Duane A. Smith

Adding 64-bit Pointer Support to a 32-bit Run-time Library

A key component of delivering 64-bit addressing on the OpenVMS Alpha operating system, version 7.0, is an enhanced C run-time library that allows application programmers to allocate and utilize 64-bit virtual memory from their C programs. This C run-time library includes modified programming interfaces and additional new interfaces yet ensures upward compatibility for existing applications. The same run-time library supports applications that use only 32-bit addresses, only 64-bit addresses, or a combination of both. Source code changes are not required to utilize 64-bit addresses, although recompilation is necessary. The new techniques used to analyze and modify the interfaces are not specific to the C run-time library and can serve as a guide for engineers who are enhancing their programming interfaces to support 64-bit pointers.

The OpenVMS Alpha operating system, version 7.0, has extended the address space accessible to applications beyond the traditional 32-bit address space. This new address space is referred to as 64-bit virtual memory and requires a 64-bit pointer for memory access.[1] The operating system has an additional set of new memory allocation routines that allows programs to allocate and release 64-bit memory. In OpenVMS Alpha version 7.0, this set of routines is the only mechanism available to acquire 64-bit memory.

For application programs to take advantage of these new OpenVMS programming interfaces, high-level programming languages such as C had to support 64-bit pointers. Both the C compiler and the C run-time library required changes to provide this support. The compiler needed to understand both 32-bit and 64-bit pointers, and the run-time library needed to accept and return such pointers.

The compiler has a new qualifier called /pointer_size, which sets the default pointer size for the compilation to either 32 bits or 64 bits. Also added to the compiler are pragmas (directives) that can be used within the source code to change the active pointer size. An application program is not required to compile each module using the same /pointer_size qualifier; some modules may use 32-bit pointers while others use 64-bit pointers. Benson, Noel, and Peterson describe these compiler enhancements.[2] The *DEC C User's Guide for OpenVMS Systems* documents the qualifier and the pragmas.[3]

The C run-time library added 64-bit pointer support either by modifying the existing interface to a function or by adding a second interface to the same function. Public header files define the C run-time library interfaces. These header files contain the publicly accessible function prototypes and structure definitions. The library documentation and header files are shipped with the C compiler; the C run-time library ships with the operating system.

This paper discusses all phases of the enhancements to the C run-time library, from project concepts through the analysis, the design, and finally the implementation. The *DEC C Runtime Library Reference Manual for OpenVMS Systems* contains user documentation regarding the changes.[4]

Starting the Project

We devoted the initial two months of the project to understanding the overall OpenVMS presentation of 64-bit addresses and deciding how to present 64-bit enhancements to customers. Representatives from OpenVMS engineering, the compiler team, the run-time library team, and the OpenVMS Calling Standard team met weekly with the goal of converging on the deliverables for OpenVMS Alpha version 7.0.

The project team was committed to preserving both source code compatibility and the upward compatibility aspects of shareable images on the OpenVMS operating system. Early discussions with application developers reinforced our belief that the OpenVMS system must allow applications to use 32-bit and 64-bit pointers within the same application. The team also agreed that for a mixed-pointer application to work effectively, a single run-time library would need to support both 32-bit and 64-bit pointers; however, there was no known precedent for designing such a library.

One implication of the decision to design a run-time library that supported 32-bit and 64-bit pointers was that the library could never return an unsolicited 64-bit pointer. Returning a 64-bit pointer to an application that was expecting a 32-bit pointer would result in the loss of one half of an address. Although typically this error would cause a hardware exception, the resulting address could be a valid address. Storing to or reading from such an address could result in incorrect behavior that would be difficult to detect.

The *OpenVMS Calling Standard* specifies that arguments passed to a function be 64-bit values.[5] If a 32-bit address is used, it is always sign extended to form a 64-bit address that can be used by the Alpha hardware. The C run-time library team exploited this fact when creating the 64-bit interface to the library.

The team also agreed that using 64-bit pointers should be as simple as possible; the simplest mode would allow the application to compile using the qualifier /pointer_size=64 without making source code changes. The design of 64-bit support must appear as a logical extension to the C programming environment in use today. Furthermore, applications written to conform strictly to the ANSI standard must be able to use 64-bit pointers while remaining conformant. For example, allocating 64-bit virtual memory would be an extension to the standard C memory management functions malloc, calloc, realloc, and free.

