], outbuf[128]
*
      Real in_null, out_null
*
*
      in_null = inbuf[2]
```

```
      Call FROM_VAX_R4( in_null, out_null, 1 )
*                                                             1..38   binary
      Call FROM_VAX_R4( inbuf[ 1], outbuf[ 1], 38 )
*                                                              39     ASCII
      If ( inbuf[39] .eq. in_null ) Then
         outbuf[39] = out_null
      Else
         outbuf[39] = inbuf[39]
      End If
*                                                             40..128  binary
      Call FROM_VAX_R4( inbuf[40], outbuf[40], 89 )
*
      Return
      End
```

Compilation

The C source code for the libvaxdata routines is in `convert_vax_data.c` in the `src` directory of the distribution kit. The C function prototypes are declared in `convert_vax_data.h` in the same directory.

To compile all routines into a single object module:

```
$ cc –c convert_vax_data.c
```

To compile a single routine into its own module, define `MAKE_`*`routine_name`*, substituting the upper-case name of the routine for *`routine_name`*, and give the object module a name. This is useful, for example, to insert the routines into a library such that a linker may extract only the routines actually needed by a particular program. For example, to compile only `from_vax_r4()`:

```
$ cc –c –o from_vax_r4.o –DMAKE_FROM_VAX_R4 \
        convert_vax_data.c
```

Two variants of `convert_vax_data.c` are available using `IS_LITTLE_ENDIAN` and `APPEND_UNDERSCORE`.

If `IS_LITTLE_ENDIAN` is defined as `0` (false), then the conversions are performed for a big-endian system; byte reordering is done for all VAX data types. If `IS_LITTLE_ENDIAN` is defined as `1` (true), then byte reordering is done for floating-point formats only; integer formats are identical to their VAX counterparts.

If `IS_LITTLE_ENDIAN` is not defined, then it is defined as `1` (true) if any of the following macros is defined:

`vax __vax vms __vms __alpha`	DEC VAX C, GNU C on a DEC VAX or a DEC Alpha, or DEC C
`M_I86 _M_IX86 __M_ALPHA`	Microsoft 80x86 C or Microsoft Visual C++ on an Intel 80x86 or a DEC Alpha
`i386 __i386`	Sun C, GNU C, or Intel C on an Intel 80x86

```
__x86_64      GNU C or Portland Group C on an AMD Opteron
__x86_64__        or an Intel EM64T
```

If `APPEND_UNDERSCORE` is defined, the entry point names are compiled with an underscore appended. This is required so that they can be called from Fortran in cases where the Fortran compiler appends an underscore to externally called routines (e.g., Sun Fortran [Sun Microsystems, 2005b]). For example, to create Fortran-callable versions of all the routines in an object module called `fconvert_vax_data.o` on a Sun SPARC system, the compiler command would be:

```
$ cc –c –o fconvert_vax_data.o –DIS_LITTLE_ENDIAN=0 \
          –DAPPEND_UNDERSCORE convert_vax_data.c
```

because a SPARC is a big-endian system and Sun Fortran appends an underscore to externally called routines.

`convert_vax_data.c` assumes an ANSI C compiler. Compilation will fail if a `char` is not 8 bits, a `short` is not 16 bits, or an `int` is not 32 bits. `convert_vax_data.c` does not use 64-bit arithmetic.[2]

Distribution Kit

The libvaxdata distribution kit includes make files and batch command files to create a (static) library of separately compiled modules for both Fortran and C programs. A single library is created, called `libvaxdata.x`, where `x` is the system suffix for object module libraries (e.g., `libvaxdata.a` on Unix).

To create the library:

1. Download or copy from CD the compressed distribution kit in a format suitable for your system (they are all identical). For example, use `libvaxdata.zip` on a Windows system.
2. Unpack the distribution kit. The most recent versions of Windows, Mac OS X, and Linux have built-in support to unpack the distribution kit directly from the desktop. (E.g., double-click the distribution kit to unpack it or open it, then drag-and-drop the contents from there.) Otherwise, a GUI tool may be available such as WinZip on Windows, or Stuffit Expander on a Macintosh. From a Linix command line, use `tar –xzf libvaxdata.tar.gz`. On Unix systems without a tar that can decompress an archive, use `zcat libvaxdata.tar.gz | tar –xf -`. You should see top-level directories named for each supported system type (e.g., `linux`, `macosx`, `win32`, etc.) and one named `src`, containing the C source files.
3. Open a terminal window (Command Prompt on Windows, MPW Shell on Mac OS 9) and navigate to the directory appropriate for your system. For example, Windows users should `cd` to the `libvaxdata\win32` directory. Follow the instructions in the `README` file there. The command to create the library will be something like:

[2] It may be possible to compile a version of libvaxdata for SMP parallel execution, since each conversion is independent. However, this has not been tried. To enable conversions in parallel across the outer loop over the conversion `count`, it may be necessary to assert that `in_array` and `out_array` are not aliased (i.e., do not overlap).

