Analysis Of

Object File Formats For Embedded Systems

Minda Zhang, Ph.D
Senior software engineer
Intel Corporation

June, 1995

1. Introduction

An object file format is the lowest level file format for any platform. It is designed with the primary goal of providing formatted binaries of machine codes, initialized data and uninitialized data for an execution vehicle. As a secondary goal, it provides source level debugging (SLD) information, such as line number information for a communication utility that links the host computer to the debugging engine.

Typically, an embedded system is a micro processor based system with an application specific software self-contained in Read-Only-Memory(ROM) and without readily-recognizable software operating system. The interface between a host machine and an embedded system is implemented by a debugger, which uses all three aspects of an object file format. The programming for an embedded system requires knowledge of how to setup text and data segments in the different memory locations and how to initialize architecture specific data structure.

From this point of view, the primary difference between an executable object file for an embedded system and for a computer system lies in absolute address resolution, although the specification of an object file format for an embedded system and a computer system is the same. An executable object file for a computer system are always relocatable, however, a finalized executable object file for an embedded system is not relocatable: the addresses for text, data segments are absolutely not changeable. As a consequence, after a source file has been translated into a relocatable object file, and bound with other object modules, which mostly are library object modules, the builder utility needs a locator, (usually assisted by a build configuration file), to fix the absolute addresses for each segment.

In this paper, we review Intel OMF and COFF, and we also describe some important tools commercially available for handling these formatted object files. At last, we discuss the ideas for converting a relocatable
COFF object file into a linkable Intel OMF object file, and propose a 6-step procedure for the conversion.

2. Linkable Intel Object Modules Format

Intel object file format has two class specifications: Intel OMF specification and Intel hexadecimal object file format specification. The Intel OMF specification defines the structure of four kinds of object files, the linkable file which contains one or more linkable modules, the loadable file which is a single module absolute object file, the bootloadable file which contains a bootloadable module, and the library file which contains a collection of linkable modules and a directory.

Intel hexadecimal object file format specification defines the format which represents an absolute binary object file in ASCII. The hexadecimal format is suitable as input to PROM programmers and hardware emulators.

In this section, we assume that Intel OMF is Intel OMF386 which is designed for the 80386 processors.

The first byte of a linkable object file is a hexadecimal number B0 which is the Intel linkable OMF file type. (cf. [1]). A linkable object file contains one or more linkable modules. As illustrated in figures 1 and 2, each linkable module must contain a first partition, since it records the most important message of the module, e.g., the formatted binaries of machine codes, initialized data and uninitialized data which are stored in the TXTFIX section, the relocation information of the module which is stored in fixup blocks of the TXTFIX section, the type checking information for symbols defined in PUBDEF and EXTDEF sections which is stored in TYPDEF section and information for binding or linking with other linkable files which is stored in SEGDEF section.

The 163-byte long linkable module header (LMH) records all the information on the module's creation environment, total length in bytes of the current module, and number of sections for segments, publics

and externals. Its structure declaration is defined as following. Here we assume that in C, numeric scalar type *char* is used for 1 byte value, *int* is used for 2-byte value and *long* is used for 4-byte value.

```
struct lmh {                        /* linkable module header */
        long    tot_length;         /* total length of the module on disk in bytes */
        int     num_segs;           /* number of SEGDEF sections in the module */
        int     num_gates;          /* number of GATDEF sections in the module */
        int     num_publics;        /* number of PUBDEF sections in the module */
        int     num_externals;      /* number of EXTDEF sections in the module */
        char    linked,             /* linked = 0, if the module was produced by a translator */
                date[8],            /* the creation date, written in the form MM/DD/YY */
                time[8],            /* the creation time, written in the form HH:MM:SS */
                mod_name[41],       /* name of the module, the first char is the string's length
*/
                creator[41],        /* the name of the program which created the module */
                src_path[46];       /* the path to the source file which produced the module */
        char    trans_id;           /* translator id, mainly for debugger */
        char    trans_vers[4];      /* translator version (ASCII) */
        char    OMF_vers;           /* OMF version  */
 };
```

The 64-byte long table of contents (TOC) for first partition describes location and length for every section in the partition. TOC plays the central role in accessing the various sections within the linkable object file. The location of a section represents a byte offset into the current module, therefore, the first field in the linkable module header starts at byte 0 of the current linkable module. See figure 2, for an illustration of the usage for fields in the TOC.

```
struct   toc_p1 {                   /* Table of contents for first partition */
        long    SEGDEF_loc,         /* all the following _loc represents location of the first byte
*/
                SEGDEF_len,         /* of the section in current module, unit is byte;
*/
                GATDEF_loc,         /* all the following _len represents the length of the
section*/
                GATDEF_len,         /* also the unit is byte.
*/
                TYPDEF_loc,
                TYPDEF_len,
                PUBDEF_loc,
                PUBDEF_len,
                EXTDEF_loc,
                EXTDEF_len,
                TXTFIX_loc,
                TXTFIX_len,
                REGINT_loc,
                REGINT_len,
                next_partition,
                reserved;
 };
```

The SEGDEF section contains one or more segment definitions. It defines the segment's name, length and alignment type. The 2-byte attributes in the field of segdef describes the rules that the segment can be combined with other logical segments at binding, linking or loading time. (cf. [1])

```
struct   segdef {                   /* segment definition */
        int     attributes;         /* need to be separated into bits to get bitwise info(cf. [1])
*/
        long    slimit,             /* the length of the segment minus one, in bytes */
                dlength,            /* the number of data bytes in the segment, only for dsc
seg*/
```

```
                speclength;             /* the total number of bytes in the segment */
        int     ldt_position;           /* the position in LDT that this segment must occupy */
        char    align;                  /* alignment requirements of the segment */
        char    combine_name[41];       /* first char is the length of the string in byte, rest is name
*/
 };
```

The GATDEF section defines an entry for each gate occurring in the module. There is a 1-byte field in the data structure which is used to identify type of gate from call gate, task gate, interrupt gate or trap gate. (cf. [1])

```
struct  gatdef {                        /* Gate definition */
        char    privilege;              /* privilege of gate */
        char    present;
        char    gate_type;
        long    GA_offset;              /* gate entry GA consists of GA_offset and GA_segment */
        int     GA_segment;
 };
```

The TYPDEF section serves two purposes: to allow Relocation and Linkage software to check the validity of sharing data across external linkages, and to provide type information to debuggers to interpret data correct. [2] provides storage size equivalence tables and lists the syntactical constructs for high level languages PL/M, PASCAL, FORTRAN and C.

```
struct  typdef {                        /* type definition */
        char            linkage;        /* is TRUE, if for public-external linkage; is FALSE, if only
                                           for debug symbols. */
        int             length;         /* the length in bytes of all the leaves in it */
        struct leaf     leaves;         /* all different leaves format */
};

 struct leaf {
        char              type;         /* an 8-bit number defines the type of the leaf */
        union {                         /* following are different kind of leaves */
              char        *string;
              int          num_2;
              long         num_4;
              ulong        num_8;
              signed int   s_2;
              signed long  s_4;
              signed ulong s_8;
              } content;
        struct leaf       *next;        /* points to next leaf */
 };
```

The PUBDEF section contains a list of public names with their general addresses for the public symbols. The 2-byte field *type_IN* specifies an internal name for a segment, gate, GDT selector or the special CONST$IN. This section serves to define symbols to be exported to other modules.

```
struct  pubdef {                        /* public definition */
        long    PUB_offset;             /* gen addr consists of PUB_offset and PUB_segment */
        int     PUB_segment;
        int     type_IN;                        /* internal name for the type of the public of
symbol */
        char    wordcount;              /* the total # of 16-bit entities of stacked parameters */
        char    *sym_name;
 };
```