This paper shows that each of the C run-time library interfaces examined falls into one of the following four categories (listed in order of added complexity to library users):

1. Not affected by the size of a pointer
2. Enhanced to accept both pointer sizes
3. Duplicated to have a 64-bit-specific interface
4. Restricted from using 64-bit pointers

One last point to come from the meetings was that many of the C run-time library interfaces are implemented by calling other OpenVMS images. For example, the Curses Screen Management interfaces make calls to the OpenVMS Screen Management (SMG) facility. It is important that the C run-time library defines the interfaces to support 64-bit addresses without looking at the implementation of the function. Consistency and completeness of the interface are more important than the complexity of the implementation. In the SMG example, if the C run-time library needs to make a copy of a string prior to passing the string to the SMG facility, this is what will be implemented.

Analyzing the Interfaces

The process of analyzing the interfaces began by creating a document that listed all the header files and the definitions in these files. A total of 50 header files that contained approximately 50 structures and 500 prototypes needed to be analyzed. Each structure or prototype had to be examined to see if a change in pointer size would affect the interface. Keep in mind that we analyzed only the interfaces; we did not examine the underlying implementation changes that would be required.

Analyzing the Structures

It is necessary to distinguish between a structure, which may contain pointers, and a pointer to the structure itself. For example, the div_t structure contains two integer fields. Although the size of the pointer to div_t does not affect the contents of the structure, the entire structure may be allocated in 32-bit or 64-bit virtual memory. Functions that accept a pointer to such a structure may need to be modified to accommodate the 64-bit case. The div_t structure is

```
typedef struct {
  int quot, rem;
} div_t;
```

Many structures used in the C run-time library interfaces are allocated by the run-time library in response to a function call. For example, a call to the fopen function returns the following pointer to the FILE structure:

```
FILE *fopen(const char *filename,
            const char *mode);
```

The C run-time library always allocates FILE structures in 32-bit virtual memory and returns a 32-bit pointer to the calling program. This important concept can dramatically impact the use of 64-bit pointers

in structures. If a FILE pointer is always a 32-bit pointer, structures that contain only FILE pointers are not affected by the choice of pointer size. We use this information when we look at the layout of structures and examine function prototypes that accept pointers to structures.

In this paper, structures that are always allocated in 32-bit virtual memory are referred to as structures bound to low memory. After determining which structures are bound to low memory, we examine the layout of each structure to decide if the structure is affected by pointer size. We keep in mind that pointer size does not affect a structure that is bound to low memory.

For upward compatibility, the analysis must always consider existing software that depends on the layout of the structure. In the case of public header file analysis, such dependence will probably always be present. An application may have executable code that, for example, fetches 4 bytes beginning at byte 12 of the structure and dereferences those 4 bytes as the address of a string.

For these public structures, the analysis must weigh the impact of forcing these structures to be 32-bit pointers. If the decision is made to allocate two different structure types, each function that either returns or is passed such a structure must have a pointer-size-specific implementation. The case analysis and further details appear in the section Pointer to Pointer-size-sensitive Structures.

Analyzing the Function Prototypes

Analyzing functions only requires looking at the function prototypes. To determine the effect of pointer size on a function, we look at each parameter and return value that involves a pointer. This section describes the types of situations that are affected by pointer size, from the simplest type to the most complex. Note that when a program passes an array of any type to a function, the array is passed as a pointer and must be considered.

Making 64-bit-friendly Parameters As mentioned in the section Starting the Project, the *OpenVMS Calling Standard* specifies that a 32-bit address is sign extended to a 64-bit address when passed as an argument to a function. This implies that existing programs that pass addresses as parameters are already sign extending those 32-bit addresses to be passed as 64-bit quantities. Each 32-bit address can, therefore, be expressed as a 64-bit address in which the top 32 bits are zero.

This sign-extending scheme allows the run-time library to have a single implementation that can be used by both 32-bit and 64-bit calling programs. This implementation would be modified to accept only 64-bit addresses. An implementation that supports parameters of either pointer size is referred to as being 64-bit friendly. The function strlen is an example of a 64-bit-friendly function.

```
size_t strlen(const char *string);
```

The *string* parameter is the only part of the strlen function that the pointer size affects. To support 64-bit addressing, the strlen function had to be modified to accept a 64-bit pointer.

Parameters Bound to Low Memory In structures bound to low memory, the addresses that the programs pass are always 32-bit addresses. One explanation is that the structures are managed by the run-time library, and the only method of creating, destroying, or obtaining the addresses of these structures is by calling a library routine. Given that a single library services both 32-bit and 64-bit calling programs, the library does not change the structures based on command qualifiers, nor does it allocate the structures in 64-bit virtual memory. For user convenience, the C run-time library implemented these pointers as 32-bit return values but 64-bit-friendly parameters.

