Record Layout for a SAS® Version 8 or 9 Data Set in SAS®Transport Format

Introduction

All Version 8-style transport data set records are 80 bytes in length. If there is not sufficient data to reach 80 bytes, then a record is padded with ASCII blanks to 80 bytes. All character data are stored in ASCII, regardless of the operating system. All integers are stored using IBM-style integer format, and all floating point numbers are stored using the IBM-style double (truncated if the variable's length < 8). (An exception to this is later noted.) Refer to the Numeric Data Fields section for information about constructing IBM-style doubles.

Record Layout

1. The first header record consists of the following character string, in ASCII:

```
HEADER RECORD*******LIBV8 HEADER RECORD!!!!!!!0000000000000000000000
000000000
```

2. The first real header record uses the following layout:

```
aaaaaaaabbbbbbbbccccccccddddddddeeeeeeee                ffffffffffffffff
```

where aaaaaaaa and bbbbbbbb are each 'SAS ' and cccccccc is 'SASLIB ', dddddddd is the version of the SAS system that created the file, and eeeeeeee is the operating system creating it. ffffffffffffffff is the datetime created, formatted as ddMMMyy:hh:mm:ss. Note that only a 2-digit year appears. If any program needs to read in this 2-digit year, be prepared to deal with dates in the 1900s or the 2000s.

Another way to consider this record is as a C structure:

```
struct REAL_HEADER {
        char sas_symbol[2][8];
        char saslib[8];
        char sasver[8];
        char sas_os[8];
        char blanks[24];
        char sas_create[16];
        };
```

3. Second real header record:

```
ddMMMyy:hh:mm:ss
```

where the string is the datetime modified. Most often, the datetime created and datetime modified will always be the same. Pad with ASCII blanks to 80 bytes.

Note that only a 2-digit year appears. If any program needs to read in this 2-digit year, be prepared to deal with dates in the 1900s or the 2000s.

4. Member header records:

Both of these occur for every member in the transport file.

```
HEADER RECORD*******MEMBV8 HEADER
RECORD!!!!!!!0000000000000000001600000000140
HEADER RECORD*******DSCPTV8 HEADER
RECORD!!!!!!!00000000000000000000000000000
```

Note the 0140 that appears in the member header record above. That value is the size of the variable descriptor (NAMESTR) record that is described later in this document.

5. Member header data:

```
aaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbcccccccddddddddeeeeeeeeffffffff
ffffffffff
```

where aaaaaaaa is 'SAS ', bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb is the data set name, cccccccc is SASDATA (if a SAS data set is being created), dddddddd is the version of the SAS System under which the file was created, and eeeeeeee is the operating system name. ffffffffffffffff is the datetime created, formatted as in previous headers. Consider this C structure:

```
struct REAL_HEADER {
        char sas_symbol[8];
        char sas_dsname[32];
        char sasdata[8];
        char sasver[8];
        char sas_osname[8];
        char sas_create[16];
        };
```

The second header record is

```
ddMMMyy:hh:mm:ss          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbb
```

where the datetime modified appears using DATETIME16. format, followed by blanks up to column 33, where the a's above correspond to a blank-padded data set label, and bbbbbbbb is the blank-padded data set type. Note that data set labels can be up to 256 characters as of Version 8 of the SAS System, but only up to the first 40 characters are stored in the second header record. Note also that only a 2-digit year appears in the datetime modified value. If any program needs to read in this 2-digit year, be prepared to deal with dates in the 1900s or the 2000s.

Consider the following C structure:

```
struct SECOND_HEADER {
        char dtmod_day[2];
        char dtmod_month[3];
        char dtmod_year[2];
        char dtmod_colon1[1];
        char dtmod_hour[2];
        char dtmod_colon2[1];
        char dtmod_minute[2];
        char dtmod_colon2[1];
        char dtmod_second[2];
        char padding[16];
        char dslabel[40];
        char dstype[8];
        };
```

6. Namestr header record:

One for each member.

```
HEADER RECORD*******NAMSTV8 HEADER     RECORD!!!!!!!000000xxxxxx
    000000000000000000
```

7. Namestr records:

Each namestr field is 140 bytes long, but the fields are streamed together and broken in 80-byte pieces. If the last byte of the last namestr field does not fall in the last byte of the 80-byte record, the record is padded with ASCII blanks ('20'x) to 80 bytes.

Here is the C structure definition for the namestr record:

```
struct NAMESTR {
short ntype;       /* VARIABLE TYPE: 1=NUMERIC, 2=CHAR */
short nhfun;       /* HASH OF NNAME (always 0) */
short nlng;        /* LENGTH OF VARIABLE IN OBSERVATION */
short nvar0;       /* VARNUM */
char8 nname;       /* NAME OF VARIABLE */
char40 nlabel;     /* LABEL OF VARIABLE */
char8 nform;       /* NAME OF FORMAT */
short nfl;         /* FORMAT FIELD LENGTH OR 0 */
short nfd;         /* FORMAT NUMBER OF DECIMALS */
short nfj;         /* 0=LEFT JUSTIFICATION, 1=RIGHT JUST */
char nfill[2];     /* (UNUSED, FOR ALIGNMENT AND FUTURE) */
char8 niform;          /* NAME OF INPUT FORMAT */
short nifl;            /* INFORMAT LENGTH ATTRIBUTE */
short nifd;            /* INFORMAT NUMBER OF DECIMALS */
long npos;         /* POSITION OF VALUE IN OBSERVATION */
char longname[32]; /* long name for Version 8-style */
short lablen;          /* length of label */
char rest[18];     /* remaining fields are irrelevant */
};
```

The variable name truncated to 8 characters goes into nname, and the complete name goes into longname. Use blank padding in either case if necessary. The variable label truncated to 40 characters goes into nlabel, and the total length of the label goes into lablen. If your label exceeds 40 characters, you will have the opportunity to write the complete label in the label section described below.

Note that the length given in the last 4 bytes of the member header record indicates the actual number of bytes for the NAMESTR structure. The size of the structure listed above is 140 bytes.

8.

If you have any labels that exceed 40 characters, they can be placed in this section. The label records section starts with this header:

```
HEADER RECORD*******LABELV8 HEADER RECORD!!!!!!!nnnnn
```

where nnnnn is the number of variables for which long labels will be defined.

Each label is defined using the following:

```
aabbccd.....e.....
```

where

aa=variable number

bb=length of name

cc=length of label

d....=name in bb bytes

e....=label in cc bytes

For example, variable number 1 named x with the 43-byte label 'a very long label for x is given right here' would be provided as a stream of 6 bytes in hex '00010001002B'X followed by the ASCII characters.

```
xa very long label for x is given right here
```

These are streamed together. The last label descriptor is followed by ASCII blanks ('20'X) to an 80-byte boundary.

If you have any format or informat names that exceed 8 characters, regardless of the label length, a different form of label record header is used:

```
HEADER RECORD*******LABELV9 HEADER RECORD!!!!!!!nnnnn
```

where nnnnn is the number of variables for which long format names and any labels will be defined.