The EXTDEF section lists all external symbols, which are then referenced elsewhere in the module by means of their internal name. The 2-byte field *seg_IN* specifies the segment that is assumed to contain

the matching public symbol and the 2-byte value of *type_IN* defines the type of the external symbol. (cf. [1])

```
struct   extdef {                        /* external definition */
         int       seg_IN;               /* internal name of segment having matched public symbol
*/
         int       type_IN;                    /* internal name for the type of the external symbol
*/
         char      allocate;                   /* not zero, if R&L needs allocate space for
external symbol*/
         union {
            int    len_2;
            long   len_4;
          } allocate_len;                /* number of bytes needed allocated for the external symbol
*/
         char      *sym_name;            /* the 1st char is length , the rest are name of the symbol*/
 };
```

The TXTFIX section consists of intermixed text block, fixup block and iterated text block. As one can see, it is the TXTFIX section that records the binaries for machine codes, initialized data and uninitialized data. TXTFIX section output by a translator under debug option will also contain SLD information.

```
struct   txtfix {                        /* text, iterated text and fixup block */
         char      blk_type;             /* 0 for text blk; 1 for fixup blk and 2 for iterated text blk */
         union {
            struct text      text_blk;          /* text block */
            struct fixup     fixup_blk;  /* fixup block */
            struct iterat    it_text_blk;/* iterated text block */
         } block;
         struct txtfix       *next;
 };
```

A text block contains binaries for code segment and data segment. These segments are relocatable. Other than that, all the SLD information is also implemented in this block by a translator under debug option. Segment MODULES in the text block is designed with the purpose of providing general information about the current module. Segment SYMBOLS provides entries for each symbol used in the module, including stack symbols, local symbols and symbols that are used as procedure or block start entries. Segment LINES consists of line offset values, each line offset is the byte offset of the start of a line in the code segment. Segment SRCLINES consists of line offsets of the source files.

```
 struct  text {                           /* text block */
         long      txt_offset;           /* gen addr consists of txt_offset and txt_IN */
         int       txt_IN;               /* internal segment name */
         long      length;               /* the length of the text content, in byte */
         union {
            char           *code;        /* CODE segment */
            char           *data;        /* DATA segment */
            struct  mod    modules;      /* MODULES segment */
            struct  sym    symbols;            /* SYMBOLS segment */
            struct  line   lines;        /* LINES segment */
            struct  src    srclines;           /* SRCLINES segment */
           } segment;
 };

 struct   mod {                           /* MODULES segment */
         int       ldt_sel;              /* a selector into the GDT for an LDT which contains the
                                             segments in this module */
         long      code_offset;          /* code segment GA consists of code_offset and code_IN */
         int       code_IN;
```

```
        long    types_offset;           /* TYPES GA consists of types_offset and types_IN */
        int     types_IN;
        long    sym_offset;             /* SYMBOLS GA consists of sym_coffset and sym_IN */
        int     sym_IN;
        long    lines_offset;           /* LINES GA consists of lines_offset and lines_IN */
        int     lines_IN;
        long    pub_offset;             /* PUBLICS GA consists of pub_offset and pub_IN */
        int     pub_IN;
        long    ext_offset;             /* EXTERNAL GA consists of ext_offset and ext_IN */
        int     ext_IN;
        long    src_offset;             /* SRCLINES GA consists of src_offset and src_IN */
        int     src_IN;
        int     first_line;             /* first line number */
        char    kind;                   /* 0 value for 286, 1 value for 386 format */
        char    trans_id;               /* same as lmh */
        char    trans_vers[4];          /* same as lmh */
        char    *mod_name;              /* same as lmh */
};

struct  sym {                           /* SYMBOLS segment */
        char    kind;                   /* kind of entries */
        union  {
          struct  blk       blk_start;  /* block start entry */
          struct  proc      prc_start;  /* procedure start entry */
          struct  sbase     sym_base;   /* symbol base entry */
          struct  symbol    s_ent;      /* symbol entry */
         } entry;
        struct sym          *next;
};

struct  blk  {                          /* block start entry */
        long    offset;                 /* offset in code segment */
        long    blk_len;                /* block length */
        char    *blk_name;              /* block name, note that first byte is the length of string */
};

struct  proc  {                         /* procedure start entry */
        long    offset;                 /* offset in code segment */
        int     type_IN;                        /* internal name of the typdef associated with the
proc */
        char    kind;                   /* specifying 16-bit or 32-bit */
        long    ebp_offset;             /* offset of return address from EBP */
        long    proc_len;               /* procedure length */
        char    *proc_name;             /* procedure name, as always, the 1st char is string length
*/
 };
 struct  sbase  {                       /* symbol base entry */
        long    offset;
        int     s_IN;
 };

 struct  symbol  {                      /* symbol entry */
        long    offset;
        int     type_IN;
        char    *sym_name;
 };

 struct  lines  {                       /* LINES segment */
        long    offset;
        struct lines        *next;
```

```
};

struct  srclines {                      /* SRCLINES segment */
        char            *src_file               /* source file name */
        int             count;
        struct lines    *src_line;
        struct srclines *next;
};
```

Fixup block contains information that allows the binder or linker to resolve (fix up) and eventually relocate references between object modules. The attributes *where_IN* and *where_offset* in the following data structures make a generalized address specifying the target for the fixup. Similarly, the attributes *what_IN* and *what_offset* make a generalized address specifying the target to which the fixup is to be applied.

There are four kinds of fixups for Intel linkable object modules. They are general fixup, intra-segment fixup, call fixup and addition fixup. The general fixup and the addition fixup have the same data structure, both provide general addresses for where_IN, where_offset, and what_IN, what_offset. The intra-segment fixup is equivalent to a general fixup with what_IN = where_IN, and the call fixup is also equivalent to a general fixup with what_offset = 0. (cf. [1])

```
struct  fixup {                         /* fixup block */
        int     where_IN;               /* specifying the  segment to which fixups should be
applied*/
        int     length;                 /* the length in bytes of the fixups */
        union {
           struct gen      general;     /* for general fixup */
           struct intra    in_seg;      /* for intra-segment fixup */
           struct cal      call_fix;    /* call fixup */
           struct ad       addition;    /* addition fixup */
         } fixups;
};

struct  gen {                           /* for general fixup */
        char    kind;                   /* specifying the kind of fixup */
        union {
           int  num2;
           long num4;
        } where_offset;                 /* 2- or 4- byte where_offset */
        union {
           int  num2;
           long num4;
        } what_offset;                  /* 2- or 4- byte what_offset */
        int     what_IN;                /* what_IN & what_offset specify the target for the fixup*/
        union fixups    *next;
};

struct  intra {                         /* for intra-segment fixup */
        char    kind;                   /* specifying the kind of fixup */
        union {
           int  num2;
           long num4;
        } where_offset;                 /* 2- or 4- byte where_offset */
        union {
           int  num2;
           long num4;
        } what_offset;                  /* 2- or 4- byte what_offset */
        union fixups    *next;
};
```

```
struct  cal {                       /* for call fixup */
        char    kind;               /* specifying the kind of fixup */
        union {
        int     num2;
        long    num4;
        }  where_offset;            /* 2- or 4- byte where-offset */
        int     what_IN;
        union fixups    *next;
};

struct  ad {                        /* for addition fixup */
        char    kind;               /* specifying the kind of fixup */
        union {
           int  num2;
           long num4;
        } where_offset;             /* specifying the target to which the fixup is to be applied */
        union {
           int  num2;
           long num4;
        }  what_offset;
        int     what_IN;
        union fixups    *next;
};

struct  iterat  {                   /* for iterated text block */
        long    it_offset;
        int     it_segment;         /*above two specify a gen addr to put 1st byte of the text */
        long    it_count;           /* the # of times the text template is to be repeated */
        struct temp     text;       /* the text template */
};

struct  temp {                      /* for the text template in the iterated text block */
        long    length;             /* the length, in bytes, of a single mem blk to be initialized
*/
        char    *value;             /* the text or data to be used to initialize any single mem
blk*/
};
```