The reason for this design became apparent while testing the 64-bit interfaces to the library. Consider the following code fragment, which exists in many applications:

```
FILE *fp;
char buffer[100];
fp = fopen("the_file", "r");
fread(array, sizeof(buffer), 1, fp);
```

The C run-time library always allocates a FILE structure in 32-bit virtual memory. When the previous code fragment is compiled using /pointer_size=64, *fp* is declared as a 64-bit pointer to a FILE structure, because using this qualifier specifies the default pointer size to be used. When the fopen function returns the 32-bit pointer, the return value is sign extended into the 64-bit FILE pointer. If the fourth parameter of the fread function had been declared as a 32-bit FILE pointer, the compiler would report an error when the 64-bit FILE pointer *fp* was passed as an argument. This example explains why the C run-time library declares structures bound to low memory as 32-bit return values but 64-bit parameters.

Parameters Restricted to Low Memory Structures restricted to low memory are similar to those bound to low memory except that the user allocates the structures and can allocate the structures in high memory. The C run-time library cannot support the allocation of such structures in 64-bit virtual memory.

An example of a parameter being restricted to a low memory address is the buffer being passed as the parameter to the function setbuf. The parameter defines this buffer to be used for I/O operations. The user expects to see this buffer change as I/O operations are performed on the file. If the run-time library made a copy of this buffer, the changes would appear in the copy and not in the original buffer that the user supplied. When the C run-time library begins to use the 64-bit OpenVMS Record Management Services (RMS) interface, this low-memory restriction will be removed.

In most cases, the run-time library is able to hide the fact that the 32-bit RMS interface is not able to interpret a 64-bit virtual memory address. Consider the *filename* parameter to the fopen function. If the parameter is a 64-bit virtual memory address, the run-time library copies this parameter to 32-bit virtual memory and passes the address of the copy to RMS. Neither the user nor RMS is aware that this copy has been made. The library may copy the data if and only if such a copy operation does not change functionality or significantly degrade performance.

Size-independent Structure Pointers Many functions receive the address of a structure whose layout is not affected by pointer size. The simplest address in this category is that of an array of integers. This array may be in either 32-bit or 64-bit virtual memory, but only one interface is needed to determine the layout of the structure. If the structure layout is independent of pointer size, then pointer-size-specific entry points are not required for this parameter. The developer would still make the parameter 64-bit friendly so that the user would have the freedom to make the allocation that is best for the application.

Pointer to Pointer Parameters It is common practice for a function to be passed a pointer to a pointer. If the pointer being pointed to is not bound or restricted to a 32-bit address, then two implementations of the function are necessary.

To understand why some functions require two implementations, consider the following strtod function:

```
double strtod(const char *string,
              char **endptr);
```

The strtod function converts a string to a floating-point double-precision number. The second parameter to this function, *endptr,* is a pointer to a memory location into which the address of the first unrecognized character is to be placed. The caller of this function has allocated either 4 or 8 bytes to store this address. Without pointer-size-specific entry points, the function has no way of determining how many bytes to write. Writing 4 bytes may truncate a pointer; writing 8 bytes may overwrite 4 bytes of user data that follows the pointer. The strtod function, therefore, has two implementations. The first expects *endptr* to be the address of a 32-bit pointer, and the second expects *endptr* to be the address of a 64-bit pointer.

Pointer to Pointer-size-sensitive Structures Many functions receive the address of a structure. If the analysis reveals that the layout of this structure is dependent upon pointer size, the functions that receive or return this structure must have pointer-size-specific entry points.

Note that the layout of the structure is separate from whether the structure is allocated in low memory or in high memory. The 32-bit-specific entry point is needed to understand the layout of the structure, but the parameter should allow this structure to be allocated in high memory.

Functions that receive the address of an array of addresses are treated in the same way, assuming that the addresses in the array are neither bound nor restricted to low memory. The function being called needs to know if the array contains 32-bit addresses or 64-bit addresses. Unlike the address of the array, the individual members of the array are not sign extended to 64-bit values.

Separate implementations are necessary only to determine the layout of what is being pointed to. The 32-bit interface handles pointers to structures containing 32-bit addresses, and the 64-bit interface handles pointers to structures containing 64-bit addresses.

Functions That Return Pointers Many functions return pointers as the value of the function. These pointers are either pointer-size specific or they are not affected by the pointer size. Similar to its specifications for 64-bit-friendly parameters, the *OpenVMS Calling Standard* indicates that return values on the OpenVMS Alpha operating system are always sign extended to 64-bit values and returned in register zero (R0).

To make an address parameter 64-bit friendly, a function allows a 64-bit address to be passed, thus enabling both 32-bit and 64-bit calling programs to use a single interface. Conversely, if a function returns a 64-bit address to a 32-bit calling program, the address is safely returned in R0 but is truncated when moved from R0 into the user's data area. A 64-bit-friendly address return value is always 32 bits. When moved from R0 into the calling program's variable, it is sign extended when the calling program is using 64-bit addresses.

