MICROSOFT™

XENIX™ STANDARD OBJECT FILE FORMAT

January 1983

CONTENTS

1. INTRODUCTION.. 1

2. X.OUT ADVANTAGES.................................... 2

3. MAIN HEADER... 3

4. EXTENDED HEADER..................................... 6

5. SYMBOL TABLE.. 7

6. LONG FORM RELOCATION................................ 8

7. SHORT FORM RELOCATION............................... 9

Appendix A: <a.out.h>................................A-1

Appendix B: <sys/relsym.h>...........................B-1

Appendix C: MC68000 Header Example.......................C-1

MICROSOFT™

XENIX™ STANDARD OBJECT FILE FORMAT

January 1983

CONTENTS

1. INTRODUCTION.. 1

2. X.OUT ADVANTAGES.................................... 2

3. MAIN HEADER... 3

4. EXTENDED HEADER..................................... 6

5. SYMBOL TABLE.. 7

6. LONG FORM RELOCATION................................ 8

7. SHORT FORM RELOCATION............................... 9

Appendix A: <a.out.h>..................................A-1

Appendix B: <sys/relsym.h>.............................B-1

Appendix C: MC68000 Header Example......................C-1

LIST OF FIGURES

Figure 1. Main header.................................... 3

Figure 2. Object file layout............................ 5

Figure 3. Seek position macros.......................... 6

Figure 4. Extended header............................... 6

Figure 5. X.out symbol structure........................ 7

Figure 6. Nlist symbol structure........................ 7

Figure 7. Xlist symbol structure........................ 8

Figure 8. Long form relocation structure................ 8

Figure 9. Short form relocation structure............... 9

The XENIX X.out Standard Object File Format

Vic Heller

Microsoft Corporation

1. INTRODUCTION

In the course of developing and porting XENIX to a variety of processors, several limitations were encountered with respect to the object file formats in popular use. These limitations were also evident in different hardware configurations using the same processor.

One of the most basic problems is that structure declarations are not necessarily binary compatible between processors. Any structure that makes use of an integer field cannot be relied upon to be readable on a processor other than the one on which it was written.

Different compiler implementations align structure elements on different boundaries within the structure. In order to be portable, a structure's layout should be designed with knowledge of all the compilers likely to be encountered.

In the past, when a change was made in the configuration of the text and data segments, a new magic number was used. Given the number of processors and the variety of implementations currently available, a better scheme for encoding the new information was needed. Overloading the magic number was simply not a clean solution.

Microsoft sought to establish a common format that would be suitable for use with all processors with which we are currently working, as well as with those that we cannot as yet anticipate. Since the need for consistency is great, we were especially interested in allowing for future flexibility without causing further upheavals.

Our response was to design and implement the x.out object file format. Through the use of a common header and magic number, object files can be dealt with easily, even while allowing for a number of different relocation and symbol table formats. By attaching a short form of relocation records to executable files, the configuration of the text and data segments can be changed without recompiling or relinking. By adding an extended header whose size is encoded in the main header, more information can be stored as needed without impacting utilities that do not need the new information.

The rest of this document discusses the advantages of the x.out format, and explains the structures used in an x.out object file. The include files may be found in the appendices.

2. X.OUT ADVANTAGES.

The x.out object file format is capable of supporting the following processors: Motorola MC68000, Intel 8086 and iAPX286, Zilog Z8000, National NS16032, Digital PDP - 11 and VAX - 11. The header, extended header, symbol table and relocation structures can be read and understood on any processor, no matter which processor actually generated them.

In order to maintain portability between processors that order bytes and words differently, a field has been reserved in the main header to indicate the target processor and the current byte and word ordering of the header, extended header, symbol table and relocation records. The position of the field in the header is not affected by the current byte and word ordering of the header; it is always readable. Using this information and fixbin(1), a utility developed by Microsoft for this purpose, the byte and word ordering of an x.out object file can be adjusted on any known processor to be readable by any known processor. This flexibility greatly simplifies cross development. Of course, the ordering of the text and data segments themselves is set by the assembler and linker, and is not changed by fixbin.

