Existing stack objects reside at fixed offsets from the frame pointer; stack heap allocation doesn't move them. No special code is needed to free dynamically allocated stack memory. The function epilogue resets the stack pointer and removes the entire stack frame, including the heap, from the stack. Naturally, a program should not reference heap objects after they have gone out of scope.

4 OBJECT FILES

ELF Header

Machine Information

For file identification in `e_ident`, the Motorola 68000 Family requires the following values.

Figure 4-1: Motorola 68000 Family Identification, `e_ident`

Position	Value
`e_ident[EI_CLASS]`	`ELFCLASS32`
`e_ident[EI_DATA]`	`ELFDATA2MSB`

The ELF header's `e_flags` member holds bit flags associated with the file. Motorola 68000 Family defines no flags; so this member contains zero. Processor identification resides in the ELF header's `e_machine` member and must have the value 4, defined as the name `EM_68K`.

Sections

Special Sections

Various sections hold program and control information. Sections in the list below are used by the system and have the indicated types and attributes.

Figure 4-2: Special Sections

Name	Type	Attributes
`.got`	`SHT_PROGBITS`	`SHF_ALLOC + SHF_WRITE`
`.plt`	`SHT_PROGBITS`	`SHF_ALLOC + SHF_EXECINSTR`

`.got`	This section holds the global offset table. See "Coding Examples" in Chapter 3 and "Global Offset Table" in Chapter 5 for more information.
`.plt`	This section holds the procedure linkage table. See "Procedure Linkage Table" in Chapter 5 for more information.

Symbol Table

Symbol Values

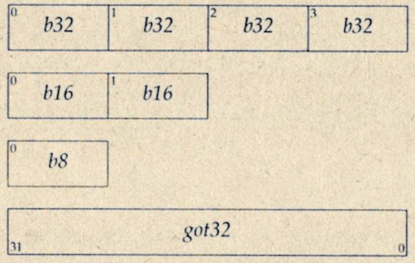
If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for that file will contain an entry for that symbol. The `st_shndx` member of that symbol table entry contains `SHN_UNDEF`. This signals to the dynamic linker that the symbol definition for that function is not contained in the executable file itself. If that symbol has been allocated a procedure linkage table entry in the executable file, and the `st_value` member for that symbol table entry is non-zero, the value will contain the virtual address of the first instruction of that procedure linkage table entry. Otherwise, the `st_value` member contains zero. This procedure linkage table entry address is used by the dynamic linker in resolving references to the address of the function. See "Function Addresses" in Chapter 5 for details.

Relocation

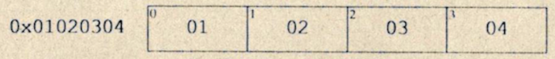
Relocation Types

Relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners; byte numbers appear in the upper box corners).

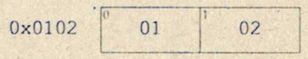
Figure 4-3: Relocatable Fields



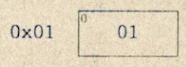
b32 This specifies a 32-bit field occupying 4 bytes with arbitrary alignment. These values use the byte order illustrated below.



b16 This specifies a 16-bit field occupying 2 bytes with arbitrary alignment.



b8 This specifies a 8-bit field occupying 1 byte with arbitrary alignment.



got32 This specifies a 32-bit field occupying 4 bytes with long word alignment. These bytes represent values in the same byte order as *b32*.

Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link editor merges one or more relocatable files to form the output. It first decides how to combine and locate the input files, then updates the symbol values, and finally performs the relocation. Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation.

A This means the addend used to compute the value of the relocatable field.

B This means the base address at which a shared object has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different.

G This means the address of the global offset table entry that will contain the address of the relocation entry's symbol during execution. See "Coding Examples" in Chapter 3 and "Global Offset Table" in Chapter 5 for more information.

G′ This means the address of entry zero in the global offset table.

L This means the place (section offset or address) of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution. See "Procedure Linkage Table" in Chapter 5 for more information.

L′ This means the address of entry zero in the procedure linkage table.

P This means the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).

S This means the value of the symbol whose index resides in the relocation entry.

A relocation entry's `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. Because the Motorola 68000 Family uses only `Elf32_Rela` relocation entries, the relocation table entry holds the addend. In all cases, the addend and the computed result use the same byte order.

Figure 4-4: Relocation Types

Name	Value	Field	Calculation
R_68K_NONE	0	none	*none*
R_68K_32	1	*b32*	S + A
R_68K_16	2	*b16*	S + A
R_68K_8	3	*b8*	S + A
R_68K_PC32	4	*b32*	S + A − P
R_68K_PC16	5	*b16*	S + A − P
R_68K_PC8	6	*b8*	S + A − P
R_68K_GOT32	7	*b32*	G + A − P
R_68K_GOT16	8	*b16*	G + A − P
R_68K_GOT8	9	*b8*	G + A − P
R_68K_GOT32O	10	*b32*	G − G′
R_68K_GOT16O	11	*b16*	G − G′
R_68K_GOT8O	12	*b8*	G − G′
R_68K_PLT32	13	*b32*	L + A − P
R_68K_PLT16	14	*b16*	L + A − P
R_68K_PLT8	15	*b8*	L + A − P
R_68K_PLT32O	16	*b32*	L − L′
R_68K_PLT16O	17	*b16*	L − L′
R_68K_PLT8O	18	*b8*	L − L′
R_68K_COPY	19	none	*none*
R_68K_GLOB_DAT	20	*got32*	S
R_68K_JMP_SLOT	21	*got32*	S
R_68K_RELATIVE	22	*b32*	B + A

Some relocation types have semantics beyond simple calculation.

