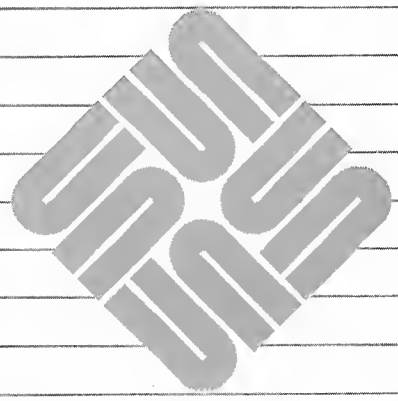
sun®
microsystems

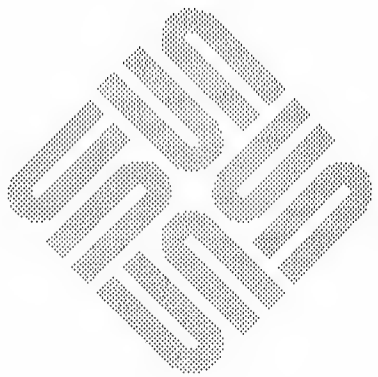
Porting Software to SPARC Systems



Part Number: 800-1796-10
Revision A of 9 May, 1988

Contents

Figures



1

Machine Architecture

1.1. Introduction

This document is intended for programmers who are porting programs written in C, FORTRAN, or Pascal from Sun-2 or Sun-3 machines to SPARC systems. The acronym SPARC stands for Scalable Processor ARChitecture. SPARC is a RISC (Reduced Instruction Set Computing) architecture easily scalable to new technologies, and is described in the *SPARC Processor Architecture* manual.

1.2. Non-Issues

Here are some common porting considerations that are not of concern here.

Word Size

Both the Sun-2, based on the Motorola MC68010 CPU, and the Sun-3, based on the MC68020, are 32-bit machines. That is, integers are 32 bits long. Since SPARC is a 32-bit architecture, word size is not an issue.

Byte Ordering

Both the MC68010 and the MC68020 have forward byte ordering but reverse bit ordering. In other words, the MC680x0 is big-endian with respect to bytes, but little-endian with respect to bits. The same is true of SPARC machines. Thus, byte ordering is not an issue.

By contrast, the VAX and the Intel 80386 have both reverse byte ordering and reverse bit ordering. In other words, they are little-endian architectures. The IBM 360, on the other hand, has both forward byte ordering and forward bit ordering. In other words, it is a big-endian architecture.

Figure 1-1 *Forward Byte and Backward Bit Ordering (MC680x0 & SPARC)*

byte 0								byte 1								byte 2								byte 3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

Figure 1-2 *Backward Byte and Bit Ordering (VAX & 80386)*

byte 3								byte 2								byte 1								byte 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

Figure 1-3 *Forward Byte and Bit Ordering (IBM 360)*

byte 0								byte 1								byte 2								byte 3							
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The bit and byte ordering of the VAX, Intel 80386, and IBM 360 are not relevant when porting from the Motorola 680x0 to SPARC systems. They are mentioned only for comparison. Also, note that the difference in bit ordering between the MC680x0 and the IBM 360 is purely notational. That is, on the MC680x0 the bit named 0 is the least significant, but on the IBM 360 the bit named 31 is the least significant. These bits have the same numeric value, but different names.

Scalar Representation

Both the MC680x0 and SPARC machines use two's-complement integers, and standard IEEE floating-point single- and double-precision representations. So scalar data representation is not an issue.

1.3. How to Read this Document

The next chapter describes issues you may encounter when porting C programs to SPARC systems. The chapter after that covers the porting of FORTRAN programs. The last chapter talks about porting Pascal programs to SPARC systems. You may read only the material that concerns you.

2

Porting C Programs

2.1. Porting Issues

Here are some architectural considerations that you should be aware of when porting C programs to SPARC machines. Fortunately you can pinpoint most of these problems with `lint -ch`. The `-c` flag detects unportable casts, and the `-h` flag performs heuristic checking.

