IA-64 gABI Proposal 74: Section Indexes

Revised 30 March 2000

Revisions and Status

[000317 All] Use 0 instead of 0xffff for overflow in ELF header fields. Specify that reserved section indexes are reserved only in symtab. Require that symtab extension indexes be valid if present.

[991007 All] Change section/flag names, move ELF header extension to section header 0.

[991005 SGI] Initial version.

Background

SGI has long been concerned about the 64K limitation on the number of sections in an object file. Although this need not normally be a problem, there are purposes for which we would like to place distinct functions, and sometimes data items, in distinct sections. When one takes into account associated sections, e.g. relocation, debug information, etc., this leads to a limitation on the order of 16K units, and threatens to be a problem for some large compilation units such as machine-generated simulators. Indeed, Compaq (DEC) reports having seen examples of this problem on their Alpha platforms.

C++ ABI efforts raise the same issue from another source. Various C++ structures are emitted under circumstances where the compiler cannot reliably identify a single compilation unit in which to emit them. Examples include common cases like class virtual tables, out-of-line copies of inline functions, and template instantiations. The favored solution is COMDAT sections, i.e. putting the potentially duplicated items in their own sections, and allowing the linker to remove the duplicates. Once again, though, this threatens to be a problem for very large compilation units.

The following proposal attempts to remove this limitation. Obviously, even if the problem is real, it will actually arise in very few compilation units. Therefore, the elements of the proposed solution are defined so as to leave unchanged object files which do not encounter the problem. We consider this compatibility objective as primary -- much more important than performance or clean definitions for the problematic object files -- particularly as it should allow vendors to merge the solution into existing tool chains at convenient times without disrupting existing programs.

Proposed ABI wording is in normal font; commentary is in italics. Section numbers are from the Intel IA-64 psABI.

Proposed gABI Changes

General Approach

The range of section indexes from 0xff00 (SHN_LORESERVE) to 0xffff (SHN_HIRESERVE) is reserved for special purposes in the symbol table, and the gABI currently forbids real sections with these indexes, restricting the valid range to 1..0xfeff. Our approach is to deal with situations where the ELF format does not have space to compatibly expand section indexes to a full 32 bits by using a reserved index as an escape value indicating that the actual index will be found elsewhere.

4.1 Elf Header

The ELF header has two relevant 16-bit fields: e_shnum contains the section count, and e_shtrndx the index of a string section. We modify their descriptions to include an overflow indicator, and put the actual values in the reserved section header at index 0 if necessary, as follows:

`ElfXX_Half e_shnum;`
: This member holds the number of entries in the section header table. Thus the product of `e_shentsize` and `e_shnum` gives the section header table's size in bytes. If a file has no section header table, `e_shnum` holds the value zero.

  If the number of sections is greater than or equal to `SHN_LORESERVE` (`0xff00`), this member has the value `SHN_UNDEF` (`0`), and the actual number of section header table entries is in the member `sh_size` of the section header at index 0. (Both entries will be zero if there are no sections.)

`ElfXX_Half e_shstrndx;`
: This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value `SHN_UNDEF`. See ``Sections" and ``String Table" below for more information.

  If the section name string table index is greater than `SHN_LORESERVE` (`0xff00`), this member has the value `SHN_XINDEX` (`0xffff`), and the actual index of the section name string table is in the member `sh_link` of the section header at index 0. (Both entries will be zero if there is no section name string table.)

4.2 Sections

The description of special section indexes needs to change to indicate that the reservation applies only in symbol table `st_shndx` fields. One approach would be to move the list of special section indexes to the symbol table section. Or, it could be modified to something like:

Some section header table indexes are reserved in contexts where index size is restricted, e.g. in symbol table `st_shndx` fields. In such contexts, an escape value indicates that the actual section index is to be found elsewhere in a larger field.

We define a new special section index as an escape value for large section indexes in the symbol table:

`SHN_XINDEX (0xffff)`
: This special section index means, conventionally, that the actual section index is too large to fit in the field where it appears, and is to be found in another location (specific to the structure where it appears).

We note here that the section header contains two fields commonly used to hold section indexes, `sh_link` and `sh_info`, but they are already defined as ElfXX_Word, and require no change.

A new section type is defined:

`SHT_SYMTAB_SHNDX (17)`
: A section of this type is paired with an SHT_SYMTAB section, if any of the symbols in that section reference a section index larger than 16 bits. It contains a table of 32-bit section indexes, one for each symbol in the symbol table section, in the same order. All of these section indexes must be valid if the section is present, whether or not the corresponding index in the SHT_SYMTAB section is `SHN_XINDEX` indicating that the actual index is to be found in this section.

  The `sh_link` field of this section contains the index of the associated `SHT_SYMTAB` section.

A new special section name is defined:

`.symtab_shndx`
: This section holds a section header index table for an associated `.symtab` section, and has type `SHT_SYMTAB_SHNDX (17)`. The section's attributes will include the `SHF_ALLOC` bit if the associated `.symtab` section does; otherwise, that bit will be off.

There is no available field to point from the `.symtab` section to its associated `.symtab_shndx` section, so we use the `sh_link` field in the latter to point back. It is recommended (but not required) that implementations place each `.symtab_shndx` section immediately after its associated `.symtab` section (in the section header table) to make it easy for the linker to find.

4.x Symbol Table

The symbol table is the most problematic. It has no convenient location for an expanded section index. Therefore, we propose that the escape value imply redirection to a separate, parallel table containing full-size section indexes.

Modify the definition of `st_shndx` as follows:

`st_shndx`

Every symbol table entry is defined in relation to some section. This member holds the relevant section header table index.

As the `sh_link` and `sh_info` interpretation table and the related text describe, section indexes in the range 0xff00 to 0xffff indicate special meanings. In particular, `SHN_XINDEX (0xffff)` indicates that the real index is too large to fit in this field, and must be found in the associated SHT_SYMTAB_SHNDX table (above).

If any of the `st_shndx` fields in a symbol table section contain the value `SHN_XINDEX (0xffff)`, there must be an associated `SHT_SYMTAB_SHNDX` section, with a `sh_link` field containing the index of this `SHT_SYMTAB` section. That section contains an array of 32-bit section indexes, matching the symbol table entries 1-1 in the same order. All entries, whether or not they correspond to `SHN_XINDEX (0xffff)` values of `st_shndx` in the symbol table, must contain the actual section header index to be used.

The .dynsym section in a linked object is completely analogous to a .symtab section in a relocatable object, and could be handled in the same way with the addition of a dynamic tag to locate it. We have not specified handling here because we expect the linking process to remove most of the section duplication process which causes the problem, e.g. leaving only a small number of .text sections.

Compatibility

There should be no compatibility impact on existing environments, since only very large section counts require object file changes. Individual vendors can postpone implementation until convenient, with no impact on typical programs.

Note, however, that any ELF consumer applications that are currently storing section indexes as 16-bit values must change.