`R_68K_GOT32` This relocation type resembles `R_68K_PC32`, except it refers to the address of the symbol's global offset table entry and additionally instructs the link editor to build a global offset table.

R_68K_GOT16	This relocation type resembles R_68K_PC16, except it refers to the address of the symbol's global offset table entry and additionally instructs the link editor to build a global offset table.
R_68K_GOT8	This relocation type resembles R_68K_PC8, except it refers to the address of the symbol's global offset table entry and additionally instructs the link editor to build a global offset table.
R_68K_GOT32O	This relocation type resembles R_68K_GOT32, except it refers to the address of the symbol's global offset table entry relative to the address of entry zero in the GOT.
R_68K_GOT16O	This relocation type resembles R_68K_GOT16, except it refers to the address of the symbol's global offset table entry relative to the address of entry zero in the GOT.
R_68K_GOT8O	This relocation type resembles R_68K_GOT8, except it refers to the address of the symbol's global offset table entry relative to the address of entry zero in the GOT.
R_68K_PLT32	This relocation type resembles R_68K_PC32, except it refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.
R_68K_PLT16	This relocation type resembles R_68K_PC16, except it refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.
R_68K_PLT8	This relocation type resembles R_68K_PC8, except it refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.
R_68K_PLT32O	This relocation type resembles R_68K_PLT32, except it refers to the address of the symbol's procedure linkage table entry relative to the address of entry zero in the PLT.
R_68K_PLT16O	This relocation type resembles R_68K_PLT16, except it refers to the address of the symbol's procedure linkage table entry relative to the address of entry zero in the PLT.

R_68K_PLT8O	This relocation type resembles R_68K_PLT8, except it refers to the address of the symbol's procedure linkage table entry relative to the address of entry zero in the PLT.
R_68K_COPY	This relocation type assists dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.
R_68K_GLOB_DAT	This relocation type resembles R_68K_32, except it is used to set a global offset table entry to the specified symbol's value. The relocation type allows one to determine the correspondence between symbols and global offset table entries. The relocated field should be aligned on a long word boundary. This relocation type does *not* use the addend.
R_68K_JMP_SLOT	This relocation type assists dynamic linking. Its offset member gives the location of a global offset table entry. This relocation type does *not* use the addend.
R_68K_RELATIVE	This relocation type assists dynamic linking. The addend member contains a value representing a relative address within a shared object. The offset member gives a location within the shared object for the final virtual address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

5. PROGRAM LOADING AND DYNAMIC LINKING

5 PROGRAM LOADING AND DYNAMIC LINKING

Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When—and if—the system physically reads the file depends on the program's execution behavior, system load, etc. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for Motorola 68000 Family segments are congruent modulo 8 K (`0x2000`) or larger powers of 2. Because 8 KB is the maximum page size, the files will be suitable for paging regardless of physical page size. Figure 5-1 is an example of an executable file.

Figure 5-1: Executable File

File Offset	File	Virtual Address
0	ELF header	
	Program header table	
	Other information	
`0x100`	Text segment	`0x80000100`
	...	
	`0x2be00` bytes	`0x8002beff`
`0x2bf00`	Data segment	`0x8004bf00`
	...	
	`0x4e00` bytes	`0x80050cff`
`0x30d00`	Other information ...	

Figure 5-2: Program Header Segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x80000100	0x8004bf00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x2000	0x2000

Although the example's file offsets and virtual addresses are congruent modulo 8 K for both text and data, up to four file pages hold impure text or data (depending on page size and file system block size).

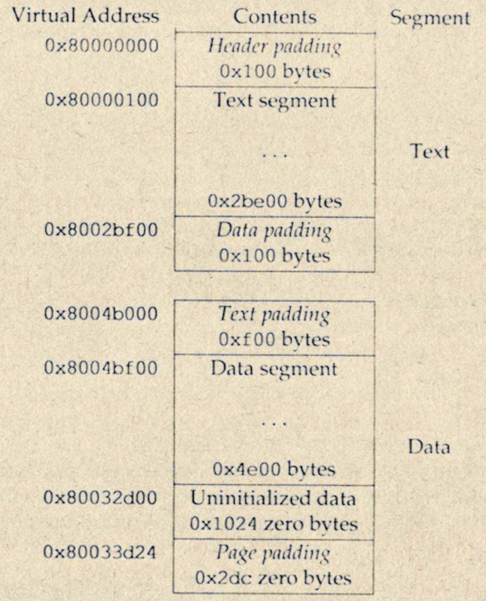
- The first text page contains the ELF header, the program header table, and other information.
- The last text page holds a copy of the beginning of data.
- The first data page has a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segments' addresses are adjusted to ensure each logical page in the address space has a single set of permissions. In the example above, the region of the file holding the end of text and the beginning of data will be mapped twice: at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file. "Impurities" in

the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for this program follows, assuming 4 KB (0x1000) pages.

