The OpenVMS Mixed Pointer Size Environment

Thomas R. Benson
Karen L. Noel
Richard E. Peterson

A central goal in the implementation of 64-bit addressing on the OpenVMS operating system was to provide upward-compatible support for applications that use the existing 32-bit address space. Another guiding principle was that mixed pointer sizes are likely to be the rule rather than the exception for applications that use 64-bit address space. These factors drove several key design decisions in the OpenVMS Calling Standard and programming interfaces, the DEC C language support, and the system services support. For example, self-identifying 64-bit descriptors were designed to ease development when mixed pointer sizes are used. DEC C support makes it easy to mix pointer sizes and to recompile for uniform 32- or 64-bit pointer sizes. OpenVMS system services remain fully upward compatible, with new services defined only where required or to enhance the usability of the huge 64-bit address space. This paper describes the approaches taken to support the mixed pointer size environment in these areas. The issues and rationale behind these OpenVMS and DEC C solutions are presented to encourage others who provide library interfaces to use a consistent programming interface approach.

Support for 64-bit virtual addressing on the OpenVMS Alpha operating system, version 7.0, has vastly increased the amount of virtual address space available for application use.[1] At the same time, fully compatible support for applications that use only 32-bit addresses (also called *pointers*) has been preserved.

An application that mixes 32-bit and 64-bit pointer sizes operates in a *mixed pointer size environment.* Mixed pointer size applications were the design center for the initial implementation of 64-bit support in the OpenVMS operating system. This paper discusses the reasons why mixing pointer sizes is expected to be a common practice and describes the design of operating system and language features that are provided to ease programming in this mixed pointer size environment.

Reasons for Mixed Pointer Sizes

To use 64-bit address space, some simple applications need only be recompiled for a uniform 64-bit pointer size. For example, self-contained DEC C applications that rely on only the C run-time library, without using system services or other libraries, can take this approach. Real-world applications are seldom this clean-cut, however. In more complex applications, where 64-bit address space is likely to be needed, mixes of languages, dependencies on system interfaces and other libraries, and reliance on third-party packages or libraries are common. These practices all lead to the mixed pointer size environment in which applications continue to use some 32-bit addresses while taking advantage of 64-bit virtual address space.

Applications that are likely to take advantage of 64-bit memory are those in which the declaration and management of a large data set can be logically separated from the rest of the program. This separation does not need to be at the source file level. It can be at a program flow level, indicating which internal and external interfaces will be given 64-bit addresses to work with.

The following sections explore the reasons for mixing pointer sizes.

OpenVMS and Language Support

Implementation choices that Digital made for this first release of the OpenVMS operating system that supports 64-bit virtual addressing will probably encourage mixed pointer size programming. These choices were driven largely by the need for absolute upward compatibility for existing programs and the goal of supporting large, dynamic data sets as the primary application for 64-bit addressing.

Dynamic Data Only OpenVMS services support dynamic allocation of 64-bit address space. This mechanism most closely resembles the malloc and free functions for allocating and deallocating dynamic storage in the C programming language. Allocation of this type differs from static and stack storage in that explicit source statements are required to manage it. For static and stack storage, the system is allocating the memory on behalf of the application at image activation time. (Of course, the allocation may be extended during execution in the case of stack storage.) This allocation continues to be from 32-bit addressable space.

Two special cases of static allocation are worth mentioning. Linkage sections, which are program sections that contain routine linkage information, and code sections, which contain the executable instructions, do not differ substantially from preinitialized static storage. As a result, these sections also reside only in 32-bit addressable memory.

Upward-compatibility Constraints The OpenVMS Alpha operating system is cautious to avoid using 64-bit memory freely where it may prevent upward compatibility for 32-bit applications. For example, the linkage section might seem to be a natural candidate for the OpenVMS system to allocate automatically in 64-bit memory. This allocation would essentially free more 32-bit addressable memory for application use; however, even if this were done only for applications relinked for new versions of the OpenVMS operating system, there is no guarantee that all object code treats linkage section addresses as 64 bits in width. A simple example is storing the address of a routine in a structure. Since a routine's address is the address of its procedure descriptor in the linkage section, moving the linkage section to 64-bit memory would cause code that stores this address in a 32-bit cell to fail.

Allocating the user stack in 64-bit space also appears to be a good opportunity to easily increase the amount of memory available to an application. Stack addresses are often more visible to application code than linkage section addresses are. For instance, a routine can easily allocate a local variable using temporary storage on the stack and pass the address of the variable to another routine. If the stack is moved to 64-bit space, this address quietly becomes a 64-bit address. If the called routine is not 64-bit capable, attempts to use the address will fail.

Focus on Services Required for Large Data Sets Not all system services could be changed to support 64-bit addresses (i.e., *promoted*) in time for the first version of the OpenVMS operating system to support 64-bit addressing. With the mixed-pointer model in mind, we focused on those services that were likely to be required for large data sets. For example, to allow I/O directly to and from high memory, it was essential that the I/O queuing service, SYS$QIO, accept a 64-bit buffer address. Conversely, the SYS$TRNLNM service for translating a logical name did not need to be modified to accept 64-bit addresses. Its arguments include a logical name, a table name, and a vector that contains requests for information about the name. These are small data elements that are unlikely to require 64-bit addressing on their own. Of course, they may be part of some larger structure that resides in 64-bit space. In this case, they can easily be copied to or from 32-bit addressable memory.

