NetWinder ELF Design Notes

Pat Bierne, `patb@corel.com` $Revision: 1.3 $, $Date: 1999/12/07 16:24:04 $

This document describes the ELF file loading and ELF relocations as implemented on ARM Linux for the NetWinder.

1 Overview

In the old days, applications were build by compiling many .c files into .o files. These files often had inter-related references that weren't resolved at compile time. The information on these references are stored in a reloc (relocation) object.

Later, at link time, the linker would merge all the .o files, building a table of where symbols are ultimately located. Then the linker would run through the set of relocs, filling them in.

A reloc consists of three parts: where in memory the fix is to be made the symbol which is involved in the fix an algorithm that the linker should use to create the fixup

The most interesting part of this paper is the latter element. The algorithm can be as simple as "use the memory location; store it in binary" (R_386_PC32 for example). Or it may be more complicated, such as "calculate the distance from here to the symbol, divide by 4, subtract 2 and add the result to the 3 lower bytes" (R_ARM_PC26 for example).

These relocs are scattered through the .o files, and are used at link time create the correct binary file. Once all the relocs are resolved, the linker is pretty well done its job.

At least this is the way things used to work, in the days of static linking.

With the introduction of run-time linking, the designers of the ELF format decided that relocs are a suitable entity to hold run-time resolution information. So now we have executable files which still have relocs in them, even after linking.

However, new algorithms are required to signal how these fixups are to be done. Hence the introduction of a new family of reloc numbers (i.e. algorithms)

The appendix of this paper analyses the existing i386 ELF relocs. [After bringing the whole ArmLinux ELF system up, it seems to me that the best design for ArmLinux is to mimic the i386 design, with a one-to-one correspondance of relocs]

2 Relocs and Memory Space

One of the targets of the ELF binary system is a separation of code and data. The code of apps and libraries is marked read-only and executable. The data is marked read-write, and not-executable.

The code is read-only so that multiple processes can use the code, having loaded the code into memory only once. Each process has its own page tables, mapping the code into its own memory. The code is NEVER modified, and appears identical in each process space. Naturally, the code must be position independent.

The code segment is allows to contain constant pointers and strings (.rodata).

The data segment is read-write and is mapped into each process space differently. [In Linux, each data segment is loaded from the same base mmap, but it is marked copy-on-write, so after the first write, each process has its own copy of the data.]

The data segment is where relocs can be realized.

This half-and-half nature of ELF binaries leads us to an interesting design point. Some of the relocs that we wish to make are in the data segment. These are easy to do: we can add relative offsets, or write absolute addresses with no problem. But the fixups in the code area are more difficult. The ELF reloc design forces us to make the code relocs "bounce off" an entry in the data area, known as the GOT (global offset table).

In other words, if code needs to refer to a global object, it instead refers to an entry in the GOT, and at run-time, the GOT entry is fixed-up to point to the intended data. In this manner, the code space need never be fixed-up at run time. If the code needs to refer to a local object, it refers to it "relative to the &GOT[0]"; this is position independent.*

Finally, ELF implements run time linking by deferring function resolution until the function is called. This means that calls to library functions go through a fixup process the first time that they are called.

*NOTE 1: Relative (GOTOFF) code is made "relative to the start of the GOT table". Instead, it could have been made "relative to the load address of the module", which would have been cleaner in my opinion. But there are reasons that other architectures chose the former, so we'll stick with it.

3 Reloc Design

Relocs are used in many places in the design cycle: a) in .o files intended for executables b) in .o files intended for shared libraries c) in executables d) in shared libraries (.so files)

a) Object files need to be able to reference external symbols. In modern architectures, we can usually get away with: a-i) relative, from "here" to a symbol (R_ARM_PC26) a-ii) abolute, to a symbol (R_ARM_32)