Figure 5-3: Process Image Segments



One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code [see "Coding Examples" in Chapter 3]. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. Thus the system uses the p_vaddr values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This lets a segment's virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments' *relative positions*. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The following table shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning.

Figure 5-4: Example Shared Object Segment Addresses

Source	Text	Data	Base Address
File	0x200	0x2a400	0x0
Process 1	0xc0080200	0xc00aa400	0xc0080000
Process 2	0xc0082200	0xc00ac400	0xc0082000
Process 3	0xd00c0200	0xd00ea400	0xd00c0000
Process 4	0xd00c6200	0xd00f0400	0xd00c6000

Dynamic Linking

Dynamic Section

Dynamic section entries give information to the dynamic linker. Some of this information is processor-specific, including the interpretation of some entries in the dynamic structure.

DT_PLTGOT	On the 68000, this entry's `d_ptr` member gives the address of the first entry in the global offset table. As mentioned below, the first three global offset table entries are reserved, and two are used to hold procedure linkage table information.

Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and shareability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries [see "Relocation" in Chapter 4]. When the dynamic linker creates memory segments for a loadable object file, it processes the relocation entries, some of which will be type R_68K_GLOB_DAT referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol `_DYNAMIC`. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image.

The dynamic linker may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

A global offset table's format and interpretation are processor-specific. For the Motorola 68000 Family an offset into the table is an *unsigned* value, allowing only non-negative "subscripts" into the array of addresses.

Function Addresses

References to the address of a function from an executable file and the shared objects associated with it might not resolve to the same value. References from within shared objects will normally be resolved by the dynamic linker to the virtual address of the function itself. References from within the executable file to a function defined in a shared object normally will be resolved by the link editor to the address of the procedure linkage table entry for that function within the executable file.

To allow comparisons of function addresses to work as expected, if an executable file references a function defined in a shared object, the link editor will place the address of the procedure linkage table entry for that function in its associated symbol table entry. [See "Symbol Values" in Chapter 4]. The dynamic linker treats such symbol table entries specially. If the dynamic linker is searching for a symbol, and encounters a symbol table entry for that symbol in the executable file, it normally follows the rules below.

1. If the `st_shndx` member of the symbol table entry is not `SHN_UNDEF`, the dynamic linker has found a definition for the symbol and uses its `st_value` member as the symbol's address.
2. If the `st_shndx` member is `SHN_UNDEF` and the symbol is of type `STT_FUNC` and the `st_value` member is not zero, the dynamic linker recognizes this entry as special and uses the `st_value` member as the symbol's address.
3. Otherwise, the dynamic linker considers the symbol to be undefined within the executable file and continues processing.

Some relocations are associated with procedure linkage table entries. These entries are used for direct function calls rather than for references to function addresses. These relocations are not treated in the special way described above because the dynamic linker must not redirect procedure linkage table entries to point to themselves.

Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the 68000, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and sharability of the program's text. Executable files and shared object files have separate procedure linkage tables.

Figure 5-5: Initial Procedure Linkage Table

```
.PLT0:   mov.l   got_plus_4,-(%sp)
         jmp     ([got_plus_8])
         nop
         nop
.PLT1:   jmp     ([name1@GOTPC,%pc])
         mov.l   &offset,-(%sp)
         bra     .PLT0
.PLT2:   jmp     ([name2@GOTPC,%pc])
         mov.l   &offset,-(%sp)
         bra     .PLT0
```

Following the steps below, the dynamic linker and the program "cooperate" to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.
2. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT1`.
3. The first instruction jumps to the address in the global offset table entry for `name1`. Initially, the global offset table holds the address of the following instruction, not the real address of `name1`.
4. Consequently, the program pushes a relocation offset (*offset*) on the stack. The relocation offset is a 32-bit, non-negative byte offset into the relocation table. The designated relocation entry will have type `R_68K_JMP_SLOT`, and its offset will specify the global offset table entry used in the previous `jmp` instruction. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, `name1` in this case.

5. After pushing the relocation offset, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `mov.l` instruction places the value of the second global offset table entry (*got_plus_4*) on the stack, thus giving the dynamic linker one long word of identifying information. The program then jumps to the address in the third global offset table entry (*got_plus_8*), which transfers control to the dynamic linker.
6. When the dynamic linker receives control, it unwinds the stack, looks at the designated relocation entry, finds the symbol's value, stores the "real" address for `name1` in its global offset table entry, and transfers control to the desired destination.
7. Subsequent executions of the procedure linkage table entry will transfer directly to `name1`, without calling the dynamic linker a second time. That is, the `jmp` instruction at `.PLT1` will transfer to `name1`, instead of "falling through" to the next instruction.

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_68K_JMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

NOTE

Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.