If the return value of a function can be a 64-bit address, this function must have pointer-size-specific entry points. If the function returns the address of a

structure that is bound to low memory, such as a FILE or WINDOW pointer, the return value does not force separate entry points.

Certain functions, such as malloc, allocate memory on behalf of the calling program and return the address of that memory as the value of the function. These functions have two implementations: the 32-bit interface always allocates 32-bit virtual memory, and the 64-bit interface always allocates 64-bit virtual memory.

Many string and memory functions have return values that are relative to a parameter passed to the same routine. These addresses may be returned as high memory addresses if and only if the parameter is a high memory address.

The following is the function prototype for strcat, which is found in the header file <string.h>:

```
char *strcat(char *s1, const char *s2);
```

The strcat function appends the string pointed to by *s*2 to the string pointed to by *s*1. The return value is the address of the latest string *s*1.

In this case, the size of the pointer in the return value is always the same as the size of the pointer passed as the first parameter. The C programming language has no way to reflect this. Since the function may return a 64-bit pointer, the strcat function must have two entry points.

As discussed earlier, the pointer size used for parameter *s*2 is not related to the returned pointer size. The C run-time library made this *s*2 argument 64-bit friendly by declaring it a 64-bit pointer. This declaration allows the application programmer to concatenate a string in high memory to one in low memory without altering the source code. The following strcat function statement shows this declaration:

```
char *strcat(char *s1, __char_ptr64 s2);
```

The data type *__char_ptr64* is a 64-bit character pointer whose definition and use will be explained later in this paper.

High-level Design

The /pointer_size qualifier is available in those versions of the C compiler that support 64-bit pointers. The compiler has a predefined macro named __INITIAL_POINTER_SIZE whose value is based on the use of the /pointer_size qualifier. The macro accepts the following values:

- 0, which indicates that the /pointer_size qualifier is not used or is not available
- 32, which indicates that the /pointer_size qualifier is used and has a value of 32
- 64, which indicates that the /pointer_size qualifier is used and has a value of 64

The C run-time library header files conditionally compile based on the value of this predefined macro. A zero value indicates to the header files that the computing environment is purely 32-bit. The pointer-size-specific function prototypes are not defined. The user must use the /pointer_size qualifier to access 64-bit functionality. The choice of 32 or 64 determines the default pointer size.

The header files define two distinct types of declarations: those that have a single implementation and those that have pointer-size-specific implementations. The addresses passed or returned from functions that have a single implementation are either bound to low memory, restricted to low memory, or widened to accept a 64-bit pointer.

Those functions that have pointer-size-specific entry points have three function prototypes defined. Using malloc as an example, prototypes are created for the functions malloc, _malloc32, and _malloc64. The latter two prototypes are the pointer-size-specific prototypes and are defined only when the /pointer_size qualifier is used. The malloc prototype defaults to calling _malloc32 when the default pointer size is 32 bits. The malloc prototype defaults to calling _malloc64 when the default pointer size is 64 bits. Application programmers who mix pointer types use the /pointer_size qualifier to establish the default pointer size but can then use the _malloc32 and _malloc64 explicitly to achieve nondefault behavior.

In addition to being enhanced to support 64-bit pointers, the C compiler has the added capability of detecting incorrect mixed-pointer usage. It is the function prototype found in the header files that tells the compiler exactly what pointer size is permitted or expected in a call. Proper use of the header files helps prevent pointer truncation.

The actual functions called in the C run-time library are either decc$malloc or decc$_malloc64, depending on the pointer size. The C run-time library does not contain an entry point called decc$_malloc32. This naming scheme was selected so that applications that link on older systems always get the 32-bit interface.

The C compiler has always looked at a table within the C run-time library shareable image for assistance in name prefixing. Using this table, the compiler knows to change calls to the malloc function into calls to the decc$malloc function and not to change calls to xyz, which is not a C run-time library function, into calls to decc$xyz.

The C run-time library and the C compiler have added new information to the table that tells the compiler which functions have pointer-size-specific entry points. When the compiler sees a call to the function _xyz32, it looks it up in the name table. If the name of the function is found, the compiler then looks at

whether the function is the 32-bit-specific entry point. If it is, the compiler forms the prefixed name by adding "decc$" to the beginning of the name but also removes the "_" and the "32." Consequently, the function name _malloc32 becomes decc$malloc, but the function name _xyz32 does not change.

Implementation

To illustrate changes that needed to be made to the header files, we invented a single header file called <header.h>. This file, which is shown in Figure 1, illustrates the classes of problems faced by a developer who is adding support for 64-bit pointers. The functions defined in this header file are actual C run-time library functions.