```
> vcmake                    Windows (Visual C++)
$ @Make                     OpenVMS (CC)
make.mrc                    Mac OS 9 (MrC)
$ make –f makefile.gcc      Unix/Linux/Mac OS X (gcc)
```

4. You can then copy the library to a system-wide directory for everyone to use, such as `/usr/local/lib` on Unix or Linux, Or, you can copy it to your own library directory, such as `~/lib` on Unix or Linux. See the `README` file for the instructions to use the library from your Fortran and C programs.

The distribution kit includes another useful routine to determine at run-time whether the system uses little-endian byte ordering:

C

Prototype	`int is_little_endian( void );`
Usage	`if ( is_little_endian() ) ...`

Fortran

Declaration	`Integer Function IS_LITTLE_ENDIAN()`
Usage	`If ( IS_LITTLE_ENDIAN() .ne. 0 ) ...`

The prototype is not defined in `convert_vax_data.h`, so it must be explicitly declared in a C program.

References Cited

ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, Institute of Electrical and Electronics Engineers, New York, NY.

Apple Computer, Inc., 2004, Inside Macintosh: PowerPC Numerics (*http://developer.apple.com/documentation/Performance/Conceptual/Mac_OSX_Numerics/Mac_OSX_Numerics.pdf*).

Apple Computer, Inc., 2005, PowerPC Runtime Architecture Guide for Mac OS X 10.4 (*http://developer.apple.com/documentation/DeveloperTools/Conceptual/PowerPCRuntime/PowerPCRuntime.pdf*).

Brunner, Richard A., Ed., 1991, VAX Architecture Reference Manual, Second Edition, Digital Press, Bedford, MA.

Hewlett-Packard Company, 2002, Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems, Order no. AA–Q66TE–TE (*http://h21007.www2.hp.com/dspp/files/unprotected/Fortran/docs /unix-um/dfum.htm*).

Hewlett-Packard Company, 2005a, HP C User's Guide for OpenVMS Systems, Order no. AA–PUNZM–TK (*http://h71000.www7.hp.com/commercial/c/docs/ug.pdf*).

Hewlett-Packard Company, 2005b, HP Fortran for OpenVMS User Manual, Order no. AA–QJRWD–TE (*http://h71000.www7.hp.com/doc/82final/6443/aa-qjrwd-te.pdf*).

Intel Corporation, 2005, IA–32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, Order no. 253665–017 (*http://download.intel.com/design/Pentium4/manuals/25366517.pdf*).

International Business Machines, Corp., 2004, XL Fortran Enterprise Edition for AIX User's Guide, Version 9.1, Order no. SC09–7898–00 (*http://www-1.ibm.com/support/docview.wss?uid=swg27005408& aid=1*).

Kane, Gerry, 1995, PA–RISC 2.0 Architecture, Prentice-Hall (*http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,2533,00.html*).

Silha, Ed, and others, 2005, PowerPC User Instruction Set Architecture, Book I, Version 2.02, International Business Machines Corp. (*ftp://www6.software.ibm.com/software/developer/library/es-ppcbook1.zip*).

Sites, Richard L., and Witek, Richard T., 1995, Alpha AXP Architecture Reference Manual, Second Edition, Digital Press (an imprint of Butterworth-Heinemann), Newton, MA.

Sun Microsystems, Inc., 2005a, Sun Studio 10, C User's Guide, Order no. 819–0494–10 (*http://docs-pdf.sun.com/819-0494/819-0494.pdf*).

Sun Microsystems, Inc., 2005b, Sun Studio 10, Fortran Programming Guide, Order no. 819–0491–10 (*http://docs-pdf.sun.com/819-0491/819-0491.pdf*).