ELF Symbol Meta-Information

Developed by Todd Snider (Texas Instruments)
in consultation with Jozef Lawrynowicz

Written by Jozef Lawrynowicz

August 2020

Contents

1 Introduction

Here we propose a new mechanism for describing additional information about ELF symbols, called Symbol Meta-Information.

Symbol Meta-Information is intended to solve the problem of how the compiler or assembler can communicate information about symbols, not supported by existing ELF constructs, to downstream tools such as the linker and other consumers of ELF files. These consumers can then change how they handle the symbols, based on the supplementary information.

A new ELF special section named `.symtab_meta` enumerates which symbols have meta-information, the type of meta-information, and the associated value of that meta-information.

The use of attributes set on symbol declarations in the source code provides the programmer with a simple interface to the new functionality.

Symbol meta-information is designed to be extensible, with plenty of room for new types of meta-information to be added, and flexible, as the value of the meta-information can take on any format.

2 Background

2.1 Motivation

The modular nature of toolchain components means that communicating information from the source code through the build process to downstream tools is not always straightforward. Of course, this is partly why formats like ELF exist, but when those formats are reaching the limit of information that is able to be precisely described by them, programmers search for alternative solutions.

Placing code and data into special named sections is the most common method used to make the linker handle specific symbols in some non-standard way. A modified linker script with knowledge of these special sections can then be used to apply specific properties to the sections, such as saving them from garbage collection or placing them at specific memory addresses.

However, it can be inconvenient for programmers to modify linker scripts:

- Entire applications can be written without consideration for the linker script, its existence perhaps acknowledged by the programmer but otherwise being an opaque part of the build process. The programmer may therefore lack knowledge of the syntax of the linker script, or the ability to leverage the full breadth of functionality available to achieve what they want.

- In the context of embedded microcontrollers, linker scripts provided by semiconductor manufacturers are usually specific to a particular device, describing a unique combination of the memory map, peripheral register addresses, vector table etc.

  - Modifying linker scripts can therefore be bothersome when an application targets different devices, each with a unique linker script, or when linker script updates from semiconductor manufacturers require merging of downstream and upstream changes.

- Linker scripts can have a large amount of boilerplate code, and modifications to this boilerplate, as a side-effect to the handling of any new special sections, can be error-prone.

Another way to supply additional information about a symbol is to give the symbol itself a special name. This requires the ELF file consumer program to have knowledge of the special name, and may not be desirable if it interferes with the way the symbol would be handled if it had its original name. Furthermore, since there is no opportunity in the gABI for the standardization of special names for code and data symbols to have some unique meaning, there is likely to be inconsistencies between processor and vendor support for any toolchains trying to make use of this mechanism.

2.2 Alternative vehicles for symbol meta-information implementation

We acknowledge some existing constructs which could be used to supply additional information about ELF symbols, and describe why they are unsuitable vehicles for the proposed symbol meta-information functionality.

New symbol types or bindings

- If a type of symbol meta-information implied only one existing symbol type or binding attribute, then the meta-information type could be implemented as a new type or binding. However, since the proposed symbol meta-information types support symbols with different types and different bindings, this approach would not work.

- There are only 3 remaining "slots" for generic symbol types and it is desirable to have more than 3 new types of symbol meta-information. There are further reserved ranges for operating system-specific and processor-specific types, but it would not be appropriate to use these for new types which have generic use.

- Fundamentally, symbol meta-information supplies additional information about symbols, and does not change the intrinsic type or binding of a symbol.

`st_other` member of symbol table entry

- `st_other` is only 8 bits in size and is used as a bit-mask. Bits 0 and 1 are reserved, with an additional proposal currently pending to reserve bit 2 as well. The remaining bits 3-7 have not been officially reserved but are all in use by a variety of targets. Therefore, there are no remaining bits which can be used without creating a conflict with some target or operating system.

- There is no standard way to provide supplemental information which gives a non-boolean value for the `st_other` field. Further modifications, such as the creation of a special section, would be required to provide non-boolean values to accompany the `st_other` value.

Solaris SymInfo

- Solaris SymInfo specifically targets dynamic symbols, and the proposed functionality should be available to targets which do not support the concept of dynamic linking. SymInfo "types" are flags that can be augmented by extracting a value from the `.dynamic` section.
  - The `.dynamic` section is identified by the `sh_info` field of the section header, and could arguably be repurposed to point to some other section in cases when there are no dynamic symbols with SymInfo entries. However, this behavior would not be well defined when there is also a `.dynamic` section in the file.
- The `si_flags` field, which describes the properties of the associated symbol, is the size of a half-word. On a target implementing 32-bit ELF, this would be 16-bits. Since the flags are implemented as a bit-mask with 10 types already implemented, there only remains space for 6 further types. This is unlikely to be enough room for all current and future meta-information types, especially once factoring in any additional vendor or processor-specific extensions.

