# King's Field II Compendium

Version 0.1

# Contents

Introduction 2
- Foreword 2
- Attributions 2
- Jargon 3
- Legal Acknowledgements 3
- License 3

**Index 4**

Chapter I: Data Types 5
- Primitive Data Types 5
- Complex Data Types 5
  - ‘fixed16’ 5
  - ‘svector’ 6
  - ‘colour16’ 6

Chapter II: Container Format (*.T) 8
- Brief 8
- Format Specification 8
  - ‘tformatheader’ 8
  - ‘tformat’ 9
- Reading and Writing 9
  - Checksum 9

Chapter III: Container Format (MIX) 11
- Brief 11
- Format Specification 11
- Sized MIX 11
- Sizeless MIX 12

Chapter IV: Model Format (*.TMD) 12

Chapter V: Model Format (*.RTMD) 12

Chapter VI: Model Format (*.MO) 13
- Brief 13
- Format Specification 13
  - Vertex Frame Decompression 14
  - Vertex Frame Interpolation 15

Chapter VII: Image Format (*.TIM) 15

Chapter VIII: Image Format (*.RTIM) 15

# 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 otherwise;* it is only fair that each contributor gets their own place in thanks for their efforts in deciphering how this obscure game works! They are:

***IvanDSM:***
Program/format reverse engineering, this document and maintainer of *KFModTool*.

***TheStolenBattenberg:***
Program/format reverse engineering, 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.
- **PsyQ**: PlayStation 1 development libraries (SDK) created by Psygnosis LTD.
- **KF**: "King's Field"

## Legal Acknowledgements

'**King's Field**' is the intellectual property of **FromSoftware Inc.** . All information provided in this documentation is for **educational purposes only**, and is not intended to facilitate piracy or wrong doing.

## License

This document is licensed under a **Creative Commons Attribution 4.0 International License**.

This license does **NOT** apply to anyone using this documentation for the purpose of learning how the game, formats or structures work in order to write tools or software. If you are one of those people you are **free to ignore this section and provide no attribution.** You will not be in breach of the license. Citation or acknowledgement is still appreciated, however.

If you wish to create a new document using any part of '*King's Field II Compendium*' please provide the required attribution to the *FromSoft Modding Committee (FSMC)* on any derivative works.

# 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 used in 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 |
| `string` | `string or char[]` | n | 8 * n |

## Complex Data Types

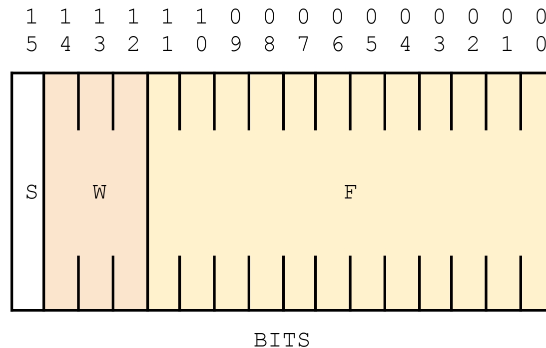
The following data types are complex (or compound), meaning that they are built from multiple primitive data types - these are used for more advanced values, such as a numerical vector. From this, you'll likely discern that in C/C++ (and other languages), a *structure* is a complex data type - a *bit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

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 also consider this a 'complex primitive type'. See ***Figure 1.0*** the bit field layout.
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*.




```
Where:
  S = Sign
  W = Whole Part
  F = Fractional Part
```

*Figure 1.0: fixed16 bit layout*

## 'svector'

```
typedef struct {
    sint16 x;
    sint16 y;
    sint16 z;
    sint16 w;    //This component is called padding in PsyQ.
} svector;
```

The 'svector' type is a numerical vector, with four components. It is an intrinsic type for PlayStation 1 hardware and libraries, and comes directly from PsyQ.

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'

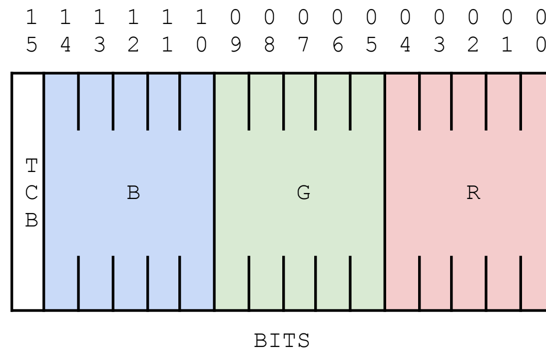
```
typedef struct {
    uint16 a : 1;
    uint16 b : 5;
    uint16 g : 5;
    uint16 r : 5;
} colour16;
```

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 'Translucency 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 TCB bit.




```
Where:
  TCB = Transparency Control Bit
  B =   Blue Component
  G =   Green Component
  R =   Red Component
```

*Figure 1.1 - colour16 bit layout*

| TCB Value | 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 TCB, colour and translucency render state combinations and their results.*

# Chapter II: Container Format (*.T)

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

## Brief

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 both impossible given the memory limitations of the PlayStation 1 and counterintuitive to streaming).