NOTE 2 see below

b) Object files which are going to be part of a library are a little different. For one thing, they must be compiled as PIC code. Next, there must be a distinction between local data/functions and global data/functions. Finally, relocs in the code/.rodata sections must use got-type relocs, because the code/.rodata area of the final libary file cannot be modified at run time. A choice of relocs might be:

in code: b-i) reference to local symbol: use the relative distance from the GOT to the local symbol (R_ARM_GOTOFF) b-ii) reference to a global symbol: create an entry in the GOT and let the run-time system deposit the symbol's address into the GOT for us (R_ARM_GOT32)

in data: b-i) reference to symbol (R_ARM_32) [NOTE: symbols which are global have a reloc that references the symbol by name; symbols which are local can have a reloc that simply references the section number, and have a section-offset contained in the reloc. See NOTE 2]

c) Executables need to be able to refer to global data (such as errno) as if there is only one copy. ELF systems do this by copying global symbols down into the application .bss space. Then the executable and all the libraries point to this single copy. To realize this, we need relocs: c-i) reach into a library to a symbol and copy down the data into our own .bss space (R_ARM_COPY) c-ii) pointer to global data (R_ARM_GLOBL_DAT) c-iii) pointer to library function (R_ARM_JMP_SLOT) Notice that all of these relocs must modifiy only the data section of the executable; the code section is read-only!

d) Shared libraries are the most complex. By the time the library is linked, all the R_ARM_GOTOFF relocs are resolved. d-i) All the R_ARM_GOT32 relocs are resolved, pointing at GOT[] entries. At link time, these GOT[] entries get relocs of their own, pointing to the global data/function. (R_ARM_GLOB_DAT/R_ARM_JMP_SLOT respectively). d-ii) There will be times when data structures need to hold absolute pointers to local data. Put the module-relative address of the symbol in the library; at run-time, add the module-load address to it (R_ARM_RELATIVE)

NOTE 3

Again, notice that all of these relocs must modifiy only the data section of the executable; the code section is read-only!

When the linker creates c) and d) above, the linker actually creates code and data that was not explicit in the .o files. There is a .plt section created in the code segment, which is an array of function stubs used to handle the run-time resolution of library calls. In libraries, there is a .got section created in the data segment, which holds pointers to global symbols. Both of these synthetic sections are "helpers" to the code segment, since the code segment cannot be modified at run-time.

To make all this happen, the object files must contain information about whether a symbol is global or local, function or data, and the object size. (The old a.out scheme did not require all this extra info)

NOTE 2 At this point, I'll mention that global relocs must neccessarily involve the three aspects of a reloc:

where in memory the reloc is to be made the symbol involved in the reloc the algorithm used to make the fixup.

However, if the symbol is local, and can be fixed in memory with respect to a memory "section", the object file is allowed to drop the symbol name, and replace it with a section-plus-offset.

For instance, in this ARM code:

```
        .section .text
        mov r0, r0              @sample code
.L2:    call _do_something
        ldr r6, .L3             @this code need a reloc!
        mov r0, r0
.L4:    .word Lextern
.L3:    .word .L2               @this read-only data needs a reloc
```

The code on the 4th line needs to be fixed up, but that's easy, since it's a PC relative fixup.

If the .o file has no idea where .Lextern is, it must neccessarily create a reloc which refers to symbol Lextern.

```
.L4     .word   0
        R_ARM_32        Lextern
```

The word at .L3 needs a fixup as well. If the .o file can determine the location of a local symbol, such as L2, then it is allowed to replace the symbol with a section-plus-offset. The offset is stored in the reloc target address, and the section is an entry in the reloc symbol table

```
.L3     .word   4
        R_ARM_32        .text
```

This reduces the number of symbols in the symbol table, making run-time linking easier.

NOTE 3 Notice that the R_ARM_GOTOFF and R_ARM_GOT32 relocs include an offset from &GOT[0], which is usually about halfway through the module. The R_ARM_RELATIVE relocs, on the other hand, contains an offset from the beginning of the module. Why? Tradition.

4 Jump Tables

As much as possible, ELF dynamic linking defers the resolution of jump/call addresses until the last minute. The technique is inspired by the i386 design, and is based on the following constraints.

1) The calling technique should not force a change in the assembly code produced for apps; it MAY cause changes in the way assembly code is produced for pic-code (i.e. libraries)

2) The technique must be such that all executable areas must not be modified; and any modified areas must not be executed.

To do this, there are three steps involved in a typical jump: 1) in the code 2) through the PLT 3) using a pointer from the GOT

When the executable or library is first loaded, the GOT entry points to code which implements dynamic name resolution and code finding. On the first invocation, the function is located and the GOT entry is replaced by the address of the real functon. Subsequent calls go through 1)-2)-3) and end up calling the real code.

1) In the code:

```
        b/bl    function_call
```

This is typical ARM code using the 26 bit relative jump or call. The target is an entry in the PLT. Note that this call is identical to a normal call.