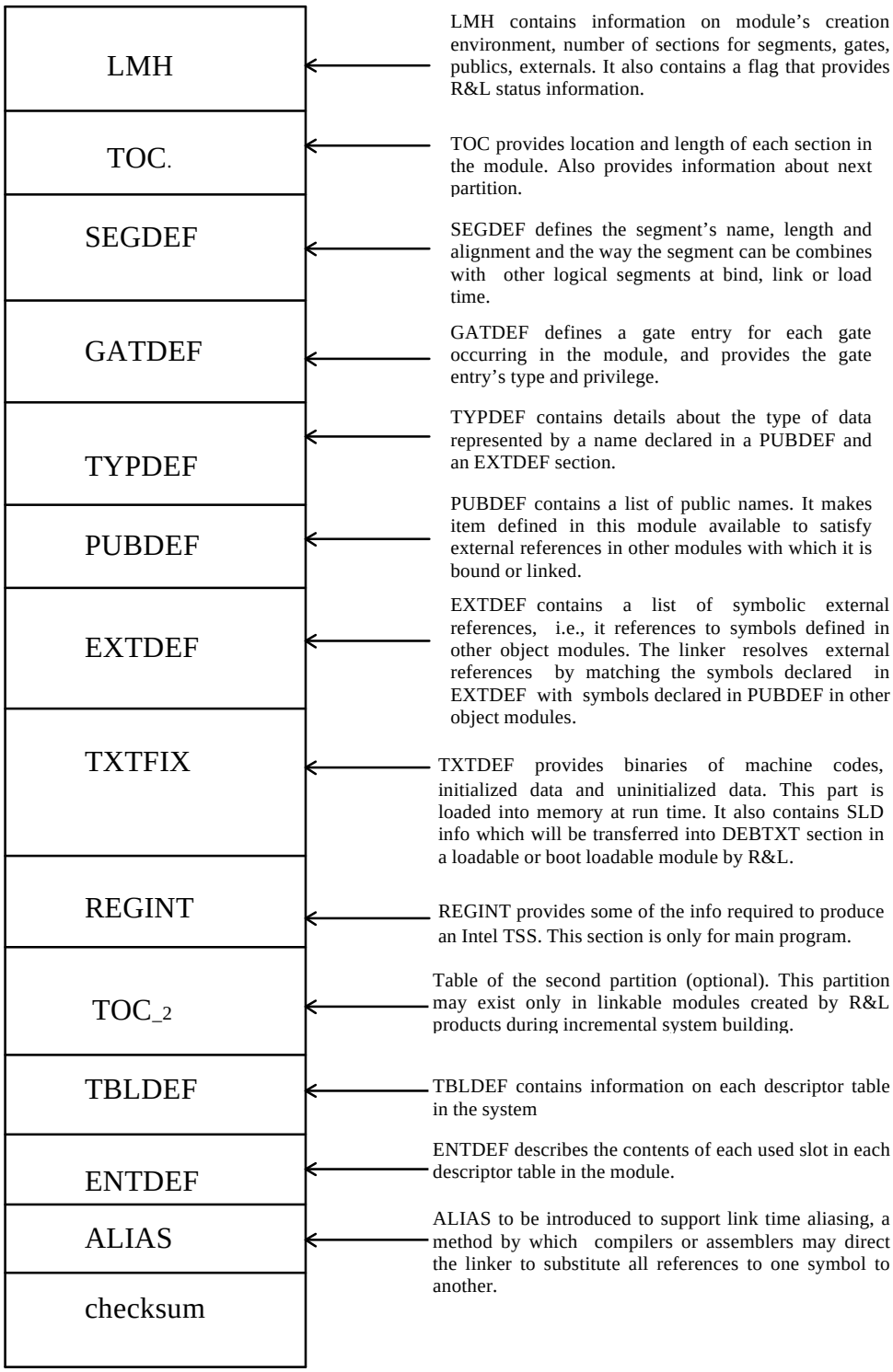

Figure 1 Linkable module basic structure and components

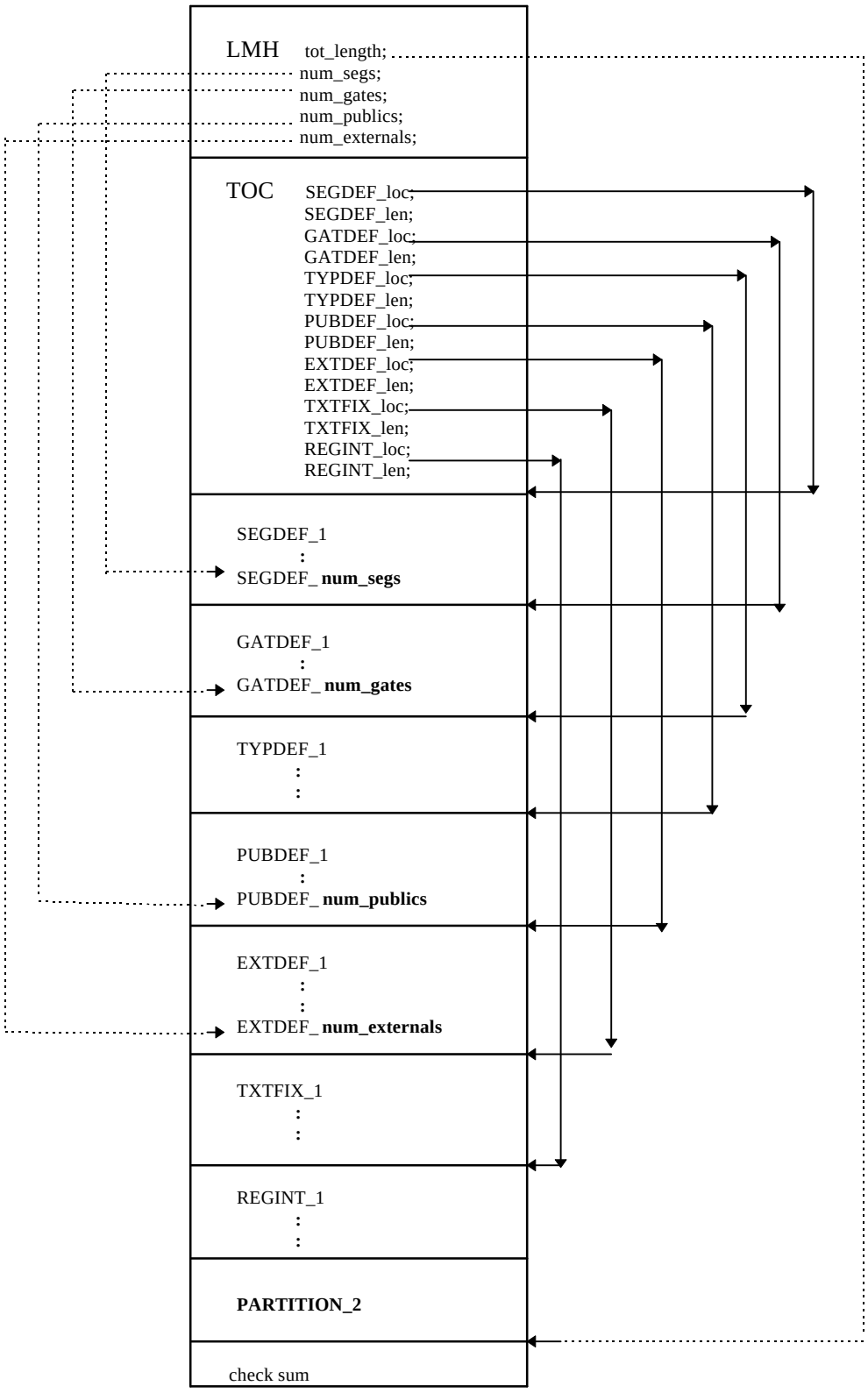

Figure 2 lmh & toc components and structural relationships

3. Common Object File Format (COFF)

COFF is one of the most well-known object file formats. It was originally designed for defining the structure of machine code files in the UNIX System V environment.(cf. [5]) Now there are several

modified version of COFF formats, for example, XCOFF is the formal definition for the structure of object files in the X window's environment.

COFF supports several types of object files, relocatable object files, fully linked executable object files and system libraries. Briefly speaking, COFF actually defines a system for both execution and debugging. The execution vehicle can access machine code section and initialized or uninitialized data sections for running the program. The debugger can access section's header, section's line number entries and symbol table entries for source level debugging.

