Strategies for migrating from OpenVMS Alpha and OpenVMS Integrity systems to VSI Open VMS x86-64

Brett Cameron, February 2025

VMS Software
Professional Services

Contents

Introduction

Moving OpenVMS to the x86-64 architecture creates a strong new future for the OpenVMS operating system and for its users, and a long-term roadmap for the operating system and layered products has also been established, further ensuring that significant user investments in the OpenVMS platform will be protected and enhanced into the future.

VSI maintains a policy of ensuring forward-source compatibility (where feasibly possible) so that "well-behaved" software applications that currently run on VSI versions of the OpenVMS operating system for Alpha and Integrity can run successfully on VSI OpenVMS x86-64. Accordingly, ISVs and other users who have application code that takes advantage of published system services and library interfaces can have a high level of confidence that their applications will move with generally minimal modification to VSI OpenVMS x86-64 and that any system services, library routines, and programming language runtime library functions used by those applications will operate in the same way as they do on Alpha and Integrity (ignoring any new functions that might have been added). In short, it is readily possible for the great majority of applications running on OpenVMS Alpha and Integrity today to be ported to VSI OpenVMS x86-64 with (in many cases) relatively few changes.

VSI OpenVMS for x86-64 preserves compatibility with the OpenVMS Alpha and Integrity user, system management, and programming environments such that for general users and for system managers, VSI OpenVMS x86-64 has the same interfaces as for Alpha and Integrity. Minor changes have been made to accommodate the new architecture, but in general terms the basic structure, capabilities, and operation of the operating system are the same. For programmers, the goal has been to come as close as possible to a "recompile, re-link, and run" model for transition to OpenVMS x86-64 (although it is sometimes not quite that simple, as will be discussed below).

The purpose of this document is to discuss the porting process in general from both a technical and a non-technical perspective, highlighting some of the more important technical considerations, as well as outlining a recommended approach to undertaking your migration project. It should be noted that the text does not attempt to address every possible scenario or problem but rather aims to provide general guidance in terms of how to undertake the migration and how to address particular types of problems. Some lower-level technical considerations are briefly discussed, and strategies for dealing with problems such as missing code or unsupported third-party products are described. Arguably the most important consideration is that while VSI have done everything possible to make porting to OpenVMS x86-64 as seamless as possible, for any large porting project involving significant volumes of code and a complex technology stack, there will always be some challenges, and careful planning, appropriate project oversight, and having a suitably skilled project team are critical factors to ensuring a successful outcome.

It should be noted that this document focusses on porting application code to OpenVMS x86-64 only and does not discuss the virtualised infrastructure that might be required to support those applications in the OpenVMS x86-64 environment. For such matters, it is recommended that the reader work with the VSI Professional Services team to determine the most appropriate

configuration for your needs. Additionally, it is assumed that the reader is familiar with software development in an OpenVMS environment and with the OpenVMS operating system in general or has experience planning and managing migration projects of this type.

Methodology

Migration projects to OpenVMS x86-64 will range in size from a few days of effort to many months of effort, depending on scale, complexity, and various other factors. For smaller projects, a rigorous methodology is generally not required; however, for larger projects a more formal and methodical approach will typically be essential to ensuring a successful result. While every migration project will be different, an approach involving the following high-level project phases can generally be applied, tailored as necessary to meet specific project requirements.

- Assessment

  As with any other type of project, the first activities are to establish project scope and boundaries, identify any potential technical and non-technical risks, create an overall phase-level schedule for the project, and to prepare a high-level work breakdown structure. The inputs to this process will at a minimum typically include a review and inventory of the application source code and build scripts, a review of all relevant documentation (functional specifications, design documents, test specifications, and so on), and seeking input from those familiar with the application environment, including users, architects, developers, and system managers. The outputs of this initial phase will typically include details of project deliverables, some form of feasibility study, risk analysis, effort estimates (time and cost), testing strategy, and a project plan for the overall project. This initial phase will generally be required for all but the most trivial of migration projects.

  From a technical perspective, it should be noted that assessment includes not only a review of the applications themselves in terms of their source modules and how they are built and tested, but also includes the identification and assessment of other software and operational infrastructure on which those applications depend, including layered products, middleware and database products, and any processes and procedures associated with the general operation and management of the application environment, including monitoring tools, job scheduling facilities, and so on. In short, assessment of the existing OpenVMS Alpha or Integrity environment should encompass every possible aspect of the application environment from development and testing through to production deployment and operation. Determining all of this information and putting it into a viable migration plan generally involves inputs from multiple parties and thorough review to ensure completeness and accuracy, however this will invariably be time well spent in terms of ensuring a successful project outcome.

