1. Introduction. This package is the library that can both be used by the compiler and the linker to create and interpret ELF artefacts. It is solely intended for use with `MMIX` programs.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mmix-arith64.h"
#include "mmix-elf64.h"
  ⟨ Global variables 9 ⟩
  ⟨ Subroutines 3 ⟩
```

2. ⟨ mmix-elf64.h 2 ⟩ ≡

```
#include <stdint.h>
  typedef uint64_t Elf64_Addr;
  typedef uint64_t Elf64_Off;
  typedef uint32_t Elf64_Word;
  typedef uint16_t Elf64_Half;
  typedef int32_t Elf64_Sword;
  typedef uint64_t Elf64_Xword;
  typedef int64_t Elf64_Sxword;
  typedef struct {
    unsigned char e_ident[16];
    Elf64_Half e_type;
    Elf64_Half e_machine;
    Elf64_Word e_version;
    Elf64_Addr e_entry;
    Elf64_Off e_phoff;
    Elf64_Off e_shoff;
    Elf64_Word e_flags;
    Elf64_Half e_ehsize;
    Elf64_Half e_phentsize;
    Elf64_Half e_phnum;
    Elf64_Half e_shentsize;
    Elf64_Half e_shnum;
    Elf64_Half e_shstrndx;
  } Elf64_Ehdr;
  typedef struct {
    Elf64_Word sh_name;
    Elf64_Word sh_type;
    Elf64_Xword sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off sh_offset;
    Elf64_Xword sh_size;
    Elf64_Word sh_link;
    Elf64_Word sh_info;
    Elf64_Xword sh_addralign;
    Elf64_Xword sh_entsize;
  } Elf64_Shdr;
  typedef struct {
    Elf64_Word p_type;
    Elf64_Word p_flags;
    Elf64_Off p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    Elf64_Xword p_filesz;
    Elf64_Xword p_memsz;
    Elf64_Xword p_align;
  } Elf64_Phdr;
  typedef struct {
```

```
  Elf64_Word st_name;
  unsigned char st_info;
  unsigned char st_other;
  Elf64_Half st_shndx;
  Elf64_Addr st_value;
  Elf64_Xword st_size;
} Elf64_Sym;
typedef struct {
  Elf64_Addr r_offset;
  Elf64_Xword r_info;
} Elf64_Rel;
typedef struct {
  Elf64_Addr r_offset;
  Elf64_Xword r_info;
  Elf64_Sxword r_addend;
} Elf64_Rela;
typedef struct {
  Elf64_Word n_strx;
  unsigned char n_type;
  unsigned char n_other;
  Elf64_Half n_desc;
  Elf64_Xword n_value;
} Elf64_Stab;
typedef struct sctstruct {
  struct sctstruct *next;
  char *name;
  int index;
  unsigned char *data;
  Elf64_Off ptr;
  Elf64_Off len;
  Elf64_Shdr *shdr;
} Elf64_Section;
typedef enum {
  ELFCLASS32 = 1, ELFCLASS64
} elf_cl;
typedef enum {
  ELFDATA2LSB = 1, ELFDATA2MSB
} elf_bo;
typedef enum {
  EM_MMIX = 80
} elf_mc;
typedef enum {
  ET_NONE = 0, ET_REL, ET_EXEC, ET_DYN, ET_CORE, ET_LOOS = #fe00, ET_HIOS = #feff,
      ET_LOPROC = #ff00, ET_HIPROC = #ffff
} elf_et;
typedef enum {
  SHF_WRITE = 1, SHF_ALLOC = 2, SHF_EXECINSTR = 4
} elf_shf;
typedef enum {
```

