System V Application Binary Interface
Motorola 68000 Supplement

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

The publisher offers discounts on this book when ordered in bulk quantities.
For more information, write:
Special Sales
Prentice-Hall, Inc.
College Technical and Reference Division
Englewood Cliffs, New Jersey 07632
or
call 201-592-2498
For single copies, call 201-767-5937

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-877663-6

1070.350

UNIX PRESS
A Prentice Hall Title

Contents

Figures and Tables

1 INTRODUCTION

Motorola 68000 Family and the System V ABI

The **System V Application Binary Interface**, or **ABI**, defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on systems that implement UNIX System V Release 4.0 or some other operating system that complies with the **System V Interface Definition, Issue 3**.

This document is a supplement to the generic **System V ABI**, and it contains information specific to System V implementations built on the Motorola 68000 processor architecture family. The generic term "Motorola 68000 family" is used in this specification to mean the Motorola MC68020, MC68030, and MC68040 processor architectures; it does not refer to the MC68000, MC68008, or MC68010.

Together, these two specifications, the generic **System V ABI** and the **Motorola 68000 Family System V ABI Supplement**, constitute a complete *System V Application Binary Interface* specification for systems that implement the architecture of the Motorola 68000 family.

How to Use the Motorola 68000 Family ABI Supplement

This document is a supplement to the generic **System V ABI** and contains information referenced in the generic specification that may differ when System V is implemented on different processors. Therefore, the generic **ABI** is the prime reference document, and this supplement is provided to fill gaps in that specification.

As with the **System V ABI**, this specification references other publicly-available reference documents, including the

- **MC68020 32-Bit Microprocessor User's Manual**, MC68020UM/AD
- **MC68030 Enhanced 32-Bit Microprocessor User's Manual**, MC68030UM/AD
- **MC68040 32-Bit Microprocessor User's Manual**, MC68040UM/AD
- **M68000 Programmer's Reference Manual**, M68000PM/AD

available from Motorola.

All the information referenced by this supplement should be considered part of this specification, and just as binding as the requirements and data explicitly included here.

Evolution of the ABI Specification

The **System V Application Binary Interface** will evolve over time to address new technology and market requirements, and will be reissued at intervals of approximately three years. Each new edition of the specification is likely to contain extensions and additions that will increase the potential capabilities of applications that are written to conform to the **ABI**.

As with the **System V Interface Definition**, the **ABI** will implement **Level 1** and **Level 2** support for its constituent parts. **Level 1** support indicates that a portion of the specification will continue to be supported indefinitely, while **Level 2** support means that a portion of the specification may be withdrawn or altered after the next edition of the **ABI** is made available. That is, a portion of the specification moved to **Level 2** support in an edition of the **ABI** specification will remain in effect at least until the following edition of the specification is published.

These Level 1 and Level 2 classifications and qualifications apply to this Supplement, as well as to the generic specification. All components of the **ABI** and of this supplement have Level 1 support unless they are explicitly labeled as Level 2.

3 LOW-LEVEL SYSTEM INFORMATION

Machine Interface

Processor Architecture

The *MC68020 32-Bit Microprocessor User's Manual*, the *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, and the *MC68040 32-Bit Microprocessor User's Manual* define the 68000 family processor architecture. The *M68000 Programmer's Reference Manual* defines the programming model. An MC68851 Paged Memory Management Unit (PMMU) may be present with the MC68020. An MC68881 or MC68882 Floating Point Coprocessor (FPCP) is assumed to be present with the MC68020 or MC68030. Programs intended to execute directly on the processor use the instruction set, instruction encodings, and instruction semantics of this architecture. A program may use only the instructions defined for the MC68040. Refer to the *M68000 Programmer's Reference Manual* for information regarding proper instruction usage for the MC68020, MC68030, and MC68040 processors.

To be ABI-conforming, the processor must implement the architecture's instructions, perform the specified operations, and produce the specified results. The ABI neither places performance constraints on systems nor specifies what instructions must be implemented in hardware. A software emulation of the architecture could conform to the ABI.

Some processors might support the 68000 and MC68881/2 FPCP architectures as subsets, providing additional instructions or capabilities. Programs that use those capabilities explicitly do not conform to the 68000 ABI. Executing those programs on machines without the additional capabilities gives undefined behavior.

Data Representation

Fundamental Types

Figure 3-1 shows the correspondence between ANSI C's scalar types and the processor's.

Figure 3-1: Scalar Types

Type	C	sizeof	Alignment (bytes)	68000
Integral	`signed char` `char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed word (2 bytes)
	`unsigned short`	2	2	unsigned word

	`int`
	`signed int`
	`long`
	`signed long`
	`enum`
		4	4	signed long word (4 bytes)

	`unsigned int` `unsigned long`	4	4	unsigned long word
Pointer	*any-type* `*` *any-type* `(*)()`	4	4	unsigned long word
Floating-point	`float`	4	4	single-precision
	`double`	8	8	double-precision
	`long double`	16	8	extended-precision

A null pointer (for all types) has the value zero.

NOTE

If a `double` or `long double` appears on the stack, not as a part of an aggregate or union, then it is aligned to 4 bytes.

Aggregates and Unions

An array assumes the alignment of its elements' type. The size of any object, including arrays, structures, and unions, always is a multiple of the object's alignment. Structure and union objects may, therefore, require padding to meet size and alignment constraints.

- The alignment of a structure or a union is the maximum of the alignment of its elements.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require *internal padding*, depending on the previous member.
- A structure's size is increased, if necessary, to make it a multiple of the structure's alignment. This may require *tail padding*, depending on the last member.

NOTE

Aggregates or unions residing on the stack only require 4-byte alignment.

In the following examples, members' byte offsets appear in the upper left corners.