System services are discussed further in the section OpenVMS System Services. The 32-bit address restriction on certain system services again emphasizes the importance of being able to logically separate large data set support from the rest of an application.

Limited Language Support Another interface point that requires care when using 64-bit addressing is at calls between modules written in different programming languages. The OpenVMS Calling Standard traditionally makes it easy to mix languages in an application, but DEC C is the only high-level language to fully support 64-bit addresses in the first 64-bit-capable version of the OpenVMS operating system.[2]

The use of 64-bit addresses in mixed-language applications is possible, and data that contains 64-bit addresses may even be shared; however, references that actually use the data pointed to by these addresses need to be limited to DEC C code or assembly language. Mixed high-level language applications are certain to be mixed pointer size applications in this version of the operating system.

Support for 32-bit Libraries

Many applications rely on library packages to provide some aspect of their functionality. Typical examples include user interface packages, graphics libraries, and database utilities. Third-party libraries may or may not support 64-bit addresses. Applications that use these libraries will probably mix 32-bit and 64-bit pointer sizes and will therefore require an operating system that supports mixed pointer sizes.

Implications of Full 64-bit Conversion

For some applications, it may be desirable to mix pointer sizes to avoid the side effects of universal 64-bit address conversion. The approach of recompiling everything with 64-bit address widths is sometimes called "throwing the switch." An obvious implication of throwing the switch is that all pointer data doubles in size. For complex linked data structures, this can be a significant overall increase in size. Increasing the pointer size may also reveal hidden dependencies on pointer size being the same as integer size. If code accesses a cell as both a 32-bit integer and a 32-bit pointer, the code will no longer work if the pointer is enlarged. Thus, universally increasing the pointer size may force changes to code that would otherwise continue to work.

There is a more compelling reason for not throwing the switch for code that is part of a shared library. Library packages must not return 64-bit addresses to users of the library unless the calling code is definitely 64-bit capable. If the library developer throws the switch when building a library written in DEC C, all memory returned by the malloc function will be in 64-bit address space. This can be a problem if the address is blindly returned to a library caller. If a library is to work in a mixed pointer size environment, and it sometimes returns pointers to memory it has allocated, it needs to use mixed pointer sizes internally.

Programming Interface Issues

The coexistence of 32-bit and 64-bit pointers raised several design questions for operating system and language support, particularly in the area of routine interfaces. When an application or library is being modified to use 64-bit address space, argument passing may be the most exposed area. In this section, we describe how mixed pointer size support affects argument-passing mechanisms and the design decisions made to ease the coexistence of mixed pointer sizes.

Argument List Width

Even before the introduction of 64-bit addressing, the OpenVMS Calling Standard defined argument list elements to be 64 bits in width. When passing a 32-bit address (that is, when passing an item in 32-bit space by reference), compilers sign extend the 32-bit value into the 64-bit argument location.[1] Passing 64-bit addresses as values works transparently without changing the calling standard, assuming, of course, that the called routine expects to receive 64-bit addresses. Passing 32-bit addresses as values to routines that expect 64-bit addresses works properly because the values have been sign extended to a 64-bit width.

Pointers by Reference

Passing the addresses of pointers requires special care when mixing pointer sizes. If the caller passes a 32-bit address by reference, and the called routine reads it as a 64-bit address from memory, the upper 32 bits will be incorrect. Similarly, if the address of a 64-bit address is passed, and the called routine reads only 32 bits from memory, it will fail when that address is used.

This is the simplest case in which support of 64-bit addresses may require a programming interface change for 64-bit callers. A single entry point that receives a pointer by reference cannot tell which size pointer it has received. Some possible solutions include a new alternate entry point for 64-bit-capable callers or a new parameter indicating the size of the address.

Pointers Embedded in Structures

Pointers passed by reference are a special case of the more general problem of passing structures that contain pointers. Again, the caller and called routine must agree on the size of the pointers contained in the structure. This case offers an option that may not require a new programming interface, however. If the structure is self-identifying, the routine may be able to tell which form of the structure it has received and dispatch to appropriate code for the corresponding pointer length.

Function Return Values

Function return values are also defined to be 64 bits in width, so no calling standard change was required to support 64-bit pointer returns. It is important that a 64-bit address not be returned blindly, though, unless it is known that the caller is 64-bit capable. Typically, this is a problem for library support routines rather than for those within an application. A library routine should return a 64-bit address only if the routine has been specifically developed for a 64-bit environment or if it can tell with certainty, based on input parameters received, that the caller is 64-bit capable.

Calling Standard Issues

The OpenVMS Calling Standard defines register usage conventions, argument list locations, data structures, and standard practices for making procedure calls that operate correctly in a multilanguage and multithreaded environment. As mentioned earlier, this standard already defined argument list elements to be 64 bits in width; however, some key data structures defined by the standard were based on 32-bit pointer sizes. The goal of upward compatibility for existing code complicated the job of extending the standard. The following sections describe how the structures were ultimately changed and illustrate some approaches to supporting mixed pointer sizes when shared structures contain pointers.