To simplify the recognition of an x.out object file, a single magic number is used. The main header has a field that encodes much of the function of the old magic number, as well as other information relating to the run-time environment. The configuration of the text, data and stack segments, the version of operating system for which the file was compiled, and the type of text and data addressing used are all kept in the header.

In order to distinguish between executable files set up for different configurations, it is necessary to keep track of the text and data base addresses. These addresses are kept in the extended header.

To allow for several different formats, a field has been added to encode the type of symbol table and relocation records attached.

Currently, two different types of relocation records are used: a long form for linkable object files, and a short form for executable files. The short form saves space while allowing an executable to be relocated to run on a different configuration of the same processor, or to be converted to or from pure text by running ld over the file with the appropriate flags.

The nlist symbol table structure is not at all portable; the value field is declared as an integer. The length of a symbol name is limited to eight characters. The addition of an underscore to all C language symbol names reduces the effective length to seven. In the x.out symbol table, arbitrary length symbols are allowed, but some utilities enforce a practical limit of fifty characters. Since no fixed amount of space is reserved, shorter symbol names do not waste space.

In the future, an expanded symbol table may be added in order to aid a symbolic debugger.

The conversion to x.out impacts the XENIX kernel and nlist(3) as well as the following utilities: adb, as, file, ld, make, mkfs, nroff and troff, prof, size, strip, and ranlib. In addition, any other utility that deals with object file formats must be modified if it needs to deal with x.out.

3. MAIN HEADER

```
struct xexec {
        unsigned short  x_magic;
        unsigned short  x_ext;
        long            x_text;
        long            x_data;
        long            x_bss;
        long            x_syms;
        long            x_reloc;
        long            x_entry;
        char            x_cpu;
        char            x_relsym;
        unsigned short  x_renv;
};
```

Figure 1. Main header

All sizes used in the header and extended header are in bytes. The text, data and bss sizes are expected to be even.

The x_magic field is always set to X_MAGIC, as defined in the include file, <a.out.h>. X_ext is set to the size of the extended header, or zero if no extended header is attached.

X_text and x_data are the sizes of the text and data segments in the object file. X_bss is the size of uninitialized data space required at the time of execution; no space is used for this segment in the file.

X_syms is the size of the symbol table; x_reloc is the total size of attached relocation records.

X_entry is set to the entry point in the text segment; this value is only valid for executable files and is usually set to the base address of the text segment.

The x_cpu field encodes the target cpu as well as the current byte and word ordering of the header, extended header, symbol table and relocation records. The text and data segments themselves are always ordered as required by the target processor. The top bit is set if the byte ordering differs from the PDP11; the second bit is set if the word ordering differs. The low six bits encode the target cpu; the processor for which the file was assembled or compiled.

The x_relsym field contains the type of symbol table and relocation records attached to the object file. The low four bits encode the type of symbol table used, and is only valid when x_syms is non-zero. The high four bits encode the type of relocation records used, and is only meaningful when x_reloc is non-zero.

The x_renv field encodes information needed at run-time. The high two bits encode the version of XENIX kernel for which the file was compiled. This enables a file compiled for Bell version 7 to to be run in compatability mode under those XENIX kernels that support it.

Bit six is set if the file was compiled for large model text addressing; bit five if large model data. Large and small model addressing is a concept that applies to processors that can run in non-segmented and segmented modes. In non-segmented mode (small model), addresses are smaller and always refer to the <u>current</u> segment. In segmented mode (large model), addresses must be expanded to include the segment. Although used on all processors, these two bits are only useful on processors such as the Intel 8086, where both large and small models of addressing have been implemented.

Bit four is set if the file is a text overlay. Bit three is set on those processors that require a fixed size stack segment; if this bit is set, the xe_stksize field in the extended header should be set to the size of the stack needed when executing. This usually involves evaluating the stack requirements of each executable file, and passing the stack size to ld on the command line.