```
  SHT_NULL, SHT_PROGBITS, SHT_SYMTAB, SHT_STRTAB, SHT_RELA, SHT_HASH, SHT_DYNAMIC, SHT_NOTE,
      SHT_NOBITS, SHT_REL, SHT_SHLIB, SHT_DYNSYM, SHT_LOOS = #60000000, SHT_HIOS = #6fffffff,
      SHT_LOPROC = #70000000, SHT_HIPROC = #7fffffff
} elf_sht;
typedef enum {
  PT_NULL, PT_LOAD, PT_DYNAMIC, PT_INTERP, PT_NOTE, PT_SHLIB, PT_PHDR, PT_LOOS = #60000000,
      PT_HIOS = #6fffffff, PT_LOPROC = #70000000, PT_HIPROC = #7fffffff
} elf_pt;
typedef enum {
  STB_LOCAL, STB_GLOBAL, STB_WEAK, STB_LOOS = 10, STB_HIOS, STB_LOPROC, STB_HIPROC
} elf_stb;
typedef enum {
  STT_NOTYPE, STT_OBJECT, STT_FUNC, STT_SECTION, STT_FILE, STT_LOOS = 10, STT_HIOS, STT_LOPROC,
      STT_HIPROC
} elf_stt;
typedef enum {
  R_MMIX_A64 = 1, R_MMIX_XR, R_MMIX_YR, R_MMIX_ZR, R_MMIX_YZOFF, R_MMIX_XYZOFF
} elf_mr;
typedef enum {
  N_UNDF = 0, N_ABS, N_ABSE, N_TEXT, N_TEXTE, N_DATA, N_DATAE, N_BSS, N_BSSE, N_FUN = 36,
      N_SLINE = 68, N_DSLINE, N_BSLINE
} stab_symtype;
Elf64_Ehdr *buildEhdr( );
Elf64_Ehdr *readEhdr(Elf64_Ehdr *hdr, FILE *file);
void writeEhdr(Elf64_Ehdr *hdr, FILE *file);
Elf64_Shdr *buildShdr( );
Elf64_Shdr *readShdr(Elf64_Shdr *hdr, FILE *file);
void writeShdr(Elf64_Shdr *hdr, FILE *file);
Elf64_Phdr *buildPhdr( );
Elf64_Phdr *readPhdr(Elf64_Phdr *hdr, FILE *file);
void writePhdr(Elf64_Phdr *hdr, FILE *file);
Elf64_Section *buildSection(char *, Elf64_Word type, Elf64_Xword flags, Elf64_Addr addr, int,
    Elf64_Xword);
Elf64_Off setSym(Elf64_Section *, Elf64_Word, unsigned char, unsigned char, unsigned
    char, Elf64_Half, Elf64_Addr, Elf64_Xword);
Elf64_Off setRel(Elf64_Section *, Elf64_Addr, Elf64_Word, Elf64_Word);
Elf64_Off setRela(Elf64_Section *, Elf64_Addr, Elf64_Word, Elf64_Word, Elf64_Sxword);
Elf64_Off setStr(Elf64_Section *, char *str);
Elf64_Off plan(Elf64_Section *, int, bool);
Elf64_Off align(Elf64_Section *, int);
Elf64_Off store(Elf64_Section *, uint64_t, int, bool);
Elf64_Off setptr(Elf64_Section *, int);
Elf64_Off setStab(Elf64_Section *, Elf64_Word, unsigned char, unsigned char, Elf64_Half,
    Elf64_Xword);
```

3. Read or write a wyde value to file or write it to a buffer.

```
⟨ Subroutines 3 ⟩ ≡
  uint16_t read16(FILE *file)
  {
    uint16_t a;
    unsigned char buf[2];
    if (fread(&buf, 1, 2, file) ≠ 2) {
      fprintf(stderr, "Error reading data\n");
      exit(−1);
    }
    if (bigendian) a = (buf[0] ≪ 8) + buf[1];
    else a = (buf[1] ≪ 8) + buf[0];
    return a;
  }
  void write16(uint16_t a, FILE *file)
  {
    unsigned char buf[2];
    if (bigendian) buf[0] = (a ≫ 8) & #ff, buf[1] = a & #ff;
    else buf[0] = a & #ff, buf[1] = (a ≫ 8) & #ff;
    if (fwrite(&buf, 1, 2, file) ≠ 2) {
      fprintf(stderr, "Error writing data\n");
      exit(−1);
    }
  }
  void set16(void *s, uint16_t a)
  {
    unsigned char *buf = s;
    if (bigendian) buf[0] = (a ≫ 8) & #ff, buf[1] = a & #ff;
    else buf[0] = a & #ff, buf[1] = (a ≫ 8) & #ff;
  }
```

See also sections 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, and 25.

This code is used in section 1.

4. Read or write a tetra value to file or write it to a buffer.