Descriptors Descriptors are structures defined by the calling standard to specify an argument's type, length, and address, along with other type or

structure-specific information. Typically, descriptors are used only for character strings, arrays, and complex data types such as packed decimal.

Descriptor types are by definition self-identifying by virtue of the type and class fields they contain. An obvious choice, therefore, for extending descriptors to handle 64-bit addresses would be to add new type constants for 64-bit data elements and extend the structure beyond the type fields to accommodate larger addresses and sizes. In practice, however, the address and length fields from descriptors are frequently used without accessing the type fields, particularly when a character string descriptor is expected.

As a result, a solution was sought that would yield a predictable failure, rather than incorrect results or data corruption, when a 64-bit descriptor is received by a routine that expects only the 32-bit form. The final design includes a separate 64-bit descriptor layout that contains two special fields at the same offsets as the length and address fields in the 32-bit descriptor. These fields are called MBO (must be one) and MBMO (must be minus one), respectively. The simplest versions of the 32-bit and 64-bit descriptors are illustrated in Figure 1.

If a routine that expects a 32-bit descriptor receives a 64-bit descriptor, it will find the value 1 in the length field. This nonzero value ensures that the address will need to be read. Otherwise, the descriptor could be treated as describing a null value, and the address would be ignored. In the address field, a 32-bit reader will find the value −1. When the reader attempts to reference this address, an access violation occurs, because the OpenVMS operating system guarantees this address to be inaccessible. This combination of values ensures that an access will also fail if the length is added to the address first, in an attempt to read the last byte of data.

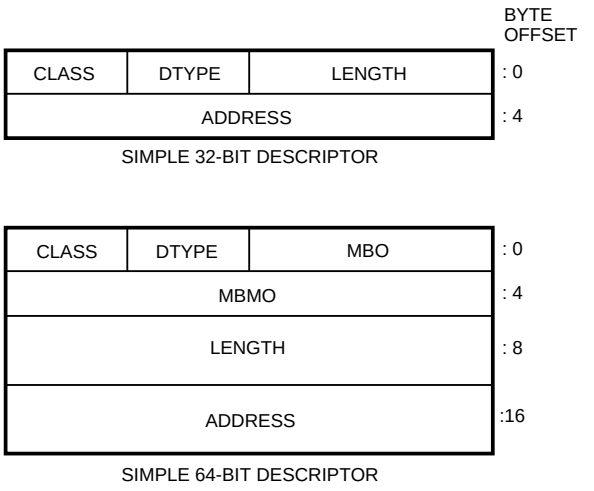

Figure 1
Simplest Versions of the 32-bit and 64-bit Descriptors

To distinguish the descriptor forms, a new routine must check the MBO and MBMO fields for the expected 64-bit descriptor values. In the OpenVMS operating system, many routines now accept either descriptor form.

Signal Arrays The signal array is a user-visible structure that is passed to condition handlers when an exception occurs. The array contains message codes, arguments specific to the conditions, and control data. Because the arguments may include one or more virtual addresses, a new format was necessary to accommodate 64-bit addresses.

The signal array could not simply be promoted to contain 64-bit addresses, because handlers in existing code often make assumptions about its format. The mechanism array, a related structure containing a snapshot of register contents, was already 64 bits in width.

The solution was to leave the original form of the signal array unchanged and create a 64-bit counterpart. The items passed to a condition handler, the 32-bit signal array address, and a 64-bit mechanism array address are the same. The mechanism array now contains a pointer to the 64-bit version of the signal array. This allows existing code to work without change, while new handlers that may require access to 64-bit addresses in exceptions can obtain the 64-bit array address from the mechanism array. Some additional work was needed in OpenVMS exception handling to keep these two arrays synchronized, because handlers are allowed to change their contents.

Sign-extension Checking

As described earlier, 32-bit addresses passed as routine arguments are sign extended into 64-bit argument locations. A safeguard that can be used in 32-bit routines that are not extended to fully support 64-bit addresses is referred to as sign-extension checking of the argument addresses. This checking consists of simply reading the low 32 bits of the argument, sign extending this value to a 64-bit width, and comparing the result to the full 64 bits of the argument. If the bits differ, the address is not one that can be represented in 32 bits. The routine can then return an error status of some kind, rather than failing in some unpredictable way. Sign-extension checking is a useful tool for ensuring robust interfaces in the mixed pointer size environment.

DEC C Language Support for Mixed Pointer Sizes

To support application programming in the mixed pointer size environment, some design work was required in the DEC C compiler. This section describes the rationale behind the final design.

It was clear that the compiler would have to provide a way for 32-bit and 64-bit pointers to coexist in the same regions of code. At the same time, customers and

internal users initially favored a simple command line switch when polled on potential compiler support for 64-bit address space. (At least one C compiler that supports 64-bit addressing, MIPSpro C, does so only through command line switches for setting pointer sizes.[3]) The motivation for using switches was to limit the source changes needed to take advantage of the additional address space, especially when portability to other platforms is desired. For cases in which mixing pointer sizes was unavoidable, something more flexible than a switch was needed.