  In short, any migration involving mission-critical systems requires wide-ranging review and planning so that all issues and opportunities affecting the business, its technologies, its staff,

and its customers are well understood in advance. A thorough assessment should cover business risk, business continuity, technical risk, technical complexity, resource issues, constraints, timescales, and anticipated costs.

- Elaboration

  This phase may or may not be required, depending on various factors such as project scale and complexity, and whether any matters arising from the initial assessment require further clarification. This phase may often be combined with the subsequent pilot project. The activities performed by the team during this elaboration phase would be to refine and complete the definition of the overall migration solution, potentially undertake some level of prototyping in order to clarify any areas of particular concern or uncertainty, start work on test plans and test cases (as per agreed test strategy), and so on. The output of this phase would be a revised project plan and revised estimates for the project.

- Pilot

  Depending on the scale and complexity of the application environment to be ported to x86-64, one or more proofs-of-concept may be required to validate certain aspects of the migration solution. This will be particularly true if there are architectural changes being made in conjunction with the platform migration (although in general such changes should be avoided where possible). The primary objective of this phase is to validate the proposed migration solution (possibly focussing on particular areas of concern or uncertainty) and will typically involve the migration of an agreed suitably representative subset of application modules (and potentially data), the preparation of test plans and test cases relating to the modules in question, testing of the ported modules, and the resolution of any defects. The outputs of this phase will be a working subset of the migrated application and a revised project plan and estimates, depending on the results of the proof-of-concept activities.

- Migration

  As its name suggests, the migration phase is the phase in which the bulk of the actual application porting work is performed, along with the preparation of test plans and test cases (based on application documentation and use-cases) and testing of the migrated application code. During this phase work may also be done to address problems such as missing code and the replacement of unsupported third-party products with alternative solutions. The outputs of the migration phase will include fully tested migrated application code, test exit report(s), release notes, and related deliverables.

  A detailed discussion of testing methodology is beyond the scope of this document, however depending on the scale and complexity of the project in question, it may be appropriate to treat acceptance testing of the migrated environment as a distinct project phase.

  If you have thoroughly analysed your application code and have carefully planned the migration process, migrating source code modules should often be fairly straightforward and in some instances you may be able to recompile many of your programs with little or no

change, and programs that do not recompile at the first attempt will frequently need only minor changes to get them built and running correctly on the target OpenVMS x86-64 system.

In general terms, migrating your application code involves the following steps:

1. Setting up the migration environment
2. Compiling your application on an Alpha or Integrity system with the latest compiler versions
3. Testing applications on the source system to establish baselines for evaluating the migration
4. Compiling and linking an application on an OpenVMS x86-64 system
5. Debugging the migrated application
6. Testing the migrated application
7. Integrating the migrated application into the OpenVMS x86-64 application environment

Note that testing effort should not be underestimated and should include both functional and non-functional testing of the ported application environment to ensure that not only does the ported code produce correct results, but also that the environment meets business requirements in terms of important non-functional requirements such as availability, security, and performance. It is not uncommon with some migration projects for testing to be the largest component of the project in terms of time and effort.

Prior to porting (and possibly as part of an elaboration step) it may prove useful to verify that you can fully build your applications in the existing OpenVMS Alpha or Integrity environment using the latest compiler versions for the platform in question, as this can often uncover compilation problems (related to enforcement by the compilers of stricter language standards) that are more readily fixed and tested in the existing environment, prior to migration. This is also a useful validation of your build procedures to verify that they are able to correctly re-build the entire application from sources. It should also be confirmed that the source code being ported to OpenVMS x86-64 is the correct version and matches what is currently running in production.

- Deployment

  The deployment phase essentially involves delivery of the ported application environment to users and deployment of the ported application environment into production, following successful completion of acceptance testing and the correction of any defects found during such testing. The deliverables are essentially a deployable set of applications and any associated items.

From an initial planning perspective, some fundamental considerations will typically include matters such as whether you have all of the application code, how the code is managed (source code control), when the applications were last modified, when the applications were last fully rebuilt and fully tested, and so on.

Other key considerations will be availability of all required products (language compilers, middleware, databases, and so on) and whether alternative products might be required. Careful consideration also needs to be given to infrastructure-related matters in terms of what the OpenVMS x86-64 compute environment will need to look like in terms of numbers of systems, clustering requirements, high-availability considerations, system management and operational tools, data migration requirements, and other such matters. The answers to these types of questions will help to drive the project plan in terms of whether further elaboration and/or proofs-of-concept are required, and in terms of the subsequent work that needs to be performed.

