King's Field II Compendium

Version 1.0

Contents

Introduction

Foreword

This document serves to be the most comprehensive collection of knowledge (a compendium) about the inner workings of the game King's Field II (released as King's Field in EU and NA territories).

Topics such as data types, file formats and internal functioning of the game will be discussed - so if you're looking for a wiki or walkthrough you've come to the wrong place.

Furthermore, this document assumes you have at least basic knowledge of programming, a basic knowledge of reverse engineering subjects and an intermediate knowledge of the PlayStation 1 Hardware.

Attributions

The knowledge contained in this document has been collected by many members of the *FSMC,* and it is only fair that each member gets their own place in thanks for their efforts in deciphering how this obscure game works! Those members are:

IvanDSM:
Program reverse engineering, maintainer of *KFModTool… Far* too much to list here.

TheStolenBattenberg:
Program reverse engineering, format reverse engineering and compiling this document

HwitVlf:
Format reverse engineering

Holy_Diver/Mick/SwordOfMoonlight:
Format reverse engineering

Mendzen:
Data structure field identification

Jargon

- **FSMC**: “FromSoft Modding Committee”
- **KFModTool**: Software created for editing and exploring King’s Field II game data.
- **FromSoft**: “*Do your own math*” - The Cold Ash, RE2 Irregular.

Legal Acknowledgements

'**King's Field**' is the intellectual property of **FromSoftware Inc.** . All information provided in this documentation is for **educational purposes only**, and the authors do not encourage nor condone piracy.

License

Chapter I: Data Types

Primitive Data Types

The following data types are simple arithmetic (or logical) byte constructs, used for the declaration of variables or to build compound data types such as structures. The primitive data types of this document are based on the ones defined in *stdint.h* from the *C Standard Library.*

Type Name	Common Type Name	Byte Size	Bit Size
`sint8`	`signed char`	1	8
`sint16`	`signed short`	2	16
`sint32`	`signed int`	4	32
`uint8`	`unsigned char`	1	8
`uint16`	`unsigned short`	2	16
`uint32`	`unsigned int`	4	32
`fixed16`	`n/a`	2	16
`bool`	`bool`	1	8

Complex Data Types

The following data types are complex (or compound), meaning that they are built from multiple primitive data types - these are used for more advanced values, such as a numerical vector. From this, you'll likely discern that in C++/C (and other languages), a *structure* is a complex data type - a *bit field* can also be classified as a complex data type.

Taking into account the above brief, the following complex data types will be described using true C structure representation, and/or a visual representation of the bit layout should the type be a bit field.

'fixed16'

The only type here that may require any further explanation, is the *fixed16* type. This is a fixed point type, which allows storage of a decimal point number as an integer. This was very much commonplace before faster FPU integration in CPUs, and absolutely necessary for the PlayStation 1 hardware, which lacked an FPU entirely. You might want to consider this a 'complex primitive type'. See ***Figure 1.0*** for more information.

An easy method to convert a fixed16 value to a floating point type, is to first read the value as a sint16, then divide it by the constant *4096.0*.

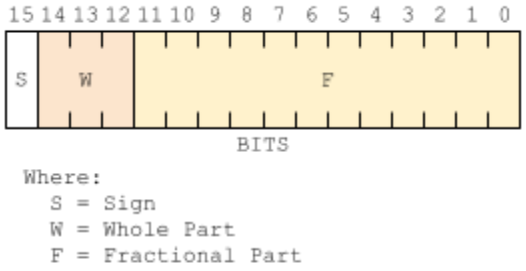

Figure 1.0: fixed16 bit layout

‘svector4’

```
typedef struct
{
    sint16 x;
    sint16 y;
    sint16 z;
    sint16 w;      //For the most part, this component is used as padding.
} svector4;
```

The ‘svector4’ type is a numerical vector, with four sint16 components. It is an intrinsic type for PlayStation 1 hardware and libraries, and comes from those very libraries.

Technically speaking, ‘***w***’ component is actually padding if you want to strictly adhere to the specifications that Sony laid out; FromSoftware didn’t really care about those for the most part and sometimes used the component to store additional data, which is likely the case for other PlayStation 1 software due to the strict memory limits of the console.

‘colour16’

```
typedef struct
{
    uint16 a : 1;
    uint16 b : 5;
    uint16 g : 5;
    uint16 r : 5;
} colour16;
```

This type represents an RGBA colour using 16 bits, it's roughly equivalent to an 'rgba5551' only with an exception to the alpha mask. This is a standard type for PlayStation 1 libraries, as the framebuffer was represented in this exact format.
The alpha (or mask) bit has different functions, depending on the following factors:

1. PlayStation 1 Render State
2. The RGB value of the colour16.

See ***Figure 1.1*** for the bit layout, and ***Figure 1.2*** for a table which describes the different modes of the alpha (mask) bit.