Figure 3-2: Structure Smaller Than a Long Word

```
struct {
    char    c;
};
```

Byte aligned, `sizeof` is 1

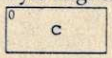


Figure 3-3: No Padding

```
struct {
    char   c;
    char   d;
    short  s;
    long   n;
};
```

Long word aligned, `sizeof` is 8

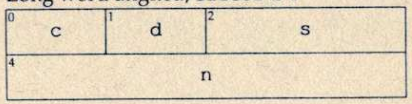


Figure 3-4: Internal Padding

```
struct {
    char   c;
    short  s;
};
```

Word aligned, `sizeof` is 4

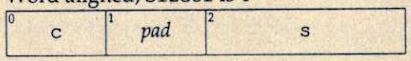


Figure 3-5: Internal and Tail Padding

```
struct {
    char   c;
    double d;
    short  s;
};
```

8-byte aligned (4-byte on stack), `sizeof` is 24

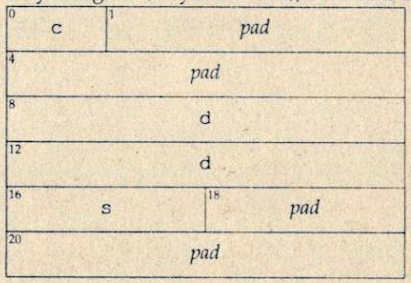
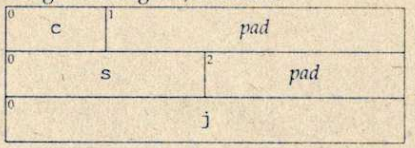


Figure 3-6: `union` Allocation

```
union {
    char   c;
    short  s;
    int    j;
};
```

Long word aligned, `sizeof` is 4



Bit-Fields

C `struct` and `union` definitions may have *bit-fields*, defining integral objects with a specified number of bits.

Figure 3-7: Bit-Field Ranges

Bit-field Type	Width w	Range
`signed char`	1 to 8	-2^{w-1} to $2^{w-1}-1$
`char`		0 to $2^{w}-1$
`unsigned char`		0 to $2^{w}-1$
`signed short`	1 to 16	-2^{w-1} to $2^{w-1}-1$
`short`		0 to $2^{w}-1$
`unsigned short`		0 to $2^{w}-1$
`signed int`	1 to 32	-2^{w-1} to $2^{w-1}-1$
`int`		0 to $2^{w}-1$
`enum`		0 to $2^{w}-1$
`unsigned int`		0 to $2^{w}-1$
`signed long`	1 to 32	-2^{w-1} to $2^{w-1}-1$
`long`		0 to $2^{w}-1$
`unsigned long`		0 to $2^{w}-1$

"Plain" bit-fields always have non-negative values. Although they may have type `char`, `short`, `int`, or `long` (which can have negative values), these bit-fields are extracted into a long word with zero fill. Bit-fields obey the same size and alignment rules as other structure and union members, with the following additions.

- Bit-fields are allocated from left to right (most to least significant).
- A bit-field must entirely reside in a storage unit appropriate for its declared type. Thus a bit-field never crosses its unit boundary.
- Bit-fields may share a storage unit with other `struct`/`union` members, including members that are not bit-fields. Of course, `struct` members occupy different parts of the storage unit.
- Unnamed bit-fields' types do not affect the alignment of a structure or union, although individual bit-fields member offsets obey the alignment constraints.

The following examples show `struct` and `union` members' byte offsets in the upper left corners; bit numbers appear in the lower corners.

Figure 3-8: Bit Numbering

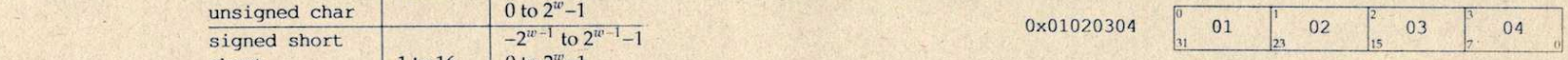


Figure 3-9: Left-to-Right Allocation

```
struct {
    int    j:5;
    int    k:6;
    int    m:7;
};
```

Long word aligned, `sizeof` is 4

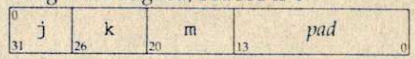


Figure 3-10: Boundary Alignment

```
struct {
    short  s:9;
    int    j:9;
    char   c;
    short  t:9;
    short  u:9;
    char   d;
};
```

Long word aligned, `sizeof` is 12

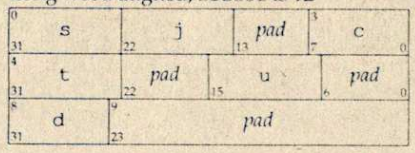


Figure 3-11: Storage Unit Sharing

```
struct {
    char    c;
    short   s:8;
};
```

Word aligned, sizeof is 2

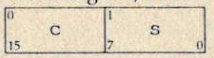


Figure 3-12: `union` Allocation

```
union {
    char    c;
    short   s:8;
};
```

Word aligned, sizeof is 2

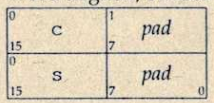
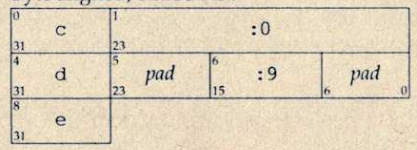


Figure 3-13: Unnamed Bit-Fields

```
struct {
    char    c;
    int     :0;
    char    d;
    short   :9;
    char    e;
    char    :0;
};
```

Byte aligned, sizeof is 9



As the examples show, `int` bit-fields (including `signed` and `unsigned`) pack more densely than smaller base types. One can use `char` and `short` bit-fields to force particular alignments, but `int` generally works better.

Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, parameter passing, etc. The operating system interface and C programs use this calling sequence. See "Coding Examples" later in this chapter for more information on C.

NOTE All of the coding examples in this section are simply illustrations of a sample implementation.

