uClinux - BFLT Binary Flat Format

uClinux uses a Binary Flat format commonly known as BFLT. It is a relatively simple and ligh executable format based on the original a.out format. It has seen two major revisions, versio more recently version 4. Version 2 is used by the m68k-coff compilers and is still in use with elf2flt converter while version 4 is used with the m68k elf2flt converter. Like many open sourc projects worked on by many individuals, little features have creped into versions without the r field changing. As a consequence what we detail in each version may not strictly be the case circumstances. Both the 2.0.x and 2.4.x kernels' bFLT loaders in the CVS support both form Earlier kernels may need to have binfmt_flat.c and flat.c patched to support version 4, should 4 binaries report the following message when loading

```
bFLT:not found.
bad magic/rev(4,need 2)
```

Each flat binary is preceded by a header of the structure shown below in listing 1. It starts with 4 ASCII bytes, "bFLT" or 0x62, 0x46, 0x4C, 0x54 which identifies the binary as conforming to the flat format. The next field designates the version number of the flat header. As mentioned there are two major versions, version 2 and version 4. Each version differs by the supported flags and the format of the relocations.

Figure 1 : Flat Fil

The next group of fields in the header specify the starting address of each segment relative to the start of the flat file. Most files start the .text segment at 0x40 (immediately after the end of the header). The data_start, data_end and bss_end fields specify the start or finish of the designated segments. With the absence of text_end and bss_start fields, it is assumed that the text segment comes first, followed immediately by the data segment. While the comments for the flat file header would suggest there is a bss segment somewhere in the flat file, this is not true. bss_end is used to represent the length of the bss segment, thus should be set to data_end + size of bss.

```
struct flat_hdr {
        char magic[4];
        unsigned long rev;          /* version */
        unsigned long entry;        /* Offset of first executable instruct
                                       with text segment from beginning of
        unsigned long data_start;   /* Offset of data segment from beginni
                                       file */
        unsigned long data_end;     /* Offset of end of data segment
                                       from beginning of file */
        unsigned long bss_end;      /* Offset of end of bss segment from b
                                       of file */

        /* (It is assumed that data_end through bss_end forms the bss segm

        unsigned long stack_size;   /* Size of stack, in bytes */
        unsigned long reloc_start;  /* Offset of relocation records from
                                       beginning of file */
        unsigned long reloc_count;  /* Number of relocation records */
        unsigned long flags;
        unsigned long filler[6];    /* Reserved, set to zero */
};
```

Listing 1 : The bFLT header structure extracted from flat.h

Following the segments' start and end pointers comes the stack size field specified in bytes normally set to 4096 by the m68k bFLT converters and can be changed by passing an argur to the elf2flt / coff2flt utility.

The next two fields specify the details of the relocations. Each relocation is a long (32 bits) w relocation table following the data segment in the flat binary file. The relocation entries are di per bFLT version.

Version 2 specified a 2 bit type and 30 bit offset per relocation. This causes a headache wit endianess problems. The 30 bit relocation is a pointer relative to zero where an absolute ad

used. The type indicates whether the absolute address points to .text, .data or .bss.

```
#define FLAT_RELOC_TYPE_TEXT 0
#define FLAT_RELOC_TYPE_DATA 1
#define FLAT_RELOC_TYPE_BSS 2

struct flat_reloc {
#if defined(__BIG_ENDIAN_BITFIELD) /* bit fields, ugh ! */
        unsigned long type : 2;
        signed long offset : 30;
#elif defined(__LITTLE_ENDIAN_BITFIELD)
        signed long offset : 30;
        unsigned long type : 2;
#endif
```

Listing 2 : Version 2 relocation structures - Not for use in newcode.

This enables the flat loader to fix-up absolute addressing at runtime by jumping to the absolu address specified by the relocation entry and adding the loaded base address to the existin address.

Version 4 removed the need of specifying a relocation type. Each relocation entry simply co pointer to an absolute address in need of fix-up. As the bFLT loader can determine the lengt .text segment at runtime (data_start - entry) we can use this to determine what segment the r is for. If the absolute address before fix-up is less that the text length, we can safety assume relocation is pointing to the text segment and this add the base address of this segment to th absolute address.

On the other hand if the absolute address before fix-up is greater than the text length, then th absolute address must be pointing to .data or .bss. As .bss always immediately follows the d segment there is no need to have a distinction, we just add the base address of the data seg and subtract the length of the text segment. We subtract the text segment as the absolute ad version 4 is relative to zero and not to the start of the data segment.

Now you could say we may take it one step further. As every absolute address is referenced we can simply add the base address of the text segment to each address needing fix-up. Th be true if the data segment immediately follows the text segment, but we now have complica msep-data where the text segment can be in ROM and the data segment in another location Therefore we can no longer assume that the .data+.bss segment and text segment will imme follow each other.

The last defined field in the header is the flags. This appeared briefly in some version 2 head (Typically ARM) but was cemented in place with version 4. The flags are defined as follows

```
#define FLAT_FLAG_RAM    0x0001 /* load program entirely into RAM */
#define FLAT_FLAG_GOTPIC 0x0002 /* program is PIC with GOT */
#define FLAT_FLAG_GZIP   0x0004 /* all but the header is compressed */
```

Listing 3 : Newflags defined in Version 4

Early version 2 binaries saw both the .text and .data segments loaded into RAM regardless. (eXecute in Place) was later introduced allowing programs to execute from ROM with only th segment being copied into RAM. In version 4, it is now assumed that each binary is loaded f ROM if GOTPIC is true and FLAT_FLAG_GZIP and FLAT_FLAG_RAM is false. A binary ca forced to load into RAM by forcing the FLAT_FLAG_RAM flag.

The FLAT_FLAG_GOTPIC informs the loader that a GOT (Global Offset Table) is present at of the data segment. This table includes offsets that need to be relocated at runtime and thus XIP to work. (Data is accessed through the GOT, thus relocations need not be made to the . residing in ROM.) The GOT is terminated with a -1, followed immediately by the data segme

The FLAT_FLAG_GZIP indicates the binary is compressed with the GZIP algorithm. The he left untouched, but the .text, .data and relocations are compressed. Some bFLT loaders do n support GZIP and will report an error at loading.

Acknowledgments to Pauli from Lineo for pointing out some minor errors.