The R2K Object Module Format

Revision History

- Minor rewordings: Mon Apr 17 12:38:58 EDT 2017 bks@cs.rit.edu
- Initial version: 2015/11/11 22:01:06 wrc@cs.rit.edu

Overview

The program translation process typically involves converting a program written in some original source language into an executable image for some specific processor architecture. This conversion is typically done in several stages: high-level language code is *compiled* into assembly language; assembly language code is *assembled* into object (machine) language modules; and object modules are *linked* into executable load modules.

There are many different formats in use for object modules. One of these is the *Executable and Linking Format* (ELF)[1], which is used in many different operating systems (including Linux®). It is an extremely flexible format, and even allows the creation of object and load module files which contain executable code for multiple architectures.

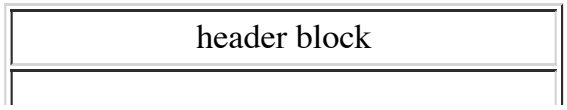
The object module format you will be working with, called the R2K object module format, was designed for use with the assembler, linker, and simulator used in the CSCI-250 (Concepts of Computer Systems) course. This course uses the MIPS R2000 (R2K) processor as its example of CPU design and assembly language programming (hence the R2K name). It is significantly simpler than the ELF format, and was designed to be easy to work with while still providing enough information to allow linking of separately-translated source programs.

When you use the `file` program to identify the file type of these binary files on the CS department systems, you get the type name `PARIX`. For example,

```
$ file sample.obj
sample.obj: PARIX object
```

The R2K Object Module Format

As with virtually all object module formats, this one begins with a header block, which is followed by information specific to the various parts of the memory image of the program. Here is the basic layout of the module:



text section (TEXT)
read-only data section (RDATA)
initialized data section (DATA)
small initialized data section (SDATA)
relocation table (RELTAB)
reference table (REFTAB)
symbol table (SYMTAB)
string table (STRINGS)

Some object modules may not have some of these sections; for instance, an object module having no read-only data would not contain the RDATA section, and one which did not refer to any global symbols (variables or functions) would not contain the REFTAB section.

One important thing to note about these values is that the MIPS R2000 is a "big-endian" architecture. This means that 16-bit and 32-bit numeric values are represented in big-endian byte order, such that the most significant byte is at the first address in memory and the least-significant byte is as the last address. For example, if the hex value 0x11223344 was in memory with its starting address 0x100, the 0x11 byte (the most-significant byte) would be at 0x100, 0x22 would be at 0x101, 0x33 at 0x102, and 0x44 (the least-significant byte) would be at 0x103. This applies to all data other than character strings.

Processing this information on a little-endian architecture (such as the x86 or x86_64 architectures) requires that the bytes be "swapped". This can be done easily using the bitwise and shifting operators, or C library functions can be used; see `"man endian"` or see `"man ntohs"` for more information.

Header Block

The header block in the object module has the following format:

```
typedef
   struct exec_s {
      uint16_t magic;     // "magic number"
      uint16_t version;   // module format version number
      uint32_t flags;     // flags describing the contents
      uint32_t entry;     // entry point (load modules only), or 0
      uint32_t data[10];  // section sizes (in bytes, or # of entries)
   }
      exec_t;
```

(Note the use of type names from `<stdint.h>` to guarantee that fields are of specific sizes.) This type is defined in a header file named `exec.h`, along with the other structure types described below.

The fields in this header have the following contents:

`magic` The "magic number" which identifies this as an R2K object module or load module. For all R2K modules, this is the 16-bit binary value `1111 1010 1100 1110`

(typically abbreviated using hexadecimal notation as 0xface).

`version` A 16-bit value which identifies the version of the R2K module format used for this module. Version numbers are the dates on which those versions were established, encoded in bits as

```
yyyyyyymmmmddddd
```

where `yyyyyyy` is the year minus 2000, `mmmm` is the month, and `ddddd` is the day. Thus, the value `0x0f22` (in binary, `0000 1111 0010 0010`) consists of the fields `0000111` (2007), `1001` (9), and `00010` (2), which is 2007/09/02.

`flags` A set of flags which describe the contents of the module. These would be used by the linker as it is combining this module's contents with the contents of other object modules to determine how the various sections should be merged.