Registers and the Stack Frame

The 68000 provides 8 data and 8 address registers, which are global to a running program. Additionally, the MC68040, or the MC68881/2 Floating Point Coprocessor, provides 8 global floating-point registers. Brief register descriptions appear in Figure 3-14; more complete information appears later.

Figure 3-14: Processor Registers

Type	Names		Usage
68000	%d0 %d1 %a0 %a1		Scratch registers
	%d2 ... %d7 %a2 ... %a5		Local register variables
	%a6	%fp	Frame pointer (if implemented)
	%a7	%sp	Stack pointer
		%pc	Program counter
		%ccr	Condition code register
MC68040, MC68881/2	%fp0 %fp1		Scratch registers
	%fp2 ... %fp7		Local register variables
		%fpcr	Floating Point Control Register
		%fpsr	Floating Point Status Register
		%fpiar	Floating Point Instruction Address Register

In addition to the registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Figure 3-15 shows the stack frame organization right after function prologue processing has allocated the frame and saved the registers (see below).

Figure 3-15: Standard Stack Frame

Base	Offset	Contents	Purpose	
	+4+4*n	argument long word *n*	incoming arguments	*High Addresses*
		...		
	+4	argument long word 0		
		return address		
%fp	(optional)	previous %fp (optional)		
		unspecified	local storage and register save area	
		...		
SP	(SP after prologue)	variable size		
		stack top, unused		*Low addresses*

Several key points about the stack frame deserve mention.

- The stack frame is long word aligned.
- Functions may save registers %d2 through %d7, %a2 through %a6, and %fp2 through %fp7 *as necessary*, without saving unused registers.
- All incoming arguments reside on the stack. "Coding Examples" below explains how variable argument lists may be implemented.
- A frame pointer may be implemented.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size, nor does it restrict how a language system uses the "unspecified" areas of the standard stack frame.

Registers %d2 through %d7, %a2 through %a6, and %fp2 through %fp7, which are visible to both a calling and a called function, "belong" to the calling function. In other words, a called function must save these registers' values before it changes them, restoring their values before it returns. Remaining registers "belong" to the called function. If a calling function wants to preserve such a register value

across a function call, it must save the value in its local stack frame.

Across function boundaries, the standard function prologue saves registers %d2 through %d7, %a2 through %a6, and %fp2 through %fp7 (as needed) and allocates stack space, including the required areas of Figure 3-15 and any private space it needs. The example below illustrates this, saving registers %fp, %d7, %a5, and %fp2 and allocating 80 bytes for local storage.

Figure 3-16: Function Prologue

```
fcn:
        link.l    %fp,&-80
        movm.l    %d7/%a5,-(%sp)
        fmovm.x   %fp2,-(%sp)
```

The movm instruction manipulates registers as part of the normal function prologue and epilogue. As explained later, the function epilogue executes a movm instruction to unwind the stack and restore the saved registers to their original condition.

NOTE

Strictly speaking, a function does not need the movm instructions if it preserves the registers as described above. Although some functions can be optimized to eliminate the movm instructions, the general case uses the standard prologue and epilogue.

Some registers have assigned roles.

%fp or %a6 — The *frame pointer*, if used by an individual function, holds the address of the local storage within a stack frame, referenced as negative offsets from %fp.

%sp or %a7 — The *stack pointer* holds the limit of the current stack frame, which is the address of the stack's topmost valid long word. The long word to which %sp points is "in" the valid stack.

%d0	*Integral return values* appear in %d0.
%a0	*Pointer return values* appear in %a0. When calling a function that returns a structure or union, the caller allocates space for the return value and sets %a0 to its address. A function that returns a structure or union value places the same address in %a0 before it returns.
%fp0	*Floating-point return values* appear in this register.

Except as specified here, %d0, %d1, %a0, %a1, %fp0, and %fp1 are scratch registers. Functions do not need to preserve their values for the caller.

Local registers %d2 through %d7, %a2 through %a5, and %fp2 through %fp7 have no specified role in the standard calling sequence.

Signals can interrupt processes [see signal(BA_OS)]. Functions called during signal handling have no unusual restrictions on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus programs and compilers may freely use all registers without the danger of signal handlers changing their values.

Integral and Pointer Arguments

As mentioned, a function receives all integral and pointer argument long words on the stack. Functions pass all integer-valued arguments as long words, expanding signed or unsigned bytes and words as needed.

NOTE The following examples use the frame pointer. Functions not using %fp would find arguments at different locations.

Figure 3-17: Integral and Pointer Arguments

Call	Argument	Callee
g(1, 2, 3, (void *)0);	1	8(%fp)
	2	12(%fp)
	3	16(%fp)
	(void *)0	20(%fp)

Floating-Point Arguments

The stack also holds floating-point arguments: single-precision values use one long word, double-precision use two, and extended-precision use four. The example below uses only double-precision arguments.

Figure 3-18: Floating-Point Arguments

Call	Argument	Callee
h(1.414, 1, 2.998e10);	long word 0, 1.414	8(%fp)
	long word 1, 1.414	12(%fp)
	1	16(%fp)
	long word 0, 2.998e10	20(%fp)
	long word 1, 2.998e10	24(%fp)

Structure and Union Arguments

As described in the data representation section, structures and unions always have the alignment of their most strictly aligned member. When passed as arguments, sizes are rounded up to the next long word size. Structure and union objects appear directly on the stack, occupying as many long words as necessary.

Figure 3-19: Structure and Union Arguments

Call	Argument	Callee
`i(1, s);`	1	8(%fp)
	long word 0, s	12(%fp)
	long word 1, s	16(%fp)
	...	...

Functions Returning Scalars or No Value

A function that returns an integral value places its result in `%d0`. A function that returns a pointer value places its result in `%a0`. A function that returns a floating-point value places its result in `%fp0`.

Functions that return no value put no specified value in any return register.