Bit two is set if the text segment is to be pure (write-protected) and shared among all users executing the same file. Bit one is set if the text and data segments use separate address spaces. Bit zero is set if the file is executable; object modules that have not yet been linked or have not had all external references resolved will not have this bit set.

```
|--------------------------------|
|          x.out header          |
|--------------------------------|
|        extended header         |
|--------------------------------|
|                                |
|          text segment          |
|                                |
|--------------------------------|
|                                |
|          data segment          |
|                                |
|--------------------------------|
|                                |
|          symbol table          |
|                                |
|--------------------------------|
|        text relocation         |
|--------------------------------|
|        data relocation         |
|--------------------------------|
```

Figure 2. Object file layout

An x.out object file has seven sections, the header, extended header, text, data, symbol table, text relocation, and data relocation (in that order). If the extended header is not present, no distinction can be made between the text and data relocation records. The symbol table and/or relocation records need not be present.

Since the type of symbol table and relocation records can be encoded in the header, the x.out format is capable of dealing with the original Bell a.out symbol table and relocation information.

The following macros define the seek positions for the various sections within an object file; they are defined in the include file and depend on the above ordering. These macros are not valid in x.out files that contain either: 1) Bell's a.out relocation information, or 2) the combined symbol table and relocation records used in an 8086 relocatable object module (not executable).

```
#define XEXTPOS(xp)            ((long) sizeof(struct xexec))
#define XTEXTPOS(xp)           (XEXTPOS(xp) + (long)(xp)->x_ext)
#define XDATAPOS(xp)           (XTEXTPOS(xp) + (xp)->x_text)
#define XSYMPOS(xp)            (XDATAPOS(xp) + (xp)->x_data)
#define XRELPOS(xp)            (XSYMPOS(xp) + (xp)->x_syms)
#define XENDPOS(xp)            (XRELPOS(xp) + (xp)->x_reloc)
#define XRTEXTPOS(xp, ep)      (XRELPOS(xp))
#define XRDATAPOS(xp, ep)      (XRELPOS(xp) + (ep)->xe_trsize)
```

Figure 3. Seek position macros

4. EXTENDED HEADER

```
struct xext {
        long    xe_trsize;
        long    xe_drsize;
        long    xe_tbase;
        long    xe_dbase;
        long    xe_stksize;
};
```

Figure 4. Extended header

The extended header currently contains five fields. Xe_trsize and xe_drsize contain the sizes of text and data relocation records attached to the file. The xe_tbase and xe_dbase fields contain the base address of the text and data segments as they will be located in memory.

The xe_stksize field contains the size of the stack segment required for execution, if the appropriate bit is set in x_renv in the main header. This field is used by those processors that cannot expand the stack dynamically.

5. SYMBOL TABLE

```
struct sym {
        unsigned short  s_type;
        unsigned short  s_pad;
        long            s_value;
};
```

Figure 5. X.out symbol structure

The sym structure is the standard x.out symbol table structure. It has been designed to be portable between processors and to remove the limitations of the nlist structure.

Each symbol in the symbol table consists of the above structure, followed by a null terminated symbol name. Using this method, long symbol names may be stored without reserving unused space in the symbol table. No attempt is made to align subsequent structures on even boundaries.

The s_type field encodes the symbol type: undefined, absolute, text, data, etc. A separate bit is provided to indicate that a symbol is external.

The s_pad field is padding to insure portability; it is not currently used. The s_value field contains the symbol's value; it is declared as a long as an aid to portability.

```
struct nlist {
        char            n_name[8];
        int             n_type;
        unsigned        n_value;
};
```

Figure 6. Nlist symbol structure

The nlist structure is supported unchanged from its original declaration. It is used by the library routine nlist(3), which has been expanded to understand x.out files. Since nlist(3) is used primarily on a native XENIX kernel to find addresses in the kernel's memory space, and since an integer field is usually large enough to store an address, the nlist structure is barely adequate.

If the library routine were to be used for cross development, it would be hopelessly inadequate, as an integer on one processor is often too small to store an address from another.

```
struct xlist {
        unsigned short  xl_type;
        unsigned short  xl_pad;
        long            xl_value;
        char           *xl_name;
};
```

Figure 7. Xlist symbol structure

The solution to the limitations of nlist(3) is the xlist structure and a new library routine, xlist(3). Xlist is used in the same manner as nlist, but allows longer symbol names and completely portable results.

