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

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

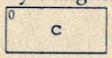


Figure 3-3: No Padding

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

Long word aligned, `sizeof` is 8

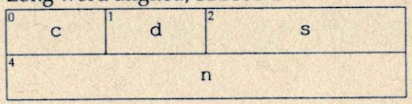


Figure 3-4: Internal Padding

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

Word aligned, `sizeof` is 4

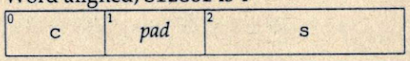


Figure 3-5: Internal and Tail Padding

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

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

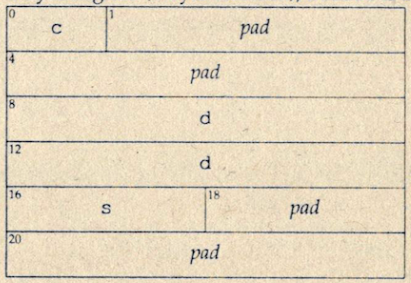
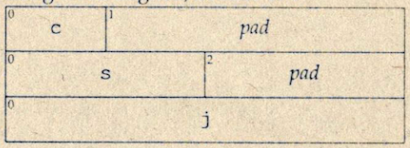


Figure 3-6: `union` Allocation

```
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

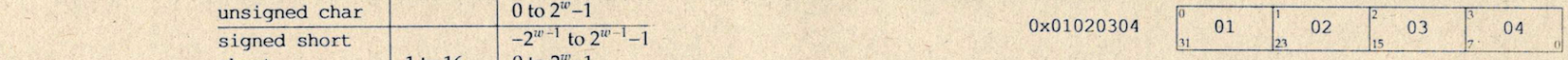


Figure 3-9: Left-to-Right Allocation

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

Long word aligned, `sizeof` is 4

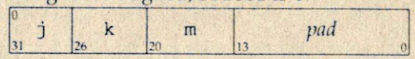


Figure 3-10: Boundary Alignment

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

Long word aligned, `sizeof` is 12

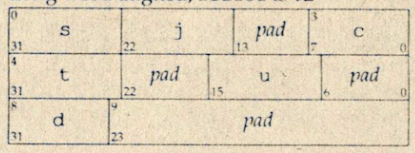


Figure 3-11: Storage Unit Sharing

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

Word aligned, sizeof is 2

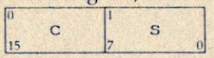


Figure 3-12: `union` Allocation

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

Word aligned, sizeof is 2

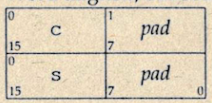
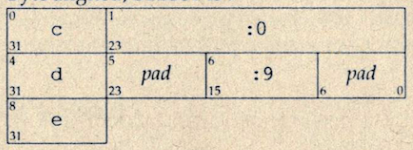


Figure 3-13: Unnamed Bit-Fields

```
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)