Just as the function prologue may save registers, the epilogue must restore those same registers before returning to the caller. To complete the example from Figure 3-16, the following function epilogue restores `%d7`, `%a5`, and `%fp2` and then returns.

Figure 3-20: Function Epilogue

```
fmovm.x    (%sp)+,%fp2
movm.l     (%sp)+,%d7/%a5
unlk       %fp
rts
```

Functions Returning Structures or Unions

As mentioned above, when a function returns a structure or union, it expects the caller to provide space for the return value and to place its address in register `%a0`. Having the caller supply the return object's space allows re-entrancy.

NOTE

Structures and unions in this context have fixed sizes. The ABI does not specify how to handle variable sized objects.

A function returning a structure or union also sets `%a0` to the value it finds in `%a0`. Thus when the caller receives control again, the address of the returned object resides in register `%a0`. Both the calling and the called functions must cooperate to pass the return value successfully. Failure of either side to meet its obligations leads to undefined program behavior.

The following example assumes the return object has been copied, and its address is in register `%a5`.

Figure 3-21: Function Epilogue

```
mov.l     %a5,%a0
fmovm.x   (%sp)+,%fp2
movm.l    (%sp)+,%d7/%a5
unlk      %fp
rts
```


Operating System Interface

Virtual Address Space

Processes execute in a 32-bit virtual address space. Memory management hardware translates virtual addresses to physical addresses, hiding physical addressing and letting a process run anywhere in the system's real memory. Processes typically begin with three logical segments, commonly called text, data, and stack. As Chapter 5 describes, dynamic linking creates more segments during execution, and a process can create additional segments for itself with system services.

Page Size

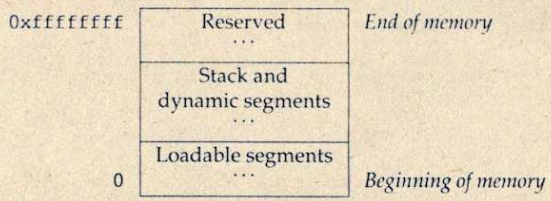
Memory is organized by pages, which are the system's smallest units of memory allocation. Page size can vary from one system to another, depending on the processor, memory management unit and system configuration. Allowable page sizes are 2K, 4K, or 8K. Processes may call `sysconf(BA_OS)` to determine the system's current page size.

Virtual Address Assignments

Conceptually, processes have the full 32-bit address space available. In practice, however, several factors limit the size of a process.

- The system reserves a configuration-dependent amount of virtual space.
- A tunable configuration parameter limits process size.
- A process whose size exceeds the system's available, combined physical memory and secondary storage cannot run. Although some physical memory must be present to run any process, the system can execute processes that are bigger than physical memory, paging them to and from secondary storage. Nonetheless, both physical memory and secondary storage are shared resources. System load, which can vary from one program execution to the next, affects the available amounts.

Figure 3-22: Virtual Address Configuration



CAUTION Programs that dereference null pointers are erroneous. Although such programs may appear to work on the 68000, they might fail or behave differently on other systems. To enhance portability, programmers are strongly cautioned not to rely on dereferencing null pointers.

Loadable segments
: Processes' loadable segments may begin at 0. The exact addresses depend on the executable file format (see Chapters 4 and 5).

Stack and dynamic segments
: A process's stack and dynamic segments reside below the reserved area. Processes can control the amount of virtual memory allotted for stack space, as described below.

Reserved
: A reserved area resides at the top of virtual space.

NOTE Although application programs may begin at virtual address 0, they conventionally begin above 0x10000 (64K), leaving the initial 64K with an invalid address mapping. Processes that reference this invalid memory (for example, by dereferencing a null pointer) generate an access exception trap, as described in the "Exception Interface" section of this chapter.

As the figure shows, the system reserves the high end of virtual space, with a process's stack and dynamic segments below that. Although the exact boundary between the reserved area and a process depends on the system's configuration, the reserved area shall not consume more than 512 MB from the virtual address space. Thus the user virtual address range has a minimum upper bound of 0xdfffffff. Individual systems may reserve less space, increasing processes' virtual memory range. More information follows in the section "Managing the Process Stack."

Although applications may control their memory assignments, the typical arrangement follows the diagram above. Loadable segments reside at low addresses; dynamic segments occupy the higher range. When applications let the system choose addresses for dynamic segments (including shared object segments), it chooses high addresses. This leaves the "middle" of the address spectrum available for dynamic memory allocation with facilities such as malloc(BA_OS).

Managing the Process Stack

Section "Process Initialization" in this chapter describes the initial stack contents. Stack addresses can change from one system to the next—even from one process execution to the next on a single system. Processes, therefore, should *not* depend on finding their stack at a particular virtual address. The stack segment has read and write permissions.

A tunable configuration parameter controls the system maximum stack size. A process also can use setrlimit(BA_OS), to set its own maximum stack size, up to the system limit. Changes in the stack virtual address and size affect the virtual addresses for dynamic segments. Consequently, processes should *not* depend on finding their dynamic segments at particular virtual addresses. Facilities exist to let the system choose dynamic segment virtual addresses.

Coding Guidelines

Operating system facilities, such as mmap(KE_OS), allow a process to establish address mappings in two ways. First, the program can let the system choose an address. Second, the program can force the system to use an address the program supplies. This second alternative can cause application portability problems, because the requested address might not always be available. Differences in virtual address space can be particularly troublesome between different architectures, but the same problems can arise within a single architecture.

Processes' address spaces typically have three segment areas that can change size from one execution to the next: the stack [through setrlimit(BA_OS)], the data segment [through malloc(BA_OS)], and the dynamic segment area [through mmap(KE_OS)]. Consequently, an address that is available in one process

execution might not be available in the next. A program that used `mmap`(KE_OS) to request a mapping at a specific address thus could appear to work in some environments and fail in others. For this reason, programs that wish to establish a mapping in their address space should let the system choose the address.