Preparing the Header File

The first pass through <header.h> resulted in a number of changes in terms of formatting, commenting, and 64-bit support. Realizing that many modifications would be made to the header files, we considered readability a major goal for this release of these files.

The initial header files assumed a uniform pointer size of 32 bits for the OpenVMS operating system. During the first pass through <header.h>, we added pointer-size pragmas to ensure that the file saved the user's pointer size, set the pointer size to 32 bits, and then restored the user's pointer size at the end of the header.

Next we formatted <header.h> to show the various categories that the structures and functions fall into. The categories and the result of the first pass through <header.h> can be seen in Figure 2. For example, the function rand had no pointers in the function prototype and was immediately moved to the section "Functions that support 64-bit pointers."

Organizing <header.h> in this way gave us an accurate reading of how many more functions needed 64-bit support. If any of the sections became empty, we did not remove the section. This approach worked well because while some engineers were doing 64-bit work, others were adding new functions. Any new functions added to a header file after the 64-bit work was done would be placed in the section "Functions that need 64-bit support." Prior to shipping the header files, we removed the empty sections.

Preparing the Source Code

After several false starts, we settled on a design for modifying the source code for 64-bit support. The expected starting design was to modify the source code by adding pointer_size pragmas and compile the source modules using the /pointer_size qualifier. Some modules would use /pointer_size=32; others would use /pointer_size=64. The major drawback to this approach was that looking at a variable declared as a pointer requires an understanding of the context in which that variable appears. No longer would "char *" be simply a character pointer. It could be a 32-bit or a 64-bit character pointer, and the implementer needed to know which one.

The design on which we decided overcomes the readability problem. By default, source files are not compiled with the /pointer_size qualifier. This means that no pointer-size manipulation occurs when including the header files. The readability of the source code is improved in that the implementers can see which pointers are 32-bit pointers and which are 64-bit pointers.

```
#ifndef __HEADER_LOADED
#define __HEADER_LOADED 1

#ifndef __SIZE_T
#  define __SIZE_T 1
   typedef unsigned int size_t;
#endif

int    execv(const char *, char *[]);
void   free(void *);
void  *malloc(size_t);
int    rand(void);
char  *strcat(char *, const char *);
char  *strerror(int);
size_t strlen(const char *);

#endif /* __HEADER_LOADED */
```

Figure 1
Original Header File <header.h>

```
#ifndef __HEADER_LOADED
#define __HEADER_LOADED 1

/*
** Ensure that we begin with 32-bit pointers.
*/
#if __INITIAL_POINTER_SIZE
#  if (__VMS_VER < 70000000)
#     error "Pointer size added in OpenVMS V7.0 for Alpha"
#  endif
#  pragma __pointer_size __save
#  pragma __pointer_size 32
#endif

/*
** STRUCTURES NOT AFFECTED BY POINTERS
*/
#ifndef __SIZE_T
#  define __SIZE_T 1
   typedef unsigned int size_t;
#endif

/*
** FUNCTIONS THAT NEED 64-BIT SUPPORT
*/
int    execv(const char *, char *[]);
void   free(void *);
void  *malloc(size_t);
char  *strcat(char *, const char *);
char  *strerror(int);
size_t strlen(const char *);

/*
** Create 32-bit header file typedefs.
*/

/*
** Create 64-bit header file typedefs.
*/

/*
** FUNCTIONS RESTRICTED FROM 64 BITS
*/

/*
** Change default to 64-bit pointers.
*/
#if __INITIAL_POINTER_SIZE
#  pragma __pointer_size 64
#endif

/*
** FUNCTIONS THAT SUPPORT 64-BIT POINTERS
*/
int rand(void);

/*
** Restore the user's pointer context.
*/
#if __INITIAL_POINTER_SIZE
#  pragma __pointer_size __restore
#endif

#endif /* __HEADER_LOADED */
```

Figure 2
First Pass through <header.h>

We created a C run-time library private header file called <wide_types.src>. This header file has the appropriate pragmas to define 64-bit pointer types used within the C run-time library, as shown in Figure 3.

This header file also contains the definitions of macros used in the implementations of the functions. Figure 4 shows the macros declared in <wide_types.src>.

Once a module includes the file <wide_types.src>, the compilation of that module changes to add the qualifier /pointer_size=32. This change improves the readability of the code because "char *" is read as a 32-bit character pointer, whereas 64-bit pointers use typedefs whose names begin with "__wide." The name of the new typedef is __wide_char_ptr, which is read as a 64-bit character pointer.

