Porting C Applications DEC OSF/1 Alpha

Lu Anne Van de Pas

The DEC OSF/1 V1.2 system takes advantage of the full 64-bit capabilities of the Alpha architecture. In doing so, it introduces a number of extended capabilities beyond 32-bit architectures that can effect the portability and interoperability of programs. Careful coding practices can help reduce these inconsistencies. The following sections will look at specific aspects of the C language and explain certain programming techniques that will help in both new program development and the porting of existing programs from DEC ULTRIX to DEC OSF/1 Alpha systems.

Overview

The ease of moving a C program from a 32-bit VAX or MIPS architecture to a full 64-bit implementation of the Alpha architecture depends upon: 1. the overall coding disciplines employed in the application, and 2. the use of nonstandard system features. For example, a program written with adherence to the ANSI C standard, using function prototypes, and having no assumption about the machine size of data types or system architectural specifics, can be ported with ease. Special care may need to be taken when mixing 64-bit and 32-bit systems through data sharing mechanisms such as networks, databases, and shared file systems.

At a high level, the general coding issues can be outlined by examining specifics in the host (build) and target (runtime) environment of the DEC OSF/1 Alpha system. Each of the areas described below affects the portability of the application. The programming techniques and examples in the remaining sections will show in detail how to code or recode your program.

- Development environment

  When you port an application to the DEC OSF/1 Alpha system you must recompile the application or use the binary translator. (See MX documentation for information on translating executables from the MIPS architecture to Alpha.)

  The DEC OSF/1 Alpha development environment has a similar compilation, linking, debugging, and performance analysis tools as RISC ULTRIX. The Alpha development environment, C compiler and related tools, has additional support for ANSI C, 64-bit data types and addresses, and shared libraries. Lint has additional features to help find 32-to-64 bit conversion problems.

- Compiler optimizations:

  The DEC OSF/1 Alpha C compiler has additional optimizations and code scheduling specific to the Alpha architecture. Because of this you may notice different error messages between ULTRIX and Alpha systems.

- Data Representation:

  In order to take advantage of the 64-bit architecture the C data types have been modified to include a 64-bit type. In the table below, 'int' is unchanged as a 32 bit entity, and long is redefined to be 64 bits. In order to extend the address space, pointers on Alpha are defined to be 64 bits.

  The DEC OSF/1 Alpha system has also defined a 'long long' data type to be 64 bits. It provides the unique name for a 64-bit data type that may provide additional interoperability between 32-bit and 64-bit systems.

  Similar to VAX and MIPS systems, the DEC OSF/1 Alpha system uses right-to-left byte order for integer types (little-endian).

Data type	32-bit MIPS or VAX system (size in bits)	64-bit Alpha system (size in bits)
char	8	8
short	16	16
int	32	32
long	32	64
long long	Not available	64
float	32 (MIPS: IEEE Single) (VAX: F float)	32 (IEEE Single)
double	64 (MIPS: IEEE Double) (VAX: G or D float)	64 (IEEE Double)
pointer	32	64

- Data access

  The VAX and MIPS architectures are both byte and word addressable. Alpha supports only memory accesses of longword (32 bits) or quadword (64 bits). Byte and word accesses are accomplished by multiple instructions which load a longword or quadword, mask, and shift to get the desired entity. This lack of a single operation for byte and word access may produce incorrect results in cases where you are accessing adjacent byte or word entities in shared memory segments. For instance, a multi-threaded application or multiple processes that has access to adjacent byte data through shared memory or shared memory-mapped files will have to use thread mutual exclusion locking functions or semaphone locks, respectively, to avoid conflicts with accesses to adjacent byte or word data items.

- Data Alignment

  On both MIPS and ALPHA systems the data alignment is implied by the data type. For instance, an *int* (32 bits) is aligned on a 4 byte boundary. On MIPS systems, a *long* (32 bits) is also aligned on a 4 byte boundary. But on Alpha systems, a *long* (64 bits) is aligned on 8 byte boundaries. If using assembly language, you will need to understand and code according to these alignment restrictions. If using a high-level language such as C, the compiler will take care of this alignment for you but it is still important that you understand these alignment differences when using long and pointer types in structure definitions that are shared between 32-bit and 64-bit systems.

- File system

  On the 32-bit systems of MIPS and VAX, files and file systems were limited to 2 gigabytes in size. This limit was imposed by the programming interface and file system which used a 32-bit integer to represent the file offset in bytes (*off_t*) when navigating within a file or file system. On a 64-bit DEC OSF/1 Alpha system, you can now build much larger files and file systems. *off_t* is defined to be a *long* on Alpha systems.