The roughly fifty or so bytes at the beginning of the COFF file contain two types of COFF headers (cf. Figure 3) The first twenty bytes is the COFF file header which holds information specifying number of sections in the object file, pointing to the symbol table with the number of symbol table entries, and indicating the data and the time at which the object file was created. The C language structural declaration for COFF file header is found in the file *filehdr.h* which is usually located at the UNIX environment */usr/include* directory. As usual, the following type *unsigned short* is two bytes, and *long* is four bytes.

```
struct filehdr    {                       /* COFF file header structural definition */
        unsigned short   f_magic;                 /* magic number to indicate the UNIX port
*/
        unsigned short   f_nscns;         /* number of sections in this object file */
        long             f_timdat;        /* # of seconds since GMT 00:00:00 Jan. 1, 1970
*/
        long             f_symptr;        /* offset of symbol table in this object file */
        long             f_nsyms;                 /* number of symbol table entries */
        unsigned short   f_opthdr;        /* 0, if there is no aout header in the file */
        unsigned short   f_flags;         /* flags */
        } ;

#define         FILHDR              struct filehdr
#define                 FILHSZ      sizeof(FILHDR).
```

The next thirty bytes are reserved for the second type COFF header, i.e., the *AOUT* header or optional header. The optional header was designed with the primary purpose of providing the linker with implementation dependent information and the necessary run-time parameters, e.g., the magic number indicating whether the COFF file is a normal executable file, all the sizes for text, initialized data and uninitailized data, and the address used by the system as the starting point for program execution, the start addresses for both text and data sections. The structural declaration for optional header is listed in the file *aouthdr.h* which again located in the UNIX environment */usr/include* directory.

```
typedef struct aouthdr    {             /* COFF optional header structure */
        short     magic;                /* 0413(octal), if it is a normal executable file */
        short     vstamp;               /* version stamp set by assembler & used by linker
*/
        long      tsize;                /* text section size in bytes */
        long      dsize;                /* data section size in bytes (initialized data)*/
        long      bsize;                /* bss section size in bytes (uninitialized data)*/
        long      entry;                /* starting point for program execution */
        long      text_start;           /* start address for text section */
        long      data_start;           /* start address for data section */
        } AOUTHDR;
```

Immediately after COFF file headers are section headers, as showed by Figure 3. It is those section headers that play the central roles in accessing the various components within the file. Section header provides the linker with the necessary information to access section's content, section's relocation entries and symbol table entries for the relocation process. These components make possible to define the areas in machine code and data that require patching with run-time addresses. It also provides source level debugger (e.g. *sdb* or *gdb*) with the necessary information to access section's content, section's line number entries and symbol table entries. These components work together to set break-points and to trace program execution.

The section header's structural declaration is defined in the UNIX environment, the heading file *scnhdr.h* which also located in the directory, */usr/include.*

```
struct scnhdr  {                                /* COFF section header structural declaration */
        char            s_name[8];              /* section's name; 8 char at most */
        long            s_paddr;                        /* physical address */
        long            s_vaddr;                        /* virtual address */
        long            s_size;                 /* section size in bytes */
        long            s_scnptr;               /* section content's file offset in bytes */
        long            s_relptr;                       /* section entries' file offset in bytes */
        long            s_lnnoptr;              /* section line number entries' file offset in bytes */
        long            s_nreloc;               /* number of relocation entries within the section
*/
        unsigned short  s_nlnno;                        /* number of line number entries within
the section */
        long            s_flags;                /* type and content flags */
        };

#define                 SCNHDR          struct scnhdr
#define                 SCNHSZ          sizeof (SCNHDR).
```

As pointed out in [5], a relocation entry is created by the assembler for every instance of an address reference that requires patching by the linker. Relocation entry fields implement the dynamics of relocation. The field *r_vaddr* is the byte-offset value relative to the start of its raw data, it specifies the area in the section's content that requires patching by the linker. The field *r_symndx* is the index into the symbol table, it points to the appropriate symbol table entry that contains run-time address information, i.e., the field *n_value* defined in the structural declaration for symbol table entry.

The relocation entries' structural declaration is defined in the UNIX environment, the heading file *reloc.h* which is in the directory */usr/include*.

```
struct reloc    {                               /* COFF relocation structural declaration */
        long            r_vaddr;                        /* byte-offset value of reference  */
        long            r_symndx;               /* index into the symbol table */
        unsigned short  r_type;                 /* relocation type */
        };
```

Line number entries are generated by a compiler only in the case that the debug option was invoked. The structure of a line number entry is used to define a point in the source file that corresponds to a break point, i.e., the point where program execution can stop. (cf. [5]) As a consequence, line number information can be created only for the text section, and that is all needed to implement symbolic or source line reference to execution points within the source code. The structural declaration for line number entries is defined in the UNIX environment, the heading file *linenum.h* which is in the directory */usr/include.*

```
struct  lineno  {                               /* COFF line number entry */
        union   {
            long        l_symndx;               /* if l_lnno==0; then l_addr takes l_symndx*/
            long        l_paddr;                        /* else l_addr takes l_paddr */
        } l_addr;
        unsigned short  l_lnno;                 /* line number relative to the source file */
        };

#define                 LINENO                  struct    lineno
#define         LINESZ          6               /* the size of a line number entry */
```

At the end of a COFF object file, there are two components, symbol table and string table. As shown by figure 3, symbol table is the symbolic information resource of the object file which provides linker

a run-time address whenever it requires patching in the relocation process, symbol table is also the resource for source level debugger to understand the structure of its source file, as well as the binary file.

If the source file has been compiled under the debug option, an individual symbol table entry completely records an individual quantum of debug information, and the sequence of symbol table entries then maps the structure of source file written in a high level language (HLL). The field *n_sclass* defines the storage class for a symbol at run time, as a consequence of this, *n_sclass* determines the meaning of the value given by *n_value*. The field *n_type* provides the symbol's type information, and the field *n_scnum* indicates the section number where the symbol is defined. The complete correspondence relation among these fields is declared in the files *storeclass.h*, *a.out.h* and *syms.h*.

```
#define         SYMNMLEN     8

struct  syment  {                                   /* symbol table entries' structural declaration */
        union {
          char          _n_name[SYMNMLEN];  /* symbol name, old COFF version */
          struct{                                   /* symbol name, new COFF version */
                long    _n_zeroes;          /* new COFF version requires this be zero */
                long    _n_offset;          /* offset into string table for the symbol name */
                } _n_n;
          char          *_n_nptr[2];          /* use for offset, allows for overlaying */
        } _n;
        long            n_value;                    /* location for the symbol */
        long            n_scnum;            /* section number the symbol is defined */
        unsigned short  n_type;             /* type and derived type info of the symbol */
        char            n_sclass;           /* storage class */
        char            n_numaux;           /* number of auxiliary entries it needs */
        };

#define                 SYMENT          struct syment.
```

Symbol table entries for linked list structure needs auxiliary entries to provide linker or debugger extra information for efficient access of symbol table data. As described in [5], the *.file* auxiliary entry is the first auxiliary entry in the symbol table which provides the ASCII string for the file's name; the *.text*, *.data*
and *.bss* auxiliary entries record the section's length, number of relocation entries and the number of line number entries in the section. The auxiliary entries for structure, union or enumeration's tag record their size and index to next structure, union or enumeration. The function definition auxiliary entry records the size, a pointer to the function's line number entry and index to the next function's auxiliary entry. The block auxiliary entry records the line numbers for the start code and end of the block as well as an index to the next block's auxiliary entry.