Each label is defined using the following:

```
aabbccddeef.....g.....h.....i.....
```

where

aa=variable number

bb=length of name in bytes

cc=length of label in bytes

dd=length of format description in bytes

ee=length of informat description in bytes

f.....=text for variable name

g.....=text for variable label

h.....=text for format description

i.....=text of informat description

Note: The FORMAT and INFORMAT descriptions are in the form used in a FORMAT or INFORMAT statement. For example, my_long_fmt., my_long_fmt8., my_long_fmt8.2. The text values are streamed together and no characters appear for attributes with a length of 0 bytes.

For example, variable number 1 is named X and has a label of 'ABC,' no attached format, and an 11-character informat named my_long_fmt with informat length=8 and informat decimal=0. The data would be

```
(hex)        (characters)
010103000d      XABCmy_long_fmt8.
```

The last label descriptor is followed by ASCII blanks ('20'X) to an 80-byte boundary.

9. Observation header:

```
HEADER RECORD*******OBSV8 HEADER RECORD!!!!!!!000000000000000000000000
000000
```

10. Data records:

Data records are streamed in the same way that namestrs are. There is ASCII blank padding at the end of the last record if necessary. There is no special trailing record.

Missing Values

Missing values are written out with the first byte (the exponent) indicating the proper missing values. All subsequent bytes are 0x00. The first byte is:

Type	Byte
._	0x5f
.	0x2e
.A	0x41
.B	0x42
	....
.Z	0x5a

A Sample Session to Show a Transport Data Set

Here is a sample SAS session that creates a SAS data set with a long variable name. The file written to the MYTEST fileref is shown. The file is read back in and compared to the original SAS data set, and PROC COMPARE shows that they are identical.

```
filename mytest temp;
data temp;
        x='01jan2012'd;
        abcdefghi='xyz';
        format x date9.;
        run;

%loc2xpt(libref=work,memlist=temp,filespec=mytest,format=auto);

data _null_; infile mytest recfm=f lrecl=80;
        input; list;
        run;

proc datasets lib=work;
        change temp=orig;
        quit;

%xpt2loc(libref=work,filespec=mytest);
```

```
proc compare data=orig compare=temp; run;
```

The contents of the file as shown by the LIST statement:

```
RULE: ----+----1----+----2----+----3----+----4----+----
5----+----6----7----+----8----+----9----+----0
1 HEADER RECORD*******LIBV8 HEADER RECORD!!!!!!!0000000000000000
0000000000000
2 CHAR SAS SAS SASLIB 9.1 LINUX... 19OCT12:11:58:27
     ZONE 5452222254522222545444223232222244455000222222222222222222
222222344533333333333
     NUMR 3130000031300000313C92009E100000C9E58000000000000000000000000
00000019F3412A11A58A27
3 19OCT12:11:58:27
4 HEADER RECORD*******MEMBV8 HEADER RECORD!!!!!!!0000000000000000
1600000000140
5 HEADER RECORD*******DSCPTV8 HEADER RECORD!!!!!!!0000000000000000
00000000000000
6 CHAR SAS TEMP SASDATA 9.1 LINUX...19OCT12:11:58:27
     ZONE 5452222254452222222222222222222222222222254544542323222224444
550003344533333333333
     NUMR 3130000045D0000000000000000000000000000313414109E100000C9E
5800019F3412A11A58A27
7 19OCT12:11:58:27
8 HEADER RECORD*******NAMSTV8 HEADER RECORD!!!!!!!0000000002000000
00000000000000
9 CHAR ........x DATE ........
     ZONE 0000000072222222222222222222222222222222222222222222222222445
422220000000022222222
     NUMR 0100080180000000000000000000000000000000000000000000000000414
500009000000000000000
10 CHAR ........x ............................abcdefgh
     ZONE 0000000072222222222222222222222222222220000000000000000000000
000000006666666622222
     NUMR 0000000080000000000000000000000000000001040100000000000000000
002000302123456780000
11 CHAR ........ ........abcdefghi
     ZONE 2222222222222222222222222222222222222222222000000002222222
200000006666666622222
     NUMR 0000000000000000000000000000000000000000000000000000000000000000
000000008123456789000
12 CHAR ....................
     ZONE 2222222222222222222000000000000000000022222222222222222222
222222222222222222222
     NUMR 0000000000000000000010101000000000000000000000000000000000000000
000000000000000000000
13 HEADER RECORD*******OBSV8 HEADER RECORD!!!!!!! 1
14 CHAR DJ1.....xyz
     ZONE 4430000077722222222222222222222222222222222222222222222222222
222222222222222222222
     NUMR 4A10000089A0000000000000000000000000000000000000000000000000000
000000000000000000000
```

Another Sample Session with Long Format Name and Long Label

Here is a second example, using the same fileref, but with a long format name and a long label. We can see that the LABELV9 label type is used.

```
proc format;
        value longfmtname 1='yes';
        run;

data temp2;
        longvarname=1;
        format longvarname longfmtname.;
        label longvarname=
            'this is a label that is over 40 characters long';
        run;

%loc2xpt(libref=work,memlist=temp2,filespec=mytest,format=auto);
        data _null_; infile mytest recfm=f lrecl=80;
        input; list;
        run;
```

The contents of the file as shown by the LIST statement:

```
RULE: ----+----1----+----2----+----3----+----4----+----5
----+----6----+---7----+----8----+----9----+----0
1 HEADER RECORD*******LIBV8 HEADER RECORD!!!!!!!000000000000000000
000000000000
2 CHAR SAS SAS SASLIB 9.1 LINUX... 19OCT12:12:07:16
     ZONE 5452222254522222545444223232222244455000222222222222222222
22222344533333333333
     NUMR 3130000031300000313C92009E100000C9E58000000000000000000000000
0000019F3412A12A07A16
3 19OCT12:12:07:16
4 HEADER RECORD*******MEMBV8 HEADER RECORD!!!!!!!000000000000000001
600000000140
5 HEADER RECORD*******DSCPTV8 HEADER RECORD!!!!!!!000000000000000000
0000000000000
6 CHAR SAS TEMP2 SASDATA 9.1 LINUX...19OCT12:12:07:16
     ZONE 5452222254453222222222222222222222222222254544542323222224445
50003344533333333333
     NUMR 3130000045D020000000000000000000000000313414109E100000C9E5
800019F3412A12A07A16
7 19OCT12:12:07:16
8 HEADER RECORD*******NAMSTV8 HEADER RECORD!!!!!!!000000000100000000
000000000000
9 CHAR ........longvarnthis is a label that is over 40
characteLONGFMTN........
     ZONE 0000000066667676766726726266666276672672676723326667667644444
4540000000022222222
     NUMR 01000801CFE7612E4893093010C125C048140930F652040038121345CFE76
D4E0000000000000000
10 CHAR ........longvarname ./..................
     ZONE 0000000066667676666222222222222222222220200000000000000000002
2222222222222222222
     NUMR 00000000CFE7612E1D500000000000000000000000F0B01000000000000000
0000000000000000000
11 HEADER RECORD*******LABELV9 HEADER RECORD!!!!!!!1
12 CHAR ...../....longvarnamethis is a label that is over 40
characters long LONGFMTNAME.
      ZONE 0000020000666676766667667267262666662766726726767233266676676
77266664444444544442
      NUMR 010B0F0C01CFE7612E1D54893093010C125C048140930F65204003812134
5230CFE7CFE76D4E1D5E
```

```
13
14 HEADER RECORD*******OBSV8 HEADER RECORD!!!!!!! 1
15 CHAR A.......
     ZONE 41000000222222222222222222222222222222222222222222222222222222222
222222222222222222
     NUMR 10000000000000000000000000000000000000000000000000000000000000000
000000000000000000
```

Differences between Version 6 and Version 8 File Types

The header records for Version 8-style transport files are all different from their Version 6 counterparts in order to ensure that the files are rejected by the XPORT engine and recognized differently by the%XPT2LOC macro.