  Given this extended capability, it is possible to build files and files systems that can not be fully accessed by 32-bit systems. This is very important to keep in mind when working in an distributed environment where file systems are shared between 32 and 64 bit systems.

Coding Guidelines

When developing or porting C code the RISC ULTRIX and DEC OSF/1 Alpha systems are similar in a number of ways. Both systems are little endian. Both support 32-bit integers, 16-bit shorts, 8-bit characters, and IEEE single and double floating point formats. And both have a similar development environment and C compiler. The major differences you need to consider in coding are in the size of addresses, the availability of 64-bit integer types, the data type alignment restrictions, byte and word accessibility, and interoperability between 32-bit and 64-bit systems.

The remainder of this chapter will go through specific C coding examples and outline areas that may need to be changed for this 64-bit architecture. Many of these changes deal with the "cleaning up" of data type usage, so that you can have code that works on both the 32-bit and 64-bits systems.

Header files - Constant definitions

On Alpha, there are a few changes to the standard header files that are directly related to 64-bit data types. These include:

/usr/include/limits.h

Constant	Definition	Value on ULTRIX	Value on Alpha
LONG_BIT	Number of bits in a long	32	64
LONG_MAX	Maximum value of a long type	0x7fffffff	0x7fffffffffffffff
LONG_MIN	Minimum value of a long type	0x80000000	0x8000000000000000
ULONG_MAX	Maximun value of an unsigned long	4294967295U	18446744073709551615U

Constants

Some constants may have different values between 32-bit and 64-bit systems. For instance, the hexadecimal value, 0xFFFFFFFF, has the value -1 on a 32-bit system and the value 4294967295 on Alpha. The table below lists a few other interesting integer constants and their values.

C constant	Value	Value on MIPS	Value on Alpha
0xFFFFFFFF	(2^32) -1	-1	4294967295
4294967296	2^32	(0)	4294967296
0x100000000	2^32	(0)	4294967296
0xFFFFFFFFFFFFFFFF	(2^64) -1	-1	-1

Definitions and Declarations

Structure Size

Structures and unions on DEC OSF/1 Alpha systems change size from 32-bit systems. This is due to the new 64-bit data sizes and the additional alignment considerations for *long* and pointer. In the example below, the structure, **TextNode**, grows in size because all of its members double in size from 4 bytes to 8 bytes for pointer types.

```
struct TextNode {
        char *text;
        struct TextNode *left;
        struct TextNode *right;
        } ;
```

This change in size is an important consideration if you are sharing data defined in structures between 32-bit and 64-bit systems. Be careful about using longs and pointers as members in shared structures. These data types now introduce sizes that are not available on 32-bit systems. One of the most portable methods is to only use typedef types in structures and set up the types as appropriate for the system. You can automatically do this by utilizing information in the limits.h header file.

Additionally, you should be careful when building unions between ints and pointers, since they are no longer the same size.

Structure Member Alignment

Members of structures and unions are aligned on their "natural" boundaries. That is, char is aligned on a byte boundary, short on a word boundary, int on a longword boundary, and longs and pointers on quadword boundaries.

This means that additional space will be used for padding member alignment in structures and unions.

```
struct  TextCountNode {
      char  *text;
      int size,
      struct TextCountNode *left;
      struct TextCountNode *right;
};
```

On 32-bit systems the size of this structure would be 16 bytes. On 64-bit Alpha systems the size of the structure would be 32 bytes: 8 bytes for each pointer and 4 bytes of padding after the member, **size**, for the alignment of the pointer, **left.**

Additional padding may also be introduced at the end of structure, to assure proper structure alignment for arrays of these structures. The structure must terminate on the same alignment boundary on which it started.

Given these additional alignment considerations, you should always use *sizeof* to determine the size of a structure. Don't assume the size of a structure is the accumulated size of all of the objects defined in it. Additional space will be taken up for padding the member alignment.

To minimize the amount of padded needed, you may want to reorder members in a structure. For example,

```
struct s {
        int count;
        struct s *next;
        int total;
}
```

has a *sizeo*f 24 bytes. This definition can be recoded to

```
struct s{
        struct s *next;
        int count;
        int total;
}
```

Which has a *sizeof* 16 bytes.