Despite these warnings about requesting specific addresses, the facility can be used properly. For example, a multiprocess application might map several files into the address space of each process and build relative pointers among the files' data. This could be done by having each process ask for a certain amount of storage at an address chosen by the system. After each process receives its own, private address from the system, it would map the desired files into memory, at specific addresses within the original area. This collection of mappings could be at different addresses in each process but their *relative* positions would be fixed. Without the ability to ask for specific addresses, the application could not build shared data structures, because the relative positions for files in each process would be unpredictable.

Processor Execution Modes

Two execution modes exist in the 68000 architecture: user and supervisor. Processes run in user mode (the least privileged). The operating system kernel runs in supervisor mode. A program executes the `trap` instruction to change execution modes.

NOTE The ABI does not define the implementation of individual system calls. Instead, programs shall use the system libraries that Chapter 6 describes. Programs with embedded system call `trap` instructions do not conform to the ABI.

Exception Interface

As the 68000 user manuals describe, the processor also changes mode to handle *exceptions*. Exceptions can be explicitly generated by a process as a result of instruction execution. The operating system defines the following correspondence between hardware exceptions and the signals specified by `signal`(BA_OS).

Figure 3-23: Exceptions and Signals

Exception Name	Signal
trap #1 (breakpoint trap)	SIGTRAP
external memory fault	see below
address error	SIGBUS
all floating-point exceptions	SIGFPE
illegal instruction	SIGILL
integer zero-divide	SIGFPE
privileged opcode	SIGILL
trace	SIGTRAP
chk, chk2 instruction	SIGFPE
cptrapcc, trapcc, trapv	SIGFPE
line 1010 emulator	SIGSYS
line 1111 emulator	SIGSYS
trap #2-15	SIGSYS

An external memory fault exception can generate various signals, depending on why the exception occurred.

SIGBUS or SIGEMT
: The process accessed a memory location in a way disallowed by the current mapping's permissions. As an example, the process tried to store a value into a location without write permission.

SIGSEGV
: The process referenced a memory address for which no valid mapping existed.

Process Initialization

This section describes the machine state that exec(BA_OS) creates for "infant" processes, including argument passing, register usage, stack frame layout, etc. Programming language systems use this initial program state to establish a standard environment for their application programs. As an example, a C program begins executing at a function named `main`, conventionally declared in the following way.

Figure 3-24: Declaration for `main`

```
extern int main(int argc, char *argv[], char *envp[]);
```

Briefly, `argc` is a non-negative argument count; `argv` is an array of argument strings, with `argv[argc]==0`; and `envp` is an array of environment strings, also terminated by a null pointer.

Although this section does not describe C program initialization, it gives the information necessary to implement the call to `main` or to the entry point for a program in any other language.

Registers

Registers `%d0` through `%d7`, `%a0` through `%a6`, and all MC68040 or MC68881/2 FPCP floating-point data registers have unspecified values at process entry. The floating-point control registers are set to provide IEEE default behavior. Consequently, a program that requires registers to have specific values must set them explicitly during process initialization. It should *not* rely on the operating system to set all registers to 0. See "Process Stack" below for information about the initial values of the stack registers.

As the architecture defines, the status register controls and monitors the processor. Application programs cannot access the entire status register directly; they run in the processor's *user mode*, and the instructions to access the entire status register are privileged. Nonetheless, a program can access the condition code register, which initially has the following value.

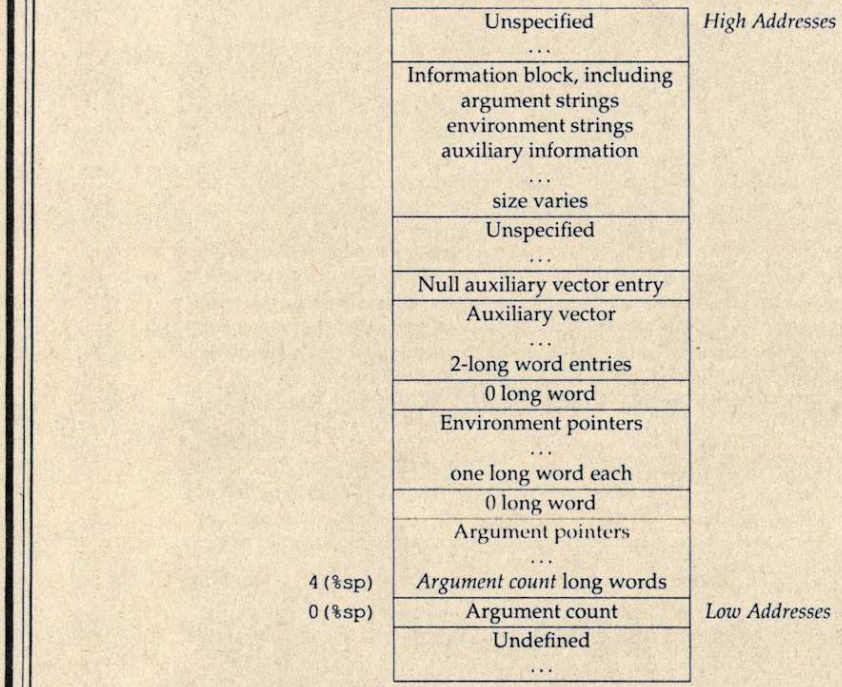
Figure 3-25: Condition Code Register Fields

Field	Value	Note
XNZVC	unspecified	Condition codes unspecified

Process Stack

When a process receives control, its stack holds the arguments and environment from exec(BA_OS).

Figure 3-26: Initial Process Stack



Every process has a stack, but the system defines *no* fixed stack address. Furthermore, a program's stack address can change from one system to another—even from one process invocation to another. Thus the process initialization code must use the address in `%sp`.

Argument strings, environment strings, and auxiliary information appear in no specific order within the information block; the system makes no guarantees about their arrangement. The system may leave an unspecified amount of memory between the null auxiliary vector entry and the start of the information block.