The Version 6 style headers are:

```
HEADER RECORD*******LIBRARY HEADER
RECORD!!!!!!!000000000000000000000000000000
HEADER RECORD*******MEMBER HEADER
RECORD!!!!!!!000000000000000001600000000140
HEADER RECORD*******DSCRPTR HEADER
RECORD!!!!!!!000000000000000000000000000000
HEADER RECORD*******NAMESTR HEADER
RECORD!!!!!!!000000000200000000000000000000
HEADER RECORD*******OBS HEADER
RECORD!!!!!!!000000000000000000000000000000

The corresponding Version 8 style headers are:
HEADER RECORD*******LIBV8 HEADER
RECORD!!!!!!!000000000000000000000000000000
HEADER RECORD*******MEMBV8 HEADER
RECORD!!!!!!!000000000000000001600000000140
HEADER RECORD*******DSCPTV8 HEADER
RECORD!!!!!!!000000000000000000000000000000
HEADER RECORD*******NAMSTV8 HEADER
RECORD!!!!!!!000000000200000000000000000000
HEADER RECORD*******OBSV8 HEADER
RECORD!!!!!!!000000000000000000000000000000
```

There are also these new headers:

```
HEADER RECORD*******LABELV8 HEADER RECORD!!!!!!!
HEADER RECORD*******LABELV9 HEADER RECORD!!!!!!!
```

The data following the DSCPTV8 record allows for a 32-character member name. In the Version 6-style format, the member name was only 8 characters.

The name descriptors use 34 previously unused bytes for the 32-character variable name and for the length of the label.

There is a new section for label descriptors, used for labels that exceed 40 characters.

The data portion can have character values that exceed 200 characters.

Numeric Data Fields

All numeric data fields in the transport file are stored as floating point numbers.

All floating point numbers in the file are stored using the IBM mainframe representation. If your application is to read from or write to transport files, it is necessary to convert native floating point numbers to or from the transport representation.

Most platforms use the IEEE representation for floating point numbers. Some of these platforms store the floating point numbers in reversed byte order from other platforms. For the sake of nomenclature, we call these platforms "big endian" and "little endian" platforms.

A big endian environment stores integers with the lowest-significant byte at a higher address in memory. Likewise, an IEEE platform is big endian if the first byte of the exponent is stored in a lower address than the first byte of the mantissa. For example, the HP series machines store a floating point 1 as 3F F0 00 00 00 00 00 00 (the bytes in hexadecimal), while an IBM PC stores a 1 as 00 00 00 00 00 00 F0 3F. The bytes are the same, just reversed. Therefore, the HP is considered big endian and the PC is considered little endian.

This is a partial list of the categories of machines on which the SAS System runs:

Hardware	Operating Systems	Float Type	Endian
IBM mainframe	MVS,CMS,VSE	IBM	Big
DEC Alpha	AXP/VMS,DEC UNIX	IEEE	Little
HP	HP-UX	IEEE	Big
Sun	Solaris I, II	IEEE	Big
RS / 6000	AIX	IEEE	Big
IBM PC	Windows,OS/2,IABI	IEEE	Little

Not included is VAX, which uses a different floating-point representation than either IBM mainframe or IEEE.

Provided Subroutines

In order to assist you in reading and/or writing transport files, we are providing routines to convert from IEEE representation (either big endian or little endian) to transport representation and back again. The source code for these routines is provided at the end of this document. Note that the source code is provided as is, and as a convenience to those needing to read and/or write transport files. The source code has been tested on HP-UX, DEC UNIX, IBM PC, and MVS.

The routine to use is cnxptiee. This converts in either direction, either to or from transport. Its usage is as follows:

```
rc = cnxptiee(from,fromtype,to,totype);
```

In this routine:

from
: is a pointer to a floating-point value.

fromtype
: is the type of floating-point value (see below).

to
: is a pointer to the target area.

totype
: is the type of target value (see below).

Floating point types:

0
: is a native floating point.

1
: is an IBM mainframe (transport representation) floating point.

2
: is a big endian IEEE floating point.

3
: is a little endian IEEE floating point.

Return codes:

```
rc = cnxptiee(from,0,to,1); native -> transport
rc = cnxptiee(from,0,to,2); native -> Big endian IEEE
rc = cnxptiee(from,0,to,3); native -> Little endian IEEE
rc = cnxptiee(from,1,to,0); transport -> native
rc = cnxptiee(from,1,to,2); transport -> Big endian IEEE
rc = cnxptiee(from,1,to,3); transport -> Little endian IEEE
rc = cnxptiee(from,2,to,0); Big endian IEEE -> native
rc = cnxptiee(from,2,to,1); Big endian IEEE -> transport
rc = cnxptiee(from,2,to,3); Big endian IEEE -> Little endian IEEE
rc = cnxptiee(from,3,to,0); Little endian IEEE -> native
rc = cnxptiee(from,3,to,1); Little endian IEEE -> transport
rc = cnxptiee(from,3,to,2); Little endian IEEE -> Big endian IEEE
```

The "native" representation is whatever is appropriate for the host machine. Most likely you will use that mode.

The testieee.c routine is supplied here to demonstrate how the cnxptiee is used. It is also useful to ensure that the cnxptiee routine works in your environment.

Note that there are several symbols that can be defined when compiling the ieee.c file. These symbols are FLOATREP, BIG_ENDIAN, and LITTLE_ENDIAN

FLOATREP should be set to one of the following strings:

```
CN_TYPE_IEEEB Big endian IEEE

CN_TYPE_IEEEL Little endian IEEE

CN_TYPE_XPORT Transport format (i.e., IBM)
```

If BIG_ENDIAN is defined, it is assumed that the platform is big endian. If LITTLE_ENDIAN is defined, it is assumed that the platform is little endian.

Do not define both of them.

If FLOATREP is not defined, the proper value is determined at run time. Although this works, it incurs additional overhead that can increase CPU time with large files. Use the FLOATREP symbol to improve efficiency. Likewise, if neither BIG_ENDIAN nor LITTLE_ENDIAN is defined, the proper orientation is determined at run time. It is much more efficient to supply the proper definition at compile time.

For example, consider this command on HP-UX:

```
cc testieee.c ieee.c -DFLOATREP=CN_TYPE_IEEEB -DBIG_ENDIAN
```

And the corresponding command on DEC UNIX:

```
cc testieee.c ieee.c -DFLOATREP=CN_TYPE_IEEEL -DLITTLE_ENDIAN
```

Here is the correct output from the testieee run:

- Native -> Big endian IEEE match count = 4 (should be 4).
- Native -> Little endian IEEE match count = 4 (should be 4).
- Native -> Transport match count = 4 (should be 4).
- Transport -> Big endian IEEE match count = 4 (should be 4).
- Transport -> Little endian IEEE match count = 4 (should be 4).
- Transport -> Native match count = 4 (should be 4).
- Big endian IEEE -> Little endian IEEE match count = 4 (should be 4).
- Big endian IEEE -> Transport match count = 4 (should be 4).
- Big endian IEEE -> Native match count = 4 (should be 4).
- Little endian IEEE -> Big endian IEEE match count = 4 (should be 4).
- Little endian IEEE -> Transport match count = 4 (should be 4).
- Little endian IEEE -> Native match count = 4 (should be 4).