Some technical considerations

This section briefly considers some specific and some of the more general technical matters that might need to be taken into consideration when porting your application code from Alpha or Integrity to OpenVMS x86-64. The text does not go into in-depth detail but is rather intended to highlight the most common situations and how generally to deal with them when porting your code.

Note that if you are porting directly from OpenVMS Alpha to OpenVMS x86-64, keep in mind that many of the considerations that apply to porting from Alpha to Integrity are equally applicable when porting from Alpha to x86-64. A detailed description of these considerations is beyond the scope of this document, and if you are looking to port directly from Alpha to x86-64 you should review the document "Porting Applications from VSI OpenVMS Alpha to VSI OpenVMS Industry Standard 64 for Integrity Servers" in addition to reading this document.

General considerations

As noted previously, many applications will compile and link on OpenVMS x86-64 without the need for significant or any code changes, and in many cases the problems that need to be fixed will be relatively straightforward to address, often equating to little more than minor modifications to code to eliminate warning messages produced by the compilers as a consequence of the new compilers on OpenVMS x86-64 more rigorously enforcing adherence to language standards or being more pedantic with regard to enforcement of good programming practices. However, any applications that depend on internal OpenVMS data structures, use undocumented system calls, use privileged interfaces, or operate at inner access modes may require code changes before they can be built successfully on OpenVMS x86-64. Likewise, any code that uses machine instructions specific to Alpha or Integrity or that makes assumptions about numbers of registers will require change, as will any code that in any way relies on the VAX, Alpha, or Integrity calling standards. Some of these topics are considered in a little more detail in various sections below.

Alpha or Integrity-specific logic (conditionalized code)

Some code (application code and/or build procedures) may include logic that assumes it is running on either Alpha or Integrity (or possibly even VAX). Such code might not be able to

correctly deal with another platform architecture, and changes to the logic will often be required. For example, a build command procedure might contain the following sequence of statements to execute different logic depending on whether the command procedure is being run on Alpha or Integrity, such that if the architecture is not Alpha, Integrity is assumed. If different logic is required for x86-64, the procedure will need to be changed accordingly.

```
$ arch := "''f$getsyi("ARCH_NAME")'"
$
$ if "''arch'" .eqs. "Alpha"
$ then
$    ! Do whatever needs to be done for Alpha
$ else
$    ! Assumes Integrity
$ endif
```

Aside from dealing with any such conditionalization, DCL command procedures written for OpenVMS Alpha and Integrity will generally continue to work on OpenVMS x86-64 without change. Certain command procedures, such as build procedures, may need to be modified to accommodate new or different compiler qualifiers and linker switches (see comments elsewhere in this document).

Similar conditionalized code might exist in a C/C++ application. For example, code such as the following does not consider that if the platform is not Integrity, it could be something other than Alpha. Again, any such code may need to be modified to appropriately accommodate the x86-64 architecture.

```
#ifndef __ia64
    /* Assume Alpha-specific code */
#endif
```

In either case, when these files are moved to OpenVMS x86-64, they may need to be modified in order to work correctly. In general, these sorts of situations are relatively straightforward to identify by performing a simple search on the application code and build procedures, and in general any such matters are typically straightforward to resolve, however care should be taken to ensure that the logic associated with the use of any such pre-processor directives and/or DCL constructs is correct and is suitably tested.

When working with C/C++ code, the macro `__x86_64` is defined for the x86-64 architecture and when working with DCL scripts, the lexical function call `f$getsyi("ARCH_NAME")` returns `x86_64` on OpenVMS x86-64 systems. The macro `X86_64` is provided by the Bliss and MACRO compiler.

Architecture-specific code

It will be necessary rewrite any code that is in any way dependent on the current Alpha or Integrity architecture. For example, assembly language code that deals with specific machine instructions or that makes assumptions about the number of registers, or the functions of specific registers, will likely have to be modified. Likewise, any code that implements user-written threading-like behaviours, such as code that performs direct stack switching, that implements a co-routine or

tasking model, or that depends on the nature of the procedure call stack frames will need to be modified to operate correctly on VSI OpenVMS x86-64.

These types of architecture-specific code are typically not found in business applications written in high-level languages but may be found in lower-level code for device drivers or other such specialist situations.

Object file format code

Any software applications that deal with object files or executable image file formats will have to be modified to handle the new OpenVMS x86-64 formats of these file types. This type of code is typically only found in compilers, debuggers, and analysis tools that directly process or manipulate object files or executable images, but some applications may for example include code to read image header information. Any such code will need to be modified to correctly handle the new x86-64 image file format.