Why Not __near and __far?

The most common suggestion for controlling individual pointer declarations was to adopt the __near and __far type qualifier syntax used in the PC environment in its transition from 16-bit to 32-bit addressing.[4] While this idea has merit in that it has already been used elsewhere in C compilers and is familiar to PC software developers, we rejected this approach for the following reasons:

- The syntax is not standard.
- The syntax requires source code edits at each declaration to be affected.
- The syntax has become largely obsolete even in the PC domain with the acceptance of the flat 32-bit address space model offered by modern 386-minimum PC compilers and the Win32 programming interface.
- Because of the vast difference in scale in choosing between 16-bit or 32-bit pointers on a PC as compared to choosing between 32-bit or 64-bit pointers on an Alpha system, there would be no porting benefit in using the same keywords. No existing source code base would be able to port to the OpenVMS mixed pointer size environment more easily because of the presence of __near and __far qualifiers.

Pragma Support

The Digital UNIX C compiler had previously defined pragma preprocessing directives to control pointer sizes for slightly different reasons than those described for the OpenVMS system.[5] By default, the Digital UNIX operating system offers a pure 64-bit addressing model. In some circumstances, however, it is desirable to be able to represent pointers in 32 bits to match externally imposed data layouts or, more rarely, to reduce the amount of memory used in representing pointer values. The Digital UNIX pointer_size pragmas work in conjunction with command line options and linker/loader features that limit memory use and map memory such that pointer values accessible to the C program can always be represented in 32 bits.

Since compatibility with the Digital UNIX compiler would have greater value if it met the needs of the OpenVMS platform, we evaluated the pragma-based approach and decided to adopt it, propagating any necessary changes back to the UNIX platform to maintain compatibility. The decision to use pragmas to control pointer size addressed the major deficiencies of the __near and __far approach. In particular, the pragma directive is specified by ISO/ANSI C in such a way that using it does not compromise portability as the use of additional keywords can, because unrecognized pragmas are ignored. Furthermore, pragmas can easily be specified to apply to a range of source code rather than to an individual declaration. A number of DEC C pragmas, including the pointer size controls implemented on the UNIX system, provide the ability to save and restore the state of the pragma. This makes them convenient and safe to use to modify the pointer size within a particular region of code without disturbing the surrounding region. The state may easily be saved before changing it at the beginning of the region and then restored at the end.

Command Line Interaction

Pragmas fit in with the initial desire of prospective users to have a simple command line switch to indicate 64 bits. As with several other pragmas, we defined a command line qualifier (/pointer_size) to specify the initial state of the pragma before any instances are encountered in the text. Unlike other pragmas, though, we also use the same command line qualifier to enable or disable the action of the pragmas altogether. In this way, a default compilation of source code modified for 64-bit support behaves the same way that it would on a system that did not offer 64-bit support. That is, the pragmas are effectively ignored, with only an informational message produced.

This behavior was adopted for consistency with the Digital UNIX behavior and also to aid in the process of adding optional 64-bit support to existing portable 32-bit source code that might be compiled for an older system or with an older compiler. In this model, a compilation of new source code using an old command line produces behavior that is equivalent to the behavior produced using an older compiler or a compiler on another platform. With one notable exception, building an application that actually uses 64-bit addressing requires changing the command line.

The exception to the rule that existing 32-bit build procedures do not create 64-bit dependencies is a second form of the pragma, named required_pointer_size. This form contrasts with the form pointer_size in that it is always active regardless of command line qualifiers; otherwise, required_pointer_size and pointer_size are identical. The intent of this second pragma is to support writing source code that specifies or interfaces to services or libraries that can only work correctly with 64-bit pointers. An example of this code might be a header file that contains declarations for both 64-bit and 32-bit memory management services; the services

must always be defined to accept and return the appropriate pointer size, regardless of the command line qualifier used in the compilation.

Pragma Usage

The use of pragmas to control pointer sizes within a range of source code fits well with the model of starting with a working 32-bit application and extending it to exploit 64-bit addressing with minimal source code edits. Programming interface and data structure declarations are typically packaged together in header files, and the primary manipulators of those data structures are often implemented together in modules.

One good approach for extending a 32-bit application would be to start with an initial analysis of memory usage measurements. The purpose of this analysis would be to produce a rough partitioning of routines and data structures into two categories: "32-bit sufficient" and "64-bit desirable." Next, 64-bit pointer pragmas could be used to enclose just the header files and source modules that correspond to the routines and data structures in the 64-bit-desirable category. After recompilation, the next step would be to respond to compiler diagnostics for pointer-type mismatches by adding pragma regions to mark sections of the 64-bit files as 32-bit and parts of the 32-bit files as 64-bit and to carefully add type casts, where necessary. This operation is likely to iterate until the compilation is clean and a debugging cycle has shown correctness. The end result is an application that takes advantage of the increased address space for the data structures that will benefit from it.