The first three fields have exactly the same meaning as in the sym structure, above. The xl_name field is a pointer to a null-terminated symbol name, which must be initialized by the calling routine.

Since many non-standard XENIX utilities depend on nlist, the expanded version will continue to be supported.

6. LONG FORM RELOCATION

```
struct reloc {
        unsigned short  r_desc;
        unsigned short  r_symbol;
        long            r_pos;
};
```

Figure 8. Long form relocation structure

The reloc structure is the standard type of relocation record attached to linkable object modules.

The relocation records for the text and data segments must be kept in separate sections of the object file because the relocation of each segment is performed separately. In addition, the relocation records are examined in sequence; each relocation must take place at progressively greater offsets in the current segment.

The r_pos field is the offset within the current segment at which the relocation is to take place.

The r_desc field contains bits to indicate the segment referenced, the number of bytes involved in the reference, and whether the reference is relative.

The two high bits encode the segment referenced as one of text, data, bss and external. The next two bits encode the number of bytes that must be relocated; one, two or four bytes are the allowed sizes. The fifth highest bit is set if the reference is relative to the current location in the text or data segment.

If the segment referenced is external, it relates to a previously undefined symbol; the r_symbol field is then used an an index into the symbol table in order to obtain the value when it becomes defined. Zero is used to index the first entry in the symbol table. When relocation is performed, and an external reference relates to a newly defined symbol, the value in the current segment is set to the symbol's value and the segment referenced bits are set to the segment to which the symbol belongs.

If the segment referenced is not external, the value in the current segment is only updated by the amount necessary to perform the relocation.

7. SHORT FORM RELOCATION

```
struct xreloc {
        long    xr_cmd;
};
```

Figure 9. Short form relocation structure

The short form of relocation is used to save space in an object file while allowing relocation. Since it is only attached to executable files, all external references must be resolved. It is also limited in that relocation can only invlove two or four bytes. References to the bss segment are recorded as references to the data segment.

The high bit is set if the reference is to the text segment; otherwise it is assumed to be a reference to data. The second bit is set if the relocation involves four bytes; otherwise it involves two bytes. The remaining thirty bits encode the offset within the current segment at which relocation is to be performed.