The C run-time library design also requires that the implementation of a function include all header files that define the function. This ensures that the implementation matches the header files as they are modified to support 64-bit pointers. For functions defined in multiple header files, this ensures that header files do not contradict each other.

```
/*
** This include file defines all 32-bit and 64-bit data types used in
** the implementation of 64-bit addresses in the C run-time library.
**
** Those modules that are compiled with a 64-bit-capable compiler
** are required to enable pointer size with /POINTER_SIZE=32.
*/
#ifdef __INITIAL_POINTER_SIZE
#  if (__INITIAL_POINTER_SIZE != 32)
#     error "This module must be compiled /pointer_size=32"
#  endif
#endif

/*
** All interfaces that require 64-bit pointers must use one of
** the following definitions. When this header file is used on
** platforms not supporting 64-bit pointers, these definitions
** will define 32-bit pointers.
*/
#ifdef __INITIAL_POINTER_SIZE
#  pragma __pointer_size __save
#  pragma __pointer_size 64
#endif

typedef char *__wide_char_ptr;
typedef const char *__wide_const_char_ptr;

typedef int *__wide_int_ptr;
typedef const int *__wide_const_int_ptr;

typedef char **__wide_char_ptr_ptr;
typedef const char **__wide_const_char_ptr_ptr;

typedef void *__wide_void_ptr;
typedef const void *__wide_const_void_ptr;

#include <curses.h>
typedef WINDOW *__wide_WINDOW_ptr;

#include <string.h>
typedef size_t *__wide_size_t_ptr;

/*
** Restore pointer size.
*/
#ifdef __INITIAL_POINTER_SIZE
#  pragma __pointer_size __restore
#endif
```

Figure 3
Typedefs from <wide_types.src>

```
/*
** Define macros that are used to determine pointer size and
** macros that will copy from high memory onto the stack.
*/
#ifdef __INITIAL_POINTER_SIZE

#  include <builtins.h>

#  define C$$IS_SHORT_ADDR(addr) \
     ((((__int64)(addr)<<32)>>32) == (unsigned __int64)addr)

#  define C$$SHORT_ADDR_OF_STRING(addr) \
     (C$$IS_SHORT_ADDR(addr) ? (char *) (addr) \
     :(char *) strcpy(__ALLOCA(strlen(addr) + 1), (addr)))

#  define C$$SHORT_ADDR_OF_STRUCT(addr) \
     (C$$IS_SHORT_ADDR(addr) ? (void *) (addr) \
     :(void *) memcpy(__ALLOCA(sizeof(* addr)), (addr), sizeof(*addr)))

#  define C$$SHORT_ADDR_OF_MEMORY(addr, len) \
     (C$$IS_SHORT_ADDR(addr) ? (void *) (addr) \
     :(void *) memcpy(__ALLOCA(len), (addr), len))

#else

#  define C$$IS_SHORT_ADDR(addr) (1)
#  define C$$SHORT_ADDR_OF_STRING(addr) (addr)
#  define C$$SHORT_ADDR_OF_STRUCT(addr) (addr)
#  define C$$SHORT_ADDR_OF_MEMORY(addr, len) (addr)

#endif
```

Figure 4
Macros from <wide_types.src>

Implementing the strerror Return Pointer

The function strerror always returns a 32-bit pointer. The memory is allocated by the C run-time library for both 32-bit and 64-bit calling programs. As shown in Figure 5, we moved the function strerror into the section "Functions that support 64-bit pointers" of <header.h> to show that there are no restrictions on the use of this function.

The "Create 32-bit header file typedefs" section of <header.h> is in the 32-bit pointer section, where the bound-to-low-memory data structures are declared. The function returns a pointer to a character string. We, therefore, added typedefs for __char_ptr32 and __const_char_ptr32 while in a 32-bit pointer context. These declarations are protected with the definition of __CHAR_PTR32 to allow multiple header files to use the same naming convention. Declarations of the const form of the typedef are always made in the same conditional code since they usually are needed and using the same condition removes the need for a different protecting name.

The strerror function could have been implemented in <header.h> by placing the function in the 32-bit section, but that would have implied that the 32-bit pointer was a restriction that could be removed later. The pointer is not a restriction, and the strerror function fully supports 64-bit pointers.

The private header file typedefs are always declared starting with two underscores and ending in either "_ptr32" or "_ptr64." These typedefs are created only when the header file needs to be in a particular pointer-size mode while referring to a pointer of the other size. The return value of strerror is modified to use the typedef __char_ptr32.

Including the header file, which declares strerror, allows the compiler to verify that the arguments, return values, and pointer sizes are correct.

Widening the strlen Argument

The function strlen accepts a constant character pointer and returns an unsigned integer (*size_t*). Implementing full 64-bit support in strlen means changing the parameter to a 64-bit constant character pointer. If an application passes a 32-bit pointer to the strlen function, the compiler-generated code sign extends the pointer. The required header file modification is to simply move strlen from the section "Functions that need 64-bit support" to the section "Functions that support 64-bit pointers."

