KSM File Format Documentation

- Version 1
- Unofficial, written as of July 2021, Kerbal Operating System release version 1.3.2.0

Contents

Preface

This document is intended to provide anyone with an interest in how KSM files are structured with a thorough explanation of exactly how and why each component of the file is there. A document like this one exists currently within the kOS GitHub repository, however it is outdated and does not explain specific key details. The kOS document also is meant for developers of kOS who are already familiar with the internal code that handles KSM, while this and accompanying documents are designed to be able to be used by someone who is not very familiar with how kOS works internally at all.

There has been in interest for several years to create a new programming language for kOS, 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If any parts of this document are outdated or incorrect, please notify us by creating a GitHub issue.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 KSM

The KerboScript Machine Code file format was developed to overcome the problem of bloated code files filled with comments and repeated data. This data is

essential to good programming practices, but is not needed for code execution. This KSM files were created.

KSM files are collections of compiled program data and instructions that is then compressed in GZIP format to further reduce file size.

Terminology

KSM - KerboScript Machine Code
Instruction - One opcode followed by its operands (look up CPU instructions on Wikipedia)
Operand - An index into the argument section that stores the data this instruction references

Overview

As mentioned above, KSM files are compressed using GZIP compression. A file stored using GZIP can be identified by the first 4 bytes of the file, also known as the "file magic": `0x1f 0x8b 0x08 0x00`, or in decimal `31, 139, 8, 0`.

Certain GZIP compressors will not output this specific file magic, and instead it will be slightly different. No manual GZIP compression tool like 7Zip has been found to create a file that kOS will accept. Nor have the settings that determine this been found.

KSM files can be decompressed using these tools though, and once extracted, can be hexdumped to find the actual contents of the file as described here.

Once decompressed, the file will have the following overall structure:

- KSM file magic
- Argument Section
- Code Section(s)
- Debug Section

KSM files have their own file magic that distinguishes them from other files: `0x6b 0x03 0x58 0x45`, or in decimal `107, 3, 88, 69`.

If converted to ASCII, this becomes: k XE, with the second byte having a value of 3 and not being ASCII text, just being a 3, therefore: k3XE, or kEXE (Kerbal Executable).

This file magic is directly followed by the argument section

Argument Section

The purpose of the argument section is to store all of the constant values that are used by each instruction in the program. This is not usually the way data like this is stored in real-life file formats like ELF, but kOS is not a real-life computer and can do what it wants. The main reason for this is to reduce

duplicated values used by instructions, as kOS storage space is much more precious than storage space on your computer.

The argument section has a header, which marks the start of the section and provides necessary information to continue reading the file. The argument section header begins with two ASCII characters:

%A

The first argument after the "%A" is a byte that stores the size of an instruction operand that is required to index an argument in the argument section. If this number is 1, then that means that only 1 byte is required to store the maximum index of an operand to an instruction, so the maxmimum index is 255. If the number is 2, then 2 bytes are required to store an index into the argument section. Because this is the width of an index, this means that this is also the width of each instruction operand in the entire file.

After the %A and the index size byte, follow the arguments.

Arguments

Each argument begins with the argument type. The argument type is 1 byte wide. kOS currently has 13 argument types:

Name	Byte	Description
Null	0	Represents a null value just like in other languages.
Boolean	1	Represents true or false. Used for arguments to kOS system calls.
Byte	2	Represents a byte. Used for a few kOS system calls.
Int16	3	16 bit integer.
Int32	4	32 bit integer.
Float	5	32 bit floating point number.
Double	6	64 bit floating point number.
String	7	String.
Argument Marker	8	Used to mark the end of function arguments.
Scalar Integer Value	9	32 bit integer. All KerboScript integers.
Scalar Double Value	10	Double-precision floating point. All KerboScript floats.
Boolean Value	11	Same as above, but a boolean.
String Value	12	Same as above, but a string.

In kOS all integer/floating point arguments are stored in little-endian format.

The values of Null and Argument Marker both have 0 size, they are completely defined by their type.

Boolean, Byte, and Boolean Value all have a size of 1 byte, not counting the argument type.

Int16 has a size of 2 bytes not counting the argument type.

Int32, Scalar Integer, and Float value are 4 bytes in size.

Double and Scalar Double Value are 8 bytes in size.

String and String Value are different than other kOS values. The format they are stored in is the same. The next byte after the argument type is a value that stores the length of the string in characters. After that comes the string's value.

The next argument starts right after the last one has ended. A program knows to stop reading the argument section when the next argument's type instead is read as the percent sign character: '%'. This character denotes the start of a code section.