A complete description for all types' auxiliary entries is declared in the following data structure. (cf. *syms.h*)

```
union  auxent  {                                    /* COFF auxiliary entry format */
        struct  {                                   /* auxiliary entry for linked list structure */
          long                  x_tagndx;           /* struct, union, or enum.  tag index */
          union {
            struct      {
                unsigned short  x_lnno;             /* declaration line number */
                unsigned short  x_size;             /* struct, union, or array size*/
                } x_lnsz;
            long                f_size;             /* size of function */
            } x_misc;
          union{
            struct      {                           /* if ISFCN, tag, or .bb */
                long            x_lnnoptr;          /* pointer to function line */
                long            x_endndx;           /* entry index past block end */
```

```
                } x_fcn;
            struct          {                       /* if ISARY, up to 4 dimension */
                unsigned short  x_dimen[DIMNUM];
                } x_ary;
            } x_fcnary;
            unsigned short      xtvndx;             /* tv index */
        } x_sym;
        struct    {                                 /* auxiliary entry for .file */
            char                x_fname[FILNMLEN];
         } x_file;
        struct    {                                 /* auxiliary entry for .text, .data, or .bss */
            long                x_scnlen;           /* section length */
            unsigned short              x_nreloc;           /* number of relocation entries */
            unsigned short              x_linno;            /* number of line number entries */
         } x_scn;
        struct    {                                 /* auxiliary entry for tv */
            long                x_tvfill;           /* tv fill value */
            unsigned short              x_tvlen;            /* length of tv */
            unsigned short              x_tvran[2];         /* tv range */
         } x_tv;
  };

#define         AUXENT          union auxent.
```

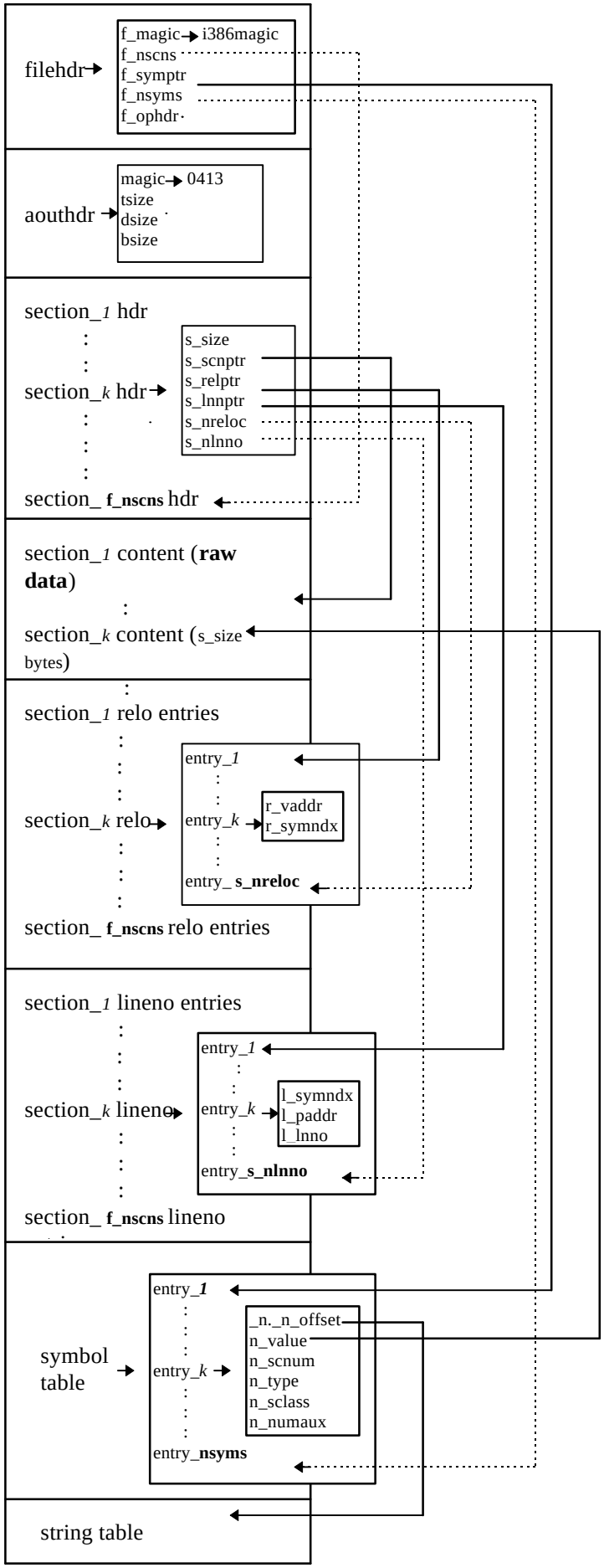

Figure 3 COFF components and structural relationships

4. Tools for object file formats

4.1. Tools for Intel OMF

DOS debugger is a useful tool for dumping any binary file. By counting the bytes of a linkable Intel OMF file displayed by DOS debugger, users can access every section of a module. However, it is a tedious process for users to dump an Intel linkable OMF file using DOS debugger. A dumping tool called *deomf.exe* [12] is designed to display fields of a linkable module header, components for toc, and components for every section in every partition for a linkable Intel OMF file.

In an embedded environment, usually the embedded system requires a development tools package and debug facilities. Commercially, there are some development tools which directly bind Intel OMF386 formatted object files with Intel OMF386 library object modules and build it into a bootloadable Intel OMF386 file, i.e., the ABS file which is ready for loading into the embedded system. Some vendors' development tool set can accept relocatable Intel OMF386 object modules and link them with their system library object modules to produce an executable file for an embedded system.

The Intel386™ processor family utilities provide total program development and system configuration resources for Intel386 processor family software. The utilities are the Librarian LIB386, the mapper MAP386, the binder BND386 and the system builder BLD386. Translators, like the assembler Intel ASM386, the C language compiler Intel IC386 etc., translate source files into Intel OMF386 formatted object modules. When building a system, collect these object modules according to their common functions and characteristics, and use BND386 to combine them with object modules in a system library generated by LIB386. Then use BLD386 which accepts an input building file to processes the modules and add the initial protected-mode data structures.

SSI (System & Software, Inc.) Link&Locate 386 package also provides a solution for the software development producing Intel OMF386 formatted object files. This package includes an embedded linker XLINK386, an ROM-image generator PROM386 and a librarian XLIB386. XLINK386 generates an output object file in Intel OMF386 format for protected-mode embedded applications.

Phar Lap Software, Inc. has an one-step linker/locator for embedded system, the LinkLoc. It accepts object files in Intel OMF386 format, and also can generate executable files in Intel OMF 386 format. LinkLoc supports the Intel 386 processor's protected-mode.

As for the debug facilities for embedded system, according to [4], commercial debugging tools could be catalogued as debugging software, ROM emulator debugger and Logic Analyzer/ In-Circuit-Emulators. A debugging software consists of debug tools that are purely software in nature, including debug kernels and source code debuggers, the target for a software debugger is usually RS-232 serial port. A ROM emulator debugger consists of ROM emulator hardware units that have direct links into ROM sockets in the target system. The Logic Analyzers and In-Circuit Emulators are the big guns in the debug arena for embedded systems programming. Essentially, stand-alone in-circuit emulators are embedded micro-processor systems that contain hardware and software capable of totally emulating the function and functional characteristics of a given CPU, hence, the target for an in-circuit emulator is the CPU socket.

As debugging software, SSI SoftProbe 386/RTD series accept Intel OMF386 absolute object files, they are DOS-hosted source-level debuggers for debugging real mode and protected mode 386 embedded C applications. Concurrent Science's Soft-Scope/CSi-Mon also supports Intel OMF386 ABS files. It is a DOS-hosted source level debugger for debugging real and protected mode C embedded applications. Basically, the Soft-Scope is a communication utility that uses DOS as a host computer's environment to load and debug in the source level for an Intel OMF386 ABS file through CSi-Mon.

For the ROM monitors, there are American Arium's RP-256, SSI SoftProbe 386/RED series, and Grammar Engine, Inc. product PromICE™. They accept Intel hex object files which can be generated by Intel tool OH386 converting from an Intel OMF386 ABS object file. In addition to an Intel ICE™ system, American Arium's EZ-Prob and LA/ICE suppport Intel486™ processors.