Here is the source code for the test program, testieee.c.

```
#define CN_TYPE_NATIVE 0

#define CN_TYPE_XPORT 1
#define CN_TYPE_IEEEB 2
#define CN_TYPE_IEEEL 3

void tohex();
#define N_TESTVALS 4

static char xpt_testvals[N_TESTVALS][8] = {
{0x41,0x10,0x00,0x00,0x00,0x00,0x00,0x00},   1
{0xc1,0x10,0x00,0x00,0x00,0x00,0x00,0x00},   -1
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},   0
{0x41,0x20,0x00,0x00,0x00,0x00,0x00,0x00}   2
};

static char ieeeb_testvals[N_TESTVALS][8] = {
{0x3f,0xf0,0x00,0x00,0x00,0x00,0x00,0x00},   1
{0xbf,0xf0,0x00,0x00,0x00,0x00,0x00,0x00},   -1
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},   0
{0x40,0x00,0x00,0x00,0x00,0x00,0x00,0x00}   2
};

static char ieeel_testvals[N_TESTVALS][8] = {
{0x00,0x00,0x00,0x00,0x00,0x00,0xf0,0x3f},   1
{0x00,0x00,0x00,0x00,0x00,0x00,0xf0,0xbf},   -1
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},   0
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40}   2
};

static double native[N_TESTVALS] =
{1,-1,0,2};
#define N_MISSINGVALS 3
static char missingvals[N_MISSINGVALS][8] = {
{0x2e,0x00,0x00,0x00,0x00,0x00,0x00,0x00},   std missing
{0x41,0x00,0x00,0x00,0x00,0x00,0x00,0x00},   .A
```

```
{0x5A,0x00,0x00,0x00,0x00,0x00,0x00,0x00}   .Z
};
  rc = cnxptiee(from,0,to,1); native -> transport
  rc = cnxptiee(from,0,to,2); native -> Big endian IEEE
  rc = cnxptiee(from,0,to,3); native -> Little endian IEEE
  rc = cnxptiee(from,1,to,0); transport -> native
  rc = cnxptiee(from,1,to,2); transport -> Big endian IEEE
  rc = cnxptiee(from,1,to,3); transport -> Little endian IEEE
  rc = cnxptiee(from,2,to,0); Big endian IEEE -> native
  rc = cnxptiee(from,2,to,1); Big endian IEEE -> transport
  rc = cnxptiee(from,2,to,3); Big endian IEEE -> Little endian IEEE
  rc = cnxptiee(from,3,to,0); Little endian IEEE -> native
  rc = cnxptiee(from,3,to,1); Little endian IEEE -> transport
  rc = cnxptiee(from,3,to,2); Little endian IEEE -> Big endian IEEE
main()
{
char to[8];
int i,matched;
char hexdigits[17];

for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(>amp;native[i],CN_TYPE_NATIVE,to,CN_TYPE_IEEEB);
    matched += (memcmp(to,ieeeb_testvals[i],8) == 0);
}
printf("Native -> Big endian IEEE match count = %d (should be %d).\n",
    matched,N_TESTVALS);

for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(&native[i],CN_TYPE_NATIVE,to,CN_TYPE_IEEEL);
    matched += (memcmp(to,ieeel_testvals[i],8) == 0);
}
printf("Native -> Little endian IEEE match count =
%d (should be %d).\n",matched,N_TESTVALS);

for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(xpt_testvals[i],CN_TYPE_XPORT,to,CN_TYPE_IEEEB);
    matched += (memcmp(to,ieeeb_testvals[i],8) == 0);
}
printf("Transport -> Big endian IEEE match count =
%d (should be %d).\n",matched,N_TESTVALS);

for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(xpt_testvals[i],CN_TYPE_XPORT,to,CN_TYPE_IEEEL);
    matched += (memcmp(to,ieeel_testvals[i],8) == 0);
}
printf("Transport -> Little endian IEEE match count = %d \
(should be %d).\n",
    matched,N_TESTVALS);

for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(xpt_testvals[i],CN_TYPE_XPORT,to,CN_TYPE_NATIVE);
    matched += (memcmp(to,ieeel_testvals[i],8) == 0);
}
printf("Big endian IEEE -> Little endian IEEE match count = %d \
(should be %d).\n",
    matched,N_TESTVALS);
```

```
for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(ieeeb_testvals[i],CN_TYPE_IEEEB,to,CN_TYPE_IEEEL);
    matched += (memcmp(to,ieeel_testvals[i],8) == 0);
}
printf("Big endian IEEE -> Little endian IEEE match count = %d \
(should be %d).\n",
    matched,N_TESTVALS);

for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(ieeeb_testvals[i],CN_TYPE_IEEEB,to,CN_TYPE_XPORT);
    matched += (memcmp(to,xpt_testvals[i],8) == 0);
}
printf("Big endian IEEE -> Transport match count = %d \
(should be %d).\n",
    matched,N_TESTVALS);

for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(ieeeb_testvals[i],CN_TYPE_IEEEB,to,CN_TYPE_NATIVE);
    matched += (memcmp(to,&native[i],8) == 0);
}
printf("Big endian IEEE -> Native match count = %d \
(should be %d).\n",
    matched,N_TESTVALS);

for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(ieeeb_testvals[i],CN_TYPE_IEEEL,to,CN_TYPE_IEEB);
    matched += (memcmp(to,ieeeb_testvals[i],8) == 0);
}
printf("Little endian IEEE -> Big endian IEEE match count = %d \
(should be %d).\n",
    matched,N_TESTVALS);

for (i=matched=0;i<N_TESTVALS;i++){
    cnxptiee(ieeel_testvals[i],CN_TYPE_IEEEL,to,CN_TYPE_XPORT);
    matched += (memcmp(to,xpt_testvals[i],8) == 0);
printf("Little endian IEEE -> Transport match count = %d (should be
%d).\n",
        matched,N_TESTVALS);

for (i=matched=0;i<N_TESTVALS;i++){
cnxptiee(ieeel_testvals[i],CN_TYPE_IEEEL,to,CN_TYPE_NATIVE);
matched += (memcmp(to,&native[i],8) == 0);
}
printf("Little endian IEEE -> Native match count =
%d (should be %d).\n",matched,N_TESTVALS);
}

void tohex(bytes,hexchars,length)
unsigned char *bytes;
char *hexchars;
int length;
{
static char *hexdigits = "0123456789ABCDEF";
int i;
for (i=;i<length;i++)  {
    *hexchars++ = hexdigits[*bytes >> 4];
    *hexchars++ = hexdigits[*bytes++ & 0x0f];
```

```
}
*hexchars = 0;
}


CN_TYPE_IEEEB Big endian IEEE
CN_TYPE_IEEEL Little endian IEEE
CN_TYPE_XPORT Transport format (i.e., IBM)

ieee.c

#define CN_TYPE_NATIVE 0
#define CN_TYPE_XPORT 1
#define CN_TYPE_IEEEB 2
#define CN_TYPE_IEEEL 3

int cnxptiee();
void xpt2ieee();
void ieee2xpt();

#ifndef FLOATREP
#define FLOATREP get_native()
int get_native();
#endif

rc = cnxptiee(from,fromtype,to,totype);
```

In this routine:

from
: is a pointer to a floating-point value.

fromtype
: is the type of floating-point value (see below).

to
: is a pointer to target area.

totype
: is the type of target value (see below).

