Position-Independent Code

Position-independent code, or *PIC*, is the name of the code generation technique used in the PowerPC environments that allows the dynamic linker to load a region of code at a non-fixed virtual memory address. Without some form of position-independent code generation, the operating system would need to place all code you wanted to be shared at fixed addresses in virtual memory, which would make maintenance of the operating system remarkably difficult. For example, it would be nearly impossible to support shared libraries and frameworks because each one would need to be preassigned an address that could never change.

Mach-O position-independent code design is based on the observation that the `__DATA` segment is always located at a constant offset from the `__TEXT` segment. That is, the dynamic loader, when loading any Mach-O file, never moves a file's `__TEXT` segment relative to its `__DATA` segment. Therefore, a function can use its own current address plus a fixed offset to determine the location of the data it wishes to access. All segments of a Mach-O file, not only the `__TEXT` and `__DATA` segments, are at fixed offsets relative to the other segments.

Note: If you are familiar with the Executable and Linking Format (ELF), you may note that Mach-O position-independent code is similar to the GOT (global offset table) scheme. The primary difference is that Mach-O code references data using a direct offset, while ELF indirects all data access through the global offset table.

Eliminating Position-Independent Code References

Position-independent code is typically required for shared libraries and bundles to allow the dynamic loader to relocate them to different addresses at load time. However, it is not required for applications that typically reside at the same address in virtual memory. GCC 3.1 introduces a new option, called `-mdynamic-no-pic`. This option both reduces the code size of application executables and improves their performance by eliminating position-independent code references, while preserving indirect calls to shared libraries and indirection to undefined symbols. If you use Xcode to create your application, this option is enabled by default. For an example of dynamic code generated without PIC, see Listing 2.

Listing 2 shows an example of the position-independent code generated for the C code in Listing 1.

Listing 1 C source code example for position-independent code

```
struct s { int member1; int member2; };


struct s bar = {1,2};


int foo(void)
{
    return bar.member2;
}
```

Listing 2 Position-independent code generated from the C example (with addresses in the left column)

```
                    .text
                    ; The function foo
                    .align 2
                    .globl _foo
0x0     _foo:       mflr r0                    ; save the link register (LR)
0x4                 bcl 20,31,L1$pb            ; Use the branch always instruction
                                               ;  that does not affect the link
                                               ;  register stack to get the address
                                               ;  of L1$pb into the LR.
0x8     L1$pb:      mflr r10                   ; then move LR to r10
0xc                 mtlr r0                    ; restore the previous LR
                                               ; bar is located at L1$pc + distance
0x10                addis r9,r10,ha16(_bar-L1$pb); L1$pb plus high 16 bits of distance
0x14                la r9,lo16(_bar-L1$pb)(r9)  ; plus low 16 of distance
                                               ; => r9 now contains address of bar
0x18                lwz r3,4(r9)               ; return bar.member2
0x1c                blr
.data
                    ; The initialized structure bar
                    .align 2
                    .globl _bar
0x20    _bar:       .long 1                    ; member1’s initialized value
0x24                .long 2                    ; member2’s initialized value
```

To calculate the address of `_bar`, the generated code adds the address of the `L1$pb` symbol (`0x8`) to the distance to `bar`. The distance to `bar` from the address of `L1$pb` is the value of the expression `_bar - L1$pb`, which is `0x18` (`0x20 - 0x8`).

Relocating Position-Independent Code

To support relocation of code in intermediate object files, Mach-O supports a section difference relocation entry format. Relocation entries are described in *OS X ABI Mach-O File Format Reference*.

Each of the add-immediate instructions is represented by two relocation entries. For the `addis` instruction (at address `0x10` in the example) the following tables list the two relocation entries. The fields of the first relocation entry (of type `scattered_relocation_info`) are:

`r_scattered`	`1`—true
`r_pcrel`	`0`—false
`r_length`	`2`—indicating 4 bytes
`r_type`	`PPC_RELOC_HA16_SECTDIFF`
`r_address`	`0x10`—the address of the `addis` instruction
`r_value`	`0x20`—the address of the symbol `_bar`

The values of the second relocation entry are:

`r_scattered`	`1`—true
`r_pcrel`	`0`—false
`r_length`	`2`—indicating 4 bytes
`r_type`	`PPC_RELOC_PAIR`
`r_address`	`0x18`—the low 16 bits of the expression (`_bar - L1$pb`)
`r_value`	`0x8`—the address of the symbol `L1$pb`

The first relocation entry for the `la` instruction (at address `0x14` in the example) is:

`r_scattered`	`1`—true
`r_pcrel`	`0`—false
`r_length`	`2`—indicating 4 bytes
`r_type`	`PPC_RELOC_LO16_SECTDIFF`
`r_address`	`0x14`—the address of the `addi` instruction
`r_value`	`0x20`—the address of the symbol `_bar`

The values of the second relocation entry are:

`r_scattered`	`1`—true
`r_pcrel`	`0`—false
`r_length`	`2`—indicating 4 bytes
`r_type`	`PPC_RELOC_PAIR`
`r_address`	`0x0`—the high 16 bits of the expression (`_bar - L1$pb`)
`r_value`	`0x8`—the address of the symbol `L1$pb`

Relocations in the x86-64 Environment

Relocations in the OS X x86-64 environment are different than relocations in other OS X environments and System V x86-64 (http://www.x86-64.org/documentation). The main differences are:.