A common approach to minimizing the spread of pragmas throughout a program is to limit them to typedefs in header files. Then, subsequent uses of the defined type do not require the pragma. A simple example appears in Figure 2.

This example defines a type called char_ptr64, which may be used to declare 64-bit pointers to character data without the use of pragmas. Of course, individual pointers within structure types may also be set to 64-bit or 32-bit sizes.

Secondary Effects

With the decision made to use pragmas and the basic semantics of how the pragmas take effect established by the Digital UNIX implementation, we needed to consider additional requirements and issues that might be specific to the OpenVMS implementation. Two major differences between the platforms are

1. On the Digital UNIX system, the linker/loader options used with mixed pointer size compilations ensure that any address value obtained by the program can be represented using 32 bits, whereas on the OpenVMS system, any program using 64-bit pointers in C will almost certainly encounter address values that cannot be represented in 32 bits.
2. On the Digital UNIX system, the scope of the use of mixed pointer sizes was expected to be quite small and not likely to grow much over time, whereas on the OpenVMS system, the scope is expected to be somewhat larger at first and grow significantly over time.

These two differences emphasized the need for effective compile-time diagnostics, debugging aids, environmental support, and clear documentation.

Diagnostics As an aid to finding bugs resulting from improper mixing of pointer sizes, the DEC C compiler provides two kinds of diagnostics. Compile-time warnings are issued for assignments from long pointers to short pointers because of the possibility of data loss. In addition, users may enable run-time checking for pointer truncation through a command line qualifier. This option causes the compiler to generate code on each conversion from a long to a short pointer, which will signal a range-check error if data truncation occurs.

Run-time checking is particularly useful in code that sometimes employs type casting to use long pointers in short pointer contexts. Since this action prevents a compile-time warning about using a long pointer where a short pointer is expected, a run-time check may be the only way to discover a coding error. The run-time check qualifier provides options distinguishing this case from checking on general assignments and parameter passing, allowing users to select for which classes of pointer-size mixing the compiler should generate checking code. Run-time checking is also available for parameters received by a routine. This allows detection of 64-bit addresses passed to routines that expect 32-bit parameters even when the caller is separately compiled or written in a different programming language. For performance reasons, it is usually desirable to remove all run-time checking once a program is debugged.

```
#pragma required_pointer_size save    /* Save the previous pointer size */
#pragma required_pointer_size 64      /* Set pointer size to 64 bits */
typedef char * char_ptr64;            /* Define a 64-bit char pointer */
#pragma required_pointer_size restore /* Restore the pointer size */
```

Figure 2
Sample Header File Code That Limits Pragmas to Defined Types

Allocation Function Mapping The command line qualifier setting the default pointer size has an additional effect that simplifies the use of 64-bit address space. If an explicit pointer size is specified on the command line, the malloc function is mapped to a routine specific to the address space for that size. For example, _malloc64 is used for malloc when the default pointer size is 64 bits. This allows allocation of 64-bit address space without additional source changes. The source code may also call the size-specific versions of run-time routines explicitly, when compiled for mixed pointer sizes. These size-specific functions are available, however, only when the /pointer_size command line qualifier is used. See "Adding 64-bit Pointer Support to a 32-bit Run-time Library" in this issue for a discussion of other effects of 64-bit addressing on the C run-time library.[6]

Header File Semantics The treatment of pointer_size pragmas in and around header files (i.e., any source included by the #include preprocessing directive) deserves special mention. Programs typically include both private definition files and public or system-specific header files. In the latter case, it may not be desirable for definitions within the header files to be affected by the pointer_size pragmas or command line currently in effect. To prevent these definitions from being affected, the DEC C compiler searches for special prologue and epilogue header files when a #include directive is processed. These files may be used to establish a particular state for environmental pragmas, such as pointer_size, for all header files in the directory. This eliminates the need to modify either the individual header files or the source code that includes them.

The compiler creates a predefined macro called __INITIAL_POINTER_SIZE to indicate the initial pointer size as specified on the command line. This may be of particular use in header files to determine what pointer size should be used, if mixed pointer size support is desirable. Conditional compilation based on this macro's definition state can be used to set or override pointer size or to detect compilation by an older compiler lacking pointer-size support. If its value is zero, no /pointer_size qualifier was specified, which means that pointer_size pragmas do not take effect. If its value is 32 or 64, pointer_size pragmas do take effect, so it can be assumed that mixed pointer sizes are in use.

Code Example

In the simple code example shown in Figure 3, suppose that the routine proc1 is part of a library that has been only partially promoted to use 64-bit addresses. This function may receive either a 32-bit address or a 64-bit address in the *argument_ptr* parameter. To demonstrate the use of the new DEC C features, proc1 has been modified to copy this character string parameter from 64-bit space to 32-bit space when necessary, so that routines that proc1 subsequently calls need to deal with only 32-bit addresses.