Data Alignment

On the MC680x0, characters are aligned on byte boundaries, and everything else, regardless of size, is aligned on halfword (even) boundaries. On SPARC machines, all quantities must be aligned on boundaries corresponding to their sizes: bytes on byte boundaries, (16-bit) halfwords on halfword boundaries, (32-bit) words on word boundaries, and (64-bit) doublewords on doubleword boundaries. If you are coding in assembly language, you must observe alignment restrictions. Otherwise, compilers normally keep track of everything for you. There are several C language constructs, however, that may lead to a bus error during execution:

- Casting a pointer to a `char` or `unsigned char` into a pointer to a larger quantity, such as a `short`, `int`, `long`, `float`, `double`, or `struct/-union` containing one of these. This includes passing a `char *` as an argument to a function expecting a pointer to a larger quantity.
- Casting a pointer to a `short` or `unsigned short` into a pointer to a larger quantity, such as an `int`, `long`, `float`, `double`, or `struct/-union` containing one of these. This includes passing a `short *` as an argument to a function expecting a pointer to a larger quantity.
- Casting a pointer to a 32-bit quantity (such as an `int`, `unsigned int`, or `float`) into a pointer to a (64-bit) `double` or `struct/union` containing a `double`. This includes passing a pointer to a 32-bit quantity as an argument to a function expecting a pointer to a `double`. C programmers should note that `float *` and `double *` are not the same.

The above constructs may work occasionally, if the pointer happens to end up on the right boundary. But more often, these constructs lead to bus errors. It is not the cast itself that causes the bus error, but rather dereferencing the resulting pointer. The use of `lint` should catch most of these problems.

Structure Alignment and Padding

Structure Alignment. On the MC680x0, each structure is aligned on a halfword (even) boundary. On SPARC machines, the alignment requirement for a structure is the same as that of its most strictly aligned component. For instance, a `struct` containing only `char` members has no alignment restrictions, whereas a `struct` containing a `double` must be aligned on an 8-byte boundary.

Internal Padding. On the MC680x0, structures are padded internally so that integers and floats always begin on an even boundary. On SPARC machines, structures are padded internally so that every element is aligned on the appropriate boundary. For instance, a `struct` containing only one `char` and then a `long` has three bytes of padding after the `char`, so that the `long` is aligned on a 4-byte boundary.

Tail Padding. On the MC680x0, structures are padded on the end to contain an even number of bytes. On SPARC machines, structures are padded on the end to the appropriate alignment boundary. For instance, a `struct` containing only `char` members is unpadded, whereas a `struct` containing an `int` but no `double` is padded out to a 4-byte boundary.

Because of the three considerations above, members of a given structure may have different offsets on SPARC machines than on the MC680x0, and the structure as a whole may have a different size. Even though data representations are identical on the MC680x0 and SPARC, binary files where raw structures have been written out may not be portable between processors. Note that structures retained in memory are fine; problems occur only when raw structures are written to disk or across the network.

Here is an example of two structures that could not be written on one processor and read on the other (though in memory they would be fine):

Figure 2-1 *Structures that Result in Non-Portable Binary Files*

	Offsets		*sizeof(struct)*	
	MC680x0	SPARC	MC680x0	SPARC
`struct chl`				
`{`				
`char c;`	+0	+0		
`long i;`	+2	+4	6	8
`};`				
`struct ccc`				
`{`				
`char c1;`	+0	+0		
`char c2;`	+1	+1		
`char c3;`	+2	+2	4	3
`};`				

There are three solutions to this problem. First, you could write a program to run on each processor to create binary files of structures according to the requirements of that processor. For example, the `makedev` program used with device-independent `troff` writes out the font information structures on each machine running `troff`.

Second, if a structure must be portable across machines, Sun's eXternal Data Representation (XDR) is the best solution. The best way to write a record on one machine that is to be read on others is to use an XDR standard representation for the data. See the section entitled "XDR Protocol Specification" in the manual *Networking Programming on the Sun Workstation.*

Third, you could manually arrange the members of a structure, from the most to least restrictive alignment requirements, then insert explicit fill (padding) elements as needed. Structures are often designed in this manner anyway, with the largest elements at the beginning.

