ELF1 1E Symbol Table Section

Young W. Lim

2022-06-29 Wed

Outline

Based on

"Study of ELF loading and relocs", 1999
http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Compling 32-bit program on 64-bit gcc

- gcc -v
- gcc -m32 t.c
- sudo apt-get install gcc-multilib
- sudo apt-get install g++-multilib
- gcc-multilib
- g++-multilib
- gcc -m32
- objdump -m i386

uninitialized global variable (1)

gcc, in C mode:

- uninitialised globals which are not declared extern
  are treated as common symbols, not weak symbols.
- common symbols are merged at link time
  so that they all refer to the same storage;
  - if more than one object attempts to *initialise* such a symbol,
    you will get a *link-time error*
  - if they are not explicitly initialised anywhere,
    they will be placed in the BSS, i.e. initialised to 0.

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

uninitialized global variable (2)

gcc, in C++ mode:

- not the same as in c mode
- there is no common symbols in C++
- Uninitialised globals which are not declared extern
  are implicitly initialised to a default value
  (0 for simple types, or default constructor).

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

uninitialized global variable (3)

- In either case,
  a weak symbol allows an initialised symbol
  to be overridden by a non-weak initialised symbol
  of the same name at link time

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

uninitialized global variable (4)

1. init

```
int global = 999;

int main(void) { ... }
```

2. uninit

```
int global;

int main(void) { ... }
```

3. uninit, extern

```
extern int global;

int main(void) { ... }
```

4. init, weak

```
int global __attribute__((weak))
    = 999;

int main(void) { ... }
```

5. another init global

```
int global = 1234;
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

uninitialized global variable (5)

1. init

```
#include <stdio.h>
int global = 999;
int main(void) {
  printf("%d\n", global);
  return 0;
}
```

2. uninit

```
#include <stdio.h>
int global;
int main(void) {
  printf("%d\n", global);
  return 0;
}
```

3. uninit, extern

```
#include <stdio.h>
extern int global;
int main(void) {
  printf("%d\n", global);
  return 0;
}
```

4. init, weak

```
#include <stdio.h>
int global __attribute__((weak))
    = 999;
int main(void) {
  printf("%d\n", global);
  return 0;    }
```

5. another init global

```
int global = 1234;
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

Case 1 init

1. init

```
$ gcc -o test uninit.c && ./test
0

$ gcc -o test uninit.c another.c && ./test
1234
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

Case 2 uninit

2. uninit

```
$ gcc -o test init.c another.c && ./test
/tmp/cc5DQeaz.o:(.data+0x0): multiple definition of ‘global’
/tmp/ccgyz6rL.o:(.data+0x0): first defined here
collect2: ld returned 1 exit status
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

Case 3 uninit, extern

3. uninit, extern

```
$ gcc -o test uninit_extern.c && ./test
/tmp/ccqdYUIr.o: In function ‘main’:
main_uninit_extern.c:(.text+0x12): undefined reference to ‘global’
collect2: ld returned 1 exit status#+end_src

$ gcc -o test main_uninit_extern.c another_def.c && ./test
1234
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-

Case 4 init, weak

4. init, weak

```
$ gcc -o test init_weak.c && ./test
999

$ gcc -o test init_weak.c another.c && ./test
1234
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

Case 5 weak, strong

weak.c

```
#include <stdio.h>

int weak; /* global, weak, zero */

int main(void) {
  printf("weak value is %d.\n", weak);
  return 0;
}
```

strong.c

```
int weak = 42; /* global, strong, 42 */
```

running

```
$ gcc weak.c
$ ./a.out
weak value is 0.
$ gcc weak.c strong.c
$ ./a.out
weak value is 42.
```

Weak symbols (1)

- Weak symbol references that remain unresolved,
  do not result in a fatal error condition,
  no matter what output file type is being generated.
- If a static executable is being generated,
  the symbol is converted to an absolute symbol
  with an assigned value of zero

https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html

Weak symbols (2)

- If a dynamic executable or shared object is being produced, the symbol is left as an undefined weak reference with an assigned value of zero
- During process execution, the runtime linker searches for this symbol.
- If the runtime linker does not find a match, the reference is bound to an address of zero instead of generating a fatal relocation error.

https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html

Weak symbols (3)

- Historically, these undefined weak referenced symbols have been employed as a mechanism to test for the existence of functionality.
- For example, the following C code fragment might have been used in the shared object `libfoo.so.1`

`https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html`

Weak symbols (4)

weak symbols

```
#pragma weak    foo

extern  void    foo(char *);

void bar(char *path)
{
        void (*fptr)(char *);

        if ((fptr = foo) != 0)
                (*fptr)(path);
}
```

https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html

Weak symbols (5)

- When building an application that references `libfoo.so.1`,
  the link-edit completes successfully
  regardless of whether a definition for the symbol `foo` is found
- If during execution of the application
  the function address tests nonzero,
  the function is called.
- However, if the symbol definition is not found,
  the function address tests zero
  and therefore is not called.