The __INITIAL_POINTER_SIZE macro is used to determine if pointer_size pragmas will be effective and, hence, whether *argument_ptr* might be 64 bits in width. If it might be a 64-bit pointer, whose actual width is unknown in this example, the pointer's value is copied to a 32-bit-wide pointer. The pointer_size pragma is used to change the current pointer size to 32 bits to declare the temporary pointer. Next, the two pointer values are compared to determine if the original pointer fits in 32 bits. If the pointer does not fit, temporary storage in 32-bit addressable space is allocated, and the argument is copied there. Note that the example uses _malloc32 rather than malloc, because malloc would allocate 64-bit address space if the initial pointer size was 64 bits. At the end of the routine, the temporary space is freed, if necessary.

A type cast is used in the assignment from *argument_ptr* to *temp_short_ptr,* even though both variables are of type char *. Without this type cast, if *argument_ptr* is a 64-bit-wide pointer, the DEC C compiler would report a warning message because of the potential data loss when assigning from a 64-bit to a 32-bit pointer.

For other examples of pointer_size pragmas and the use of the __INITIAL_POINTER_SIZE macro, see Duane Smith's paper on 64-bit pointer support in run-time libraries.[6]

OpenVMS System Services

The OpenVMS operating system provides a suite of services that perform a variety of basic operating system functions.[7] Design work was required to maximize the utility of these routines in the new mixed pointer size environment. Issues that needed to be addressed included the following, which are discussed in subsequent sections:

- Several services pass pointers by reference and, hence, required an interface change.
- Because of resource constraints, not all system services could be promoted to handle 64-bit addresses in the first version of the 64-bit-capable OpenVMS operating system.
- Since the services provide mixed levels of support, it is important to indicate those that support 64-bit addresses and those that do not.
- Certain new services seemed desirable to improve the usability of 64-bit address space.

Services That Are 64-bit Friendly

Services that can be promoted to support 64-bit addresses without any interface change are called 64-bit friendly. If a service receives an address by reference, the service is typically not 64-bit friendly, and a separate

```
void proc1(char * argument_ptr)
{
#if __INITIAL_POINTER_SIZE != 0
   #pragma pointer_size save
   #pragma pointer_size 32
   char * temp_short_ptr;
   temp_short_ptr = (char *)argument_ptr;
   if (temp_short_ptr != argument_ptr) {
       temp_short_ptr = _malloc32(strlen(argument_ptr) + 1);
       strcpy(temp_short_ptr,argument_ptr);
       argument_ptr = temp_short_ptr;
   }
   else {
      temp_short_ptr = 0;
   }
   #pragma pointer_size restore
#endif

/*
    The actual body of proc1 is omitted. Assume that it calls
    routines that operate on the data pointed to by argument_ptr
    and that the routines are not yet prepared to handle 64-bit
    addresses.
*/

#if __INITIAL_POINTER_SIZE != 0
if (temp_short_ptr != 0)
    free(temp_short_ptr);
#endif
}
```

Figure 3
Code Example of Pointer_size Pragmas and the __INITIAL_POINTER_SIZE Macro

entry point is required to support 64-bit addresses. A single routine cannot distinguish whether the address at the specified location is 32 bits or 64 bits in width.

If a service does not receive or return an address by reference, the service is usually 64-bit friendly. Even descriptor arguments present no problem, because the 32-and 64-bit versions can be distinguished at run time. The majority of services fall into this category.

The services that are not 64-bit friendly include the entire suite of memory management system services, since they access address ranges passed by reference. Other such services include those that receive a 32-bit vector as an argument, which may include the address of a pointer as an element. A good example from this group is SYS$FAOL, which accepts a 32-bit vector argument for formatted output. For all these services, new interfaces were designed to accommodate 64-bit callers.

Promotion of Services

The OpenVMS project team explored the idea of promoting all system services to support 64-bit addresses. Since the majority of OpenVMS system service routines are written in the MACRO-32 assembly language or the Bliss-32 programming language, the internals of the routines could not be promoted to handle 64-bit addresses without modifications. We could not take advantage of the throw-the-switch approach, and we did not want to because many pointers used internally in the OpenVMS operating system remain at 32 bits.

We considered using 64-bit jacket routines to copy 64-bit arguments to the stack in 32-bit space, which would then call the 32-bit internal routine to perform the requested function. However, this approach would fail for context arguments such as asynchronous system trap (AST) routine parameters, where the address of the argument is stored for subsequent use. This approach would also prevent services from operating on any true 64-bit addresses. It was clear that at least some routines would have to be modified internally.

The idea of using jacket routines was ultimately rejected for several reasons. First, the jackets would need to be custom written to ensure correct parameter semantics. There could not be a "common jacket" that could have saved time and lowered risk. Second, there would be an undesirable performance impact for 64-bit callers. Third, we decided that having a complete 64-bit system service suite was not essential for usable 64-bit support. We could define a subset that would meet the needs of 64-bit address space users, while lowering our risk and implementation costs.

The services selected for 64-bit support fall into four categories.

1. Memory management services.
2. Performance-critical services. This group includes services that are typically sensitive to the addition of

even a few cycles of execution time. Requiring that a 64-bit address user do any additional work, such as copying data to 32-bit space, is undesirable. An example of this type of service is SYS$ENQ, which is used for queuing lock requests.