2) In the PLT:

The PLT is a synthetic area, created by the linker. It exists in both executables and libraries. It is an array of stubs, one per imported function call. It looks like this:

```
PLT[n+1]:
        ldr     ip, 1f          @load an offset
        add     ip, pc, ip      @add the offset to the pc
        ldr     pc, [ip]        @jump to that address
1:      .word   GOT[n+3] - .
```

The add on the second line makes ip = &GOT[n+3], which contains either a pointer to PLT[0] (the fixup trampoline) or a pointer to the actual code.

The first PLT entry is slightly different, and is used to form a trampoline to the fixup code.

```
PLT[0]:
        str     lr, [sp, #-4]!  @push the lr
        ldr     lr, [pc, #16]   @load from 6 words ahead
```

```
add     lr, pc, lr      @form an address
ldr     pc, [lr, #8]!   @jump to the contents of that addr
```

The lr is pushed on the stack and used for calculations. The load on the second line loads lr with &GOT[3] - . - 20. On the third line, the addition leaves

```
lr = (&GOT[3] - . - 20) + (. + 8)
lr = (&GOT[3] - 12)
```

On the fourth line, the pc and lr are both updated, so that

```
pc = GOT[2]
lr = &GOT[2]
```

3) In the GOT: The GOT (global offset table) contains helper pointers for both PLT fixups and GOT fixup. The first 3 entries are special. The next M entries belong to the PLT fixups. The next D entries belong to various data fixups.

The GOT is also a synthetic area, created by the linker. It exists in both executables and libraries.

When the GOT is first set up, all the GOT entries relating to PLT fixups are pointing to code back at PLT[0].

The special entries in the GOT are:

GOT[0] = linked list pointer used by the dyn-loader GOT[1] = pointer to the reloc table for this module GOT[2] = pointer to the fixup/resolver code

The first invocation of function call comes through and uses the fixup/resolver code. On the entry to the fixup/resolver code:

```
ip = &GOT[n+3]
lr = &GOT[2]
stack[0] = lr of the function call
[r0, r1, r2, r3 are still caller data]
```

This is enough information for the fixup/resolver code to work with. Before the fixup/resolver code returns, it actually calls the requested function and repairs &GOT[n+3]

NOTE: PLT[0] borrows an offset .word from PLT[1]. I know this is a little "tight", but allows us to keep all the PLT entries the same size.

5 Memory & Load Addresses

In a typical Linux system, the addresses 0-3fff.ffff (3 gigs) are available for the user program space.

Exectuable binary files include header information that indicates a load address. Libraries, because they are position-independent, don't need a load address, but contain a 0 in this field.

Our proposed design has normal executables loading like this:

```
Start           Len     Usage
0               4k      zero page
0000.1000       32M     not used
0200.0000       960M    app code/data space
                        after the app is the small malloc space
(sys_brk)
4000.0000       1G      mmap space
                        includes library load space (code & data)
                        & large malloc space
8000.0000       1G      stack space, working down from bfff.ffe0
```

The kernel has a preferred location for mmap data objects, at 0x4000.0000. Since the libraries are loaded by mmap, they end up here.

The library that we are using for malloc handles small mallocs by calling sys_brk(), which extends the data area after the app, at 0x0200.0000+sizeof(app). Large mallocs are realized by creating a mmap, so these end up in the pool at 0x4000.0000.

As the mmap pool grows upward, the stack grows downward. Between them, they share 2G bytes.

There is a separate case. The shared library design usually has the app loading first, then the loader notices that it need support, and loads the dyn-loader library (ld.so.1 or ld-linux.so.1) at 0x4000.0000. Other libraries are loaded after ld.so.1.

There is a diagnostic case where the app is invoked by

```
ld.so.1 foo_app foo_arg ....
```

In this case, the ld.so.1 is loaded as an app. Since it is a library, it tries to load a 0. In ArmLinux, this is forbidden, so the kernel pushes it up to 0x1000. Once ld.so.1 loads, it reads it argv[1] and loads the foo_app at its preferred location (0x0200.0000). Other libraries are loaded up at the mmap area. So, in this case, the user memory map appears as:

```
Start           Len     Usage
0               4k      zero page
0000.1000       32M     ld.so.1
                        after it the small malloc space (sys_brk)
0200.0000       960M    app code/data space
4000.0000       1G      mmap space
                        includes library load space (code & data)
                        & large malloc space
8000.0000       1G      stack space, working down from bfff.ffe0
```

Notice that the small malloc space is much smaller in this case, but this is supposed to be for load-testing and diagnostics, so it's not too bad.