Floating point types:

0
: is a native floating point.

1
: is an IBM mainframe (transport representation) floating point.

2
: is a big endian IEEE floating point.

3
: is a little endian IEEE floating point.

```
rc = cnxptiee(from,0,to,1); native -> transport
rc = cnxptiee(from,0,to,2); native -> Big endian IEEE
rc = cnxptiee(from,0,to,3); native -> Little endian IEEE
rc = cnxptiee(from,1,to,0); transport -> native
rc = cnxptiee(from,1,to,2); transport -> Big endian IEEE
rc = cnxptiee(from,1,to,3); transport -> Little endian IEEE
rc = cnxptiee(from,2,to,0); Big endian IEEE -> native
rc = cnxptiee(from,2,to,1); Big endian IEEE -> transport
rc = cnxptiee(from,2,to,3); Big endian IEEE -> Little endian IEEE
```

```
  rc = cnxptiee(from,3,to,0); Little endian IEEE -> native
  rc = cnxptiee(from,3,to,1); Little endian IEEE -> transport
  rc = cnxptiee(from,3,to,2); Little endian IEEE -> Big endian IEEE

int cnxptiee(from,fromtype,to,totype)
char *from;
int fromtype;
char *to;
int totype;
{
char temp[8];
int i;

if (fromtype == CN_TYPE_NATIVE) {
    fromtype = FLOATREP;
}
switch(fromtype) {
    case CN_TYPE_IEEEL :
        if (totype == CN_TYPE_IEEEL)
            break;
        for (i=7;i>=0;i--) {
        temp[7-i] = from[i];
}
        from = temp;
        fromtype = CN_TYPE_IEEEB;


Break intentionally omitted.
case CN_TYPE_IEEEB :
Break intentionally omitted.

case CN_TYPE_XPORT :
break;
default:
return(-1);
}
if (totype == CN_TYPE_NATIVE) {
    totype = FLOATREP;
}
switch(totype) {
    case CN_TYPE_XPORT :
    case CN_TYPE_IEEEB :
    case CN_TYPE_IEEEL :
        break;
    default:
        return(-2);
}
if (fromtype == totype) {
    memcpy(to,from,8);
    return(0);
}
switch(fromtype) {
    case CN_TYPE_IEEEB :
        if (totype == CN_TYPE_XPORT)
            ieee2xpt(from,to);
        else memcpy(to,from,8);
        break;
    case CN_TYPE_XPORT :
```

```
        xpt2ieee(from,to);
        break;
}
if (totype == CN_TYPE_IEEEL) {
    memcpy(temp,to,8);
    for (i=7;i>=0;i--) {
        to[7-i] = temp[i];
}
return(0);
}

int get_native() {

static char float_reps[][8] = {
{0x41,0x10,0x00,0x00,0x00,0x00,0x00,0x00},
{0x3f,0xf0,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0xf0,0x3f}
};

static double one = 1.00;

int i,j;
j = sizeof(float_reps)/8;
for (i=0;i<j;i++)  {
    if (memcmp(&one,float_reps+i,8) == 0)
        return(i+1);
    }
return(-1);
}

#ifdef BIG_ENDIAN
#define REVERSE(a,b)
#endif

#ifdef LITTLE_ENDIAN
#define DEFINE_REVERSE
void REVERSE();
#endif

#if !defined(DEFINE_REVERSE) && !defined(REVERSE)
#define DEFINE_REVERSE
void REVERSE();
#endif

void xpt2ieee(xport,ieee)

unsigned char *xport;
unsigned char *ieee;

{

char temp[8];
register int shift;
register int nib;
unsigned long ieee1,ieee2;
unsigned long xport1 = 0;
unsigned long xport2 = 0;
```

```
memcpy(temp,xport,8);
memset(ieee,0,8);

if (*temp && memcmp(temp+1,ieee,7) == 0) {
    ieee[0] = ieee[1] = 0xff;
    ieee[2] = ~(*temp);
    return;
    }

memcpy(((char *)&export1)+sizeof(unsigned long)-4,temp,4);
REVERSE(&export1,sizeof(unsigned long));
memcpy(((char *)&export2)+sizeof(unsigned long)-4,temp+4,4);
REVERSE(&export2,sizeof(unsigned long));

/*************************************************************/
/* Translate IBM format floating point numbers into IEEE */
/* format floating point numbers. */
/* */
/* IEEE format: */
/* */
/* 6 5 0 */
/* 3 1 0 */
/* */
/* SEEEEEEEEEEEMMMM ............ MMMM */
/* */
/* Sign bit, 11 bits exponent, 52 bit fraction. Exponent is */
/* excess 1023. The fraction is multiplied by a power of 2 of */

/* the actual exponent. Normalized floating point numbers are */
/* represented with the binary point immediately to the left */
/* of the fraction with an implied "1" to the left of the */
/* binary point. */
/* */
/* IBM format: */
/* */
/* 6 5 0 */
/* 3 1 0 */
/* */
/* SEEEEEEEMMMM ......... MMMM */
/* */
/* Sign bit, 7 bit exponent, 56 bit fraction. Exponent is */
/* excess 64. The fraction is multiplied bya power of 16 of */
/* the actual exponent. Normalized floating point numbers are */
/* represented with the radix point immediately to the left of*/
/* the high order hex fraction digit. */
/* */
/* How do you translate from IBM format to IEEE? */
/* */
/* Translating back to ieee format from ibm is easier than */
/* going the other way. You lose at most, 3 bits of fraction, */
/* but nothing can be done about that. The only tricky parts */
/* are setting up the correct binary exponent from the ibm */
/* hex exponent, and removing the implicit "1" bit of the ieee*/
/* fraction (see vzctdbl). We must shift down the high order */
/* nibble of the ibm fraction until it is 1. This is the */
/* implicit 1. The bit is then cleared and the exponent */
```

```
/* adjusted by the number of positions shifted. A more */
/* thorough discussion is in vzctdbl.c. */

/* Get the first half of the ibm number without the exponent */
/* into the ieee number */
ieee1 = xport1 & 0x00ffffff;

/* get the second half of the ibm number into the second half */
/* of the ieee number . If both halves were 0. then just */
/* return since the ieee number is zero. */
if ((!(ieee2 = xport2)) && !xport1)
return;

/* The fraction bit to the left of the binary point in the */
/* ieee format was set and the number was shifted 0, 1, 2, or */
/* 3 places. This will tell us how to adjust the ibm exponent */
/* to be a power of 2 ieee exponent and how to shift the */
/* fraction bits to restore the correct magnitude. */

if ((nib = (int)xport1) & 0x00800000)
shift = 3;
else
if (nib & 0x00400000)
shift = 2;
else
if (nib & 0x00200000)
shift = 1;
else

shift = 0;

if (shift)
    {

    /* shift the ieee number down the correct number of places */
    /* then set the second half of the ieee number to be the */
    /* second half of the ibm number shifted appropriately, */
    /* ored with the bits from the first half that would have */
    /* been shifted in if we could shift a double. All we are */
    /* worried about are the low order 3 bits of the first */
    /* half since we're only shifting by 1, 2, or 3. */
ieee1 >>= shift;
ieee2 = (xport2 >> shift) |
((xport1 & 0x00000007) << (29 + (3 - shift)));
}

/* clear the 1 bit to the left of the binary point */
ieee1 &= 0xffefffff;

    /* set the exponent of the ieee number to be the actual */
    /* exponent plus the shift count + 1023. Or this into the */
    /* first half of the ieee number. The ibm exponent is excess */
    /* 64 but is adjusted by 65 since during conversion to ibm */
    /* format the exponent is incremented by 1 and the fraction */
/* bits left 4 positions to the right of the radix point. */
    ieee1 |=
(((((long)(*temp & 0x7f) - 65) << 2) + shift + 1023) << 20) |
```

```
(xport1 & 0x80000000);

REVERSE(&ieee1,sizeof(unsigned long));
memcpy(ieee,((char *)&ieee1)+sizeof(unsigned long)-4,4);