New ABI-mandated "Special Sections"

- A new type of ELF "special section" could be created for each of the proposed new types of symbol meta-information. ELF file consumers such as the linker would then handle these sections in a specific way, without assistance from the linker script. However, this has some downsides:
  - The user may not want to put a symbol in it's own section just to make use of the desired functionality.
  - A special section for the symbol obscures the fact that the meta-information is for a symbol, not a section.
  - If the `sh_info` member is used to provide an accompanying value for the meta-information type, then only one value can be specified per section, meaning symbols with the same type might not be able to be grouped together in a section.
  - An application making use of a large amount of new special sections to describe symbol meta-information could pollute the section header table.

3 Design

Abbreviations

metasym Any type of meta-information symbol

locsym A meta-information symbol with type `SMT_LOCATION`

3.1 Symbol Meta-Information Table

ELF relocatable and executable files may contain a new section named `.symtab_meta`. This section can be omitted from ELF files if there is no meta-information for any symbols, but if present, there can only be one section with this name and type.

Table 1: Section types, `sh_type`

Name	Value
`SHT_SYMTAB_META`	`19`

Table 2: `sh_link` and `sh_info` interpretation

Name	sh_link	sh_info
`SHT_SYMTAB_META`	The section header index of the associated symbol table.	The format version number of the symbol meta-information table (`ELFxx_SMH_VER`), and the section header index of the `.strtab_meta` string table used by entries in this section (`ELFxx_SMH_STR`).

(a) Accessors for the `sh_info` field

```
#define ELF32_SMH_STR(i)    ((i)>>8)
#define ELF32_SMH_VER(i)    ((unsigned char)(i))
#define ELF32_SMH_INFO(s,v) (((s)<<8)+(unsigned char)(v))

#define ELF64_SMH_STR(i)    ((i)>>32)
#define ELF64_SMH_VER(i)    ((i)&0xffffffffL)
#define ELF64_SMH_INFO(s,v) (((s)<<32)+((v)&0xffffffffL))
```

(b) `.symtab_meta` versions

Value	Meaning
0	Invalid Version
1	There is no header at the beginning of `.symtab_meta`.
2	A header containing the hash of `.symtab` is at the beginning of `.symtab_meta`.

Table 3: Special Sections

Name	Type	Attributes
`.symtab_meta`	`SHT_SYMTAB_META`	None
`.strtab_meta`	`SHT_STRTAB`	None

Version 2 of the table has a short header, and a list of symbol meta-information entries follows.

symtab_hash For version $>=$ 2, a 20-byte SHA-1 hash of the entire contents of `.symtab` (taken once the symbol table indices have been finalized) is

Figure 1: Structure of the `.symtab_meta` header

```
typedef struct {
  unsigned char symtab_hash[20];
} Elf32_SMhdr;

typedef struct {
  unsigned char symtab_hash[20];
} Elf64_SMhdr;
```

used to verify `.symtab` has not been modified by tools which do not recognize `.symtab_meta`. These tools would not update the symbol index stored in the symbol meta-information table entry when making changes to the program, possibly corrupting the state of `.symtab_meta`.

3.2 Symbol Meta-Information Table Entries

Symbol meta-information table entries describe the symbol that the meta-information applies to, the type of meta-information, and the associated value of the meta-information.

The format of symbol meta-information table entries is physically identical to ELF `Rel` relocation entries. The `smi_info` field encodes the symbol table index of the corresponding symbol and the type of meta-information in the same way that the symbol table index and type of a relocation are encoded in the `r_info` field of relocation entries.

Figure 2: Structure of a `.symtab_meta` entry

```
typedef struct {
  Elf32_Addr smi_info;
  Elf32_Word smi_value;
} Elf32_SymMetaInfo;

typedef struct {
  Elf64_Addr  smi_info;
  Elf64_Xword smi_value;
} Elf64_SymMetaInfo;
```

smi_info This field describes both the symbol table index of the ELF symbol this symbol meta-information this applies to, and the type of meta-information entry this is. A number of generic types are pre-defined. There are also reserved ranges for processor-specific and application-specific (i.e. vendor-specific) types.