Floating-point formats

There have been no changes to floating point formats between OpenVMS Integrity and OpenVMS x86-64, however if you are migrating directly from OpenVMS Alpha to OpenVMS x86-64 you may need to take differences in floating point formats into consideration when rebuilding and testing your applications on OpenVMS x86-64.

The Alpha architecture supports both IEEE and VAX floating-point formats in hardware, and OpenVMS compilers generate code using the VAX floating point formats by default, with options to use IEEE formats. The Itanium and x86-64 architectures implement floating-point arithmetic in hardware using the IEEE floating-point formats, including IEEE single-precision and IEEE double-precision.

If an application was originally written for OpenVMS VAX or OpenVMS Alpha using the default floating-point formats, it can be ported to OpenVMS x86-64 in one of two ways:

1. For programs written in languages other than C++ (which does not support VAX floating point formats), you can continue to use VAX floating-point formats, utilizing the conversion features of the VSI compilers on OpenVMS x86-64, or
2. You can choose to convert the application to use IEEE floating-point formats.

VAX floating-point formats can (for example) continue to be used in those situations where access to previously generated binary floating-point data is required, however it is typically recommended that IEEE floating-point formats be used in situations where VAX floating-point formats are not specifically required, as apart from anything else, the use of IEEE floating-point formats will generally result in more efficient code. General purpose conversion routines such as `CVT$CONVERT_FLOAT()` can be used to assist with floating-point format conversions, where necessary.

Particular care should be taken when working with floating-point data that is transferred between OpenVMS and external systems.

It is possible to mix-and-match both VAX and IEEE floating-point types in the same image, but in general this is not recommended (and is generally not necessary). Linker map files on OpenVMS Integrity and OpenVMS x86-64 (V9.2-2 and higher) show which floating-point format is used in each object module (the Integrity and x86-64 compilers record this information in the object module). It should be noted that this feature does not exist on Alpha.

Some important C++ differences

For OpenVMS on x86-64, VSI has adopted the open-source LLVM compiler backend. For all languages other than C++ this change to the compiler architecture is of little or no consequence from a development perspective, however for C++ there are a number of important differences that will often need to be considered when porting C++ code from Alpha or Integrity to OpenVMS x86-64. These differences largely stem from the fact that the C++ compiler on OpenVMS x86-64 (or more specifically the `CXX` command) is essentially a wrapper on top of the Clang compiler front-end, without any extensive modifications (other than those required in order to support operation in the OpenVMS environment). The following list identifies some of the more common issues that can arise because of this change when porting existing C++ code from OpenVMS Alpha and Integrity.

- Default pointer size

  By default, the C++ compiler on OpenVMS x86-64 uses 64-bit pointers, which is not the case on Alpha and Integrity where, by default, the C++ compiler uses 32-bit pointers. When porting C++ code from Alpha or Integrity to OpenVMS x86-64 it will therefore often be necessary to use the `/POINTER_SIZE` qualifier to explicitly instruct the compiler to use 32-bit pointers. For applications that are written purely in C++, it may be possible to build such applications using 64-bit pointers, however in mixed-language situations, or in situations where the C++ code makes calls into third-party or open-source libraries built with 32-bit pointers, it will generally be necessary to compile your C++ using 32-bit pointers, or to make localised code changes to accommodate any 32-bit interfaces. Calls to some system services and library routines may also be impacted by this change.

- Sizes of `long`, `size_t`, and `off_t` data types

  When porting C++ code from Alpha or Integrity to OpenVMS x86-64, it is important to be aware that the size of the `long`, `size_t`, and `off_t` types is 8 bytes as opposed to 4 bytes on Alpha and Integrity, irrespective of the specification of any compiler options. These differences can cause problems if you application code makes assumptions regarding the sizes of any of these types, or if you are passing structures containing any of these types between C and C++ code. Aside from potentially causing runtime errors and incorrect program results, these changes can cause various compilation errors and warnings.

- External function names

  For the compilers on Alpha and Integrity and for the compilers other than C++ on OpenVMS x86-64, the maximum length of external symbol names is 31 characters, with names longer

than this being truncated in accordance with the algorithm described for the C compiler `/NAMES` qualifier, and symbol names by default are converted to uppercase. For C++ on OpenVMS x86-64, symbol names can be longer than 31 characters, and the default behaviour of the C++ compiler is to preserve case, rather than to convert symbol names to uppercase. These changes in default behaviour may need to be taken into consideration, particularly when working with mixed-language applications, or when developing new C++ applications that need to link with existing code that might not have been compiled in the same way. When porting mixed language applications to x86-64, it will often be necessary to specify `/NAMES=(UPPERCASE,SHORTENED)` when compiling C++ modules.