The steps necessary for the source code to support 64-bit addressing are as follows:

1. Ensure that the module includes header files that declare strlen.

```
#ifndef __HEADER_LOADED
#define __HEADER_LOADED 1

/*
** Ensure that we begin with 32-bit pointers.
*/
#if __INITIAL_POINTER_SIZE
#  if (__VMS_VER < 70000000)
#     error "Pointer size added in OpenVMS V7.0 for Alpha"
#  endif
#  pragma __pointer_size __save
#  pragma __pointer_size 32
#endif

/*
** STRUCTURES NOT AFFECTED BY POINTERS
*/
#ifndef __SIZE_T
#  define __SIZE_T 1
   typedef unsigned int size_t;
#endif

/*
** FUNCTIONS THAT NEED 64-BIT SUPPORT
*/

/*
** Create 32-bit header file typedefs.
*/
#ifndef __CHAR_PTR32
#  define __CHAR_PTR32 1
   typedef char *__char_ptr32;
   typedef const char *__const_char_ptr32;
#endif

/*
** Create 64-bit header file typedefs.
*/
#ifndef __CHAR_PTR64
#  define __CHAR_PTR64 1
#  pragma __pointer_size 64
   typedef char *__char_ptr64;
   typedef const char *__const_char_ptr64;
#  pragma __pointer_size 32
#endif

/*
** FUNCTIONS RESTRICTED FROM 64 BITS
*/
int execv(__const_char_ptr64, char *[]);

/*
** Change default to 64-bit pointers.
*/
#if __INITIAL_POINTER_SIZE
#  pragma __pointer_size 64
#endif

/*
** The following functions have interfaces of XXX, _XXX32,
** and _XXX64.
**
** The function strcat has two interfaces because the return
** argument is a pointer that is relative to the first arguments.
**
** The malloc function returns either a 32-bit or a 64-bit
** memory address.
*/
#if __INITIAL_POINTER_SIZE == 32
# pragma __pointer_size 32
#endif
```

Figure 5
Final Form of <header.h>

```
void *malloc(size_t __size);
char *strcat(char *__s1, __const_char_ptr64 __s2);

#if __INITIAL_POINTER_SIZE == 32
#  pragma __pointer_size 64
#endif

#if __INITIAL_POINTER_SIZE && __VMS_VER >= 70000000
#  pragma __pointer_size 32
   void *_malloc32(size_t);
   char *_strcat32(char *__s1, __const_char_ptr64 __s2);
#  pragma __pointer_size 64
   void *_malloc64(size_t);
   char *_strcat64(char *__s1, const char *__s2);

#endif

/*
** FUNCTIONS THAT SUPPORT 64-BIT POINTERS
*/
void   free(void *__ptr);
int    rand(void);
size_t strlen(const char *__s);

__char_ptr32 strerror(int __errnum);

/*
** Restore the user's pointer context.
*/
#if __INITIAL_POINTER_SIZE
#  pragma __pointer_size __restore

#endif

#endif /* __HEADER_LOADED */
```

Figure 5
Continued

2. Add the following line of code to the top of the module: `#include <wide_types.src>`.
3. Change the declaration of the function to accept a __wide_const_char_ptr parameter instead of the previous const char * parameter.
4. Visually follow this argument through the code, looking for assignment statements. This particular function would be a simple loop. If local variables store this pointer, they must also be declared as __wide_const_char_ptr.
5. Compile the source code using the directive /warn=enable=maylosedata to have the compiler help detect pointer truncation.
6. Add a new test to the test system to exercise 64-bit pointers.

Restricting execv from High Memory

Examination of the execv function prototype showed that this function receives two arguments. The first argument is a pointer to the name of the file to start. The second argument represents the argv array that is to be passed to the child process. This array of pointers to null terminated strings ends with a NULL pointer.

Initially, the execv function was to have had two implementations. The parameters passed to the execv function are used as the parameters to the main function of the child process being started. Because no assumptions could be made about that child process (in terms of support for 64-bit pointers), these parameters are restricted to low memory addresses.

To illustrate that the argv passing was a restriction, we place that prototype into the section "Functions restricted from 64 bits" of <header.h>. The first argument, the name of the file, did not need to have this restriction. The section "Create 64-bit header file typedefs" was enhanced to add the definition of __const_char_ptr64, which allows the prototypes to define a 64-bit pointer to constant characters while in either 32-bit or 64-bit context.

Returning a Relative Pointer in strcat

The strcat function returns a pointer relative to its first argument. We looked at this function and determined that it required two entry points. In addition, we widened the second parameter, which is the address of the string to concatenate to the second, to allow the application to concatenate a 64-bit string to a 32-bit string without source code changes.