- Scattered relocations are not used
- Compiler-generated code uses mostly external relocations
- Mach Object (Mach-O), not Executable and Linkable Format (ELF), is used as the executable file format

This section describes how relocations are implemented in the OS X x86-64 environment.

When the assembler generates relocations, if the target label is a local label (it begins with `L`), the previous nonlocal label in the same section is used as the target of the external relocation. An addend (that is, the `4` in `_foo + 4`) is used with the distance from that nonlocal label to the target label. The assembler uses an internal relocation only when there is no previous nonlocal label in the section.

The addend is encoded in the instruction (Mach-O does not have `RELA` relocations). For PC-relative relocations, the addend is stored in the instruction. This practice is different than in other OS X environments, which encode the addend minus the current section offset. The x86-64 relocation types are described in *OS X ABI Mach-O File Format Reference*.

Listing 3 shows assembly instructions and the relocation and section content that they generate.

Listing 3 Example assembly instructions and their corresponding relocations

```
call    _foo
r_type=X86_64_RELOC_BRANCH, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
E8 00 00 00 00

call    _foo+4
r_type=X86_64_RELOC_BRANCH, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
E8 04 00 00 00

movq _foo@GOTPCREL(%rip), %rax
r_type=X86_64_RELOC_GOT_LOAD, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
48 8B 05 00 00 00 00

pushq _foo@GOTPCREL(%rip)
r_type=X86_64_RELOC_GOT, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
FF 35 00 00 00 00

movl _foo(%rip), %eax
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
8B 05 00 00 00 00

movl _foo+4(%rip), %eax
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
8B 05 04 00 00 00

movb  $0x12, _foo(%rip)
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
C6 05 FF FF FF FF 12
```

```
movl  $0x12345678, _foo(%rip)
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
C7 05 FC FF FF FF 78 56 34 12

.quad _foo
r_type=X86_64_RELOC_UNSIGNED,r_length=3, r_extern=1,r_pcrel=0, r_symbolnum=_foo
00 00 00 00 00 00 00 00

.quad _foo+4
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_foo
04 00 00 00 00 00 00 00

.quad _foo - _bar
r_type=X86_64_RELOC_SUBTRACTOR,r_length=3,r_extern=1, r_pcrel=0,r_symbolnum=_bar
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1, r_pcrel=0,r_symbolnum=_foo
00 00 00 00 00 00 00 00

.quad _foo - _bar + 4
r_type=X86_64_RELOC_SUBTRACTOR,r_length=3, r_extern=1,r_pcrel=0,r_symbolnum=_bar
r_type=X86_64_RELOC_UNSIGNED,r_length=3, r_extern=1,r_pcrel=0,r_symbolnum=_foo
04 00 00 00 00 00 00 00

.long _foo - _bar
r_type=X86_64_RELOC_SUBTRACTOR,r_length=2,r_extern=1,r_pcrel=0,r_symbolnum=_bar
r_type=X86_64_RELOC_UNSIGNED,r_length=2,r_extern=1,r_pcrel=0,r_symbolnum=_foo
00 00 00 00

lea L1(%rip), %rax
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_prev
48 8d 05 12 00 00 00
// Assumes that _prev is the first nonlocal label 0x12 bytes before L1.

lea L0(%rip), %rax
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=0, r_pcrel=1, r_symbolnum=3
48 8d 05 56 00 00 00
// Assumes that  L0 is in third section, and has an address of 0x00000056
// in .o file, and no previous nonlocal label.

.quad L1
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1,r_pcrel=0, r_symbolnum= _prev
12 00 00 00 00 00 00 00
// Assumes that _prev is the first nonlocal label 0x12 bytes before L1.
```

```
.quad L0
r_type=X86_64_RELOC_UNSIGNED,r_length=3, r_extern=0, r_pcrel=0, r_symbolnum= 3
56 00 00 00 00 00 00 00
// Assumes that L0 is in third section, and has address of 0x00000056
// in .o file, and no previous nonlocal label.

.quad _foo - .
r_type=X86_64_RELOC_SUBTRACTOR,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_prev
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_foo
EE FF FF FF FF FF FF FF
// Assumes that _prev is the first nonlocal label 0x12 bytes
// before this .quad

.quad _foo - L1
r_type=X86_64_RELOC_SUBTRACTOR,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_prev
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_foo
EE FF FF FF FF FF FF FF
// Assumes that  _prev is the first nonlocal label 0x12 bytes before L1.

.quad L1 - _prev
// No relocations. This is an assembly time constant.
12 00 00 00 00 00 00 00
// Assumes that _prev is the first nonlocal label 0x12 bytes before L
```