`entry` For object modules, this is the value 0. For load modules, this is the entry point of the program (i.e., the address of the first instruction to be executed when the program runs).

`data` This is an array of section sizes, with one entry per section of the module. The entries in this array are either the sizes of the sections in bytes (for executable code, data areas, and the string table), or the number of entries in the section (for the relocation table, reference table, and symbol table). See below for a description of each section.

Object Module Sections

Each module consists of up to 10 sections, not all of which are represented in the module file itself. These sections are identified here by the index into the `data` array which holds the size of the section.

Index 0: TEXT

This section contains all the executable code for the module, in the form of machine language instructions in binary.

This section is represented in the object module by the binary memory image of the executable code, as an array of bytes; its entry in the `data` array is the size of the section, in bytes.

Indices 1, 2, and 3: RDATA, DATA, and SDATA

These sections contain initialized global data. RDATA contains data that is marked as "read-only" (string literals, `const` globals, etc.); SDATA contains "short" initialized data (no arrays or complex structures); and DATA contains everything else.

These sections are represented in the object module by the binary memory image of the data in the section, as an array of bytes; their entries in the `data` array are the size of the section, in bytes.

Indices 4 and 5: SBSS and BSS

These sections contain uninitialized global data. As with the data sections, SBSS contains "short" data items, while BSS contains everything else. The memory occupied by these sections is traditionally set to 0 by the operating system.

Unlike the other data sections, these sections have no memory image in the object module; their entries in `data` indicate the size of the section, in bytes.

Indices 6, 7, and 8: RELTAB, REFTAB, and SYMTAB

These sections contain the information needed by the linker to link this module with other object modules. The "size" value for these sections is the number of elements in an array of structures that is specific to that section (see below). If the size is non-zero, the appropriate section of the object module will contain this array; otherwise, that section of the object module will be empty (i.e., the section will not occupy any space in the object module).

Relocation entries indicate where the text and initialized data sections contain things which hold values that must be adjusted (relocated) when the module is linked. This includes things such as initialized global pointer variables (which contain the addresses of other variables) and instructions which contain the addresses of other sections of code (the entry points of functions, or other parts of the current function). Each entry has the section (text, data, etc.) and address within that section of the instruction or variable containing the value that must be relocated.

Reference entries are where the text and data sections contain things which refer to global symbols. This includes references to global variables and calls to functions. Each entry contains the section (text, data, ...) and address within that section where the reference occurs, and an index into the string table (see below) to identify the referred-to-symbol.

Symbol table entries exist for all of the global symbols (function names, names of global variables, etc.) which the programmer defined in the module. Each entry contains information about the symbol, and an index into the string table (see below).

Index 9: STRINGS

All the names of symbols in the symbol table are found here as NUL-terminated strings, one after another. This is done to save space in the symbol table entries.

The string table is essentially a large array of characters. Each symbol table and reference table entry contains an index into the string table which is where that specific symbol is found in the table. Once the string table has been read into memory, the base address of the table can be added to the index to create a character pointer.

The Linking Process

As the linker processes object modules, it combines each section from each object module into a single "result" instance of that section; for example, the text sections of all the object modules being linked are combined into a single result text section which is then put into the created object (or load) module. This occurs for all sections from the object module, including the text and data sections as well as the relocation, reference, and symbol tables.

Combining the Object Modules

For the tables, the contents of the table from each object module is entered into a master version of that table within the linker (e.g., the master relocation table contains the entries from all the relocation tables in all the object modules being linked).

For the text and data sections, in order to keep track of where these sections wind up in the resulting object (or load) module, the linker maintains a *load point* for each result section. The load point indicates where this section from the next object module will be placed in the result section; essentially, it's the offset into the result section where the section from the object module will be placed.

Initially, the load point for each result section is 0. The first object module section to be added to the result section begins at the load point, and the load point is incremented by the length of the first section; The section from the second object module is then placed at the current load point (i.e., immediately after the first) and the load point is incremented by the length of this object module's section. This process continues until all object modules have been processed.

How the Modules are Linked

Once the result sections have been created, the actual "linking" begins. The relocation, reference, and symbol tables are used by the linker as part of the linking process, as described below.