`https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html`

Weak symbols (6)

- Compilation systems view this address comparison technique as having undefined semantics, which can result in the test statement being *removed* under optimization.
- In addition, the runtime symbol binding mechanism places other *restrictions* on the use of this technique.
- These *restrictions prevent* a consistent model from being made available for *all* dynamic objects

`https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html`

Common symbols (1)

- common symbols allow a programmer
  to *define* several variables
  of the same name in different source files
- the other way is to define a variable
  once in *one* source file, and
  reference it everywhere else
  in *other* source files,
  using `extern`

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

Common symbols (2)

- when common symbols are used,
  the linker will merge all symbols of the same name
  into a single memory location
- the size of which is the largest type of
  the individual common symbol definitions.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

Common symbols (3)

- `fileA.c` defines an uninitialized 32-bit integer `myint`
- `fileB.c` defines an 8-bit character `myint`,
- then in the final executable,
  references to `myint` from both files
  will point to the same memory location (common location),
  and the linker will reserve 32 bits for that location.

`https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml`

Common symbols (4)

- COMMON symbols are contained
  only in relocatable object files,
  not in executable object files.
- they are generated by the compiler / assembler
  when creating an object file from a single source file.
- later, the linker will need to interpret these symbols.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

Common symbols (5)

- Remember that ELF reserves
  a special section header table index
  for referring to a COMMON section:
- the index COM
  - just like the special indices ABS and UND,
  - these sections do not physically exist in the file
- common symbols defined in the symbol table
  of relocatable object files
  have their section index member set to COM

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

Common symbols (6)

- common symbols first appeared
  as a feature of the FORTRAN language.
- common symbols are present only
  for backward-compatibility
  with *old* source files
  where `extern` is not used
- nowadays, the best practice is to make use of
  only one definition of a variable,
  and use `extern` in all *other source files*
  that reference it

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

Common symbols (7)

- if a global variable is not initialized
  in a C source file,
- after compiling, we would expect the variable to go to
  the .bss section in the relocatable object file
- However, *by default*, GCC will put the symbol
  in the COMMON section of the file;
- that is, the option -fcommon is the default behaviour.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(1a) COMMON

- *by default*, the uninitialized variable `un_a`
  is put in the common section.
  `int un_a` → `COM un_a` (common)
- when compile with `-fno-common`,
  the section of `un_a` is now at index 3,
  which is the `.bss` section
  of the `main.o` relocatable object file
  `int un_a, -fno-common` → `3 un_a` (`.bss`)

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(1b) COMMON

main.c

```
int un_a;      // common

int main() {
     return 0;
}
```

script

```
gcc -c -o main.o main.c
readelf -s main.o

gcc -c -o main.o main.c -fno-common
readelf -s main.o
```

results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a
$> gcc -c -o main.o main.c -fno-common
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(2a) .data

- if the variable is initialized to a certain value,
  then it is placed in the .data section in the output file
  by the compiler (actually the assembler)
- note that section index 2 corresponds
  to the .data section here:
  int un_a=9 → 2 un_a (.data)
- use -SW options to readelf to list all sections

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(2b) .data

main.c

```
int un_a = 9;      // .data

int main() {
     return 0;
}
```

script

```
$> gcc -c -o main.o main.c

$> readelf -s main.o
```

results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3a) .bss

- If we define a global variable,
  and explicitly initialise it to zero,
  then it will be put in the .bss section
  int un_a=0 → 2 un_a (.bss)
- although it is initialized and
  logically should go into .data,
  the compiler knows it is optimal to put it in the .bss,
  as in any case it will become initialized to zero at runtime,
  and in .bss will not consume *file space*

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3b) .bss

main.c

```
int un_a = 0;      // .bss

int main() {
     return 0;
}
```

script

```
$> gcc -c -o main.o main.c

$> readelf -s main.o
```

results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

Two object files defining the same symbol

main.c		swap.c		prog
int un_a	COM	int un_a=0	.bss	.bss
int un_a	COM	extern int un_a	UND	.bss
int un_a	COM	int un_a	COM	.bss
int un_a=0	.bss	int un_a=9	.data	multiple definition
int un_a=10	.data	extern int un_a	UND	.data