```
⟨ Subroutines 3 ⟩ +≡
  uint32_t read32(FILE *file)
  {
    uint32_t a;
    unsigned char buf[4];
    if (fread(&buf, 1, 4, file) ≠ 4) {
      fprintf(stderr, "Error␣reading␣data\n");
      exit(−1);
    }
    if (bigendian) a = (buf[0] ≪ 24) + (buf[1] ≪ 16) + (buf[2] ≪ 8) + buf[3];
    else a = (buf[3] ≪ 24) + (buf[2] ≪ 16) + (buf[1] ≪ 8) + buf[0];
    return a;
  }
  void write32(uint32_t a, FILE *file)
  {
    unsigned char buf[4];
    if (bigendian)
      buf[0] = (a ≫ 24) & #ff, buf[1] = (a ≫ 16) & #ff, buf[2] = (a ≫ 8) & #ff, buf[3] = a & #ff;
    else buf[0] = a & #ff, buf[1] = (a ≫ 8) & #ff, buf[2] = (a ≫ 16) & #ff, buf[3] = (a ≫ 24) & #ff;
    if (fwrite(&buf, 1, 4, file) ≠ 4) {
      fprintf(stderr, "Error␣writing␣data\n");
      exit(−1);
    }
  }
  void set32(void *s, uint32_t a)
  {
    unsigned char *buf = s;
    if (bigendian)
      buf[0] = (a ≫ 24) & #ff, buf[1] = (a ≫ 16) & #ff, buf[2] = (a ≫ 8) & #ff, buf[3] = a & #ff;
    else buf[0] = a & #ff, buf[1] = (a ≫ 8) & #ff, buf[2] = (a ≫ 16) & #ff, buf[3] = (a ≫ 24) & #ff;
  }
```

5. Read or write an octa value to file or write it to a buffer.

```
⟨ Subroutines 3 ⟩ +≡
  uint64_t read64(FILE *file)
  {
    uint64_t a;
    unsigned char buf[8];
    if (fread(&buf, 1, 8, file) ≠ 8) {
      fprintf(stderr, "Error reading data\n");
      exit(−1);
    }
    if (bigendian)
      for (int i = 0; i < 8; i++) a ≪= 8, a += buf[i];
    else
      for (int i = 0; i < 8; i++) a ≪= 8, a += buf[8 − i];
    return a;
  }
  void write64(uint64_t a, FILE *file)
  {
    unsigned char buf[8];
    if (bigendian)
      for (int i = 0; i < 8; i++) buf[7 − i] = a & #ff, a ≫= 8;
    else
      for (int i = 0; i < 8; i++) buf[i] = a & #ff, a ≫= 8;
    if (fwrite(&buf, 1, 8, file) ≠ 8) {
      fprintf(stderr, "Error reading data\n");
      exit(−1);
    }
  }
  void set64(void *s, uint64_t a)
  {
    unsigned char *buf = s;
    if (bigendian)
      for (int i = 0; i < 8; i++) buf[7 − i] = a & #ff, a ≫= 8;
    else
      for (int i = 0; i < 8; i++) buf[i] = a & #ff, a ≫= 8;
  }
```

6. The ELF header. Each ELF file needs a header to indicate what type of file is be worked on. All ELF files should start with the sequence `#7f`, `'E'`, `'L'`, `'F'`. Since the `MMIX` architecture is a 64 bit big endian architecture, the class must be set to `ELFCLASS64` and the data encoding must be set to `ELFDATA2MSB`. The attribute *e_machine* is set to EM˙MMIX which is the code assigned to the `MMIX` architecture.

```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Ehdr *buildEhdr( )
  {
    Elf64_Ehdr *hdr = (Elf64_Ehdr *) calloc(1, sizeof(Elf64_Ehdr));
    if (hdr ≠ Λ) {
      hdr→e_ident[0] = #7f;
      hdr→e_ident[1] = 'E';
      hdr→e_ident[2] = 'L';
      hdr→e_ident[3] = 'F';
      hdr→e_ident[4] = ELFCLASS64;
      hdr→e_ident[5] = ELFDATA2MSB;
      hdr→e_ident[6] = 1;
      hdr→e_type = ET_NONE;
      hdr→e_machine = EM_MMIX;
      hdr→e_version = 1;
      hdr→e_ehsize = sizeof(Elf64_Ehdr);
    }
    return hdr;
  }
```

7.

```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Ehdr *readEhdr(Elf64_Ehdr *hdr, FILE *file)
  {
    if (¬hdr) hdr = (Elf64_Ehdr *) calloc(1, sizeof(Elf64_Ehdr));
    if (hdr ≠ Λ) {
      if (fread(hdr→e_ident, 1, 16, file) ≠ 16) {
        fprintf(stderr, "Error␣reading␣data\n");
        exit(−1);
      }
      hdr→e_type = read16(file);
      hdr→e_machine = read16(file);
      hdr→e_version = read32(file);
      hdr→e_entry = read64(file);
      hdr→e_phoff = read64(file);
      hdr→e_shoff = read64(file);
      hdr→e_flags = read32(file);
      hdr→e_ehsize = read16(file);
      hdr→e_phentsize = read16(file);
      hdr→e_phnum = read16(file);
      hdr→e_shentsize = read16(file);
      hdr→e_shnum = read16(file);
      hdr→e_shstrndx = read16(file);
      bigendian = (hdr→e_ident[5] ≡ ELFDATA2MSB);
    }
    return hdr;
  }
```

8. ⟨ Subroutines 3 ⟩ +≡