Code Sections

A code section does what it says on the tin, it holds the actual code that is executed when you run a program inside of kOS. The first code section always encountered is a function code section, denoted in the file as the characters "%F".

There are 3 types of code sections, function sections, initialization sections, and main sections.

There can be multiple function sections in one KSM file, and they each store one user-defined function's code.

There can be multiple initialization sections defined in one file as well, and they are used to set default kOS settings before your script starts up or things kOS needs to do strange magic things with LOCKs for example.

Main sections can only appear once per KSM file, and represent the code that will be run when the program starts executing. When kOS loads your program, the entry point is the first instruction in the main section of the loaded file.

Initialization sections and main sections are denoted in the file as "%I" and "%M" respectively.

kOS's KerboScript compiler generally creates KSM files in a non-intuitive way. Each function section does not stand on it's own, but is *always* followed by initialization and main sections, even if another function is the next code in the file. To be more clear, this means that after each function section can be empty initialization and main sections, as shown in a pseudo-hex-dump:

```
%F(insert some hex codes of instructions here)%I%M%F(more...)
```

It is possible that instead of being empty, the initialization section is also populated with code, so don't rely on this pattern.

The main section will(should) always be last.

The best way to parse code sections are to keep reading until a % sign is reached, end that code section and start another. Check if each section is empty, if it

is, ignore it ever existed. (Unless parsing the debug section, but that will be discussed later)

Instructions

Directly after the code section header (the percent sign thing), starts the list of instructions. Instructions in KSM are variable length. The first byte of the instruction is the instruction's opcode. A full list of instruction opcodes is provided in these docs in another file, however a few will be used for examples.

Three instructions will be used as examples:

Instruction	Opcode Value
Add	0x3c
Push	0x4e
Bcsp (Begin scope)	0x5a

Some instructions take operands, others do not. One example of an instruction that does not take any operands is the Add instruction.

Here is an example of a function section that only contains a complete Add intruction:

```
0x25 0x46 0x3c
^^^  ^^^   ^
 %    F   Add
```

The Add instruction is always the same size: only 1 byte. You may be asking what the add instruction does, how does it add two numbers if none are provided? The answer is that kOS is what is called a stack machine. If you understand what a stack machine is, the next example instruction will certainly strike you as important, the Push instruction.

An example of the Push instruction in binary form alone is:

```
0x4e 0x02
^^^   ^

Push  2?
```

This instruction takes one operand, and in this case the value of that operand is 5. First of all, the reason that this operand's size is only 1 byte is because in this example, we are assuming that argument index width from the Argument Section above, is 1. If that argument index width were 2, this Push instruction would be:

```
0x4e 0x0500
```

Note that the operand, if multiple bytes, is in little-endian format.

Now for the value of the operand. The value of any instruction's operand does not represent the value that will be used, it represents a *byte index into the file's Argument Section of the value*. A key thing to note is that the Argument Section's header **is included** in the index. To be clear, that means that because the Argument Section's header is 3 bytes long (%A then number), the first index an argument can have is 3, and that refers to the first argument in the section.

Finally, if an instruction has more than one operand, they are simple added one after another. Currently as of writing this documentation, kOS only has up to 2 operands per instruction.

An example of the Bscp instruction which takes two arguments is shown below using an index size of 1:

```
0x5a 0x03 0x06
```

The only way to know the size an instruction will be when reading it is to keep track of how many operands each instruction takes. As stated above a file documenting this is provided alongside this one.

Because an instruction's opcode value is not allowed to be 0x25 which is the percent sign in ASCII, the end of the current code section can be detected by checking if the next byte is 0x25, or '%'.

Debug Section

Each KSM file contains a debug section. This section allows kOS to find out what source code line an error occurred on without having the code available.

The Debug Section, like the other sections starts with a header that begins with %: "%D".

The Debug Section also has another part of its header, a single byte representing the size of an index into the object file. 1 for 1 byte, 2 for 2 bytes, etc.

The Debug Section consists of several Debug Entries that map a line number to one or more Instructions that make up that source line.

A Debug Entry contains the following data:

- Source code line number, a 16-bit signed integer.
- The number of Instruction ranges
- 1 or more Instruction ranges that represent the instructions that make up the source line.

An example would be a debug entry like this:

Line Number	Number of Ranges	Range 1
1	1	2 - 7

This means that the Instructions within the range 2 through 7 make up line 1, and nothing else.

Or like this for a more complicated source line:

Line Number	Number of Ranges	Range 1	Range 2
3	2	7 - 8	10 - 11

This means that the Instructions within range 7 to 8, and 10 to 11, but **not** byte 9 (for whatever reason) make up line 3 of the source.