Figure 3: Accessors for the `smi_info` field

```
#define ELF32_SMI_SYM(i)    ((i)>>8)
#define ELF32_SMI_TYPE(i)   ((unsigned char)(i))
#define ELF32_SMI_INFO(s,t) (((s)<<8)+(unsigned char)(t))

#define ELF64_SMI_SYM(i)    ((i)>>32)
#define ELF64_SMI_TYPE(i)   ((i)&0xffffffffL)
#define ELF64_SMI_INFO(s,t) (((s)<<32)+((t)&0xffffffffL))
```

smi_value The interpretation depends on the associated type. The value could be interpreted as a boolean, symbol table index, address, string table index etc.

Figure 4: Symbol Meta-Information Types

Value	Type	Format of Value
0	`SMT_NONE`	None
1	`SMT_RETAIN`	Boolean
2	`SMT_LOCATION`	Address
3	`SMT_NOINIT`	Boolean
4	`SMT_PRINTF_FMT`	Integer
0xC0	`SMT_LOPROC`	Processor-specific
0xDF	`SMT_HIPROC`	
0xE0	`SMT_LOUSER`	Vendor-specific
0xFF	`SMT_HIUSER`	

SMT_NONE This indicates an invalid or incomplete entry.

SMT_RETAIN A value of 1 indicates the associated symbol should be retained in the output executable file, even it appears unused and so the linker would normally garbage collect it. Other values result in the type being ignored.

SMT_LOCATION The VMA of the associated symbol in the output executable file should be set to the specified the value.

SMT_NOINIT A value of 1 indicates the associated data symbol should not be initialized by the runtime support code at program startup. Other values result in the type being ignored.

SMT_PRINTF_FMT The value indicates a byte offset into the `.strtab_meta` section. The section header table index of `.strtab_meta` is extracted from the `sh_info` value of `.symtab_meta`, using the `ELFxx_SMH_STR` accessor. The null-terminated string extracted from the string table is a de-duplicated

list of format specifiers used by calls to `printf`-like functions, in the function whose symbol is pointed to by this entry.
For example, the following C code:
`printf (“%d / %d = %f\n”, ...);`
would generate the following string in `.strtab_meta`:
“`%d%f`”.

SMT_LOPROC..SMT_HIPROC Values in this range are reserved for processor-specific semantics.

SMT_LOUSER..SMT_HIUSER Values in this range are reserved for vendor-specific semantics.

3.3 Symbol Meta-Information Values

3.3.1 Restrictions on applying symbol meta-information types to symbols

Symbol meta-information entries are always tied to a symbol in the symbol table, so there are no special rules regarding different symbols with the same name; the standard symbol binding rules apply.

No two entries in `.symtab_meta` can have the same `smi_info` value - each symbol must only have one value for a given meta-information type.

Figure 5: Symbol bindings and types permitted for metasyms

Symbol Meta-Information Type	Permitted Symbol Binding	Permitted Symbol Type
`SMT_RETAIN`	`Any < STB_LOOS`	`STT_FUNC` or `STT_OBJECT` or `STT_COMMON`
`SMT_LOCATION`	`Any < STB_LOOS`	`STT_FUNC` or `STT_OBJECT` or `STT_COMMON`
`SMT_NOINIT`	`Any < STB_LOOS`	`STT_OBJECT` or `STT_COMMON`
`SMT_PRINTF_FMT`	`Any < STB_LOOS`	`STT_FUNC`

3.3.2 SMT_NOINIT use case

When a piece of data is not initialized to a constant value, but does not need to be zero-initialized, `SMT_NOINIT` indicates that it can be skipped by runtime startup code that would normally initialize it, to save time when starting the program.

Alternatively, when a piece of data is initialized to a constant value when the program is loaded, but should not be re-initialized when the processor resets, `SMT_NOINIT` can also be applied.

3.3.3 SMT_PRINTF_FMT use case

When the size of an application is a concern to the programmer, limiting the format specifiers supported by `printf`-like functions can reduce the code and data usage of these functions in the application.

By storing the required format specifiers in the symbol meta-information table, the linker can examine each of the `SMT_PRINTF_FMT` entries for functions that will be used in the final linked executable, and link in the minimal implementation of the `printf` function required to support all the format specifiers used by the application.

3.3.4 Considerations for placement of SMT_LOCATION meta-information symbols (locsyms)

Locsyms are intended to augment a well-defined linker script. The linker validates the address provided for the locsym by examining the permissions of the segment (`p_flags`) which contains the specified VMA. For example, the linker must ensure that a locsym for a read/write symbol with type `STT_OBJECT` is not placed in a segment without write (`PF_W`) permissions, and emit an error if the segment containing the address is invalid.