## Format Specification

```
#define BLOCK_SIZE 2048

typedef struct
{
    uint16 offsetCount;
    uint16[] offsetTable;   //In blocks (BLOCK_SIZE * offsetTable[n])
    uint16 endOfFileOffset; //Also in blocks
} tformatheader;

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

*Figure 2.0: C-style pseudocode for the T container format.*
TSB: *As you can see, what is meant by a 'processing efficient manner' in the brief is actually 'minimal as fuck'.*

### 'tformatheader'

This is a dynamic-size structure, which contains a basic TOC for the T file. Starting from the top, '***offsetCount***' will tell you the length of '***offsetTable***' in elements. The offset table is an array of type *uint16*, with each element storing a **block offset**.

Something to be aware of at this point, is that KF2 uses hardcoded T indices within its code (for example, when trying to look up the game database, it will always use index 48). As a result of this, every index must point towards the file the game originally expects it to point to - which brings up an annoying aspect of T Files - duplicate block offsets, which are an intentional inclusion in order to not break those hardcoded indices (see *Figure 2.1*).

The final value is the '***endOfFileOffset***' (again, a block offset) which quite simply is the end of the file. Although seemingly uninteresting, this is actually a key feature of the format which we'll go over in the '*Reading and Writing*' section.

```
Offset (x) 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

0x00000000 46 00 01 00 22 00 30 00 32 00 54 00 62 00 65 00
0x00000010 87 00 95 00 97 00 B9 00 C7 00 CA 00 EB 00 F9 00
0x00000020 FD 00 1E 01 2C 01 2F 01 51 01 5F 01 62 01 82 01
0x00000030 90 01 94 01 94 01 94 01 94 01 94 01 94 01 94 01
```

*Figure 2.1: Annoying duplicates in KF2's FDAT.T*

### ‘tformat’

This is a dynamic-size structure, which contains the header and a large chunk of data. We’ve already covered the 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 ‘***BLOCK_SIZE***’, the header is padded with zeros until a full block can be filled. This is actually what happens with all the files before they are appended to the end of the bin. The total size is rounded up to the next multiple of BLOCK_SIZE. Any additional bytes added with the padding is simply zero filled, and the last 4 bytes are sometimes reserved for a custom checksum (see section: ‘*Checksum*’, in ‘*Reading and Writing*’).

From observing many T files, it's noted that all have less than 1022 files referenced in the TOC, which means in every known case **'bin'** will begin at offset 0x800 (using Figure 2.0 again, you can see that the first entry of the offset table has a block offset value of '01 00', or '1') - knowing this information, we can actually assume that should a T file have more than 1022 entries, 'bin' would simply begin at offset 0x1000, and the first entry in the offset table would be '02 00', or '2'.

## Reading and Writing

Here is a brief overview of how you could read or write T files, with code samples and reference images provided. First, we’ll go over some of the necessary concepts.

### Checksum

The last four bytes of the last block of each file contain a checksum for that file. The exceptions to this rule are VAB.T and FDAT.T, which we presume don’t have checksums because adding a checksum would overwrite a part of the files data, since .VH/.VB (split VAB header and body files) can fill every single byte of a block.

The algorithm itself is simple, and likely leads to many collisions, but it does verify that the data has loaded correctly which is believed to be the reason a checksum is included and not an anti-tamper solution. To calculate this checksum for a given file, start a *uint32* variable with the hexadecimal value 0x12345678, and add every 4 bytes of the file to it (casting to *uint32*). Some pseudocode for this operation follows.

```
uint32 get_checksum(void* file, uint32 size)
{
    uint32 checksum = 0x12345678;
    for (uint32 offset = 0; offset < size; offset++)
        checksum += *(uint32*) void + offset;
    return checksum;
}
```

*Figure 2.1: C-style pseudocode for the T container checksum calculation.*

# Chapter III: Container Format (MIX)

*Theorised/Possible file extension meaning: **MIX***

## Brief

MIX is a simple container format originally designed for King's Field I that does not do any padding or alignment. There are two variants of MIX used in King's Field II - one where files contain their size and one where they don't.

## Format Specification

```
// MIX with sizes
typedef struct
{
    uint32 fileSize;
    uint8[] fileContents;
} sizedMixSubfile;

typedef struct
{
    sizedMixSubfile[] files;
} sizedMix;

// MIX without sizes - not much of a structure really.
typedef struct
{
    uint8[] data;
} sizelessMix;
```

*Figure 3.0: C-style pseudocode for both variations of the MIX container format.*
*IDSM: Calling the second one a structure or even pseudocode is really stretching it...*

## Sized MIX

The most common type of MIX file in King's Field II is the sized MIX. It is a simple sequence of files, where each file has its length prefixed to it as a *uint32* value. There is no table of contents or any other fancier features. Reading and writing them is straightforward, but we provide slightly more abstract pseudocode for the read operation nonetheless.