The ranges are the Debug Entry fields that are affected by that index width in the Debug Section header. So if these two examples were encoded into hexadecimal, the first one having an index width of 1, and the second having an index width of 2:

```
0x01 0x00  0x01  0x02 0x07
^^^^^^^^^^   ^       ^^^^^^

Line 1     1 range   2 - 7

0x03 0x00  0x02  0x0700 0x0800 0x0a00 0x0b00
^^^^^^^^^^   ^        ^^^^^^^^       ^^^^^^^

Line 3     2 ranges   7 - 8         10 - 11
```

The ranges are expressed as *byte indexes* into the KSM file. The index 0 begins at the first byte after the *Argument Section*. And continues on until the Debug Section is reached. That means that all bytes in the file from the Argument Section up to that point are counted, including code section headers like %F and %I%M. So the first valid Instruction index in the file is actually index 2, because it will have come after one code section header. The debug section ends when the end of the file is reached.

Example

Here is the full hexdump of the `test.ksm` file provided in the kerbalobjects.rs repository:

```
00000000  6b 03 58 45 25 41 01 07  07 70 72 69 6e 74 28 29  |k.XE%A...print()|
00000010  07 00 09 02 00 00 00 08  07 05 40 30 30 30 31 03  |..........@0001.|
00000020  01 00 03 00 00 25 46 25  49 25 4d f0 14 5a 1b 1e  |.....%F%I%M..Z..|
00000030  60 4e 13 4e 0e 4e 0e 3c  4c 0c 03 4f 5b 1b 25 44  |`N.N.N.<L..O[.%D|
00000040  01 01 00 01 06 18                                 |......|
00000046
```

As you can see the first 4 bytes of the file are what are expected: 6b 03 58 45, or as you can see on the left side (mostly) converted to text: `k.XE`. The next 2 bytes are the expected %A, followed by a byte with the value of 1, meaning the

Argument Section can be indexed using only 1 byte, or more specifically, that Instruction operands will be 1 byte wide.

The next byte starts the Argument Section data, and is a 7. Going back to the list of kOS value types, that is a String. This means that the next byte stores how long the string is, 7 bytes. Then after that follows the string: 70 72 69 6e 74 28 29. Or more readable to us on the right hand side: `print()`. ”print()” is the string used to call the built-in kOS print function.

Using the kOS value reading logic, we get the following arguments in total:

Type	Value	Index
String	”print()”	0x03
String	””	0x0c
Scalar Int Value	2	0x0e
ArgMarker	n/a	0x13
String	”@0001”	0x14
Int16	1	0x1b
Int16	0	0x1e

Then we hit another %-sign and read: %F%I%M. This means that there is an empty function section, empty initialization section, and a main section. Then follow the Instructions.

The first byte in the Main Code Section is 0xf0. This is the opcode for the Label Reset Instruction. This Instruction takes 1 operand, and that operand is 0x14. Checking the kOS value at Argument Section index 0x14, we get:

```
lbrt @0001
```

This instruction resets the internal kOS label and marks the first Instruction as the first in the program, and tells kOS where to start running the program.

The next byte after that is 0x5a, which is the opcode for the Begin Scope Instruction. This Instruction takes two operands, and the next two bytes are 0x1b and 0x1e. These indexes correspond to the values 1 and 0 respectively, so the instruction encoded here is:

```
bscp 1, 0
```

Then the rest of the Instructions are read this way:

Instruction	Operand(s)
argb	
push	ArgMarker
push	2
push	2
add	

Instruction	Operand(s)
call	"", "print()"
pop	
escp	1

Wow! Reading through that, this program adds 2 + 2 and prints the result! Sounds like a useful program.

At the end of that, the Debug Section is reached. %D is read, and the next byte after that is a 1, meaning that range indexes will only be 1 byte wide.

In this case there is only one Debug Entry. The next two bytes are 0x01 and 0x00, which encodes the line number of 1. Then the next byte is 0x01, meaning that there is only 1 range. That range is 0x06 to 0x18, or indexes 6 to 24. If you do the math, you will find out that bytes 6 through 24 means all of the Code Sections, from the Argument Section to the Debug Section, or the entire code of the file. This encodes that all of the code in this file is from source line 1.

The KerboScript that this KSM File could have been created by compiling could be:

```
PRINT 2 + 2.
```

That is quite a large file to store such a simple program. But when the source code starts becoming larger, the cost savings of using KSM files to store it rockets (pun intended), especially when things like comments come into play.

This concludes the description of the kOS KSM file format.

Notes

The best way to get hands-on experience with KSM files is to run the kOS compiler by running `COMPILE (your file).ks.` and looking at the output.