Appendix A: <a.out.h>

```
struct xexec {                          /* x.out header */
        unsigned short  x_magic;   /* magic number            */
        unsigned short  x_ext;     /* extended header size */
        long            x_text;    /* size of text segment */
        long            x_data;    /* size of data segment */
        long            x_bss;     /* size of bss required */
        long            x_syms;    /* size of symbol table */
        long            x_reloc;   /* size of relocation   */
        long            x_entry;   /* entry point          */
        char            x_cpu;     /* cpu, byte/word order */
        char            x_relsym;  /* reloc & symbol type  */
        unsigned short  x_renv;    /* run-time environment */
};

struct xext {                           /* x.out header extension */
        long        xe_trsize;     /* text relocation size */
        long        xe_drsize;     /* data relocation size */
        long        xe_tbase;      /* text relocation base */
        long        xe_dbase;      /* data relocation base */
        long        xe_stksize;    /* stack size, if XE_FS */
};
```

```
/*
 *  Definitions for xexec.x_magic, HEX (short).
 */

#define ARCMAGIC    0xff65       /* 0177545, archive      */
#define X_MAGIC     0x0206       /* x.out magic number    */


/*
 *  Definitions for xexec.x_cpu, cpu type (char).
 *
 *  b                set if high byte first in short
 *   w               set if low word first in long
 *    cccccc         cpu type
 */

/*
 *  Bytes or words are "swapped", if not stored in
 *  PDP11 ordering.
 */
#define XC_BSWAP    0x80       /* bytes swapped       */
#define XC_WSWAP    0x40       /* words swapped       */

#define XC_NONE     0x00       /* none                */
#define XC_PDP11    0x01       /* pdp11               */
#define XC_23       0x02       /* 23fixed from PDP11  */
#define XC_Z8K      0x03       /* Z8000               */
#define XC_8086     0x04       /* I8086               */
#define XC_68K      0x05       /* M68000              */
#define XC_Z80      0x06       /* Z80                 */
#define XC_VAX      0x07       /* VAX 780/750         */
#define XC_16032    0x08       /* NS16032             */
#define XC_CPU      0x3f       /* cpu mask            */
```

```
/*
 *  Definitions for xexec.x_relsym (char).
 *
 *  rrrr            relocation table format
 *      ssss        symbol table format
 */

        /* relocation table format */
#define XR_RXOUT     0x00     /* x.out long form, linkable  */
#define XR_RXEXEC    0x10     /* short form, executable     */
#define XR_RBOUT     0x20     /* b.out format               */
#define XR_RAOUT     0x30     /* a.out format               */
#define XR_R86REL    0x40     /* 8086 relocatable format    */
#define XR_R86ABS    0x50     /* 8086 absolute format       */
#define XR_REL       0xf0     /* relocation format mask     */

        /* symbol table format */
#define XR_SXOUT     0x00     /* x.out, struct sym        */
#define XR_SBOUT     0x01     /* b.out, struct bsym       */
#define XR_SAOUT     0x02     /* struct asym (nlist)      */
#define XR_S86REL    0x03     /* 8086 relocatable format */
#define XR_S86ABS    0x04     /* 8086 absolute format     */
#define XR_SUCBVAX   0x05     /* separate string table    */
#define XR_SYM       0x0f     /* symbol format mask       */


/*
 *  Definitions for xexec.x_renv (short).
 *
 *  vv                  version compiled for
 *    xxxxxx            extra (zero)
 *
 *          r           reserved
 *           t          set if large model text
 *            d         set if large model data
 *             o        set if text overlay
 *              f       set if fixed stack
 *               p      set if text pure
 *                s     set if separate I & D
 *                 e    set if executable
 */

#define XE_V2        0x4000     /* up to and including 2.3 */
#define XE_V3        0x8000     /* after version 2.3       */
#define XE_VERS      0xc000     /* version mask            */

#define XE_RES       0x0080     /* reserved                */
#define XE_LTEXT     0x0040     /* large model text        */
#define XE_LDATA     0x0020     /* large model data        */
#define XE_OVER      0x0010     /* text overlay            */
```

```
#define XE_FS        0x0008    /* fixed stack            */
#define XE_PURE      0x0004    /* pure text              */
#define XE_SEP       0x0002    /* separate I & D         */
#define XE_EXEC      0x0001    /* executable             */

#define XEXTPOS(xp)           ((long) sizeof(struct xexec))
#define XTEXTPOS(xp)          (XEXTPOS(xp) + (long)(xp)->x_ext)
#define XDATAPOS(xp)          (XTEXTPOS(xp) + (xp)->x_text)
#define XSYMPOS(xp)           (XDATAPOS(xp) + (xp)->x_data)
#define XRELPOS(xp)           (XSYMPOS(xp) + (xp)->x_syms)
#define XENDPOS(xp)           (XRELPOS(xp) + (xp)->x_reloc)

#define XRTEXTPOS(xp, ep)     (XRELPOS(xp))
#define XRDATAPOS(xp, ep)     (XRELPOS(xp) + (ep)->xe_trsize)
```

```
struct aexec {                          /* a.out header */
        unsigned short  xa_magic; /* magic number         */
        unsigned short  xa_text;  /* size of text segment */
        unsigned short  xa_data;  /* size of data segment */
        unsigned short  xa_bss;   /* size of bss segment  */
        unsigned short  xa_syms;  /* size of symbol table */
        unsigned short  xa_entry; /* entry point          */
        unsigned short  xa_unused;/* not used             */
        unsigned short  xa_flag;  /* relocation stripped  */
};


/*
 *  Definitions for aexec.xa_magic, obsolete.
 */

#define FMAGIC       0407          /* normal             */
#define NMAGIC       0410          /* pure, shared text  */
#define IMAGIC       0411          /* separate I & D     */
#define OMAGIC       0405          /* text overlays      */

#define A_MAGIC1     FMAGIC
#define A_MAGIC2     NMAGIC
#define A_MAGIC3     IMAGIC
#define A_MAGIC4     OMAGIC


#define ATEXTPOS(ap)       ((long) sizeof(struct aexec))
#define ADATAPOS(ap)       (ATEXTPOS(ap) + (long)(ap)->xa_text)
#define ARTEXTPOS(ap)      (ADATAPOS(ap) + (long)(ap)->xa_data)
#define ARDATAPOS(ap)      (ARTEXTPOS(ap) + ((long) \
                               ((ap)->xa_flag? \
                                   0 : (ap)->xa_text)))
#define ASYMPOS(ap)        (ATEXTPOS(ap) + \
                              (((ap)->xa_flag? 1L : 2L) * \
                               ((long) (ap)->xa_text + \
                                (long) (ap)->xa_data)))
#define AENDPOS(ap)        (ASYMPOS(ap) + (long) (ap)->xa_syms)
```

```
struct bexec {                  /* b.out header */
        long    xb_magic;       /* magic number              */
        long    xb_text;        /* size of text segment      */
        long    xb_data;        /* size of data segment      */
        long    xb_bss;         /* size of bss segment       */
        long    xb_syms;        /* size of symbol table      */
        long    xb_trsize;      /* size of text relocation */
        long    xb_drsize;      /* size of data relocation */
        long    xb_entry;       /* entry point               */
};

#define BTEXTPOS(bp)        ((long) sizeof(struct bexec))
#define BDATAPOS(bp)        (BTEXTPOS(bp) + (bp)->xb_text)
#define BSYMPOS(bp)         (BDATAPOS(bp) + (bp)->xb_data)
#define BRTEXTPOS(bp)       (BSYMPOS(bp) + (bp)->xb_syms)
#define BRDATAPOS(bp)       (BRTEXTPOS(bp) + (bp)->xb_trsize)
#define BENDPOS(bp)         (BRDATAPOS(bp) + (bp)->xb_drsize)


/*
 *  nlist symbol table structure.
 *
 *  Used to provide compatibility with nlist(3).
 */

struct nlist {
        char        n_name[8];      /* symbol name */
        int         n_type;         /* type flag */
        unsigned    n_value;        /* value */
};


/*
 *  xlist symbol table structure, used by xlist(3).
 */

struct xlist {
        unsigned short  xl_type;        /* symbol type         */
        unsigned short  xl_pad;         /* for transient use */
        long            xl_value;       /* symbol value        */
        char           *xl_name;        /* pointer to name     */
};
```

