KO File Format Documentation

- Version 1
- Written as of July 2021

Contents

Preface

This document is intended to provide anyone with an interest in how KO files are structured with a thorough explanation of exactly how and why each component of the file is there.

There has been an interest for several years to create a new programming language for Kerbal Operating System, and many would benefit from being a compiled language rather than a transpiled one. This document seeks to be able to provide developers with a strong starting point to be able to begin that undertaking of their own.

If any parts of this document are outdated or incorrect, please notify us by creating a GitHub issue.

There has been a tool created and maintained called KDump, which is the equivalent to KSM and KO files as objdump or readelf are to ELF files.

About KerbalObjects

The Kerbal Object file format was developed to overcome the problem of wanting to make a complex compiled programming language for kOS. KSM files are the equivalent of executable files in real computer programming. But when most compiled languages like C or C++ are turned into binary, they are turned into what are called object files first. These object files store code and other associated information. Each compiled .c or .cpp file is turned into its own separete object file. Then when the programmer wants to create code that can run, a program called a linker is run, which creates the file that can be run, an executable.

kOS does not have a format for object files, only executables. So KerbalObject files were born. They are based on a simplified version of the real-life ELF file format. These are meant to be read by a specially made linker such as KLinker.

This makes development for kOS much more like the real-life compiler toolchains, bridging the gap between traditional compilers and assemblers, and kOS.

KerbalObject files are based on the ELF file format and provide ELF file like capabilities such as symbols, and more sections than KSM files. Mostly though these features allow an assembler targeting KerbalObject files such as KASM to be able to export more advanced formats which are more sophisticated than what can be accomplished with normal kOS files.

Terminology

KO - KerbalObject
Instruction - One opcode followed by its operands (look up CPU instructions on Wikipedia)

Overview

KerbalObject files consist of a KO header, a section header table, then any number of sections:

- KO header
- Section header table
- Section 1
- Section 2
- Section 3
- Section n

All values in KO files are stored in little-endian format unless otherwise specified.

KO files, like KSM files contain 4 bytes of "magic" that let any program reading the file know that this is a KO file. These bytes are: `0x6b 0x01 0x6f 0x66`, or in decimal `107, 1, 111, 102`.

If converted to ASCII, this becomes: k of, with the second byte having a value of 1, and not being ASCII text, just being a 1. Therefore: k1of, or klof. klof stands for Kerbal Linkable Object Format, which is what this format was originally called before being shortened to just KerbalObject files.

The next byte after the first 4 encodes the version of KO file that this is. As of writing this document the version number will always be 3. KerbalObject files have undergone several large revisions before being completely usable externally. This version should be used to identify if this file is still one that can be read correctly.

The next two bytes are a 16 bit unsigned integer that stores the number of entries there will be in the section header table.

The next two bytes store another 16 bit unsigned integer that stores the index into the section header table that represents the Section Header String Table.

Next comes the section header table.

Section Header Table

KerbalObject files consist of multiple sections. These sections have types, and can store many different kinds of data. In order to know how to read each of these sections, and where they are in the file, a table of Section Headers are created and stored.

Because the number of entries in the Section Header Table has already been stored, the Section Header Table simply contains those entries. The first Section Header is always a "null header", the format of which will be explained. Each Section Header has the following format:

- Name index - 32 bit unsigned integer
- Section Kind - 1 byte
- Section Size (in bytes) - 32 bit unsigned integer

Here is a table of the currently supported Section Kinds:

Section Kind	Value
Null	0
Symbol Table	1
String Table	2
Function	3
Data	4
Debug	5
Relocation Data	6

- Null - Used only for the first Section Header, the "null header". The null header's name index, section kind, and section size are all 0. This Section Header's index is 0, being the first one in the table. This allows symbols or other constructs that reference a Section Header to specify "none" which is a reference to the null header.
- Symbol Table - The symbol table contains KO Symbols which are used when a linker is performing relocation on the file. Symbols and relocation will be explained in further sections of these docs.

- String Table - The string table just contains many null-terminated strings. These strings are indexed like a list, 0 being the first string, 1 being the second, and so on.
- Function - These sections store the actual code that will be put in the executable.
- Data - These sections store the kOS values that are used by instructions and referenced by symbols. Basically the equivalent of Argument Sections from KSM files.
- Debug - Currently not supported. Could be used to store debug information someday.
- Relocation Data - These sections store the places where instruction operands need to be replaced with correct values during linking.