Function Return Values

On SPARC machines, if a function is going to return a structure by value, both the calling function and the called function must agree on its type. If the called function returns a structure by value but the calling function doesn't use it, no harm is done. The value is returned, but the calling function ignores it. If the called function does not return a structure by value but the calling function expects one, you get an "Unimplemented Instruction Trap" at runtime upon return from the called function. The use of `lint` should catch these problems.

Passing Mismatched Parameter Types

The C language does not define what happens when you pass a list of variables to a routine that receives a `struct` by value, or vice versa. This just happened to work with Sun's MC680x0 C compilers. On SPARC machines, it does not work. Here is an example that won't work on SPARC:

```
struct thing {
    int x, y;
};
int a, b;

routine(s)
struct thing s;
    .
    .
routine(a, b);
```

Likewise, on SPARC machines, passing a `union` by value is not equivalent to passing one of its elements (use of `lint` should catch this). Here is a construct that won't work on SPARC:

```
union thing {
    int i; double x;
} combo;

routine(x)
double x;
    .
    .
routine(combo);
```

Parameter Passing: `varargs()`

Taking the address of a function parameter and manipulating it to access other parameters (or other data) on the stack is possible with Sun's MC680x0 C compilers. With SPARC compilers, however, routines that receive a context-dependent number of arguments of varying types must be written using the macros defined in `<varargs.h>`. These macros permit you to write even such routines as `printf()` portably. See the *varargs*(3) manual page for details.

Order of Parameter Evaluation

The order of evaluation of parameters to a function is not defined by the C language, and is different in SPARC C compilers than in Sun's MC680x0 C compilers. Let's consider this example:

```
func(x * i, y / i, i++);
```

Since the arguments of `func()` are evaluated in a different order on SPARC systems than on Sun-2 or Sun-3 machines, the side effect caused by `i++` is going to yield different results on different machines. It is never a good idea to make assumptions about the order of parameter evaluation. The best strategy is to write C code that does not depend on any side effects of parameter evaluation.

Passing Union Arguments to `semctl()`

Users of the System V semaphore facility may have to modify code that worked on other machines for SPARC. With the `semctl`(2) system call, the subcommands `SETVAL`, `GETALL`, `SETALL`, `IPC_STAT`, and `IPC_SET` require a fourth argument `semun`, which is a `union`. Programs that call `semctl()` with these subcommands must pass the `union` itself, rather than an element of the `union`, or a constant such as 0 (zero). Programs that call `semctl()` with other subcommands should omit the fourth argument, rather than pass a constant such as 0 (zero). As usual, `lint` helps you spot problems of this kind.

Stack Allocation with `alloca()`

On SPARC systems, users of the stack allocation routine `alloca()` must include the header file `<alloca.h>` before using the routine. Furthermore, since `alloca()` is now built in to the compiler, it cannot be assigned to an `int (*)()` variable, nor can it be passed as a procedure-type parameter.

Out-of-Range Shifts

Using the C language bit-wise shift operators <<, >>, <<=, or >>= with a right-hand operand greater than or equal to the size of the left-hand operand (in bits) yields machine-dependent results. Many programmers are not aware of this, and assume that an unsigned shift by a large amount yields zero. This is often true on MC680x0 machines, because the shift count is interpreted modulo 64. This is true less often on SPARC machines, because the shift count is interpreted modulo 32. The best strategy is to avoid shift counts greater than the size of the affected operand. A negative shift count won't yield sensible results on either machine, of course.

Uninitialized Automatic Variables

Naturally, beware of uninitialized local variables; they may have different values on different machines, and in different calls to the same function. The use of uninitialized automatic variables continues to be a poor programming practice. Fortunately, the use of `lint` should detect such problems.

2.2. Conclusion

Well-written portable C programs should compile and run on SPARC machines as well as on other machines. Non-portable programs, by definition, may present problems when transported to SPARC machines, or to any other machine. There is no substitute for good program design and judicious use of `lint`.