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(1a) COMMON, .bss

1. int un_a;, int un_a=0; (COMMON, .bss)

main.c

```
int un_a;      // common

int main() {
     return 0;
}
```

swap.c

```
int un_a=0;    // .bss

int swap() {
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(1b) COMMON, .bss

1. int un_a;, int un_a=0; (COMMON, .bss)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 un_a

$> gcc -o prog main.o swap.o
$> readelf -s prog | grep 'un_a'
    49: 0000000000601030     4 OBJECT  GLOBAL DEFAULT   25 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(1c) COMMON, .bss

1. `int un_a;`, `int un_a=0;` (COMMON, .bss)
   - the linker creates the final variable `un_a`
     in the `.bss` section of the executable object file
     (index 25 is the index of `.bss` as shown by `readelf -SW`)
   - In the file `swap.c`, `un_a` were initialised with a non-zero value,
     the variable would have been in the `.data` of the relocatable `swap.o`,
     and the linker would have then placed it
     in the `.data` section of the executable.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(2a) COMMON, undefined

2. int un_a;, extern int un_a; (COMMON, undefined)

main.c

```
int un_a;

int main() {

     return 0;
}
```

swap.c

```
extern int un_a;

int swap() {
     int a = un_a;
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(2b) COMMON, undefined

2. int un_a;, extern int un_a; (COMMON, undefined)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND un_a

$> gcc -o prog main.o swap.o
$> readelf -s prog | grep 'un_a'
   49: 0000000000601030     4 OBJECT  GLOBAL DEFAULT   25 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(2c) COMMON, undefined

2. `int un_a;`, `extern int un_a;` (COMMON, undefined)
   - As we see, in the final executable, our variable is located in the `.bss` section.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3a) COMMON, COMMON

3. int un_a;, int un_a; (COMMON, COMMON)

main.c

```
int un_a;

int main() {
     return 0;
}
```

swap.c

```
int un_a;

int swap() {
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3b) COMMON, COMMON

3. int un_a;, int un_a; (COMMON, COMMON)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a

$> gcc -o prog main.o swap.o
$> readelf -s prog | grep 'un_a'
    49: 0000000000601030     4 OBJECT  GLOBAL DEFAULT   25 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3c) COMMON, COMMON

3. `int un_a;`, `int un_a;` (COMMON, COMMON)
   - We see here also, the variable is located in `.bss` of final executable.

`https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb`

(4a) .bss, .data

4. int un_a=0;, int un_a=9; (.bss, .data)

main.c

```
int un_a=0;

int main() {
     return 0;
}
```

swap.c

```
int un_a=9;

int swap() {
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(4b) .bss, .data

4. int un_a=0;, int un_a=9; (.bss, .data)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 un_a
$> gcc -o prog main.o swap.o
swap.o:(.data+0x0): multiple definition of `un_a'
main.o:(.bss+0x0): first defined here
collect2: error: ld returned 1 exit status
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(4c) .bss, .data

4. `int un_a=0;`, `int un_a=9;` (`.bss`, `.data`)
   - a linking error.
     The codes from both files each reference
     their own data location, initialized differently.
     - if it takes the value 9 and puts it in the `.data` section
       for symbol `un_a`, then the code in `main.c` may behave *incorrectly*
       as it was written assuming the initial value of `un_a` to be `0`.
     - if it were to choose to put `un_a` in `.bss`,
       such that at runtime the initial value of `un_a` will be `0`,
       the code in `main.c` will work correctly
       but code from `swap.c` will likely not function well.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(5a) .data, undefined

5. int un_a=10;, extern int un_a; (.data, undefined)

main.c

```
int un_a=10;      // .data

int main() {
     return 0;
}
```

swap.c

```
extern int un_a;    // undefined

int swap() {
     int a = un_a;
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(5b) .data, undefined

5. int un_a=10;, extern int un_a; (.data, undefined)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND un_a

$> gcc -o prog main.o swap.o
$> readelf -s prog | grep 'un_a'
    49: 000000000060102c     4 OBJECT  GLOBAL DEFAULT   24 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(5c) .data, undefined

5. `int un_a=10;`, `extern int un_a;` (.data, undefined)
   this example shows a normal and common case.
   We see that finally the variable is defined
   in the .data of the executable (section index 24).

`https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml`

TOC: Symbol table section

- uninitialized global variables
- Symbol table
- Global and weak symbols

TOC: Symbol table section

Symbol table (1)

- An object file's symbol table holds
  information needed to *locate* and *relocate*
  a program's symbolic definitions and references
- A symbol table index is
  a subscript into this array.
- Index 0 both designates
  the first entry in the table and
  serves as the undefined symbol index

`https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html`

Symbol table (2)

- the first byte is index zero,
  holds a null character (\0)
- the last byte holds a null character (\0)
  ensuring null termination for all strings.
- A string with zero index specifies
  either no name or a null name,
  depending on the context.

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

Elf32_Sym structure type