Whereas the argument and environment vectors transmit information from one application program to another, the auxiliary vector conveys information from the operating system to the program. This vector is an array of the following structures, interpreted according to the `a_type` member.

Figure 3-27: Auxiliary Vector

```
typedef     struct
{
            int         a_type;
            union {
                        long        a_val;
                        void        *a_ptr;
                        void        (*a_fcn)();
            } a_un;
} auxv_t;
```

Figure 3-28: Auxiliary Vector Types, `a_type`

Name	Value	a_un
AT_NULL	0	ignored
AT_IGNORE	1	ignored
AT_EXECFD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val

Figure 3-28: Auxiliary Vector Types, a_type (continued)

Name	Value	a_un
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr

AT_NULL — The auxiliary vector has no fixed length; instead its last entry's a_type member has this value.

AT_IGNORE — This type indicates the entry has no meaning. The corresponding value of a_un is undefined.

AT_EXECFD — As Chapter 5 describes, exec(BA_OS) may pass control to an interpreter program. When this happens, the system places either an entry of type AT_EXECFD or one of type AT_PHDR in the auxiliary vector. The entry for type AT_EXECFD uses the a_val member to contain a file descriptor open to read the application program's object file.

AT_PHDR — Under some conditions, the system creates the memory image of the application program before passing control to the interpreter program. When this happens, the a_ptr member of the AT_PHDR entry tells the interpreter where to find the program header table in the memory image. If the AT_PHDR entry is present, entries of types AT_PHENT, AT_PHNUM, and AT_ENTRY must also be present. See Chapter 5 in both the System V ABI and the processor supplement for more information about the program header table.

AT_PHENT — The a_val member of this entry holds the size, in bytes, of one entry in the program header table to which the AT_PHDR entry points.

AT_PHNUM — The a_val member of this entry holds the number of entries in the program header table to which the AT_PHDR entry points.

AT_PAGESZ — If present, this entry's a_val member gives the system page size, in bytes. The same information also is available through sysconf(BA_OS).

AT_BASE — The a_ptr member of this entry holds the base address at which the interpreter program was loaded into memory. See "Program Header" in the System V ABI for more information about the base address.

AT_FLAGS — If present, the a_val member of this entry holds one-bit flags. Bits with undefined semantics are set to zero. No flags are defined for the 68000.

AT_ENTRY — The a_ptr member of this entry holds the entry point of the application program to which the interpreter program should transfer control.

Other auxiliary vector types are reserved.

In the following example, the stack resides at an address below 0xf0000000, growing toward lower addresses. The process receives three arguments.

- cp
- src
- dst

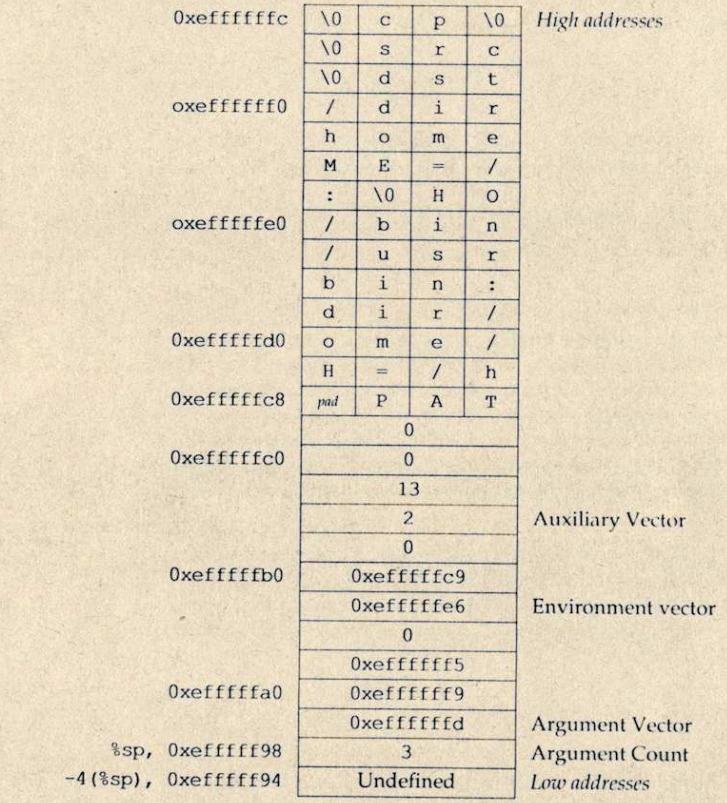
It also inherits two environment strings (this example is not intended to show a fully configured execution environment).

- HOME=/home/dir
- PATH=/home/dir/bin:/usr/bin:

Its auxiliary vector holds one non-null entry, a file descriptor for the executable file.

- {AT_EXECFD, 13}

Figure 3-29: Example Process Stack



Coding Examples

This section discusses example code sequences for fundamental operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. Previous sections discuss how a program may use the machine or the operating system, and they specify what a program may and may not assume about the execution environment. The information here illustrates how operations *may* be done, not how they *must* be done.

As before, examples use the ANSI C language. Other programming languages may use the same conventions displayed below, but failure to do so does *not* prevent a program from conforming to the ABI. Two main object code models are available.

- *Absolute code*. Instructions can hold absolute addresses under this model. To execute properly, the program must be loaded at a specific virtual address, making the program's absolute addresses coincide with the process's virtual addresses.
- *Position-independent code*. Instructions under this model hold relative addresses, *not* absolute addresses. Consequently, the code is not tied to a specific load address, allowing it to execute properly at various positions in virtual memory.

The following sections describe the differences between these models. Code sequences for the models (when different) appear together, allowing easier comparison.

NOTE Examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences nor to reproduce compiler output.

NOTE When other sections of this document show assembly language code sequences, they typically show only the absolute versions. Information in this section explains how position–independent code would alter the examples.