The name index is the index into the *Section Header String Table* that stores the actual string name of this section. The Section Header String Table is a normal string table with a special name. Each section has an associated name. These names are used to identify specifically which section a given section is and how it must be used. A list of currently used special section names is below.

Section Name	Section Type
.shstrtab	String Table
.symtab	Symbol Table
.symstrtab	String Table
.data	Data Section
.reld	Relocation Data Section
.comment	String Table
_init	Function
_start	Function

- .shstrtab - This section marks a specific string table as being the table that all Section Headers' name indexes refer to.
- .symtab - This section stores symbols that are referred to be instructions and handled during linking.
- .symstrtab - KO Symbols have names just like sections do, and this String Table stores those.
- .data - This section is the official place that KO file Instruction data is held. Basically an Argument Section.
- .reld - This section stores the relocation data that the linker uses to perform symbol relocation. This is explained more below, and in the KLinker doccs.
- .comment - This is a string table whose sole purpose is to hold 1 string, known as a comment. This string can store information like "This file was generated using the KASM assembler" or "Kerbals are cool". This will also be included in the final KSM file if this section exists.
- _init - This specific function is treated as the file's initialization code section, like in KSM. The code here happens before _start.
- _start - The entry point of the program. This function is like a "main" function in most programming languages. The program starts running here.

Each Section Header contains the size of the section in order to know where to start and stop reading certain sections.

String Tables

String tables are used to store character strings. Each String Table begins with one byte, with a value of 0x00. This represents a zero-length string, which can be used whenever another part of the KO file wants to express that it has no name, for example. The strings are null-terminated. Unlike in the ELF file format, indexes into the String Table represent which string is being referenced, and not a location in bytes. For example, here is a small list of strings that could represent a String Table:

Index	String
0	""

Index	String
1	"Kerbals"
2	"Like"
3	"Cake!"

A hexdump of that String Table would look like this:

```
 00000000  00 4b 65 72 62 61 6c 73  00 4c 69 6b 65 00 43 61  |.Kerbals.Like.Ca|
00000010  6b 65 21 00                                       |ke!.|
00000014
```

The special String Tables are the Section Header String Table, (named .shstrtab), the Symbol String Table (.symstrtab) and the Comment Section (.comment).

The Section Header String Table stores the names of all of the sections in the KO file, including the Section Header String Table. The Section Header name indexes refer to that table.

The Symbol String Table stores the names of symbols. The format of KOSymbols will be described in the next section.

The Comment Section simply stores one null-terminated string that will be placed in the final KSM file and the KDump utility can read it in order to provide information about where the KSM file came from.

Symbol Tables

Symbol Tables are used to store things called KOSymbols. If you are familiar with the C or C++ programming languages, you already know about symbols to some extent. When a function is declared in C or C++, normally function declaractions are stored in header files like this:

```
int foo(int);
```

The function declaraction is a symbol. In this case because the function can be accessed outside of the current .c or .cpp file by including the header file, it is called a global symbol. In this case it is a global symbol, whose type is a function, and whose name is "foo".

Global variables can also be symbols, like if this was declared outside of any functions in C:

```
int bar = 2;
```

In this case, this is a global symbol, whose type is a "no type" which simply means a variable, whose name is "bar".

In C and C++, there are also what are called local symbols as well. These exist in KO files as well. A local function in C can be defined by:

```
static int func();
```

The "static" keyword in C creates local symbols. That means that this function can only be accessed inside of a single .cpp file.

Finally, KOSymbols in KO files have the following format:

- Name index - 32 bit unsigned integer
- Value index - 32 bit unsigned integer
- Size - 16 bit unsigned integer
- Symbol Binding - 1 byte
- Symbol Type - 1 byte
- Section header index - 16 bit unsigned integer

The name index is an index into the Symbol String Table (.symstrtab).

The value index is an index into the Data Section (.data) which will be documented in the next section.

The size represents the size of whatever this KOSymbol represents, it could be the size of a function in bytes, or the size of a string value.

The symbol binding represents the visibility discussed before, global and local. There is also extern which specifies that the symbol is defined somewhere else, so values or functions can be used without them needing to be defined.