```
void writeEhdr(Elf64_Ehdr *hdr, FILE *file)
{
  fwrite(hdr→e_ident, 1, 16, file);
  write16(hdr→e_type, file);
  write16(hdr→e_machine, file);
  write32(hdr→e_version, file);
  write64(hdr→e_entry, file);
  write64(hdr→e_phoff, file);
  write64(hdr→e_shoff, file);
  write32(hdr→e_flags, file);
  write16(hdr→e_ehsize, file);
  write16(hdr→e_phentsize, file);
  write16(hdr→e_phnum, file);
  write16(hdr→e_shentsize, file);
  write16(hdr→e_shnum, file);
  write16(hdr→e_shstrndx, file);
}
```

9. ⟨ Global variables 9 ⟩ ≡

```
bool bigendian = true;
```

This code is used in section 1.

10. Section header. Object files have sections for different purposes. The linker combines the sections to create segments that can be loaded in storage. Each section needs a header to describe the section.

```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Shdr *buildShdr( )
  {
    Elf64_Shdr *hdr = (Elf64_Shdr *) calloc(1, sizeof(Elf64_Shdr));
    return hdr;
  }
```

11.
```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Shdr *readShdr(Elf64_Shdr *hdr, FILE *file)
  {
    if (¬hdr) hdr = (Elf64_Shdr *) calloc(1, sizeof(Elf64_Shdr));
    if (hdr) {
      hdr→sh_name = read32(file);
      hdr→sh_type = read32(file);
      hdr→sh_flags = read64(file);
      hdr→sh_addr = read64(file);
      hdr→sh_offset = read64(file);
      hdr→sh_size = read64(file);
      hdr→sh_link = read32(file);
      hdr→sh_info = read32(file);
      hdr→sh_addralign = read64(file);
      hdr→sh_entsize = read64(file);
    }
    return hdr;
  }
```

12.
```
⟨ Subroutines 3 ⟩ +≡
  void writeShdr(Elf64_Shdr *hdr, FILE *file)
  {
    write32(hdr→sh_name, file);
    write32(hdr→sh_type, file);
    write64(hdr→sh_flags, file);
    write64(hdr→sh_addr, file);
    write64(hdr→sh_offset, file);
    write64(hdr→sh_size, file);
    write32(hdr→sh_link, file);
    write32(hdr→sh_info, file);
    write64(hdr→sh_addralign, file);
    write64(hdr→sh_entsize, file);
  }
```

13. Program segment header. Executable files have segments that can be loaded in storage.

```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Phdr *buildPhdr( )
  {
    Elf64_Phdr *hdr = (Elf64_Phdr *) calloc(1, sizeof(Elf64_Phdr));
    return hdr;
  }
```

14.

```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Phdr *readPhdr(Elf64_Phdr *hdr, FILE *file)
  {
    if (¬hdr) hdr = (Elf64_Phdr *) calloc(1, sizeof(Elf64_Phdr));
    if (hdr) {
      hdr→p_type = read32(file);
      hdr→p_flags = read32(file);
      hdr→p_offset = read64(file);
      hdr→p_vaddr = read64(file);
      hdr→p_paddr = read64(file);
      hdr→p_filesz = read64(file);
      hdr→p_memsz = read64(file);
      hdr→p_align = read64(file);
    }
    return hdr;
  }
```

15.

```
⟨ Subroutines 3 ⟩ +≡
  void writePhdr(Elf64_Phdr *hdr, FILE *file)
  {
    write32(hdr→p_type, file);
    write32(hdr→p_flags, file);
    write64(hdr→p_offset, file);
    write64(hdr→p_vaddr, file);
    write64(hdr→p_paddr, file);
    write64(hdr→p_filesz, file);
    write64(hdr→p_memsz, file);
    write64(hdr→p_align, file);
  }
```

16. Miscellaneous. Object files and/or executable file may have sections in special formats containing tables with symbols, relocation information and more. These routines help to fill the sections properly.