Code Model Overview

When the system creates a process image, the executable file portion of the process has fixed addresses, and the system chooses shared object virtual addresses to avoid conflicts with other segments in the process. To maximize text sharing, shared object libraries conventionally use position-independent code, in which instructions contain no absolute addresses. Shared object text segments can be loaded at various virtual addresses without having to change the segment images. Thus multiple processes can share a single shared object text segment, even though the segment resides at a different virtual address in each process.

Position-independent code relies on two techniques.

- Control transfer instructions hold addresses relative to the program counter (PC). A PC-relative branch or function call computes its destination address in terms of the current program counter, *not* relative to any absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in the instructions, the compiler generates code to calculate an absolute address during execution.

Because the processor architecture provides PC-relative call and branch instructions, compilers can satisfy the first condition easily.

A *global offset table* and a *procedure linkage table* provide information for address calculation. Position-independent object files (executable and shared object files) have these tables in their data segment. When the system creates the memory image for an object file, the table entries are relocated to reflect the absolute virtual addresses as assigned for an individual process. Because data segments are private for each process, the table entries can change—unlike text segments, which multiple processes share.

Assembly language examples below show the explicit notation needed for position-independent code.

name@GOT
: This expression denotes the displacement in the global offset table of the entry for the symbol *name*.

name@PLT
: This expression denotes the displacement in the procedure linkage table of the entry for the symbol *name*.

name@GOTPC
: This expression denotes a PC-relative reference to the global offset table entry for the symbol *name*.

name@PLTPC
: This expression denotes a PC-relative reference to the procedure linkage table entry for the symbol *name*.

Position-Independent Function Prologue

This section describes the function prologue for position-independent code. A function's prologue first allocates the local stack space. Position-independent functions also set register %a5 to the global offset table's address, accessed with the symbol _GLOBAL_OFFSET_TABLE_. Because %a5 is private for each function and preserved across function calls, a function calculates its value once at the entry.

NOTE As a reminder, this entire section contains examples. Using %a5 is a convention, not a requirement; moreover, this convention is private to a function. Not only could other registers serve the same purpose, but different functions in a program could use different registers.

NOTE When an instruction uses _GLOBAL_OFFSET_TABLE_@GOTPC, it sees the offset between the current instruction and the global offset table as the symbol value.

Figure 3-30: Position-Independent Function Prologue

```
name:
    link.l    %fp,&-80
    movm.l    %a5,-(%sp)
    lea       (%pc,_GLOBAL_OFFSET_TABLE_@GOTPC),%a5
```

Data Objects

This discussion excludes stack-resident objects, because programs always compute their virtual addresses relative to the stack and frame pointers. Instead, this section describes objects with static storage duration. Symbolic references in absolute code put the symbols' values—or absolute virtual addresses—into instructions.

Figure 3-31: Absolute Load and Store

C

```
extern int src;
extern int dst;
extern int *ptr;
ptr = &dst;

*ptr = src;
```

Assembly

```
.global   src,dst,ptr


mov.l     &dst,ptr

mov.l     src,([ptr])
```

Position-independent code cannot contain absolute addresses. Referencing global symbols must be done with a base register and global offset table index (as in these examples). Alternatively, instructions that reference symbols hold the PC-relative offsets into the global offset table. Combining the offset with the PC gives the absolute address of the table entry holding the desired address.

Figure 3-32: Position-Independent Load and Store

C

```
extern int src;
extern int dst;
extern int *ptr;
ptr = &dst;

*ptr = src;
```

Assembly

```
.global src,dst,ptr


mov.l   (%a5,dst@GOT),([%a5,ptr@GOT])

mov.l   ([%a5,ptr@GOT]),%a4
mov.l   ([%a5,src@GOT]),(%a4)
```

Function Calls

Function calls in absolute code put the symbols' values—or absolute virtual addresses—into instructions. Programs use the `jsr` or `bsr` instructions to make function calls. For absolute code, the destination operand is an absolute address. Even when the code for a function resides in a shared object, the caller uses the same assembly language instruction sequence, although in that case control passes from the original call, through an indirection sequence, to the desired destination. See "Procedure Linkage Table" in Chapter 5 for more information on the indirection sequence.

Figure 3-33: Absolute Direct Function Call

C

```
extern void function();
function();
```

Assembly

```
.global   function
jsr       function
```

For position-independent code, the `bsr` instruction's destination operand is a PC-relative value. Typically, the destination will be an entry in the procedure linkage table mentioned above.

Figure 3-34: Position-Independent Direct Function Call

C

```
extern void function();
function();
```

Assembly

```
.global   function
bsr       function@PLTPC
```

Indirect function calls also use the `jsr` instruction.

Figure 3-35: Absolute Indirect Function Call

C

```
extern void (*ptr)();
extern void name();
ptr = name;

(*ptr)();
```

Assembly

```
.global   ptr,name

mov.l     &name,ptr

jsr       ([ptr])
```

For position-independent code, the global offset table supplies absolute addresses for all required symbols, whether the symbols name objects or functions.

Figure 3-36: Position-Independent Indirect Function Call

C

```
extern void (*ptr)();
extern void name();
ptr = name;

(*ptr)();
```

Assembly

```
.global   ptr,name

mov.l     (%a5,name@GOT),([%a5,ptr@GOT])

mov.l     ([%a5,ptr@GOT]),%a0
jsr       (%a0)
```

Branching

Programs use branch instructions to control their execution flow. As defined by the architecture, branch instructions can hold a PC-relative value with a range that covers the entire address space.

Figure 3-37: Branch Instruction, Both Models

C

```
label:
        ...
        goto label;
```

Assembly

```
.L01:
        ...
        bra.l   .L01
```

C `switch` statements provide multiway selection. When the case labels of a `switch` statement satisfy grouping constraints, the compiler implements the selection with an address table. The following examples use several simplifying conventions to hide irrelevant details:

- The selection expression resides in register %d0;
- case label constants begin at zero;
- case labels, `default`, and the address table use assembly names `.Lcase`*i*, `.Ldef`, and `.Ltab`, respectively.