Structure Alignment

In order to have specific members aligned on their required boundaries, structures themselves need to have the alignment of the strictest aligned member.

```
struct  {
        char *text;
        int count;
        } CountedString;
```

In the example above, the sizeof(**CountedString**) is 16 bytes (***text** = 8 bytes, **count** = 4 bytes, tail padding = 4 bytes.) This structure needs to be aligned on a quadword boundary because the pointer requires quadword alignment. This means that a definition that has **CountedString** as a member will have each reference to **CountedString** on a quadword boundary.

```
CountedString CsArray[10]
```

```
struct {
        char line[MAX_LINE];
        struct CountedString string;
}TextAndString;
```

In each of the above examples, the **CountedString** structure will force alignment of the beginning of the structure to be on a quadword boundary. In the first declaration above, no additional padding (beyond 4 bytes of tail padding) will be introduced because **CountedString** will naturally align on a quadword boundary. In the second, additional padding maybe introduced (depending upon the value of MAX_LINE) to insure proper quadword alignment for the structure member, **string**.

Variable Declarations

With the changes in *long* and pointer type you should be careful to code your application so that it can work on both 32-bit and 64-bit systems. Check your *int* and *long* declarations. If you have specific variables that you need to be 32 bits in size on both DEC OSF/1 on MIPS and Alpha then define the type to be *int.* If the variable should be 32 bits on DEC OSF/1 on MIPS and 64 bits on Alpha then define the variable to be *long*. Remember if the type specifier is missing from a declaration, it defaults to *int* type. For example, here are six declarations which declare the variables to be of size *int* and the function to be returning type *int*.

```
extern e;
register n;
static x;
unsigned i;
const c;
funtion();
```

Bit-Fields

Bit fields are allowed on any integral type on Alpha. (ANSI C only requires, bit-fields with int, signed int, and unsigned int types.) In a C declaration, if a bit-field immediately follows another in a structure declaration the following bit-field will be packed into adjacent bits of the former unit. Since long is now 64 bits in length on Alpha, adjacent declarations of bit-fields of type long may contain multiple bit-field definitions in cases that previously did not on RISC or VAX. This change may cause different results in operations on these bit-fields. To insure the same behavior when porting code, change bit-field definitions of type *long* to *int*.

Statements and Expressions

Variable Assignments and Function Arguments

On DEC OSF/1 Alpha, since *int* and *long* are no longer defined to be the same size, you can not freely interchange their use without the possibility of truncation of significant digits. Use the lint utility to help you find these problems. You should avoid assignments such as

```
int i;
long l;

i = l;
```

Also, you should avoid passing long arguments to functions expecting type *int*.

```
int  toascii(int);
int i;
long l;

i= toascii(l)
```

Pointers and ints should not be freely exchanged on DEC OSF/1 Alpha. Assigning a pointer to an int, then assigning back to a pointer, and dereferencing the pointer will result in a bus error.

```
int i ;
char *buffer;

buffer = (char *)malloc(MAX_LINE)
i = (int)buffer;
buffer = (char*)i;
```

Similarly, passing a pointer to a function expecting an *int* argument will result in lost information.

```
void f();
char *cp;

f(cp);
```

This nonportable function declaration will produce a compiler warning if you use ANSI C prototypes, such as:

```
void f(int);
char *cp;

f(cp);
```

You can also find these pointer to int assignments by using the -h flag of the lint(1) command, which will find pointer to *int* argument passing and assignments. Even though a pointer can be converted to a type *long* on Alpha without a loss of information, you should use the *void ** type if you need to use a generic pointer type.

Additionally, watch out for poor programming practices such as different multiple definitions of the same object. For instance, two structures that are used in different areas of your code to refer to the same object in different ways. Such as

```
struct node {
   int src_addr, dst_addr;
   char *name;
  } ;


struct node {
   struct node *src, *dst;
   char * name;
 }
```

This type of nonstandard coding should be replace with a union declaration. Be thorough when porting this type of code to a 64-bit system, the interdependency and incompatibilities between these two structures may be difficult to find.

Integer and Long Constants - Assignment and Argument Passing

In C, an integer constant is specified like, 543210. To specify a long int constant you use the suffix L or l. To specify a unsigned long you use the UL or ul suffix. (L is preferred since l is easily confused with 1). Note the example where three different constants are passed to the function, *labs()*:

```
labs(543210)
labs(543210L)
labs(543210UL)
```

On DEC OSF/1 on MIPS, 543210 would be passed as a 4 byte constant in all three examples. On an Alpha system , 543210 would be treated as a 4 byte constant, and 543210L or 543210UL would be treated as a 8 byte constant. If the *labs()* function was expecting a *long* argument each of these invocations would work as expected since the *int* constants would be converted to *long.* Problems happen if the function *labs()* was expecting type int. In this case the long constant would be truncated to an integer constant. This truncation would result in the loss of significant digits if the constant was greater then maximum integer constant (INT_MAX) of +2147483647, or less then the minimum integer constant (INT_MIN) of -2147483648, or for unsigned constants greater then the maximum unsigned integer constant (UINT_MAX) of 4294967295. This problem would also be present in an assignment expression where a long integer constant was assigned to a variable of type int. In these cases it is important to explicitly use the L or UL suffix and make sure the function arguments or variables being assigned to are of the appropriate *long* type.