3. Design center services. The primary design center for 64-bit support was database applications. Database architects and consultants were polled to determine which services were most needed by their products. Many of these services, for example SYS$QIO for queuing I/O requests, were also in the performance-critical set.
4. Other useful basic services. This set includes services to ease the transition to 64 bits with minimal change to program structure. For example, the SYS$CMKRNL service accepts a routine address and a vector of 32-bit arguments and invokes the routine in kernel mode, passing those arguments. Without a new 64-bit version of SYS$CMKRNL, a caller could not pass a 64-bit address to the kernel mode routine without changing the form of the argument block, such as passing a structure that SYS$CMKRNL would not interpret as a vector.

Several steps were taken to ease programming to this subset implementation.

- For all 64-bit services, *all* pointer arguments may be in 64-bit space. Extending only individual arguments for some services would have been confusing and difficult to document.
- The 64-bit-capable system services are clearly listed in the OpenVMS documentation, and the documentation for individual services clearly calls out their capabilities.[7,8]
- For C programmers, the header file that defines function prototypes for system services (STARLET.H) defines the expected pointer size for service arguments. This file can be used for compile-time type checking for correct argument pointer sizes.
- A strict naming convention has been adhered to for 64-bit services. If a routine was 64-bit friendly, i.e., it required no interface change, its name was not changed. If a new entry point was required because, for example, an address is passed by reference, a "_64" suffix was added to the name to identify the new entry point.
- Sign-extension checking is performed in routines that do not accept 64-bit addresses.

Centralized Sign-extension Checking

For services that have not been promoted to accept arguments in 64-bit space, centralized sign-extension checking takes place. As described in the section Sign-extension Checking, such checking prevents errors that occur when a 64-bit address is erroneously passed to a routine that uses only 32-bit addresses. This centralized checking is part of the system service dispatcher, which returns the error status SS$_ARG_GTR_32_BITS when the error is discovered. Performing the checking at this common point minimized the implementation effort, while protecting sensitive inner mode services. No changes were necessary to the modules that contain the 32-bit service code. The internal vector of services contains a flag for each service indicating whether checking should be done.

Naturally, it is best for mixed-size errors to be discovered at compile time. The DEC C compiler issues a warning message when a 64-bit pointer is used as a parameter to a routine whose function prototype specifies that the parameter should be a 32-bit pointer. Run-time sign-extension checking works for any language, though, including MACRO-32.

This support can also be used to allow a run-time decision to be made to copy data from 64-bit space to 32-bit space. For example, a routine could call a system service, passing along an address that it had received as a parameter. If the service returns SS$_ARG_GTR_32_BITS, the calling routine can then copy the argument to the stack and retry the service. In this way, the overhead of copying can be avoided if copying is unnecessary. When the system service is promoted to handle 64-bit addresses in a future version of the OpenVMS operating system, no change will be needed in this caller; the data copying code will never be invoked. This approach may be appropriate for a run-time library that needs to be fully 64-bit capable today on OpenVMS Alpha version 7.0, if that library will not be rereleased for a future version of the OpenVMS operating system.

Memory Management System Services

The OpenVMS memory management system services are not 64-bit friendly because they pass 32-bit input and output address arguments by reference. This set of services includes SYS$EXPREG (expand program/control region), SYS$MGBLSC (map global section), SYS$CRMPSC (create and map section), and SYS$PURGWS (purge working set), among others.

The guiding principle in promoting these services was that the new 64-bit services had to perform the same functions as their 32-bit counterparts but not necessarily with an identical interface. Since 32-bit addresses can be expressed as 64-bit addresses with sign-extension bits in the upper 32 bits, it made sense to accommodate 32-bit addresses in the 64-bit interfaces, making the new services a superset of the 32-bit forms. For example, the SYS$CRMPSC service was split into multiple 64-bit-capable services, because it handles a variety of types of sections. The new services can operate on either 32-bit or 64-bit addresses and have simpler

interfaces than the 32-bit-only SYS$CRMPSC. The original SYS$CRMPSC is still present so that existing code may function without change.

Some new feature requests were considered as part of the 64-bit effort, but, to maintain the focus of the release, these features were not implemented. The 64-bit memory management services were designed to more easily accommodate new features in the future. For example, the new services check the argument count for both too many and too few supplied arguments. In this way, new optional arguments can be added later to the end of the list without jeopardizing backward compatibility.

Virtual Regions

One new feature that was added to the suite of 64-bit memory management services is support for new entities called virtual regions. A virtual region is an address range that is reserved by a program for future dynamic allocation requests. The region is similar in concept to the program region (P0) and the control region (P1), which have long existed on the OpenVMS operating system.[9] A virtual region differs from the program and control regions in that it may be defined by the user by calling a system service and may exist within P0, P1, or the new 64-bit addressable process-private space, P2.[1] When a virtual region is created, a handle is returned that is subsequently used to identify the region in memory management requests.

