Apollo Object Code Format Specification v1.0.0

Seth Poulsen
sp@sandwichman.dev

Contents

1 Introduction

Apollo is the official object code format for the Aphelion ISA, inspired by ELF and specialized for Aphelion code. Apollo is designed to accelerate and streamline the linking and assembly process. A notable feature of Apollo is lossless linking. Sections and symbols keep information regarding their object file of origin, so no information is lost. This allows incremental linking, object grouping & packaging, and the ability to unlink/decompose an Apollo file into its original components.

This document uses the syntax of the Odin programming language to define its structures when necessary. Odin syntax should be readable for all programmers. For more information, visit odin-lang.org.

2 File Structure

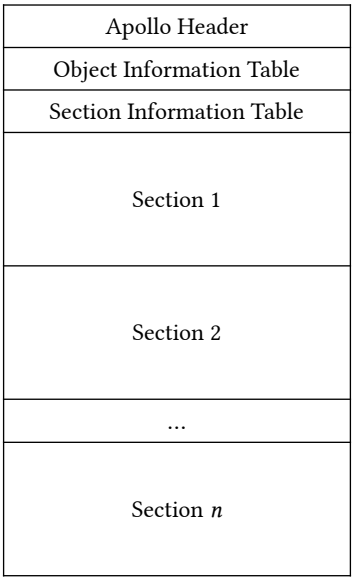

Figure 1: Apollo File Structure

The apollo header sits at the beginning of the file and contains the file's signature, version info, and information about the object and section information table.

The object information table contains information about the file's constituent objects, and is used as an anchor point for sections/symbols to reference.

The section information table contains information regarding each section's properties, size, and location within the file.

The rest of the file is dedicated to the sections themselves, which contain the majority of the file's data, including program code, the symbol table, and more.

All structures are packed with no padding added for type alignment.

2.1 Apollo Header

```
t_apollo_header :: struct {
    magic          : [4]u8,
    version        : [3]u8,
    aphel_version  : [3]u8,
    object_count   : u64,
    section_count  : u64,
}
```

`magic`	Apollo's file signature, `{0×B2, 'a', 'p', 'o'}`.
`version`	Apollo specification version, following `{MAJOR, MINOR, PATCH}`.
`aphel_version`	Aphelion specification version, following `{MAJOR, MINOR, PATCH}`.
`object_count`	Number of entries in the object information table. Entries are a constant size, so the size of the entire table can be determined by multiplying this by the size of a single entry.
`section_count`	Number of entries in the section information table. The size of the entire table can be determined in the same fashion as above.

2.2 Object Information Table

The object information table is made up of entries, structured like so:

```
t_object_info_entry :: struct {
    ident_offset : u64,
    ident_size   : u64,
}
```

`obj_ident_offset`	Offset of this object's identifier in this file's metadata pool section.
`obj_ident_size`	Size of this object's identifier.

The object information entries themselves do not hold much information, They are used as points for sections and symbols to reference.

2.3 Section Information Table

The section information table is made up of entries, structured like so:

```
t_section_info_entry :: struct {
    type         : e_section_type,
    ident_offset : u64,
    ident_size   : u64,
    obj_index    : u64,
    offset       : u64,
    size         : u64,
}
```

`type`	The kind of content this section contains. Represented with a `u8`.
`ident_offset`	Offset of this section's identifier in this file's metadata pool section.
`ident_size`	Size of this section's identifier.
`obj_index`	Index in the object information table that this section is associated with. Index `0×FFFF_FFFF` is reserved for sections not associated with a specific object.
`offset`	Offset of this section's content from the start of the section array.
`size`	Size of this section's content.

2.4 Sections

Apollo supports 6 kinds of sections:

```
e_section_type :: enum u8 {
    program  = 1,
    blank    = 2,
    info     = 3,
    symtab   = 4,
    reftab   = 5,
    metapool = 6,
}
```

`program`	Program-defined information, such as code and data.
`blank`	Uninitialized program data. `s_offset` is ignored for this type, as this kind of section does not take up space in the section pool.
`info`	A table of key-value pairs for storing arbitrary metadata with objects.
`symtab`	Symbol table, stores information about symbols the code uses, and how to relocate those symbols.

`reftab`	Reference table, stores information about where symbols are referenced and how to modify those references.
`metapool`	Index of the file's metadata pool, which is used to store internal data, such as identifier strings.