4.2. Tools for COFF

Tools for COFF, GOFF and ELF/DWARF could be classified as dumping tools, *dump*, tools for listing symbols from object files, *nm*, archive tools, *ar*, which creates or modifies an archive, and even extracts some object modules from an archive, tools for linking object modules into an executable file, and also debugging tools. The tool *dump* under certain options can dump the respective component of an object file, e.g., if it is under option *-f*, it dumps the file header, the detailed description for the options is listed in the man page for the tool *dump* in any UNIX environment.

Recently, GNU proposed a Binary File Descriptor (BFD) library which allows a program to operate on object files (e.g. *ld* or *GDB*) to support many different object formats in a clean way. Currently, BFD supports COFF, GOFF and ELF. GNU linker *gld* is a cross linker which accesses object and archive files using the BFD libraries to cover a broad range of object formats. It accepts a command file which uses linker scripts to specify everything necessary for describing the target. Basically, the command file controls input files, file formats, output file format, address of sections and placement of common blocks. Commercially, there are only CYGNUS supporting the GNU developer's kit. It is worth pointing out that there are also a lot of supports for the cross GNU linker and GNU debugger *GDB* over the internet.

5. Conversion analysis

The idea for converting an object file from one format to the other is not new. In the UNIX world, there already exists a converter *cof2elf*, which converts a relocatable COFF object file into a relocatable ELF object file. Also, as mentioned in the previous section, GNU uses BFD to process object files in a wide range of file formats. To do this, GNU takes an object file in one file format to fill in a standardized data structure, recording all the components of the object file, then all the processes are carried out based on this data structure. The idea behind this procedure is quite clear: they convert the input object file into the standardized form.

It is quite practical to convert a linkable (or relocateable) object file from one file format to another, although it seems more direct if an executable file can be converted. The direct conversion of executable files is problematic, however, due to conversion of architecture-specific initialization and alignment constraints. These constraints for a linkable file will be resolved during binding and building processes. Since all embedded development tool kits support building an executable file from linkable object modules and static shared libraries' object modules, translation at the executable file level is unnecessary.

In this section, we discuss the idea for converting a relocatable object file in COFF format into a linkable Intel OMF386 file. To make our narrative simpler, we propose a six-step procedure to illustrate the basic idea for converting from a relocatable COFF object file into a linkable Intel OMF object file. There are two conversion levels and one is to convert all the text and data segments, and process all the relocation entries. The second level needs extra effort in converting all the SLD information within the object file. Steps 0, 1, 2, and 5 in this procedure carry out the translation at first level, and steps 3 and 4 are designed for dealing with line number entries and symbol table entries. Consequently, the following six-step procedure as a whole carries out both levels of conversion.

Step 0 is an initialization step. It initializes all the fields in the *struct lmh* to be zeros, and also sets all the fields in the *struct toc_p1* to be zeros. The fields for recording the lengths or sizes of segments and sections will be updated during the steps in processing these segments or sections. However, the fields for pointing to locations of segments and sections in an Intel OMF object file will be fixed at the last step of the procedure.

Step 1 is to process all COFF sections' raw data content and also to convert their relocation entries in a relocatable COFF object file. This step will build SEGDEF and TXTFIX sections, but they are not ready to be flushed into the output object file, i.e. the Intel OMF object file, since they are not completed at this step. During steps 3 and 4, there will be more segments added into the TXTFIX

section. As a consequence of this, the SEGDEF section needs to catch the definitions for the associated segments in the TXTFIX section.

From the COFF file header, it is easy to get all the offsets in the input object file for all section headers (c.f. figure 3). This step starts at processing each section header to record definitions into the SEGDEF section for the segments being added into the TXTFIX section, then converting the COFF section's raw data content, which is pointed by the field *s_scnptr* in the section header structure, into a segment in the TXTFIX section of the output object file. At the end of this step, process all the section's relocation entries whose first entry is pointed by *s_relptr* in the data structure for the section's header, and map them into the fixup blocks for the previous segment in the TXTFIX section. Remember to update the lengths of SEGDEF section and TXTFIX section, as well as the *num_segs* for *struct lmh*.

To define a segment in SEGDEF section, first increment *num_segs*, i.e. *num_segs++*, then define the associated segment in the SEGDEF data structure. Mainly, set *slimit=s_size-1*, and let *combine_name* record the segment's name provided by *s_name* in the COFF section's header. All other fields in the structure for SEGDEF temporary are set to zeros. Remember to update the length for SEGDEF section, i.e., *SEGDEF_len += sizeof (SEGDEF_num_segs).*

To convert the COFF section's raw data content into a segment in the TXTFIX section of Intel OMF object file, allocate storage for a structure of *txtfix* and let it be pointed by the field *next* of current *txtfix*. Set *blk_type=0,* because it is a text block to be added. Consequently, the *union block* goes to the *text_blk*. In the *struct text_blk*, set *txt_offset=0*, and the internal segment name *txt_IN* is determined by the segment's name. i.e., *combine_name* which was defined in the SEGDEF section (cf. Table 1). The field *length* of the *struct text* should record the length in bytes of the text content, therefore, it equals *s_size*.

Segment	*txt_IN*
CODE	0x2001
DATA	0x2002
MODULES	0x2004
SYMBOLS	0x2005
LINES	0x2007
SRCLINES	0x2008

Table 1. Internal name definition for segments in a TXTFIX section of an Intel OMF object file

If it is a CODE segment, then the *union segment* goes to *char *code*, let *code* point to a string copied from COFF section's raw data content.

The last stage of this step is designed for processing COFF section's relocation entries. In the COFF section header, it provides a pointer which points to the first relocation entry for the section, also it provides the total number of relocation entries in the section. Consequently, a loop is required to process each relocation entry in this section.

To convert a relocation entry into a fixup block for the previous segment in the TXTFIX section, allocate storage for *txtfix* and also let it be pointed by the field *next* of current s*truct txtfix*, set *blk_type=1*, since it is for the fixup block. Consequently, the *union block* goes to *fixup_blk*, allocate storage for *fixup_blk*, set the field *where_IN* to equal to *txt_IN*, the internal name for the target segment which requires patching by a linker to resolve the address reference for the symbol.

The field *r_symndx* in the *struct reloc* for a COFF relocation entry points to the symbol table entry which contains relocation information for the symbol. The *struct syment* for the COFF symbol table

entry tells the section where the symbol is defined, i.e. *n_scnum* specifies this host section. Back to the section's header as to get *s_size* which is required information for the field *length* in the *fixup_blk*. The symbol table entry uses *n_scnum, n_type* and *n_sclass* to define symbolic meaning for this symbol. Using the symbolic language dialect declared in the files *storeclass.h, a.out.h* and *syms.h* in the UNIX environment, */usr/include*, it is easy to interpret the type for the relocation entry as a general fixup, an intra-segment fixup or a calling fixup, then set the *union fixups* of the *fixup_blk* to point to its field accordingly. Basically, the *union fixups* requires information regarding *where_offset, what_IN* and *what_offset*.

As described in the section 2.1.1, *where_offset* in a fixup block is the byte-offset value in the target segment where the symbol is referenced, while *what_offset* is the byte-offset value in a host segment where the symbol is defined, and *what_IN* is the internal name for the host segment. Consequently, set *where_offset* to equal to *r_addr* in the *struct reloc* for a COFF relocation entry which is the byte-offset value of reference, set *what_offset* to equal to *n_value* in the *struct syment* for the COFF symbol table entry which is the byte-offset value in the host section for the starting byte of a code block which defines the symbol. Then, set *what_IN* to be *txt_IN*, the internal name for the host segment, if needed. Finally, update the length of TXTFIX section, i.e., *TXTFIX_len += sizeof(struct txtfix)*.

Step 2 is designed for mapping a subset of a symbol table and string table entries in a COFF relocation object file into sections for PUBDEF, EXTDEF and even for TYPDEF in a linkable Intel OMF object file. As described in the section 2.1.1, PUBDEF section defines all public names with their general address for public symbols defined in the current object module, while EXTDEF lists all external symbols to be referenced in the current object module. Both sections are critical for a linker to define and resolve the symbols reference across different object modules. The TYPDEF section is designed to record storage size and type interpretation by the compiler for the symbols defined in the current module. Also TYPDEF provides type information for type checking.