```
// Assume filesOut is a vector we want to fill with pointers to buffers for
// each file.
void read_mix(FILE* file, vector* filesOut)
{
    uint32 subFileSize;
    while (feof(file) == 0)
    {
        fread(&subFileSize, 4, 1);
        void* buffer = malloc(subFileSize);
```

```
            fread(buffer, 1, subFileSize);
            vector_append(filesOut, buffer);
        }
}
```

*Figure 3.1: C-style pseudocode for reading MIX files.*

### Sizeless MIX

Sizeless MIX files are much less common in King's Field II, though not completely absent. The files in the OP directory with the .D extension are sizeless MIX files, and so are the subfiles in RTIM.T and RTMD.T. They are simply walls of data, files concatenated into a huge mess. Therefore, sizeless MIX files are hard to parse and require the developer to scan the file on the fly, doing type detection to match a subfile's type and then have specialized handling for each type it may contain to obtain the subfile's size, which is then used as an offset on which to restart the operation. We do not provide pseudocode for this as just the filetype handling itself would require this compendium to be a multi-volume ordeal.

# Chapter IV: Model Format (*.TMD)

---

*Theorised/Possible file extension meaning: **T**RANSFORMED **M**O**D**EL*

to-do

# Chapter V: Model Format (*.RTMD)

---

*Theorised/Possible file extension meaning: **R**EADY **T**RANSFORMED **M**O**D**EL*

to-do

# Chapter VI: Model Format (*.MO)

*Theorised/Possible file extension meaning: **MO**TION; **MO**DEL*

## Brief

MO files are a masterclass in how not to design a file format. Working with these does tend to lead to absolute insanity.

## Format Specification

```
#define MO_COPY_PACKET_ID 0xFFFF

typedef struct {
    uint32 size;
    uint32 numAnim;
    uint32 offTmdBase;
    uint32 offVertexFrames;
    uint32 offAnimationTable;
} moheader;

typedef struct {
    uint16 playMode;  //Only known use is 'reverse' in KF1.
    uint16 playRate;
    uint16 mainTargetIndex;
    uint16 numJourneyTargets;
    uint16[] journeyTargetIndices;
} moframe;

typedef struct {
    uint32 numFrame;
    uint32[] frameOffsets;
} moanimation;

typedef struct {    //Size is either 4b or 6b, depending on packet type.
    union {
        uint16 copyPacketID;
        sint16 vx;
    } p1;
    union {
        uint16 copyCount;
        sint16 vy;
    } p2;
    sint16 vz;  //Only when 'copyPacketID != MO_COPY_PACKET_ID'
} moframepacket;

typedef struct {
    uint16 packetCount;
    moframepacket[] packets;
} movertexframe;
```

```c
typedef struct {
    moheader header;
    uint32[] animationOffsets;
    moanimation[] animations;
    moframe[] animationFrames;
    uint32[] vertexFrameOffsets;
    movertexframe[] vertexFrames;
    uint8[] tmd;
} moformat;
```

*Figure 6.0 - C-style pseudocode for the MO model format.*

### Vertex Frame Decompression

Vertex frames in KF2 have a very simple type of compression applied to them, which must be decoded in order to interpolate the vertices. The TL;DR of this, is that if a vertex (or many vertices) doesn't change - it is simply copied from the last frame (or base vertices if no previous frame exists), only changed vertices are stored directly.

```c
void DecompressVertexFrame(uint16* mo, uint32 off, sint16* out, sint16* last)
{
    uint16 numPackets = mo[off++];
    uint16 numVertices = 0;

    while(numPackets > 0)
    {
        if(mo[off] == MO_COPY_PACKET_ID)
        {
            uint16 numToCopy = mo[off + 1];

            while(numToCopy > 0)
            {
                out[(3 * numVertices) + 0] = last[(3 * numVertices) + 0];
                out[(3 * numVertices) + 1] = last[(3 * numVertices) + 1];
                out[(3 * numVertices) + 2] = last[(3 * numVertices) + 2];

                numVertices++;
                numToCopy--;
            }

            off += 2;
        }
        else
        {
            out[(3 * numVertices) + 0] = (sint16)mo[off + 0];
            out[(3 * numVertices) + 1] = (sint16)mo[off + 1];
            out[(3 * numVertices) + 2] = (sint16)mo[off + 2];
            off += 3;

            numVertices++;
        }
        numPackets--;
    }
```

```
}
```

*Figure 6.1 - C-style pseudocode for vertex frame decompression.*

### Vertex Frame Interpolation

In an ordinary vertex animation format, this would be a standard procedure of interpolating the vertices of *frame N* and *frame N+1*, but due to file format constraints (the frame compression), this becomes a somewhat more lengthy procedure.

# Chapter VII: Image Format (*.TIM)

*Theorised/Possible file extension meaning: **T**EXTURE/**IM**AGE*

to-do

# Chapter VIII: Image Format (*.RTIM)

*Theorised/Possible file extension meaning: **R**EADY **T**EXTURE/**IM**AGE*

to-do