```
typedef struct {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half    st_shndx;
} Elf32_Sym;
```

- st_name :
  An index into the object file's symbol string table

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

Elf32_Sym field types (1) st_name, st_value

- st_name
  - an index into the object file's symbol string table, which holds the character representations of the symbol names.
  - if the value is nonzero, the value represents a string table index that gives the symbol name.
  - otherwise, the symbol table entry has no name.
- st_value
  - the value of the associated symbol.
  - the value can be an absolute value or an address, depending on the context. See Symbol Values.

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

Elf32_Sym fields type (2) st_size, st_info

- st_size
  - Many symbols have associated sizes.
  - For example, a data object's size is the number of bytes that are contained in the object.
  - This member holds the value zero if the symbol has no size or an unknown size.
- st_info
  - The symbol's type and binding attributes.
  - A list of the values and meanings appears in Table

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

Elf32_Sym fiedls type (3) st_shndx

- st_shndx
  - every symbol table entry is defined in relation to some section
  - st_shndx member holds the relevant section header table index
- Some section indexes indicate special meanings
  - If this member contains SHN_XINDEX,
    then the actual section header index is
    too large to fit in this member.
  - The actual value is contained in the associated section
    of type SHT_SYMTAB_SHNDX

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

Elf32_Sym fiedls type (4) st_other

- st_other
  - A symbol's visibility
  - Other bits are set to zero, and have no defined meaning.
- symbol binding

STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOOS	10
STB_HIOS	12
STB_LOPROC	13
STB_HIPROC	15

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

ELF Symbol binding (1)

- STB_LOCAL: Local symbol.
  - These symbols are not visible outside the object file containing their definition.
  - Local symbols of the same name can exist in multiple files without interfering with each other.
- STB_GLOBAL: Global symbols.
  - These symbols are visible to all object files being combined.
  - One file's definition of a global symbol satisfies another file's undefined reference to the same global symbol.
- STB_WEAK: Weak symbols.
  - These symbols resemble global symbols, but their definitions have lower precedence.

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

ELF Symbol binding (2)

- STB_LOOS - STB_HIOS
  - Values in this inclusive range are reserved for operating system-specific semantics.
- STB_LOPROC - STB_HIPROC
  - Values in this inclusive range are reserved for processor-specific semantics.

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

Global and weak symbols (1)

- When the link-editor combines several relocatable object files, it does not allow *multiple definitions* of STB_GLOBAL symbols with the same name.
- On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name will not cause an error
- The link-editor honors the global definition and ignores the weak ones.

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

Global and weak symbols (2)

- Similarly, if a common symbol exists,
  the appearance of a weak symbol with the same name
  does not cause an error
- The link-editor uses the common definition
  and ignores the weak one.
- A common symbol has the `st_shndx` field holding `SHN_COMMON`

`https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html`

Global and weak symbols (3)

- When the link-editor searches archive libraries
  it extracts archive members that contain
  definitions of undefined or tentative global symbols.
- The member's definition can be either a global or a weak symbol.
- The link-editor, by default, does not extract archive members
  to resolve undefined weak symbols.
- Unresolved weak symbols have a zero value.
- The use of `-z weakextract` overrides this default behavior.
- It enables weak references to cause the extraction of archive members.

`https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html`

Section of type SHT_SYMTAB, SHT_DYNSYM (1)

sh_type = SHT_SYMTAB, SHT_DYNSYM

- identifies a symbol table
- typically a SHT_SYMTAB section
  provides symbols for link-editing
- as a complete symbol table, it can contain
  many symbols unnecessary for dynamic linking
- Consequently, an object file can also contain
  a SHT_DYNSYM section, which holds
  a minimal set of dynamic linking symbols,
  to save space

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

Section of type SHT_SYMTAB, SHT_DYNSYM (2)

sh_type = SHT_SYMTAB, SHT_DYNSYM

- sh_link
  - The section header index of the associated string table
- sh_info
  - One greater than the symbol table index of the last local symbol (binding STB_LOCAL) .

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html

the section header index of the associated symbol table

sh_type =

- SHT_HASH
- SHT_REL, SHT_RELA
- SHT_GROUP
- in these sections, sh_link represents
  the section header index of the associated symbol table

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblh/index.html