Address space within virtual regions is allocated in the same manner as in the default P0, P1, and P2 regions, with allocation defined to expand space toward either ascending or descending addresses. As in the default regions, allocation is in multiples of pages. The OpenVMS operating system keeps track of the first free virtual address within the region. A region can be created such that address space is created automatically when a virtual reference is made within the region, just as the control region in P1 space expands automatically to accommodate user stack expansion. When a virtual region is created within P0, P1, or P2, the remainder of that containing region decreases in size so that it does not overlap with the virtual region.

Virtual regions were added to the OpenVMS Alpha operating system along with the 64-bit addressing capability so that the huge expanse of 64-bit address space could be more easily managed. If a subsystem requires a large portion of virtually contiguous address space, the space can be reserved within P2 with little overhead. Other subsystems within the application cannot inadvertently interfere with the contiguity of this address space. They may create their own regions or create address space within one of the default regions.

Another advantage of using virtual regions is that they are the most efficient way to manage sparse address space within the 64-bit P2 space. Furthermore, no quotas are charged for the creation of a virtual region. The internal storage for the description of the region comes from process address space, which is the only resource used.

Summary

This paper presents the reasons behind the new OpenVMS mixed pointer size environment and the support added to allow programming within this environment. The discussion touches on some of the new support designed to simplify the use of the 64-bit address space.

The approaches discussed yielded full upward compatibility for 32-bit applications, while allowing other applications access to the huge 64-bit address space for data sets that require it. Promotion of all pointers to 64-bit width is not required to use 64-bit space; the mixed pointer size environment was considered paramount in all design decisions. A case study of adding 64-bit support to the C run-time library also appears in this issue of the *Journal*.[6]

Acknowledgments

The authors wish to thank the other members of the 64-bit Alpha-L Team who helped shape many of the ideas presented in this paper: Mark Arsenault, Gary Barton, Barbara Benton, Ron Brender, Ken Cowan, Mark Davis, Mike Harvey, Lon Hilde, Duane Smith, Cheryl Stocks, Lenny Szubowicz, and Ed Vogel.

References

1. M. Harvey and L. Szubowicz, "Extending OpenVMS for 64-bit Addressable Virtual Memory," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 57–71.

2. *OpenVMS Calling Standard* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBBA-TE, 1995).

3. *MIPSpro 64-Bit Porting and Transition Guide,* Document No. 007-2391-002 (Mountain View, Calif.: Silicon Graphics, Inc., 1996).

4. *C Language Reference for MS-DOS and Windows Operating Systems* (Redmond, Wash.: Microsoft Corporation, 1991) and "Declarations and Types," chap. 3, and "Expressions and Assignments," chap. 4, in *Microsoft C/C++ Version 7.0* (Redmond, Wash.: Microsoft Corporation, 1991).

5. *Digital UNIX Programmer's Guide* (Maynard, Mass.: Digital Equipment Corporation, 1996).

6. D. Smith, "Adding 64-bit Pointer Support to a 32-bit Run-time Library," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 83–95.

7. *OpenVMS System Services Reference Manual: A-GETMSG* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBMA-TE, 1995) and *OpenVMS System Services Reference Manual: GETQUI-Z* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBN-TE, 1995).

8. *OpenVMS Alpha Guide to 64-Bit Addressing* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBCA-TE, 1995).

9. T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford, Mass.: Digital Press, 1987).

Biographies

Thomas R. Benson
A consulting engineer in the OpenVMS Engineering Group, Tom Benson is one of the developers of 64-bit addressing support. Tom joined Digital's VAX Basic project in 1979 after receiving B.S. and M.S. degrees in computer science from Syracuse University. After working on an optimizing compiler shell used by several VAX compilers, he joined the VMS Group where he led the VMS DECwindows FileView and Session Manager projects, and brought the Xlib graphics library to the VMS operating system. Tom holds three patents on the design of the VAX MACRO-32 compiler for Alpha and recently applied for two patents related to 64-bit addressing work.

Karen L. Noel
A principal engineer in the OpenVMS Engineering Group, Karen Noel is one of the developers of 64-bit addressing support. After receiving a B.S. in computer science from Cornell University in 1985, Karen joined Digital's RSX Development Group. In 1990, she joined the VMS Group and ported several parts of the VMS kernel from the VAX platform to the Alpha platform. As one of the principal designers of OpenVMS Alpha 64-bit addressing support, she has recently applied for six software patents.

Richard E. Peterson
Rich Peterson joined Digital's DEC C/C++ team in 1992. He was the project leader for the development of the C and C++ compilers that joined the Microsoft front ends to the GEM back end. These compilers were used to build and deliver the first release of the Windows NT operating system on the Alpha platform and later were used in Visual C++ for Alpha. A principal software engineer in the Core Technologies Group, Rich is currently the project leader for DEC C on the Digital UNIX and OpenVMS platforms. Prior to joining Digital, Rich worked at Intermetrics on a number of compiler projects, including HAL/S for the Space Shuttle and Ada for IBM/370 and MIL-STD 1750A. Rich also worked at COMPASS, where he was the project leader for the Thinking Machines Fortran compiler and Digital's initial MPP compiler effort. He received a B.S. in English from the California Institute of Technology and has applied for one patent on Alpha OpenVMS 64-bit compiler work.