REVERSE(&ieee2,sizeof(unsigned long));
memcpy(ieee+4,((char *)&ieee2)+sizeof(unsigned long)-4,4);
return;

/*-------------------------------------------------------------*/
/* Name: ieee2xpt */
/* Purpose: converts IEEE to transport */
/* Usage: rc = ieee2xpt(to_ieee,p_data); */
/* Notes: this routine is an adaptation of the wzctdbl routine */
/* from the Apollo. */
/*-------------------------------------------------------------*/

void ieee2xpt(ieee,xport)

unsigned char *ieee; /* ptr to IEEE field (2-8 bytes) */
unsigned char *xport; /* ptr to xport format (8 bytes) */
{

register int shift;
    unsigned char misschar;
    int ieee_exp;
    unsigned long xport1,xport2;
    unsigned long ieee1 = 0;
    unsigned long ieee2 = 0;

char ieee8[8];

memcpy(ieee8,ieee,8);

    /*------get 2 longs for shifting-----------------------------*/
memcpy(((char *)&ieee1)+sizeof(unsigned long)-4,ieee8,4);
REVERSE(&ieee1,sizeof(unsigned long));
memcpy(((char *)&ieee2)+sizeof(unsigned long)-4,ieee8+4,4);
REVERSE(&ieee2,sizeof(unsigned long));

memset(xport,0,8);

    /*-----if IEEE value is missing (1st 2 bytes are FFFF)-----*/
if (*ieee8 == (char)0xff && ieee8[1] == (char)0xff) {
misschar = ~ieee8[2];
*xport = (misschar == 0xD2) ? 0x6D : misschar;
return;
}

/*************************************************************/
/* Translate IEEE floating point number into IBM format float */
/* */
/* IEEE format: */
/* */
/* 6 5 0 */
/* 3 1 0 */
/* */
/* SEEEEEEEEEEEMMMM ........ MMMM */
```

```
/* */
/* Sign bit, 11 bit exponent, 52 fraction. Exponent is excess */
/* 1023. The fraction is multiplied by a power of 2 of the */
/* actual exponent. Normalized floating point numbers are */
/* represented with the binary point immediately to the left */
/* of the fraction with an implied "1" to the left of the */
/* binary point. */
/* */
/* IBM format: */
/* */
/* 6 5 0 */
/* 3 5 0 */
/* */
/* SEEEEEEEMMMM ......... MMMM */
/* */
/* Sign bit, 7 bit exponent, 56 bit fraction. Exponent is */
/* excess 64. The fraction is multiplied by a power of 16 of */
/* of the actual exponent. Normalized floating point numbers */
/* are presented with the radix point immediately to the left */
/* of the high order hex fraction digit. */
/* */
/* How do you translate from local to IBM format? */
/* */
/* The ieee format gives you a number that has a power of 2 */
/* exponent and a fraction of the form "1.<fraction bits>". */
/* The first step is to get that "1" bit back into the */
/* fraction. Right shift it down 1 position, set the high */
/* order bit and reduce the binary exponent by 1. Now we have */

/* a fraction that looks like ".1<fraction bits>" and it's */
/* ready to be shoved into ibm format. The ibm fraction has 4 */
/* more bits than the ieee, the ieee fraction must therefore */
/* be shifted left 4 positions before moving it in. We must */
/* also correct the fraction bits to account for the loss of 2*/
/* bits when converting from a binary exponent to a hex one */
/* (>> 2). We must shift the fraction left for 0, 1, 2, or 3 */
/* positions to maintain the proper magnitude. Doing */
/* conversion this way would tend to lose bits in the fraction*/
/* which is not desirable or necessary if we cheat a bit. */
/* First of all, we know that we are going to have to shift */
/* the ieee fraction left 4 places to put it in the right */
/* position; we won't do that, we'll just leave it where it is*/
/* and increment the ibm exponent by one, this will have the */
/* same effect and we won't have to do any shifting. Now, */
/* since we have 4 bits in front of the fraction to work with,*/
/* we won't lose any bits. We set the bit to the left of the */
/* fraction which is the implicit "1" in the ieee fraction. We*/
/* then adjust the fraction to account for the loss of bits */
/* when going to a hex exponent. This adjustment will never */
/* involve shifting by more than 3 positions so no bits are */
/* lost. */

/* Get ieee number less the exponent into the first half of */
/* the ibm number */

xport1 = ieee1 & 0x000fffff;
```

```
    /* get the second half of the number into the second half of */
    /* the ibm number and see if both halves are 0. If so, ibm is */
    /* also 0 and we just return */

if ((!(xport2 = ieee2)) && !ieee1) {
    ieee_exp = 0;
    goto doret;
    }

/* get the actual exponent value out of the ieee number. The */
/* ibm fraction is a power of 16 and the ieee fraction a power*/
/* of 2 (16 ** n == 2 ** 4n). Save the low order 2 bits since */
/* they will get lost when we divide the exponent by 4 (right */
/* shift by 2) and we will have to shift the fraction by the */
/* appropriate number of bits to keep the proper magnitude. */
shift = (int)
(ieee_exp = (int)(((ieee1 >> 16) & 0x7ff0) >> 4) - 1023)
& 3;
/* the ieee format has an implied "1" immdeiately to the left */
/* of the binary point. Show it in here. */
xport1 |= 0x00100000;
if (shift)
{
/* set the first half of the ibm number by shifting it left */

/* the appropriate number of bits and oring in the bits */
/* from the lower half that would have been shifted in (if */
/* we could shift a double). The shift count can never */
/* exceed 3, so all we care about are the high order 3 */
/* bits. We don't want sign extention so make sure it's an */
/* unsigned char. We'll shift either5, 6, or 7 places to */
/* keep 3, 2, or 1 bits. After that, shift the second half */
/* of the number the right number of places. We always get */
/* zero fill on left shifts. */
xport1 = (xport1 << shift) |
    ((unsigned char) (((ieee2 >> 24) & 0xE0) >>
    (5 + (3 - shift))));

xport2 <<= shift;
}

/* Now set the ibm exponent and the sign of the fraction. The */
/* power of 2 ieee exponent must be divided by 4 and made */
/* excess 64 (we add 65 here because of the poisition of the */
/* fraction bits, essentially 4 positions lower than they */
/* should be so we incrment the ibm exponent). */

xport1 |=

(((ieee_exp >>2) + 65) | ((ieee1 >> 24) & 0x80)) << 24;
/* If the ieee exponent is greater than 248 or less than -260, */
/* then it cannot fit in the ibm exponent field. Send back the */
/* appropriate flag. */

doret:
if (-260 <= ieee_exp && ieee_exp <= 248) {
REVERSE(&xport1,sizeof(unsigned long));
```

```
memcpy(xport,((char *)&xport1)+sizeof(unsigned long)-4,4);
REVERSE(&xport2,sizeof(unsigned long));
memcpy(xport+4,((char *)&xport2)+sizeof(unsigned long)-4,4);
return;
}
memset(xport,0xFF,8);

if (ieee_exp > 248)
    *xport = 0x7f;
    return;

}

#ifdef DEFINE_REVERSE
    void REVERSE(intp,l)
    char *intp;
    int l;
    {
    int i,j;
    char save;
    static int one = 1;

#if !defined(BIG_ENDIAN) && !defined(LITTLE_ENDIAN)
if (((unsigned char *)&one)[sizeof(one)-1] == 1)

return;
#endif

j = l/2;
    for (i=0;i<j;i++) {
    save = intp[i];
    intp[i] = intp[l-i-1];
    intp[l-i-1] = save;
    }
    }
    #endif
```