Symbol Binding values:

Symbol Binding	Value
Local	0
Global	1
Extern	2

Symbol Type values:

Symbol Type	Value
NoType	0
Object	1
Func	2
Section	3
File	4

- NoType - This symbol represents a value such as the constant 2, or "Hello world".
- Object - Currently unused, copied from ELF file format.
- Func - This symbol represents a function. Every function requires a KOSymbol representing it to be present in the Symbol Table. This symbol stores the name of the function, as well as if it is global, local, or extern. The section index field of the KOSymbol is used to specify which Function Section in the KOFile this symbol represents.
- Section - This symbol represents a reference to a section in this KOFile, this type is currently unused but again pulled from ELF.
- File - This symbol represents the current KOFile. The name of this symbol is defined to be the source file's name. For example if a file called "mathlib.src" was compiled into this KOFile, the name of the File Symbol would be "mathlib.src". There should only be one File symbol per KOFile.

The section header index is an index into the KOFile's Section Header Table which specifies which section this symbol's data is stored in. For Function Symbols, that is the function section that contains the code. For NoType Symbols, that is the Data Section that stores the data this symbol refers to.

For information on how KOSymbols are treated in the linking step, see the docs folder under the KLinker repo.

Data Sections

Data sections are KOFiles' more useful version of KSM files' Argument Section. Data Sections store constant values that are used by Instructions inside of Function Sections, or global constants that are defined by KOSymbols. These values are stored in the exact same format as KSM files. See KSM file documents under "Arguments" for how to store these values.

Data Section values are indexed similarly to String Tables are. The first value in the Data Section has index 0, the second's index is 1, and so on.

An example Data section could be represented like so:

Index	Value
0	String("print()")
1	Int16(42)
2	ScalarIntValue(10000)

A hexdump of the Data Section would be:

```
 00000000  07 07 70 72 69 6e 74 28  29 03 2a 00 09 10 27 00  |..print().*...'.|
00000010  00                                                |.|
00000011
```

Function Sections

A Function Section stores Instruction that make up parts of a program. Function Section Instructions are slightly different from KSM Instructions, because KO file instructions have constant width. The opcodes are the same as KSM instructions, but each operand is always a 32 bit unsigned integer that represents an entry in the Data Section .data. As mentioned above, Function Sections on their own do not contain the function's visibility to other files, or the function's name. A KOSymbol representing the function should be entered into the KO file's Symbol Table with a section index that is the function section that contains the code.

Even though KSM Code Sections are extremely similar to KO file Instructions, below are examples of a few Instructions:

```
 0x4e  0x01 0x00 0x00 0x00
Push (value at index 1)

0x5a  0x03 0x00 0x00 0x00  0x06 0x00 0x00 0x00
Bscp (value at index 3, value at index 6)
```

Instructions that reference KOSymbols should encode that operand as all zeroes. The KOSymbol reference is not stored in the operand, and is instead stored in the Relocation Data Section (.reld).

Relocation Data Sections

Relocation Data Sections store the data required to relocate KOSymbol data to the proper instructions that need them when the KO files are being linked. In simpler terms, when an Instruction needs to reference a symbol in the Symbol Table, instead of the Instruction storing that, this section does. It might be easier to understand if the format for a Relocation Data Entry is shown:

- Section index - 32 bit unsigned integer
- Instruction index - 32 bit unsigned integer
- Operand index - 1 byte
- Symbol index - 32 but unsigned integer

The section index stores the section to which this relocation applies, which is always a Function Section. The instruction index of course stores which Instruction this entry applies to, which instruction references a symbol. The operand index just stores which operand, the first, or second (if there is one). The symbol index then stores the index into the Symbol Table which is the KOSymbol that needs to eventually be placed in that Instruction's operand.

If both of an Instruction's operands reference symbols, two Relocation Data Entries must be created, one for each operand.

An example of a Relocation Data Entry is showm here:

```
 0x07 0x00 0x00 0x00  0x02 0x00 0x00 0x00  0x00  0x01 0x00 0x00 0x00
^^^^^^^^^^^^^^^^^^^  ^^^^^^^^^^^^^^^^^^^   ^^   ^^^^^^^^^^^^^^^^^^^
    Section 7           Instruction 2     Op 1        Symbol 1
```

Linking Notes

A more or less full explanation of the linking process is describe in the docs folder of the KLinker repository. However a small explanation of what happens is provided here.

- Files are read
- Symbols are resolved
    - This means that external symbols are found, and replaced with the defined versions.
- Functions are given offsets into the file
- Relocations are preformed, symbols are replaced with their values
- KSM instructions and Argument Section are generated
- The file is written