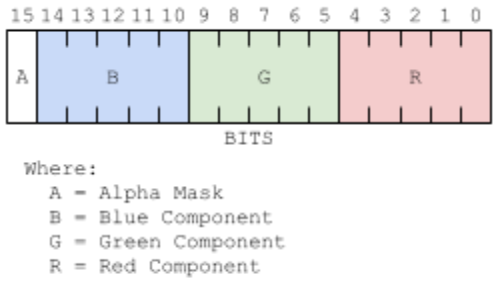

Figure 1.1: colour16 bit layout

Mask Value (A)	Colour (R, G, B)	Translucency Render State	**Result**
0	0, 0, 0	Translucency Off	Transparent
0	X, X, X	Translucency Off	Opaque
0	0, 0, 0	Translucency On	Transparent
0	X, X, X	Translucency On	Opaque
1	0, 0, 0	Translucency Off	Opaque
1	X, X, X	Translucency Off	Opaque
1	0, 0, 0	Translucency On	Opaque
1	X, X, X	Translucency On	Semi-Transparent

Figure 1.2: colour16 alpha (mask) bit result table

Chapter II: Archive Format (*.T)

Described in this chapter is the first of many file formats we will cover. This particular format is arguably one of the most important that this document will explain since without being able to parse it any further format documentation is basically redundant, as most of the data of King's Field II is stored in one of these files.

*Theorised/Possible file extension meaning: **T**OC; **T**ABLE*

Brief

While most of the structures in this document are written to actually compile (assuming of course that you defined the primitive data types), format specification structures will for the most part use a pseudo-c, which will not compile. This is primarily for readability reasons, as maintaining true compilability would require the inclusion of a lot more code, and providing source code is not the goal of this documentation.

The goal of this particular format is to store game data in an 'processing efficient manner', relying very heavily on sector alignment. This is absolutely essential for King's Field II, as level streaming is only possible with fast disc access or by keeping all the files in memory at once (which is not only impossible given the memory limitations of the PlayStation 1, but counterintuitive to the purpose of streaming data in the first place, given that it's not streaming at that point.)

Overall Format Structure

```
#define SECTOR_SIZE 2048

typedef struct
{
    uint16 offsetCount;
    uint16[] offsetTable;
    uint16 endOfFileOffset;
} tformatheader;

typedef struct
{
    tformatheader header;
    uint8[] bin;
} tformat;
```

As you can see, what is meant by a 'processing efficient manner' in the brief is actually 'minimal as fuck'. Now we'll begin breaking down each structure.

‘tformatheader’

This is a dynamic-size structure, which contains the basic TOC for the T file. Starting from the top, ‘***offsetCount***’ will tell you exactly how long the ‘***offsetTable***’ in elements. The ‘***offsetTable***’ is an array of *uint16* type, where each element stores a **sector offset** of some data that belongs to a file.

Whether a result of the tools FromSoft were using at the time to construct these files, or an intentional inclusion in order to not break hardcoded offset references used in the game code - these files contain (annoying) duplicate sector offsets (see ***Figure 2.0*** for an example) that you have to be very aware of when you’re attempting to work with these files.

The final value is the ‘***endOfFileOffset***’ (again, a **sector offset**) which quite simply is the end of the file. Although seemingly uninteresting, this is actually a key feature of the format which makes working with T files a little easier.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	46	00	01	00	22	00	30	00	32	00	54	00	62	00	65	00
00000010	87	00	95	00	97	00	B9	00	C7	00	CA	00	EB	00	F9	00
00000020	FD	00	1E	01	2C	01	2F	01	51	01	5F	01	62	01	82	01
00000030	90	01	94	01	94	01	94	01	94	01	94	01	94	01	94	01
00000040	B5	01	C3	01	C6	01	C6	01	C6	01	C6	01	C6	01	C6	01
00000050	C6	01	C6	01	C6	01	C6	01	C6	01	C6	01	C6	01	C6	01
00000060	C6	01	BC	02	F2	02	F9	02	FF	02	0A	03	16	03	22	03
00000070	2D	03	38	03	38	03	38	03	4B	03	57	03	60	03	6F	03
00000080	81	03	96	03	B5	03	BE	03	CC	03	CC	03	CC	03	CC	03

Figure 2.0: Annoying duplicates in an example T file

‘tformat’

This is a dynamic-size structure, which contains only the header and a binary chunk of data. We’ve already covered what ‘***header***’ in the previous section, but in the context of a whole file it’s worth noting that if the total size of the header is less than the size of a sector (2048 bytes), the header is padded with zeros until the size is equal to ‘***SECTOR_SIZE***’.

All files observed in the wild have been shown to have less than 1022 offsets, which means in all known cases ‘***bin***’ will always begin at **offset 0x800** (using ***Figure 2.0*** again, we can see that the second uint16 has a **sector offset** value of *one*). It is however assumed that if a T file were to have more than 1022 offsets, the ‘***bin***’ would simply begin at **offset 0x1000** (a **sector offset** of two).