Translating from Local to IBM Format

The IEEE format gives you a number that has a power of 2 exponent and a fraction of the form "1.<fraction bits>".

The first step is to get that "1" bit back into the fraction. Right shift it down 1 position, set the high order bit and reduce the binary exponent by 1. Now we have a fraction that looks like "1.<fraction bits>". and it is ready to be shoved into IBM format. The IBM fraction has 4 more bits than the IEEE, the IEEE fraction must therefore be shifted left 4 positions before moving it in. We must also correct the fraction bits to account for the loss of 2 bits when converting from a binary exponent to a hexadecimal one (>> 2). We must shift the fraction left for 0, 1, 2, or 3 positions to maintain the proper magnitude. Doing conversion this way would tend to lose bits in the fraction, which is not desirable or necessary if we cheat a bit.

First of all, we know that we are going to have to shift the IEEE fraction left 4 places to put it in the right position; we will not do that, we will just leave it where it is and increment the IBM exponent by one, this will have the since we have 4 bits in front of the fraction to work with, we will not lose any bits. We set the bit to the left of the fraction which is the implicit "1" in the IEEE fraction. We then adjust the fraction to

account for the loss of bits when going to a hexadecimal exponent. This adjustment will never involve shifting by more than 3 positions so no bits are lost.

Get IEEE number less the exponent into the first half of the IBM number:

```
xport1 = ieee1 & 0x000fffff;
```

Get the second half of the number into the second half of the IBM number and see if both halves are 0. If so, IBM is also 0 and we just return:

```
if ((!(xport2 = ieee2)) && !ieee1) {
ieee_exp = 0;
goto doret;
}
```

Get the actual exponent value out of the IEEE number. The IBM fraction is a power of 16 and the IEEE fraction a power of 2 (16 ** n == 2 ** 4n). Save the low order 2 bits since they will get lost when we divide the exponent by 4 (right shift by 2) and we will have to shift the fraction by the appropriate number of bits to keep the proper magnitude.

```
shift = (int)
(ieee_exp = (int)(((ieee1 >> 16) & 0x7ff0) >> 4) - 1023)
& 3;
```

The IEEE format has an implied "1" immediately to the left of the binary point. Show it in here:

```
xport1 |= 0x00100000;

if (shift)
        {
```

Set the first half of the IBM number by shifting it left the appropriate number of bits and oring in the bits from the lower half that would have been shifted in (if we could shift a double). The shift count can never exceed 3, so all we care about are the high order 3 bits. We don't want sign extension so make sure it is an unsigned char. We'll shift either 5, 6, or 7 places to keep 3, 2, or 1 bits. After that, shift the second half of the number the right number of places. We always get zero fill on left shifts.

```
xport1 = (xport1 << shift) |
        ((unsigned char) (((ieee2 >> 24) & 0xE0) >>
            (5 + (3 - shift))));
xport2 <<= shift;
}
```

Now set the IBM exponent and the sign of the fraction. The power of 2 IEEE exponent must be divided by 4 and made excess 64 (we add 65 here because of the position of the fraction bits, essentially 4 positions lower than they should be so we increment the IBM exponent).

```
xport1 |=
(((ieee_exp >>2) + 65) | ((ieee1 >> 24) & 0x80)) << 24;
```

If the IEEE exponent is greater than 248 or less than -260, it cannot fit in the IBM exponent field. Send back the appropriate flag.

doret:

```
if (-260 <= ieee_exp && ieee_exp <= 248) {
        REVERSE(&export1,sizeof(unsigned long));
        memcpy(xport,((char *)&export1)+sizeof(unsigned long)-4,4);
        REVERSE(&export2,sizeof(unsigned long));
        memcpy(xport+4,((char *)&export2)+sizeof(unsigned long)-4,4);
        return;
}
```

```
memset(xport,0xFF,8);

if (ieee_exp > 248)
        *xport = 0x7f;
return;

}

#ifdef DEFINE_REVERSE
void REVERSE(intp,l)
char *intp;
int l;
{
int i,j;
char save;
static int one = 1;

#if !defined(BIG_ENDIAN) && !defined(LITTLE_ENDIAN)
if (((unsigned char *)&one)[sizeof(one)-1] == 1)
        return;
#endif
j = l/2;
for (i=0;i<j;i++)  {
        save = intp[i];
        intp[i] = intp[l-i-1];
        intp[l-i-1] = save;
        }
}
#endif
```

If BIG_ENDIAN is defined, it is assumed that the platform is big endian. If LITTLE_ENDIAN is defined, it is assumed that the platform is little endian. Do not define both of them.

If FLOATREP is not defined, the proper value is determined at run time. Although this works, it incurs additional overhead that can increase CPU time with large files. Use the FLOATREP symbol to improve efficiency. Likewise, if neither BIG_ENDIAN nor LITTLE_ENDIAN is defined, the proper orientation is determined at run time. It is much more efficient to supply the proper definition at compile time.