3

Porting FORTRAN Programs

3.1. Porting Issues

In general, there are fewer potential areas of concern in porting FORTRAN programs to SPARC than there are porting C programs. Data alignment is not a problem, because FORTRAN has no type casting mechanism. Binary reads and writes are done byte-by-byte, so structure padding is not a concern. FORTRAN has no structures, no unions, and no mechanism for variable-length argument lists, so these do not pose portability problems, either.

The `EQUIVALENCE` statement and the `COMMON` block, and the order of parameter evaluation, are perhaps the only potential problem areas.

The `EQUIVALENCE` Statement

The use of `EQUIVALENCE` can force double-precision variables to be misaligned, as in the following FORTRAN code:

```
REAL A(5)
DOUBLE PRECISION D(4)

EQUIVALENCE (A(2),D(1))
```

Note that the 8-byte doubleword `D(1)` does not begin on an 8-byte boundary owing to the `EQUIVALENCE`, even though it would be much more efficient for `D(1)` to be aligned on an 8-byte boundary.

Figure 3-1 *Alignment Problems with* `EQUIVALENCE`

	8 bytes		8 bytes		8 bytes		8 bytes	
A:	1	2	3	4	5			
D:		1		2		3		4

Because this usage of `EQUIVALENCE` is standard FORTRAN, the FORTRAN compiler must deal with it. When an `EQUIVALENCE` statement skews alignment, the compiler generates code to access double-precision variables as pairs of single-precision variables. These variables are loaded and stored with word instructions, rather than with doubleword instructions. Unfortunately this slows down execution somewhat, so for the sake of efficiency, it is best not to `EQUIVALENCE` variables without regard for data alignment.

The COMMON Block

The placement of odd-length single-precision arrays before double-precision arrays in a COMMON block can also force double-precision variables to be misaligned, as in the following FORTRAN code:

```
REAL A(3)
DOUBLE PRECISION D(3)

COMMON  A,D
```

Note that the 8-byte word `D(1)` does not begin on an 8-byte boundary because of the COMMON block ordering, even though it would be much more efficient for `D(1)` to be aligned on an 8-byte boundary.

Figure 3-2 *Alignment Problems with* COMMON

	8 bytes		8 bytes			8 bytes		
A:	1	2	3					
D:				1		2		3

Because this usage of COMMON is standard FORTRAN, the FORTRAN compiler generates code to access double-precision elements of array `D` as pairs of single-precision variables. These variables are loaded and stored with word instructions, rather than with doubleword instructions. Unfortunately this slows down execution somewhat. The best fix is to get into the habit of placing double-precision variables first in a COMMON block.

Order of Parameter Evaluation

The order of evaluation of parameters to a FORTRAN function or subroutine is different on SPARC than on the Sun-2 or Sun-3. Let's consider this example:

```
call tally(f(x), g(x))
```

Since the functions `f()` and `g()`, which are arguments of subroutine `tally()`, are evaluated in a different order on SPARC systems than on the MC680x0, the value of `x` had better not change between function calls.

The best strategy is to write FORTRAN code that does not depend on the order of parameter evaluation.

4

Porting Pascal Programs

4.1. Porting Issues

Here are some architectural considerations that may cause problems when porting Pascal programs to SPARC.

Data Alignment

Since Pascal has no type casting mechanism like the one in C, there should never be data alignment problems caused by casting pointers to small objects into pointers to larger objects.

However, it is possible to simulate the effect of type casts by the use of the variant record mechanism. For example, the following program may fail to work as you would expect:

```
program WontWork;

type
    foo = record
        case boolean of
            false :
                (Iptr : ^integer);
            true :
                (Cptr : ^char)
        end;
var
    bar : foo;

begin
    new(bar.Cptr);
    bar.Iptr^ := 0;
    .
    .
end.
```

Record Alignment and Padding

On the MC680x0, each `record` is aligned on halfword (even) boundaries. On SPARC machines, the alignment requirement of a `record` is the same as that of its most strictly aligned component. For instance, a `record` containing only `char` members has no alignment restrictions, whereas a `record` containing a `real` must be aligned on an 8-byte boundary.