Relocation Table Entries

The relocation section of the object module is an array of this structure:

```
typedef
   struct relent_s {
      uint32_t addr;   // address whose contents must be relocated
      uint8_t section; // which section contains this entry
      uint8_t type;    // how to do the relocation
   }
      relent_t;
```

Each entry in this table contains the address of a 32-bit word (the "item") in either the text or data areas of the object module whose contents must be adjusted when this module is linked with other modules. The `section` field indicates which section contains the item; `addr` is the address of the item within the section. This address is relative to the beginning of the section, with the first byte of the section having address 0.

Important note: the value in the `section` field is *not* the same as the index into the header block's `data` array that corresponds to this section; instead, it is that index plus one, as indicated here:

Section name:	TEXT	RDATA	DATA	SDATA	SBSS	BSS
Index into `data`:	0	1	2	3	4	5
Value in `section` field:	1	2	3	4	5	6

Thus, a relocation table entry with a section number of 1 is the text section; one with a number of 2 is the read-only data section; etc. (This is done in order to reserve section number 0 while not requiring that the `data` array be 11 elements long.) This also applies to entries in the reference table, described below.

"Adjusting" the contents of the item consists of adding the load point (see below) of this module to the item's contents in one of several different ways, indicated by the `type` field of the relocation entry:

Value	Meaning
1	Add the lower 16 bits of the load point to the lower 16 bits of the item.
2	Add the lower 16 bits of the load point to the lower 16 bits of the item, and the upper 16 bits of the load point to the lower 16 bits of the 32-bit word following the item.
3	Add the 32-bit load point value to the 32 bits of the item.
4	Add the lower 26 bits of the load point to the lower 26 bits of the item.

Reference Table Entries

The reference table section is an array of the following structure:

```
typedef
   struct refent_s {
      uint32_t addr;   // location of the reference
      uint32_t sym;    // symbol name index into the string table
      uint8_t section; // section number (text, etc.)
      uint8_t type;    // how to fill in the reference
   }
      refent_t;
```

Each reference table entry contains the address of an item in the text or data sections of the object module which refers to an external symbol. The `section` field indicates which section contains the item; `addr` is the address of the item within the section. The `sym` field is the index into the string table of the first character of the string containing the name of the symbol being referred to at this address; this allows the linker to look the symbol up in the symbol table to determine the address associated with it, which can then be "plugged into" this item.

As with relocation table entries, the value in the `section` field is the index of the corresponding entry in the header block `data` array plus one.

The `type` field is an encoded value which describes how the address of the referenced symbol is to be "plugged into" this item. It has the format

```
iimmtttt
```

The `ii` field (the upper two bits of the 8-bit field) are bit flags indicating the state of this entry.

The `mm` field (bits 5 and 4) indicates the operation to be used to combine the symbol's address and the item, as follows:

- 0x00: add the address to the item
- 0x10: replace the item with the address
- 0x20: subtract the address from the item

The `tttt` field (bits 3 to 0) indicate how the combination is to be performed:

Value	Meaning
1	The item is a 32-bit value; add the lower 16 bits of the symbol address to the lower 16 bits of the item.
2	The item is a 16-bit value; add the lower 16 bits of the symbol address to the item.
3	Add the lower 16 bits of the load point to the lower 16 bits of the item, and the upper 16 bits of the load point to the lower 16 bits of the 32-bit word following the item.
4	Add the 32-bit load point value to the 32 bits of the item.
5	Add the lower 26 bits of the load point to the lower 26 bits of the item.

Symbol Table Entries

The symbol table section is an array of the following structure:

```
typedef
   struct syment_s {
      uint32_t flags;  // description of this symbol
      uint32_t value;  // value associated with the symbol
      uint32_t sym;    // symbol name index into the string table
   }
      syment_t;
```

Each symbol table entry contains information about a single global symbol defined in the object module. The `flags` field is a collection of individual bit fields, each of which describes some aspect of the symbol's definition. The `value` field contains a value associated with the symbol; in the case of symbols which label things in memory, this value is the address of the thing being labeled by the symbol. As with reference table entries, the `sym` field is the offset into the string table of the first character of this symbol's name string.

---

[1] *See https://refspecs.linuxbase.org/elf/elf.pdf for details.*