Address table entries for absolute code contain virtual addresses; the selection code extracts an entry's value and jumps to that address. Position-independent table entries hold offsets; the selection code computes a destination's absolute address.

Figure 3-38: Absolute `switch` Code

C

```
switch (j)
{
case 0:
     ...
case 2:
     ...
case 3:
     ...
default:
     ...
}
```

Assembly

```
            cmp.l   %d0,&3
            bhi     .Ldef
            asl.l   &2,%d0
            jmp     ([%pc,%d0.l,.Ltab])
            ...
.Ltab:      .long   .Lcase0
            .long   .Ldef
            .long   .Lcase2
            .long   .Lcase3
```


Figure 3-39: Position-Independent `switch` Code

C

```
switch (j)
{
case 0:
     ...
case 2:
     ...
case 3:
     ...
default:
     ...
}
```

Assembly

```
            cmp.l   %d0,&3
            bhi     .Ldef
            mov.l   (%pc,%d0.w*4,.Ltab),%d0
            jmp     (%pc,%d0,.Ltab)
            ...
.Ltab:      .long   .Lcase0-.Ltab
            .long   .Ldef-.Ltab
            .long   .Lcase2-.Ltab
            .long   .Lcase3-.Ltab
```

C Stack Frame

Figure 3-40 shows the C stack frame organization. It conforms to the standard stack frame with designated roles for unspecified areas in the standard frame.

Figure 3-40: C Stack Frame

Base	Offset	Contents	Purpose	
	+4+4*n	argument long word *n*	incoming arguments	*High Addresses*
		...		
	+4	argument long word 0		
SP'	(SP before call)	return address		
		unspecified	local storage and register save area	
		...		
SP	(SP after call)	variable size		
		outgoing arguments		
		stack top, unused		*Low addresses*

Variable Argument List

Previous sections describe the rules for passing arguments. Unfortunately, some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments reside on the stack, and 2) arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many machines, including the 68000. Nonetheless, portable C programs should use the header files `<stdarg.h>` or `<varargs.h>` to deal with variable argument lists (on 68000 and other machines as well).

Allocating Stack Space Dynamically

Unlike some other languages, C does not need dynamic stack allocation *within* a stack frame. Frames are allocated dynamically on the program stack, depending on program execution. The architecture supports dynamic allocation for those languages that require it, and the standard calling sequence and stack frame support it as well. Thus languages that need dynamic stack frame sizes can call C functions, and vice versa.

Figure 3-40 shows the layout of the C stack frame. The double line divides the area allocated by the compiler from the dynamically allocated memory. Dynamic space is allocated below the line as a downward growing heap whose size changes as required. Typical C functions have no space in the heap. All areas above the double line in the current frame have a known size to the compiler. Dynamic stack allocation thus takes the following steps.

1. Stack frames are long word aligned; dynamic allocation should preserve this property. Thus the program rounds (up) the desired byte count to a multiple of 4.
2. The program decreases the stack pointer by the rounded byte count, increasing its frame size. At this point, the "new" space resides just below the register save area at the bottom of the stack.

Even in the presence of signals, dynamic allocation is "safe." If a signal interrupts allocation, one of three things can happen.

- The signal handler can return. The process then resumes the dynamic allocation from the point of interruption.
- The signal handler can execute a non-local goto, or `long jmp` [see `set jmp`(BA_LIB)]. This resets the process to a new context in a previous stack frame, automatically discarding the dynamic allocation.
- The process can terminate.

Regardless of when the signal arrives during dynamic allocation, the result is a consistent (though possibly dead) process.

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

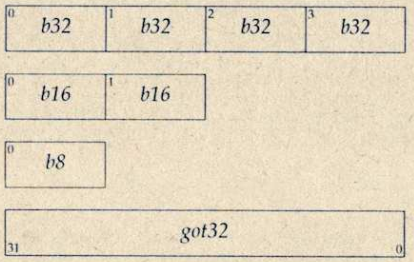
If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for that file will contain an entry for that symbol. The `st_shndx` member of that symbol table entry contains `SHN_UNDEF`. This signals to the dynamic linker that the symbol 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

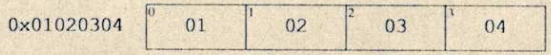
Relocation Types

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).

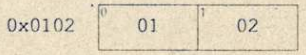
Figure 4-3: Relocatable Fields



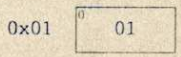
b32 This specifies a 32-bit field occupying 4 bytes with arbitrary alignment. These values use the byte order illustrated below.



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

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

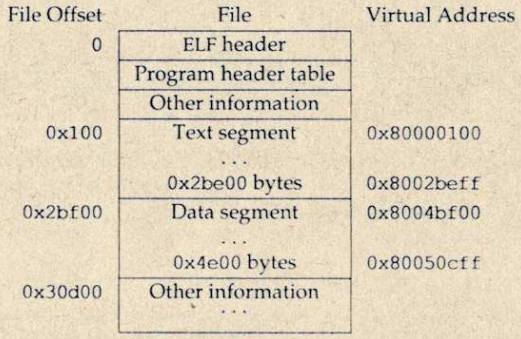


Figure 5-2: Program Header Segments

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

Virtual Address	Contents	Segment
0x80000000	*Header padding* 0x100 bytes	
0x80000100	Text segment ... 0x2be00 bytes	Text
0x8002bf00	*Data padding* 0x100 bytes	
0x8004b000	*Text padding* 0xf00 bytes	
0x8004bf00	Data segment ... 0x4e00 bytes	Data
0x80032d00	Uninitialized data 0x1024 zero bytes	
0x80033d24	*Page padding* 0x2dc zero bytes	

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 sharability 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.