2.4.1 `program` | Program

A `program` section is for unstructured program-defined information, such as executable code and data.

2.4.2 `blank` | Blank

A `blank` section is for a blank, uniform block of uninitialized space, most often used for uninitialized static variables in programs. They are listed in the section information table, but do not correspond to an actual block of data in the section pool, so `(section_info_entry).offset` may be ignored.

2.4.3 `info` | Information

An `info` section is an array of entries:

```
t_info_entry :: struct {
    key_offset   : u64,
    key_size     : u64,
    value_offset : u64,
    value_size   : u64,
}
```

`key_offset`	Offset of key in this file's metadata pool section.
`key_size`	Size of key.
`value_offset`	Offset of value in this file's metadata pool section.
`value_size`	Size of value.

2.4.4 `symtab` | Symbol Table

The symbol table is an array of symbol table entries:

```
t_symtab_entry :: struct {
    ident_offset : u64,
    ident_size   : u64,
    sect_index   : u64,
    value        : u64,
    size         : u64,
    type         : t_sym_type,
    bind         : t_sym_bind,
    reloc        : t_sym_reloc,
}
```

`ident_offset`	Offset of identifier in this file's metadata pool section.
`ident_size`	Size of identifier.
`sect_index`	Index of associated section in section information table.
`value`	Value of the symbol, usually the offset from the start of its associated section.
`size`	Size of associated data, if applicable.
`type`	Type of associated data. Represented with a `u8`.
`bind`	Scope and precedence information. Represented with a `u8`.
`reloc`	Information for how to change the symbol's value during linking. Represented with a `u8`.

```
e_sym_type :: enum u8 {
    untyped  = 1,
    function = 2,
    variable = 3,
}
```

`untyped`	No type information available.
`function`	Program code, like a function or some other executable data.
`variable`	Program data, like a variable, array, struct, etc.
`section`	Stores the base address for the associated section for relocation purposes. Must always have `bind` set to `t_sym_bind.local` and `reloc` to `t_sym_reloc.base`.

```
e_sym_bind :: enum u8 {
    undefined = 1,
    global    = 2,
    local     = 3,
    weak      = 4,
}
```

`undefined`	Symbol is referenced but not defined.
`global`	Symbol is defined and visible to every object.
`local`	Symbol is defined and only visible within the object it is defined in.
`weak`	Symbol is defined and visible to every object, but may be overwritten by a `global` symbol with the same identifier.

```
e_sym_reloc :: enum u8 {
    absolute = 1,
    base     = 2,
    location = 3,
}
```

`absolute`	Symbol value is absolute and does not change during relocation.
`base`	Symbol value is the base address of its associated section.
`location`	Symbol value relocates to maintain its offset from the section base.

2.4.5 `reftab` | Reference Table

The reference table is an array of reference table entries:

```
t_reftab_entry :: struct {
    sym_index   : u64,
    sect_index  : u64,
    offset      : u64,
    bit_offset  : u8,
    width       : u8,
    type        : e_ref_type,
}
```

`sym_index`	Index of symbol referenced in symbol table.
`sect_index`	Index of section where symbol is referenced.

`offset`	Location where the reference should be inserted, relative to the start of its associated section.
`bit_offset`	Bit offset from `byte_offset` where reference should be inserted. May not be greater than or equal to 8.
`width`	Bit width of the reference. May not be greater than 64.
`type`	How the symbol's value should be modified/transformed before insertion.

```
e_ref_type :: enum u8 {
    pc_offset      = 1,
    pc_offset_div4 = 2,
    absolute       = 3,
}
```

`pc_offset`	Current position is subtracted from symbol value
`pc_offset_div4`	Current position is subtracted from symbol value, and divided by four. Used by branch and jump instructions. Should throw error if result is not an integer.
`absolute`	Do not modify the symbol's value.

2.4.6 `metapool` | Metadata Pool

The metadata pool is not explicitly structured, like the other sections. It is a repository of raw data, most often in the form of text, though this is not enforced. The metadata pool stores the data for all strings used throughout the file, such as object, section, and symbol identifiers. It also stores keys and values used in an information section, where the value may be an integer or other data type/structure. The metadata pool is permitted to be optimized and bytes may belong to multiple data items (ex. two strings, `"object_one"` and `"this_object"` may overlap), so information should be extracted/detangled before editing, and the metadata pool should be rebuilt afterwards.