From the COFF file header, get the byte-offset value *f_symptr* of the first symbol table entry and total number of symbol table entries *f_nsyms* in the input COFF file. Start a loop on the symbol table to process each symbol table entry.

COFF symbol table entry uses *n_sclass* to define the storage class of its symbol, when *n_sclass* equals C_EXT, it represents that the current symbol is either a external symbol, if its section number field *n_scnum* equals N_UNDEF; or the current symbol is a public symbol, if *n_scnum* gives you a positive number to specify the section in the COFF file where the symbol is defined. From this point of view, a screening over the entire symbol table is needed. If the current symbol is a public symbol, i.e., *n_sclass == CEXT && n_scnum > 0,*. first increment *num_publics*, i.e., *num_publics++*. To define the general address for this public symbol, set *PUB_offset = n_value*, and set *PUB_segment* to be *txt_IN*, the internal name for the segment in TXTDEF section which is converted from the section specified by *n_scnum*. The field *n_type* in the *struct syment* provides the type declaration information of the symbol which should be converted into *type_IN*, an internal name for the type of this public symbol (cf. [1] & [5]). Finally, copy the string in the string table for the symbol's name pointed by *_n._n_offset* in *struct syment* into *sym_name* in the current PUBDEF section. Remember to update the length for PUBDEF, i.e., *PUBDEF_len += sizeof(PUBDEF_num_publics).*

If the symbol is an external symbol, i.e., *n_sclass == CEXT && n_scnum == N_UNDEF*, then *struct syment* should be converted into an EXTDEF section in the output object file. To do this, also first to increment *num_externals*, i.e., *num_externals++*, then to convert *n_type* in *struct syment* into *type_IN*, for defining the type of this external symbol. Also copy the string in the string table for the symbol's name pointed by *_n._n_offset* in *struct syment* into *sym_name* in the current EXTDEF section. Since it is not available for the information regarding segment having matched public symbol which is not in the current object module, we set *seg_IN* and *allocate_len* to be zeros, leave them to be resolved by a binder in a binding process. To complete building the EXTDEF section, we need to update the length for EXTDEF, i.e., *EXTDEF_len += sizeof(EXTDEF_num_externals)*.

To build TYPDEF section for the output object file at this step, we only need to record type information for both public and external symbols. The basic information which TYPDEF is inquiring is leaves' message which is classified as null leaf, string leaf, index leaf and numeric leaf. Especially for numeric leaves(c.f. [1]), it corresponds to the scalar type declaration for the symbol provided by

n_type and its auxiliary entries in *struct syment*. After building TYPDEF section, record its length, i.e., *TYPDEF_len = sizeof(TYPDEF)*.

Step 3 is to build segments LINES and SRCLINES in TXTFIX section of a linkable Intel OMF object file by processing all line number entries in a text section of a COFF relocatable object file. In the mean time, update SEGDEF section as to catch the definitions for the LINES and SRCLINES segments.

The segment LINES consists of offset values. Each offset is the byte offset of the start of a line in the code segment. The segment SRCLINES records information for the byte offset value for the start of each line in the source file. However, COFF line number entries only have offset values for the code segment provided by either *l_paddr* in a *lineno* entry data structure, or by *n_value* in *struct syment* pointed by *l_symndx* in *struct lineno*. It needs an extra tool to process a source file as to translate each *l_lnno* in *struct lineno* into byte-offset values of the source lines.

Before building LINES segment and SRCLINES segment, allocate storage structures for *lines* and *srclines*, also allocate memory space for a structure of *src_line* as a field of *struct srclines*. Set *src_file* in *struct srclines* to record the source file name and initialize *count* in *struct srcline* to be zero. Start from COFF file header to process each sections' header, if *s_nlnno > 0*, go to a component in COFF file which contains the section's lineno entries. Then start a loop to split each *lineno* entry into an offset record in LINES segment and a source line offset record for SRCLINES segment, i.e., for LINE segment, allocate storage for *lines* and let it be pointed by the field *next* of current *struct lines*, then set its field *offset = l_paddr*, if it is appropriate; for SRCLINES segment, increment *count*, i.e., *count++*, then process the source file to convert the *l_lnno* into a byte-offset value for the start of the line in the source file and let it be *offset* in *srcline*. At the end of each loop, update *TXTDEF_len*, i.e., *TXTDEF_len += sizeof(lines) +sizeof(srclines)*.

At the last stage of this step, we need to update SEGDEF section to define LINES and SRCLINES segments in the TXTFIX section. To do this, first to increment *num_segs*, i.e., *num_segs++*, let the associated SEGDEF section to record LINES segment, i.e., set *slimit = sizeof(lines)*, and *combine_name = "LINES"*, update the length for *SEGDEF*, i.e., *SEGDEF_len += sizeof(SEGDEF_num_segs)*. To record the definition for SRCLINES segment, again need to increment *num_segs*, also use current SEGDEF section SRCLINES segment, i.e., set *slimit = sizeof(srclines)*, and *combine_name = "SRCLINES"*, update the length for *SEGDEF*.

Step 4 is to build SYMBOLS segment in TXTFIX section from both COFF symbol table and COFF string table. In this step, we also need to build MODULES segment in TXTFIX section and then as usual to update SEGDEF to record the definitions for both segments added into TXTFIX section.

To record a block start entry for the SYMBOLS segment, we need both *.file* entry and its auxiliary entry in the COFF symbol table. At first, set the field *kind* in *struct sym* to be zero, since it is for a block start entry. The *union entry* then goes to *blk_start*, allocate memory space for *struct blk* to *blk_start*, set its field *offset* to be zero and *blk_len* to be the sum of *tsize*, *dsize* and *bsize* from the COFF optional header, then copy the *x_fname*, provided by *.file* auxiliary entry, into *blk_name*.