```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Off setSym(Elf64_Section *s, Elf64_Word name, unsigned char i1, unsigned char
          i2, unsigned char other, Elf64_Half idx, Elf64_Addr val, Elf64_Xword size)
  {
    unsigned char *p = s→data + s→ptr;
    set32(p, name), p += 4;
    *p++ = (i1 ≪ 4) + i2;
    *p++ = other;
    set16(p, idx), p += 2;
    set64(p, val), p += 8;
    set64(p, size);
    s→ptr += sizeof(Elf64_Sym);
    return s→ptr;
  }
```

17.
```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Off setRel(Elf64_Section *s, Elf64_Addr offset, Elf64_Word idx, Elf64_Word rel)
  {
    Elf64_Xword info;
    unsigned char *p = s→data + s→ptr;
    set64(p, offset), p += 8;
    info = ((Elf64_Xword) idx ≪ 32) + rel;
    set64(p, info), p += 8;
    s→ptr += sizeof(Elf64_Rel);
    return s→ptr;
  }
```

18.
```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Off setRela(Elf64_Section *s, Elf64_Addr offset, Elf64_Word idx, Elf64_Word rel,
          Elf64_Sxword addend)
  {
    Elf64_Xword info;
    unsigned char *p = s→data + s→ptr;
    set64(p, offset), p += 8;
    info = ((Elf64_Xword) idx ≪ 32) + rel;
    set64(p, info), p += 8;
    set64(p, addend);
    s→ptr += sizeof(Elf64_Rela);
    return s→ptr;
  }
```

19.
```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Off setStr(Elf64_Section *s, char *str)
  {
    char *p = (char *) s→data + s→ptr;
    strcpy(p, str), s→ptr += strlen(str) + 1;
    return s→ptr;
  }
```

20. ⟨ Subroutines 3 ⟩ +≡

```
Elf64_Off setStab(Elf64_Section *s, Elf64_Word strx, unsigned char type, unsigned char
        other, Elf64_Half desc, Elf64_Xword value)
{
  unsigned char *p = s->data + s->ptr;
  set32(p, strx), p += 4;
  *p++ = type;
  *p++ = other;
  set16(p, desc), p += 2;
  set64(p, value);
  s->ptr += sizeof(Elf64_Stab);
  return s->ptr;
}
```

21. The routine plan is used to advance the pointer of a section without writing information. This is used to determine the required size of a section.

⟨ Subroutines 3 ⟩ +≡

```
Elf64_Off plan(Elf64_Section *s, int n, bool align)
{
  if (align) s->ptr = (s->ptr + n - 1) & ~(n - 1);
  s->ptr += n;
  return s->ptr;
}
```

22. Align adds extra space to a section to make sure the field will start at the required boundary.

⟨ Subroutines 3 ⟩ +≡

```
Elf64_Off align(Elf64_Section *s, int n)
{
  s->ptr = (s->ptr + n - 1) & ~(n - 1);
  return s->ptr;
}
```

23. Store writes the information in a section. The section must be planned and allocated before store can be used.

⟨ Subroutines 3 ⟩ +≡

```
Elf64_Off store(Elf64_Section *s, uint64_t d, int n, bool align)
{
  if (align) s->ptr = (s->ptr + n - 1) & ~(n - 1);
  switch (n) {
  case 1: s->data[s->ptr] = d & #ff;
    break;
  case 2: set16(s->data + s->ptr, d);
    break;
  case 4: set32(s->data + s->ptr, d);
    break;
  default: set64(s->data + s->ptr, d);
    break;
  }
  s->ptr += n;
  return s->ptr;
}
```

24. Set the pointer explicitly to a value.

```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Off setptr(Elf64_Section *s, int n)
  {
    s→ptr = n;
    return s→ptr;
  }
```

25.

```
⟨ Subroutines 3 ⟩ +≡
  Elf64_Section *buildSection(char *s, Elf64_Word type, Elf64_Xword flags, Elf64_Addr addr, int
        index, Elf64_Xword align)
  {
    Elf64_Section *sct = (Elf64_Section *) calloc(1, sizeof(Elf64_Section));
    if (sct) {
      sct→name = strdup(s);
      sct→shdr = buildShdr( );
      if (*s) {
        sct→index = index;
        sct→shdr→sh_type = type;
        sct→shdr→sh_flags = flags;
        sct→shdr→sh_addr = addr;
        sct→shdr→sh_addralign = align;
        if (type ≡ SHT_RELA) sct→shdr→sh_entsize = sizeof(Elf64_Rela);
        if (type ≡ SHT_SYMTAB) sct→shdr→sh_entsize = sizeof(Elf64_Sym);
      }
    }
    return sct;
  }
```

⟨ Global variables 9 ⟩ Used in section 1.
⟨ Subroutines 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25 ⟩ Used in section 1.
⟨ `mmix-elf64.h` 2 ⟩

MMIX-ELF64