6 Appendix

6.1 Analysis of the Intel i386 ELF Relocation Design

in .o files; these are the old relocs......

Reloc Number Reloc Name Meaning 1 R_386_32 simply deposit the absolute memory of "symbol" into a dword 2 R_386_PC32 determine the distance from this memory location to the "symbol", then read this dword and add it to said distance deposit the result back into this dword; this is a relative jump or call

These four were introduced with dynamic libraries; they are found only in .o files which are going to be part of a library (pic code):

Reloc Number Reloc Name Meaning 3 R_386_GOT32 this reloc is going to persist through the link stage; the linker should change this reloc into a R_386_GLOB_DATA in the library file a R_386_GOTPC determine the distance from here to the _GLOBAL_OFFSET_TABLE and deposit the difference as a dword into this location 9 R_386_GOTOFF determine the distance from the .got to the "symbol" (local symbol) store that distance as a dword at this location; create an entry in the .got table; change this reloc into a R_386_RELATIVE and point it at the .got entry 4 R_386_PLT32 create a new entry in the .plt table and .got; determine the distance from here to the .plt entry, and store that distance as a dword at this location; rename the reloc to R_386_JMP_SLOT (still the same "symbol") and point it at the .got entry

Executable files that are built "static" have no relocs in them. They run standalone.

In executable files which are intended to run with shared libraries......

Reloc Number Reloc Name Meaning 7 R_386_JMP_SLOT at load time, deposit &.plt[0] into this dword at dynamic link time, deposit the address of the "symbol" subroutine into this dword 5 R_386_COPY read a string of bytes from the "symbol" address and deposit a copy into this dword; the "symbol" object contains the length; this is used to copy initialized data from a library to the main app data space

In dynamic library files...

Reloc Number Reloc Name Meaning 7 R_386_JMP_SLOT resolved as above 6 R_386_GLOB_DATA at load time, deposit the address of "symbol" into this dword; the "symbol" is in the main app; this is, in a sense, the complement of R_386_COPY above* 8 R_386_RELATIVE at dynamic-link time: read the dword at this location; add to it the run-time start address of this module; deposit the result back into the dword

Note that R_386_32 relocs can appear in libraries as well. These must be executed carefully!

*The reason I phrased it this way is the following. Suppose you have a global data object defined in a dynamic library. The library will have the binary version of the object in its .data space. When the application is built, the linker puts a R_386_COPY reloc in there to copy the data down to the application's .bss space. In turn, the library never references the original global object; it references the copy that is in the application space, through a corresponding R_386_GLOB_DATA. Wierd, huh? After loading, the original data is never used; only the copy.

To make the whole dynamic linking operation happen, the linker introduces several "synthetic" constructs into the target when you build an app or a library:

.got Global Offset Table This is a small section of data memory where run-time fixups are made; there is only one of these per-app or per-library

_GLOBAL_OFFSET_TABLE_ A pointer to the .got

.plt Procedure Lookup Table This is a small section of code memory which helps the run-time resolution work properly

The compiler can signal to the assembler that it wants to trigger one of the above constructs by:

implicit func i386 syntax ARM syntax .got pointer var&GOT(%ebs) var(GOT) .got data var&GOTOFF(%ebx) var(GOTOFF) _GLOBAL_OFFSET_TABLE_ same same .plt func&PLT func(PLT)

Note that the C/C++ programmer does not allocate this memory; it is created by, and used by the linker.

To make the job of the linker a bit easier, the relocs are clustered together in the app-file or the library-file.

.rel.bss section contains all the R_386_COPY type relocs .rel.plt section contains all the R_386_JMP_SLOT type relocs .rel.got section contains all the R_386_GLOB_DATA type relocs .rel.data section contains all the R_386_32 and R_386_RELATIVE type relocs

7 Miscellaneous

7.1 Author

The original author of the ELF Notes is Pat Bierne of Corel Corporation.

The maintainer of the NetWinder ELF Notes is Scott Bambrough (scottb@netwinder.org). Please send any comments, additions, or corrections so they can be included in the next release. The latest version of this document may be obtained from `http://www.netwinder.org/~scottb/notes/Elf-Notes.html` .

7.2 History

May 7, 1998: The original version of these notes was in the form of an email from Pat Bierne.

October 21, 1999 (version 1.0): Converted email text to SGML, and cleaned up the content.

7.3 Copyright Notice