As an example, consider this command on HP-UX:

```
cc testieee.c ieee.c -DFLOATREP=CN_TYPE_IEEEB -DBIG_ENDIAN
```

and the corresponding command on DEC UNIX:

```
cc testieee.c ieee.c -DFLOATREP=CN_TYPE_IEEEL -DLITTLE_ENDIAN
```

Here is the correct output from the testieee run:

```
Native -> Big endian IEEE match count = 4 (should be 4).
Native -> Little endian IEEE match count = 4 (should be 4).
Native -> Transport match count = 4 (should be 4).
Transport -> Big endian IEEE match count = 4 (should be 4).
Transport -> Little endian IEEE match count = 4 (should be 4).
Transport -> Native match count = 4 (should be 4).
Big endian IEEE -> Little endian IEEE match count = 4 (should be 4).
Big endian IEEE -> Transport match count = 4 (should be 4).
Big endian IEEE -> Native match count = 4 (should be 4).
Little endian IEEE -> Big endian IEEE match count = 4 (should be 4).
Little endian IEEE -> Transport match count = 4 (should be 4).
```

```
Little endian IEEE -> Native match count = 4 (should be 4).
```

Here is the source code for the test program, testieee.c

```
#define CN_TYPE_NATIVE 0
#define CN_TYPE_XPORT 1
#define CN_TYPE_IEEEB 2
#define CN_TYPE_IEEEL 3
void tohex();
#define N_TESTVALS 4
static char xpt_testvals[N_TESTVALS][8] = {
{0x41,0x10,0x00,0x00,0x00,0x00,0x00,0x00}, /* 1 */
{0xc1,0x10,0x00,0x00,0x00,0x00,0x00,0x00}, /* -1 */
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* 0 */
{0x41,0x20,0x00,0x00,0x00,0x00,0x00,0x00} /* 2 */
};
static char ieeeb_testvals[N_TESTVALS][8] = {
{0x3f,0xf0,0x00,0x00,0x00,0x00,0x00,0x00}, /* 1 */
{0xbf,0xf0,0x00,0x00,0x00,0x00,0x00,0x00}, /* -1 */
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* 0 */
{0x40,0x00,0x00,0x00,0x00,0x00,0x00,0x00} /* 2 */
};
static char ieeel_testvals[N_TESTVALS][8] = {
{0x00,0x00,0x00,0x00,0x00,0x00,0xf0,0x3f}, /* 1 */
{0x00,0x00,0x00,0x00,0x00,0x00,0xf0,0xbf}, /* -1 */
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* 0 */
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40} /* 2 */
};
static double native[N_TESTVALS] =

{1,-1,0,2};
#define N_MISSINGVALS 3
static char missingvals[N_MISSINGVALS][8] = {
{0x2e,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* std missing */
{0x41,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* .A */
{0x5A,0x00,0x00,0x00,0x00,0x00,0x00,0x00} /* .Z */
};
/* rc = cnxptiee(from,0,to,1); native -> transport */
/* rc = cnxptiee(from,0,to,2); native -> Big endian IEEE */
/* rc = cnxptiee(from,0,to,3); native -> Little endian IEEE */
/* rc = cnxptiee(from,1,to,0); transport -> native */
/* rc = cnxptiee(from,1,to,2); transport -> Big endian IEEE */
/* rc = cnxptiee(from,1,to,3); transport -> Little endian IEEE */
/* rc = cnxptiee(from,2,to,0); Big endian IEEE -> native */
/* rc = cnxptiee(from,2,to,1); Big endian IEEE -> transport */
/* rc = cnxptiee(from,2,to,3); Big endian IEEE -> Little endian IEEE */
/* rc = cnxptiee(from,3,to,0); Little endian IEEE -> native */
/* rc = cnxptiee(from,3,to,1); Little endian IEEE -> transport */
/* rc = cnxptiee(from,3,to,2); Little endian IEEE -> Big endian IEEE */
main()
{
char to[8];
int i,matched;
char hexdigits[17];

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(>amp;native[i],CN_TYPE_NATIVE,to,CN_TYPE_IEEEB);
```

```
    matched += (memcmp(to,ieeeb_testvals[i],8) == 0);
    }
printf("Native -> Big endian IEEE match count = %d (should be %d).\n",
matched,N_TESTVALS);

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(&native[i],CN_TYPE_NATIVE,to,CN_TYPE_IEEEL);
    matched += (memcmp(to,ieeel_testvals[i],8) == 0);
}
printf("Native -> Little endian IEEE match count = %d (should be %d).\n",
matched,N_TESTVALS);

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(&native[i],CN_TYPE_NATIVE,to,CN_TYPE_XPORT);
    matched += (memcmp(to,xpt_testvals[i],8) == 0);
}
printf("Native -> Transport match count = %d (should be %d).\n",
matched,N_TESTVALS);

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(xpt_testvals[i],CN_TYPE_XPORT,to,CN_TYPE_IEEEB);
    matched += (memcmp(to,ieeeb_testvals[i],8) == 0);
    }
printf("Transport -> Big endian IEEE match count = %d (should be %d).\n",
    matched,N_TESTVALS);

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(xpt_testvals[i],CN_TYPE_XPORT,to,CN_TYPE_IEEEL);
    matched += (memcmp(to,ieeel_testvals[i],8) == 0);
    }
printf("Transport -> Little endian IEEE match count = %d \
(should be %d).\n",
matched,N_TESTVALS);

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(xpt_testvals[i],CN_TYPE_XPORT,to,CN_TYPE_NATIVE);
    matched += (memcmp(to,&native[i],8) == 0);
    }
printf("Transport -> Native match count = %d (should be %d).\n",
matched,N_TESTVALS);

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(xpt_testvals[i],CN_TYPE_XPORT,to,CN_TYPE_NATIVE);
    matched += (memcmp(to,&native[i],8) == 0);
    }
printf("Transport -> Native match count = %d (should be %d).\n",
matched,N_TESTVALS);
}
for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(ieeeb_testvals[i],CN_TYPE_IEEEB,to,CN_TYPE_IEEEL);
    matched += (memcmp(to,ieeel_testvals[i],8) == 0);
    }
printf("Big endian IEEE -> Little endian IEEE match count = %d \
(should be %d).\n",
    matched, N_TESTVALS);
```

```
    cnxptiee(ieeeb_testvals[i],CN_TYPE_IEEEB,to,CN_TYPE_XPORT);
    matched += (memcmp(to,xpt_testvals[i],8) == 0);
    }
printf("Big endian IEEE -> Transport match count = %d (should be %d).\n",
    matched,N_TESTVALS);

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(ieeel_testvals[i],CN_TYPE_IEEEL,to,CN_TYPE_IEEEB);
    matched += (memcmp(to,ieeeb_testvals[i],8) == 0);
}
printf("Little endian IEEE -> Big endian IEEE match count = %d \
(should be %d).\n",
    matched,N_TESTIVALS;

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(ieeel_testvals[i],CN_TYPE_IEEEL,to,CN_TYPE_XPORT);
    matched += (memcmp(to,xpt_testvals[i],8) == 0);
}
printf("Little endian IEEE -> Transport match count = %d (should be
%d).\n",
    matched,N_TESTVALS;
}

for (i=matched=0;i<N_TESTIVALS;i++) {
    cnxptiee(ieeel_testvals[i],CN_TYPE_IEEEL,to,CN_TYPE_NATIVE);
    matched += (memcmp(to,&native[i],8) == 0);
}
printf("Little endian IEEE -> Native match count = %d (should be %d).\n",
    matched,N_TESTVALS;
}
void tohex(bytes,hexchars,length)
unsigned char *bytes;
char *hexchars;
int length;
{
static char *hexdigits = "0123456789ABCDEF";

int i;
for (i=0;i<length;i++)  {
    *hexchars++ = hexdigits[*bytes >> 4];
    *hexchars++ = hexdigits[*bytes++ & 0x0f];
}
*hexchars = 0;
}

--------------ieee.c---------------------------------------
#define CN_TYPE_NATIVE 0
#define CN_TYPE_XPORT 1
#define CN_TYPE_IEEEB 2
#define CN_TYPE_IEEEL 3
int cnxptiee();
void xpt2ieee();
void ieee2xpt();
#ifndef FLOATREP
#define FLOATREP get_native()
int get_native();
#endif
```

```
/*-------------------------------------------------------------------*/
```