Appendix B: <sys/relsym.h>

```
/*
 *  Symbol table for x.out.
 *  The "sym" structure replaces the old "asym" (nlist)
 *  structure used by a.out.  Each symbol in the table has
 *  the below structure, followed immediately by its name
 *  in the form of a null terminated string.
 *  Note that no effort is made to word align subsequent
 *  "sym" structures in the symbol table.
 */

struct sym {                            /* symbol management */
        unsigned short  s_type;     /* symbol type          */
        unsigned short  s_pad;      /* portability padding */
        long            s_value;    /* symbol value         */
};


#define SYMLENGTH  50        /* Maximum symbol name length */


/*  Definitions for sym.s_type:  */

#define S_UNDEF     0x0000          /* undefined             */
#define S_ABS       0x0001          /* absolute              */
#define S_TEXT      0x0002          /* text                  */
#define S_DATA      0x0003          /* data                  */
#define S_BSS       0x0004          /* bss                   */
#define S_COMM      0x0005          /* for internal use only */
#define S_REG       0x0006          /* undefined             */
#define S_COMB      0x0007          /* for internal use only */
#define S_TYPE      0x001f          /* type mask             */
#define S_FN        0x001f          /* file name symbol      */
#define S_EXTERN    0x0020          /* external bit          */

#define FORMAT      "%08lx"         /* symbol value format   */
#define FWIDTH      8               /* symbol format width   */
```

```
/*
 *  Symbol table for a.out.
 *
 *  Modified from nlist for portability.
 */

struct asym {
        char               sa_name[8];     /* symbol name  */
        unsigned short     sa_type;        /* symbol type  */
        unsigned short     sa_value;       /* symbol value */
};


/*
 *  Definitions for asym.sa_type and nlist.n_type.
 */

#define N_UNDF     0       /* undefined        */
#define N_ABS      01      /* absolute         */
#define N_TEXT     02      /* text symbol      */
#define N_DATA     03      /* data symbol      */
#define N_BSS      04      /* bss symbol       */
#define N_TYPE     037     /* type mask        */
#define N_REG      024     /* register name    */
#define N_FN       037     /* file name symbol */
#define N_EXT      040     /* external bit     */


/*
 *  Symbol table for b.out.
 *
 *  The same as x.out, except that it uses 6 bytes
 *  on most machines.
 */

struct bsym {
        char   sb_type;    /* symbol type  */
        long   sb_value;   /* symbol value */
};
```

```
/*
 *  Relocation table entry for x.out, long form.
 *  This form is normally attached to ".o" files.
 *  Bit-wise compatible with the b.out format on
 *  machines that allocate bitfields from the high
 *  end of a word. (68k)
 */

struct reloc {
        unsigned short  r_desc;     /* descriptor          */
        unsigned short  r_symbol;   /* external symbol id  */
        long            r_pos;      /* position in segment */
};


/*
 *  Definitions for reloc.r_desc (short).
 *
 *  ss                    segment
 *    ss                  size
 *      d                 displacement
 *       xxx              extra
 *          xxxxxxxx      extra
 */

#define RD_TEXT     0x0000
#define RD_DATA     0x4000
#define RD_BSS      0x8000
#define RD_EXT      0xc000
#define RD_SEG      0xc000

#define RD_BYTE     0x0000
#define RD_WORD     0x1000
#define RD_LONG     0x2000
#define RD_SIZE     0x3000

#define RD_DISP     0x0800

/*
 *  Definitions for reloc.r_desc, compatible with bitfield
 *          allocation from the low end of a word (pdp11).
 */

#define RD_BTEXT    0x0000
#define RD_BDATA    0x0001
#define RD_BBSS     0x0002
#define RD_BEXT     0x0003
#define RD_BSEG     0x0003
```

```
#define RD_BBYTE   0x0000
#define RD_BWORD   0x0004
#define RD_BLONG   0x0008
#define RD_BSIZE   0x000c

#define RD_BDISP   0x0010


/*
 *  Relocation table entry for x.out, short form.
 *  This form is normally attached to executable files.
 *  Currently used on the 68k.
 */

struct xreloc {
        long   xr_cmd;     /* reloc command */
};


/*
 *  Definitions for xreloc.xr_cmd (long).
 *
 *  c                                        set if code segment
 *   l                                       set if long operand
 *    oooooooooooooooooooooooooooooo         offset
 */

#define XR_CODE    0x80000000      /* code/data segment  */
#define XR_LONG    0x40000000      /* long/short operand */
#define XR_OFFS    0x3fffffff      /* 30 bit offset mask */
```