It is also important to note that when you are passing zero to a pointer argument and no function prototype is visible, always use NULL (Defined in stdio.h). Using *0* will result in using a 4 byte zero instead of a 8 byte zero (0L). (In a comparison, an assignment, or a function call where the correct function prototype is in scope, standard C promotion rules will be in effect and the correct value will be assigned.)

Integer and Long Constants - Shift operations

A bit shift operation on a integer constant will yield an 32-bit constant. If you need a result of type long then you need to use the L or UL suffix for long integer constants. For example,

```
long value;

value = 10 << 2;
```

results in **value** getting assigned a 32-bit constant. The top 32 bits of **value** will depend on the type of the value shifted. Signed values are sign-extended; unsigned values are zero extended. If you want a 64-bit constant then be sure to use the L or UL suffix. (Note that only the left operand of a shift operator determines the result type. The type of shift count operand is irrelevant.)

Sizeof expression

The result of the *sizeof* operator is of type **size_t**, which is of an *unsigned long* on Alpha.

Pointer Subtraction

The length of the integer required to hold the difference between two pointers to members of the same array, ptrdiff_t (stddef.h), is an *signed long* on Alpha.

Functions with a variable number of arguments

When writing a routine that receives a variable (context-dependent) number of arguments you must use the stdargs (stdarg.h) or varargs (varargs.h) mechanism. See the varargs(3) reference page for more information on the use of these macros.

Library Calls

printf, scanf functions

When using the printf-type function conversion specifiers for longs use the "l" (lower-case letter L) size specification, with the d, u, o, and x operations to specify assignment of type long or unsigned long. For instance, when printing a long as signed decimal use the %ld instead of %d, when printing a long as a unsigned decimal use the %lu instead of %u. If the letter l size specification is not used the type is assumed to be int, unsigned int, or int * depending upon the conversion specification. In which case the long types will be converted to the smaller int types and information may be lost.

When printing a pointer use %p. If you want to print the pointer as a specific representation then the pointer should be cast to an appropriate integer type (long for Alpha) before using the desired format specifier. For example, to print a pointer as a long unsigned decimal number use %lu:

```
char *p;
printf ( "%p %lu\n",  (void *)p,  (long)p );
```

For a portable way to print an integer of arbitrary size, case the integer to long or unsigned long, then use the %L conversion specifier. For example:

```
printf ("%ld\n", (unsigned long) sizeof (num));
```

malloc, calloc functions

Memory allocation library functions such as *ma*lloc guarantee to return data aligned to the maximum alignment of any object. On Alpha, *malloc* returns a pointer to memory that is quadword aligned.

lseek function

When calling the *lseek* system call for setting the current position in a file, use the *off_t* type defined in *types.h* for the file offset. Passing an int or long constant may work but it is not the portable and is not guarantee to continue to work. The following example shows correct uses of *lseek*.

lseek function:

```
#include <unistd.h>

off_t offset, pos;
...
pos = lseek( fd, offset, SEEK_SET );
```

```
pos = lseek( fd, (off_t)0, SEEK_CUR);
```

fsetpos, fgetpos functions

When setting or getting the file postions for a file with the ANSI C functions of *fsetpos()* or *fgetpos()* respectively, the file position is specified by a value of type *fpos_t*. This type is defined as a *long* on Alpha.

Development Tools

The DEC OSF/1 Alpha system supplies a number of development tools that help migrate applications to this 64-bit system. These include an enhanced lint tools, special linker flags to help with truncation of 64-bit addresses, and a 32-bit compatibility mode.

lint

The lint(1) utility on both DEC OSF/1 MIPS and Alpha have been enhanced to find int and pointer assignments and argument passing.

ld

The Alpha linker, ld(1), loads the program text and data in the high 64 bit virtual address space of the process by default. This means that no valid addresses will be in the range of the 32-bit address. Therefore. unintended pointer truncations will trap into the kernel and cause a runtime error. This diagnostic feature assists in porting applications from 32 bit to 64 bit environments. To override this behavior you can use the -T/-D ld options to move the program in the address space.