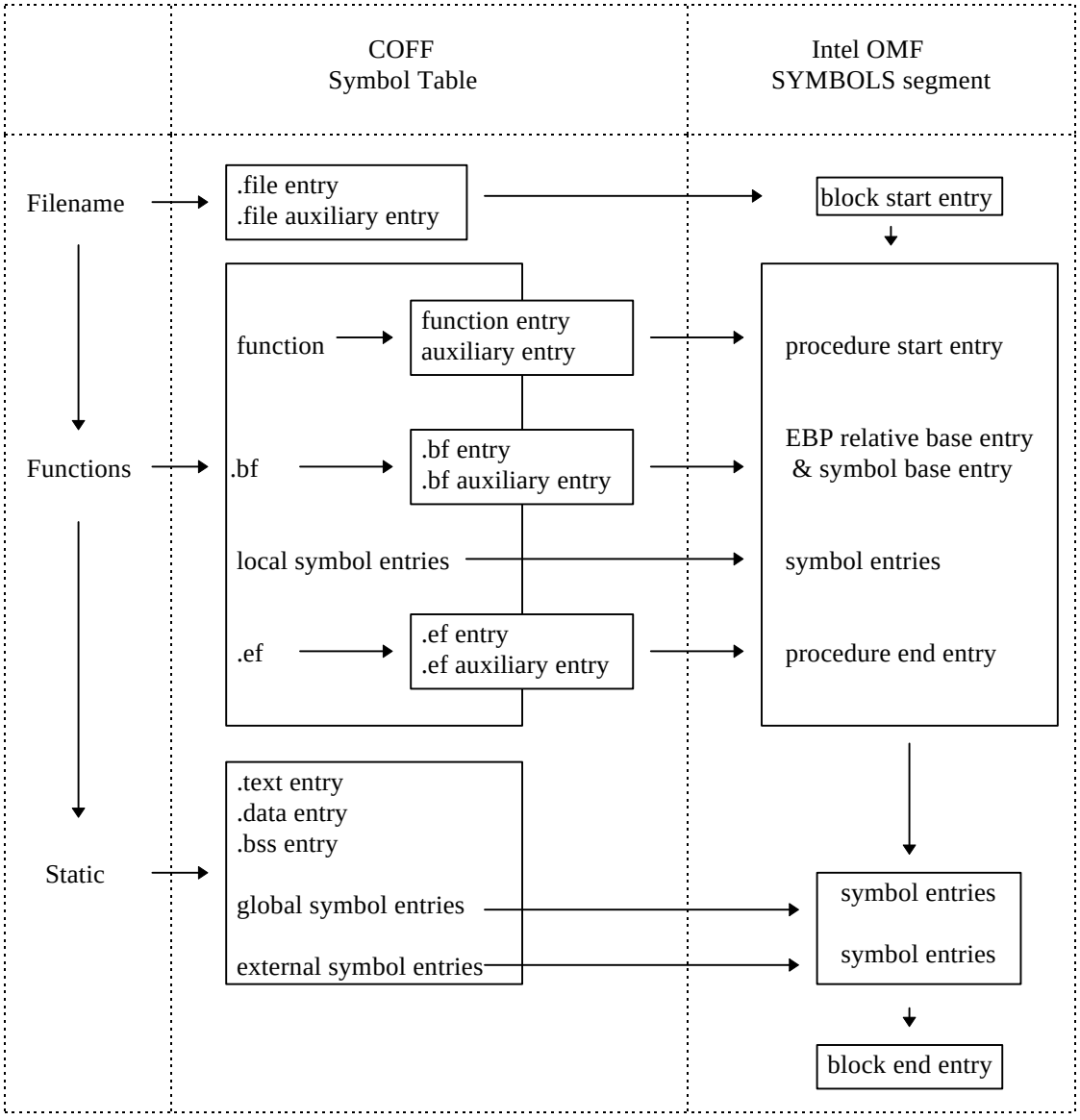

Figure 4 COFF symbol table sequence vs. Intel OMF SYMBOLS segment sequence & their correspondence relationship

To record a procedure start entry for the SYMBOLS segment, we need use function entry and its auxiliary entry from the COFF symbol table. Allocate storage for *struct sym* and let it be pointed by *next* of the current *sym*. Then set *kind = 1*, let the *union entry* points to *prc_start.* Allocate storage for *struct proc* and let it be pointed by *prc_start*, set *offset* to be *n_value* of the current *syment*, convert *n_type* of *syment* into *type_IN* for defining the internal name of the TYPDEF associated with the procedure, set *kind* to be 0x3 for specifying 32-bit near, set *ebp_offset* to be 0x4, also set *proc_len* to be *x_fsize* from the function's auxiliary entry, finally copy the string in a string table, pointed by *_n._n_offset* in *syment*, into *proc_name*.

The *.bf* entry and its auxiliary entry in a COFF symbol table should be converted into an EBP relative base entry and a symbol base entry. For an EBP relative base entry, simply set *struct sym* 's field *kind = 4*. However, we need to define *offset* and *s_IN* for the symbol base entry. Here, the field *offset* and *s_IN* in *struct sbase*, which is the data struct for a symbol base entry, gives a general address for the start byte of the current function. Therefore, set *s_IN = txt_IN*, the internal name for the text segment converted from the COFF section which *n_scnum* in *syment* specifies. Also set *offset* to be the byte-offset value in a COFF code section which can be converted from *x_lnno* in the *.bf* auxiliary entry.

All local symbol entries, global symbol entries and external symbol entries in a COFF symbol table should be converted into symbol entries in a SYMBOLS segment for an Intel OMF object file. To do this, allocate storage for *struct sym* and let it be pointed by *next* of the current *sym*, set its field *kind = 5*, since it is for a symbol entry. The *union entry* goes to *s_ent* which is declared as *struct symbol*. The field *offset* for a symbol entry should record the byte-offset value in the text segment where this symbol is defined, hence it equals *n_value* of *syment*. Similiar to the way we dealt with *type_IN*, we need to convert *n_type* provided in COFF *syment* into *type_IN* for defining an internal name of the TYPDEF associated with this symbol. Finally, copy the string in a COFF string table which contains the symbol's name into *sym_name*.

The procedure or block end entry in SYMBOLS segment is very simple, what we need to set is to let the field *kind* in *struct sym* equal to 0x2.

After building the SYMBOLS segment, we need to setup the MODULES segment in the TXTFIX section of the linkable Intel OMF file. From *struct mod*, it is quite clear that a MODULES segment inquires information on general addresses for segments or sections in this object module. We leave local descriptor table (LDT) selector and all the general addresses to be resolved during the binding and building processes. What we need to set at this stage is to set *kind =1*, (since it is for 386 architecture,) to set *first_line = 1*, to bring *trans_id* into correspondence with the translator which generates this object module(cf. [1]). Also set the *trans_vers* to record the translator's version number and the *mod_name* to record the module name.

The final stage of this step is to update SEGDEF to define the SYMBOLS and MODULES segments defined in the TXTFIX section, as well as to update the sizes for SEGDEF and TXTFIX sections.

Step 5 is a wrapping up step. In this step, we just need to update *struct lmh*, the linkable module header, and fix all fields in *struct toc_p1*, the table of contents for first partition. To build *struct toc_p1*, set *SEGDEF_len = 0xE4*, set both *GATDEF_loc* and *GATDEF_len* to be zeros, set *TYPDEF_loc* to be the sum of *SEGDEF_loc* and *SEGDEF_len*. Similarly, set PUBDEF_loc to be the sum of *TYPDEF_loc* and *TYPDEF_len*, *EXTDEF_loc* to be the sum of *TYPDEF_loc* and *TYPDEF_len*, and also set *TXTFIX_loc* to be the sum of *EXTDEF_loc* and *EXTDEF_len*. The rest of the fields in the *toc_p1* remain to be zeros. To update *struct lmh*, only we need to make sure is that the *tot_length* equals the total length of the entire current module, i.e., it is the sum of the length for *struct lmh*, *struct toc_p1* and all the lengths for the sections in this module.

At the final stage of the step, we need a tool to setup the byte for the checksum.

Acknowledgments

I would like to thank all the team members of the Embedded Software Technology group for the encouragement for writing this paper. Deepti Gupita, Sadhana Kapur, Joe Li, Kin Liang, Greg Mather, and Chak Sriprasad gave me many useful ideas for presenting this paper. Especially, I would like to thank Greg Mather for helping the justification for writing this paper.

I would especially like to acknowledge Kevin Priest, Chak Sriprasad and Joe Li who provided thorough content reviews that helped improve this paper significantly. I also recognize Jim Schrand for the discussion of defining an embedded system.

References

[1] Intel Corporation: "Specification 386™ Object Modules Format", 1990, 1991.

Order Number

482991-002

[2] Intel Corporation: “Type Checking, Guidelines for Constructing and Using Intel® Type Definition Records”, 1990, 1991, Order Number 482994-002

[3] Intel Corporation: “Simple Bootloadable Files in 386™ Object Modules Format”, 1990, 1991 Order Number 483164-001

[4] Larry Mittag: “Cross Debugging: ‘ Using Tools Other Than Your Nose’ “, Embedded System Programming Product, Fall, 1994

[5] Gintaras R. Gircys: “Understanding and Using COFF”, O’ Reilly & Associates, Inc., Nov., 1988

[6] TIS Committee, “TIS Formats Specification for Windows”, 1993, Order Number 241597-001

[7] TIS Committee, “Portable Formats Specification”, 1993, Order Number 241597-002

[8] Roland H. Pesch, “The GNU Binary Utilities”, May, 1993

[9]Jack G. Ganssle, “The Art of Programming Embedded Systems”, Academic Press, INC., 1992

[10] John Forrest Brow: “Embedded Systems Programming in C and Assembly”, VNR Computer Library, 1994

[11] Steven Armbrust and Ted Forgeron: “.OBJ lessons”, PC Tech. Journal, Vol. 3, No. 10, 1985

[12] Minda Zhang: “Dumping tool for Intel linkable OMF object files”, Project Report, 1994

[13] Minda Zhang: “Analysis of Object File Formats for Embedded Systems Using the Intel386™ Archtecture” APPLICATION NOTE AP-713 Intel Corporation, April 1995. Order # 272678-001