The linker may need to place the input section of a locsym within an output section, within which it would not normally be placed. For example, consider an application with a large `.text` output section, which spans most of ROM. If a locsym corresponding to a piece of read-only data has an address within range of that `.text` section, and there is no way to offset the `.text` section within ROM such that the read-only data can be placed directly at the location, that read-only data can be placed amongst the `.text` input sections at the requested address. As long as the output section flags are not changed by adding the new input section, there should not be any problems mixing sections in this way.

3.3.5 Initialization of locsyms at program startup

Data which requires initialization at program startup (e.g. copying data from their LMA to VMA) has long been handled by the associated runtime library. When all data requiring initialization is within a range of addresses defined by known `__*_start` and `__*_end` symbols, only a fixed number of target-dependent initialization functions need to be run. However, when code and data can reside alone at disparate locations in memory, there must be a mechanism to initialize each of these as required. The procedure for initializing this data is not enforced by this ABI. It is expected that an entry in `.init_array` is created for a function which will run through entries in a table describing how to copy data or initialize variables as required.

Note that this functionality can be leveraged to easily allow functions to be executed from a memory region without persistent storage e.g. RAM. When the linker sees that the segment containing the VMA of the function has a different LMA and VMA, a copy table entry is created, and the runtime startup code will copy the contents of this function from the LMA to VMA, in the same way it would with a piece of data.

4 Using Symbol Meta-Information

4.1 Usage example

The programmer does not need to be aware of the symbol meta-information mechanism itself to be able to make use of the different types and apply special handling to symbols. An attribute set in the source code will cause the compiler to emit an assembler directive describing the meta-information, the assembler then creates the `.symtab_meta` section, which the linker absorbs, performs any required actions, and then outputs a new `.symtab_meta` section with all accumulated metasyms from input object files.

Figure 6: Example

Compiler source code:

```
    uint16_t __attribute__((retain,location(0x1000)))
      core0_key = 0x1234;
```

Compiled assembly code:

```
            .global core0_key
            .type   core0_key, @object
            .sym_meta_info  core0_key, SMT_RETAIN, 1
            .sym_meta_info  core0_key, SMT_LOCATION, 0x1000
```

`.symtab_meta` dump from assembled object file:

```
    SYMBOL META-INFORMATION TABLE:
    Idx     Kind            Value       Sym idx Name
    0:      SMT_RETAIN      0x1         7       core0_key
    1:      SMT_LOCATION    0x1000      7       core0_key
```

5 Conclusion

5.1 Symbol meta-information benefits

Ease of use The application of an attribute to a symbol declaration in the source code is now enough to achieve what previously required both source code and linker script modifications. For programmers without strong knowledge of linker script functionality, there is an even clearer benefit as functionality which may have previously seemed overwhelming to implement is now possible without leaving the source code. Many toolchains supporting ELF are very powerful, and in the hands of an experienced user, behavior supported by symbol meta-information can already be achieved. In this case, symbol meta-information will at least reduce the number of steps the programmer must take to implement the desired behavior.

Record of operations In relocatable files, the symbol meta-information table serves as a list of transformations to be made later in the build process. In executable files, the table shows which transformations have been made. With the assistance of a dump program which has understanding of the format of `.symtab_meta`, a formatted dump of the table makes it clear which symbols have supplemental information.

When linker script modifications are used to alter the handling of certain symbols, that file has to be studied by the programmer, possibly in conjunction with the source code, to understand what special handling is going to be applied. The standard boilerplate linker script code required for regular operation is likely to further obscure which symbols have supplemental information.

Clear, defined purpose Each symbol meta-information type has a specific purpose. When putting symbols into sections with the aim of having them later be treated in some special way by the linker script, it may not always be clear what is trying to be achieved without examining the relationship between the section and symbol at different stages of the build process.

No limitations A type of symbol meta-information can be implemented such that its value describes an offset into the string table, or the section number of a section containing additional information. Therefore, since the true value is not limited to the size of the value in the symbol meta-information table itself, there are many possibilities for what can be accomplished using the meta-information.

5.2 Symbol meta-information as an extension to the ELF gABI

As for why this functionality should be added to the generic ABI, and not a processor-specific or vendor-specific ABI, we see this functionality helping other targets and vendors solve problems previously requiring non-standard and inventive solutions.

Initial versions of this functionality are already implemented for the MSP430 target within the MSP430-GCC fork, and for TI ARM targets in Texas Instruments' Clang/LLVM fork. By making this available in the gABI and introducing the changes to the upstream mainline branches, other targets and vendors can leverage the generic functionality immediately. The overall meta-information mechanism can then be extended in generic, processor-specific, or vendor-specific ways, as required, to further improve the toolchain's feature-set.