Figure 5 shows the changes made to support functions that have pointer-size-specific entry points. The prototypes of functions XXX, _XXX32, and _XXX64 begin in 64-bit pointer-size mode. Since the unmodified function name (strcat, XXX) is to be in the pointer size specified by the /pointer_size qualifier, the pointer size is changed from 64 bits to 32 bits if and only if the user has specified /pointer_size=32. At this point, we are not certain of the pointer size in effect. We know only that the size is the same as the size of the qualifier. The second argument to strcat uses the __const_char_ptr64 typedef in case we are in 32-bit pointer mode. Notice the declaration of _strcat64 does not use this typedef because we are guaranteed to be in 64-bit pointer context. Figure 6 shows the implementation of both the 32-bit and the 64-bit strcat functions.

The 64-bit malloc Function

The implementation of multiple entry points was discussed and demonstrated in the strcat implementation. Although multiple entry points are typically added to avoid truncating pointers, functions such as memory allocation routines have newly defined behavior.

The functions decc$malloc and decc$_malloc64 use new support provided by the OpenVMS Alpha operating system for allocating, extending, and freeing 64-bit virtual memory. The C run-time library utilizes this new functionality through the LIBRTL entry points. The LIBRTL group added new entry points for each of the existing memory management functions. The LIBRTL includes an additional second entry point for the free function. Since our implementation of the free function simply widens the pointer, we end up with a single, C run-time library function that must choose which LIBRTL function to call.

```
int free(__wide_void_ptr ptr) {
   if (!(C$$IS_SHORT_ADDR(ptr)))
       return(c$$_free64(ptr));
   else return(c$$_free32((void *) ptr);
}
```

Concluding Remarks

The project took approximately seven person-months to complete. The work involved two months to determine what we wanted to do, one month to figure out how we were going to do it, and four person-months to modify, document, and test the software.

During the initial two months, the technical leaders met on a weekly basis and discussed the overall approach to adding 64-bit pointers to the OpenVMS environment. Since I was the technical lead for the C run-time library project, this initial phase occupied between 25 and 50 percent of my time.

The one month of detailed analysis and design consumed more than 90 percent of my time and resulted in a detailed document of approximately 100 pages. The document covered each of the 50 header files and 500 function interfaces. The functions were grouped by type, based on the amount of work required to support 64-bit pointers.

The first month of implementation occupied nearly all of my time, as I made several false starts. Once I worked out the final implementation technique, I completed at least two of each type of work. As coding deadlines approached, I taught two other engineers on my team how to add 64-bit pointer support, pointing out those functions already completed for reference. They came up to speed within one week. Together, we completed the work during the final month of the project.

```
#include <string.h>
#include <wide_types.src>

/*
** STRCAT/_STRCAT64
**
** The 'strcat' function concatenates 's2', including the
** terminating null character, to the end of 's1'.
*/

__wide_char_ptr _strcat64(__wide_char_ptr s1, __wide_const_char_ptr s2)
{
  (void) _memcpy64((s1 + strlen(s1)), s2, (strlen(s2) + 1));
  return(s1);
}

char *_strcat32(char *s1, __wide_const_char_ptr s2) {
  (void) memcpy((s1 + strlen(s1)), s2, (strlen(s2) + 1));
  return(s1);
```

Figure 6
Implementation of 32-bit and 64-bit strcat Functions

Acknowledgments

The author would like to acknowledge the others who contributed to the success of the C run-time library project. The engineers who helped with various aspects of the analysis, design, and implementation were Sandra Whitman, Brian McCarthy, Greg Tarsa, Marc Noel, Boris Gubenko, and Ken Cowan. Our writer, John Paolillo, worked countless hours documenting the changes we made to the library.

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. T. Benson, K. Noel, and R. Peterson, "The OpenVMS Mixed Pointer Size Environment," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 72–82.
3. *DEC C User's Guide for OpenVMS Systems* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-PUNZE-TK, 1995).
4. *DEC C Runtime Library Reference Manual for OpenVMS Systems* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-PUNEE-TK, 1995).
5. *OpenVMS Calling Standard* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBBA-TE, 1995).

Biography

Duane A. Smith
As a consulting software engineer, Duane Smith is currently architect and project leader of the C run-time library for the OpenVMS VAX and Alpha platforms. He joined Digital in 1981 and has worked on a variety of projects, including the A-to-Z Database Manager and the Language-Sensitive Editor. Duane received his B.S. in engineering from the University of Connecticut in 1981 and his M.S. in software engineering from Wang Institute of Graduate Studies in 1987. He pursued his master's degree through Digital's Graduate Engineering Education Program (GEEP). Duane holds one U.S. patent issued for the DECwindows Structured Visual Navigation (SVN) widget.