Appendix C: MC68000 Header Example

The following C code illustrates how the header and extended header fields would be set up for an object file that is executable, is byte and word ordered for the MC68000, has the MC68000 as its target processor, has a fixed stack, and does not have pure text, separate I & D or text overlays.

Tsize, dsize, bsize, and stksize are the sizes of the text, data, bss and stack segments. Ssize is the size of the symbol table. Tbase is the base address of the text segment. Ntrel and ndrel are the number of relocations to be performed in the text and data segments.

```
xexec.x_magic = X_MAGIC;
xexec.x_ext = sizeof(struct xext);
xexec.x_text = tsize;
xexec.x_data = dsize;
xexec.x_bss = bsize;
xexec.x_syms = ssize;
xexec.x_reloc = (ntrel + ndrel) * sizeof(struct xreloc);
xexec.x_entry = tbase;
xexec.x_cpu = XC_BSWAP | XC_68K;
xexec.x_relsym = XR_SXOUT | XR_RXEXEC;
xexec.x_renv = XE_LTEXT | XE_LDATA | XE_FS | XE_EXEC;

xext.xe_trsize = ntrel * sizeof(struct xreloc);
xext.xe_drsize = ndrel * sizeof(struct xreloc);
xext.xe_tbase = tbase;
xext.xe_dbase = tbase + tsize;
xext.xe_stksize = stksize;
```