On the MC680x0, `records` are padded internally so that integers and reals always begin on an even boundary. For instance, a `record` containing only one

`char` and then an `integer` has three bytes of padding after the `char`, so that the `integer` is aligned on a 4-byte boundary.

On the MC680x0, `records` are padded on the end to contain an even number of bytes. On SPARC machines, `records` are padded on the end to the appropriate alignment boundary. For instance, a `record` containing only `char` members is unpadded, whereas a `record` containing an `integer` but no `real` is padded out to a 4-byte boundary.

Because of the three considerations above, members of a given record may have different offsets on SPARC machines than on the MC680x0, and the record as a whole may have a different size. Even though basic data types (machine types) are represented identically on the two machines, constructed data types may be different. Note that records retained in memory are fine; problems occur only when records are actually written out.

Here is an example of two records that could not be written on one processor and read on the other (though in memory they would be fine):

Figure 4-1 *Records that Result in Non-Portable Binary Files*

	Offsets		*sizeof(struct)*	
	MC680x0	SPARC	MC680x0	SPARC
`type`				
`  cint = record`				
`    c : char;`	+0	+0		
`    i : integer;`	+2	+4	6	8
`  end;`				
`  ccc = record`				
`    c1 : char;`	+0	+0		
`    c2 : char;`	+1	+1		
`    c3 : char;`	+2	+2	4	3
`  end;`				

There are three solutions to this problem. First, you could write a program to run on each processor that would create binary files of records according to the requirements of that processor. Pascal versions of TEX, for example, come with programs to create font metric files on different machines.

Second, if a record must be portable across machines, Sun's eXternal Data Representation (XDR) would be the best solution. The best way to write a record on one machine that is to be read on others is to use an XDR standard representation for the data. See the section entitled "XDR Protocol Specification" in the manual *Networking on the Sun Workstation*. Unfortunately calling XDR routines from Pascal is not yet supported.

Third, you could manually arrange the elements of a record, from the most to least restrictive alignment requirements, then insert explicit fill elements as needed. Records are often designed in this manner anyway, with the largest elements at the beginning.

Order of Parameter Evaluation

The order of evaluation of parameters to a procedure or function is not defined by the Pascal language, and is different in SPARC Pascal compilers than in Sun's MC680x0 Pascal compilers. Let's consider this example:

```
tally(func(x), eval(x))
```

Since the functions `func()` and `eval()`, which are arguments of procedure `tally()`, are evaluated in a different order on SPARC systems than on the MC680x0, the value of `x` had better not change between function calls.

It is never a good idea to make assumptions about the order of parameter evaluation. The best strategy is to write Pascal code that does not depend on any side effects of parameter evaluation.

Out-of-Range Shifts

Sun's Pascal compiler has non-standard extensions to perform bit operations on integral types. The logical shift (right and left) is analogous to shifting `unsigned` quantities in C, whereas the arithmetic shift (right and left) is analogous to shifting `signed` quantities in C.

Table 4-1 *Bitwise Operations in Pascal and C*

Pascal	*C*
`lsl(x, count)`	`unsigned x << count;`
`lsr(x, count)`	`unsigned x >> count;`
`asl(y, count)`	`int y << count;`
`asr(y, count)`	`int y >> count;`

With any of these operations, a `count` greater than or equal to the size of the left-hand operand yields machine-dependent results. Many programmers are not aware of this, and assume that a right logical shift by a large amount yields zero. This is often true on MC680x0 machines, because the shift count is interpreted modulo 64. This is true less often on SPARC machines, because the shift count is interpreted modulo 32.

The best strategy is to avoid shift counts greater than the size of the affected operand. A negative shift count won't yield sensible results on either machine, of course.

Uninitialized Local Variables

Naturally, beware of uninitialized local variables; they may come up with different values on different machines. The use of uninitialized variables is a poor programming practice. The Pascal compiler `pc` flags uninitialized local variables, but not uninitialized global variables, since external procedures may initialize any global variable.

4.2. Conclusion

Well-written portable Pascal programs should run on SPARC machines as well as on any other machine. Non-portable programs, by definition, may present problems when transported to SPARC machines, or to any other machine. There is no substitute for good program design.