The above list is by no means definitive, and it is not uncommon to encounter a variety of other issues when porting your C++ code from OpenVMS Alpha and Integrity to OpenVMS x86-64, however most other such issues tend to be related to enhancements to the compiler in terms of stricter adherence to language standards. Additional information regarding the above topics and other C++ porting considerations can be found in https://docs.vmssoftware.com/vsi-cxx-user-s-guide-for-openvms-systems/#porting.

Some other architectural differences to consider

The following text provides a high-level description of some of the more fundamental differences that may need to be considered when porting some applications from OpenVMS Alpha or Integrity to OpenVMS x86-64 and provides references to additional information. Note that for most applications, many (if not all) of the differences described below will be of no consequence, however a general understanding of these differences may prove useful to solving some porting issues when recompiling and relinking your application code.

- Logical names that point to the location of shareable and loadable images are different. For example, the logical name `X86$LIBRARY` is used on OpenVMS x86-64 while `IA64$LIBRARY` is used on OpenVMS Integrity to identify the location of library files. Likewise, the logical name `X86$LOADABLE_IMAGES` is used on OpenVMS x86-64 systems while the system logical names `IA64$LOADABLE_IMAGES` and `ALPHA$LOADABLE_IMAGES` are used on I64 and Alpha systems, respectively.
- On OpenVMS x86-64, a symbol vector entry is a pair of quadwords while on Integrity a symbol vector entry is a single quadword.
- For OpenVMS x86-64 images, a function symbol's value is always a code address. There is no GP (global pointer) and no short data segment. For OpenVMS x86-64 images, the entry in the symbol vector with the index value of 1 contains the values 00002080 and 80000020.
- Alignment is on the default target page size, which is 8KB for linking on OpenVMS x86-64 and 64KB for Integrity. This default can be overridden using the `/BPAGE` qualifier when linking.
- On OpenVMS x86-64, all segments within a shareable image must have the same positions relative to each other that they were given by the linker. The image activator is free to load segments in a shareable image independently of each other. To allow segments to be loaded

independently, OpenVMS compilers generate code that uses indirect addressing. In this way, the only segments whose relative positions have to be maintained are the code segment, the unwind segment, and the Global Offset Table segment. The linker flags those segments. A segment which requires the linker-given position to the preceding segment is flagged as "fixed-offset". In an image with code segments in multiple clusters, each cluster will have its own unwind segment and Global Offset Table so that their relationship can be maintained. The image activator and the `INSTALL` utility maintain the relative positions of these various segments.

- The OpenVMS x86-64 linker places code segments in the P2 memory region by default and uses the default page size of 2000 hexadecimal. The `/SEGMENT=CODE=P0` option can be specified to place code segments in the P0 region. Non-code segments (without the `ALLOC_64BIT` attribute specified) are placed in the P0 region by default. On OpenVMS Alpha and Integrity, the linker places segments in the P0 region by default and uses the default page size of 10000 hexadecimal. The `/SEGMENT=CODE=P2` option can be specified to place segments in the P2 region. On OpenVMS x86-64 systems, the first P0 segment is placed at 2000 hexadecimal. On OpenVMS Integrity systems, the first P0 segment is placed at 10000 hexadecimal, leaving the first page unused as a guard page.

  It should be noted that just because code is loaded into 64-bit space, this does not mean a 64-bit pointer is needed to point to it. On OpenVMS Alpha and Integrity, function values are pointers to descriptors rather than the actual code. The code address is one of the fields in those descriptors and this has always been a 64-bit pointer. These descriptors are allocated in 32-bit memory and can therefore be pointed to using a 32-bit pointer. However, on x86-64, function values are pointers to code. In order to deal with this difference without requiring potentially considerable and problematic changes to applications and to operating system code, on OpenVMS x86-64 the linker creates small trampoline routines that are allocated in the 32-bit P0 address range that then transfer control to the actual code that resides in 64-bit space.

- On OpenVMS Integrity, you can set or clear the `SHORT` symbol vector attributes. For OpenVMS x86-64, setting or clearing this attribute is ignored by the linker.

- On OpenVMS Integrity, to move all code into P2 space you can (as mentioned previously) use the linker option `/SEGMENT_ATTRIBUTE=CODE=P2`, and on OpenVMS x86-64, to move all code into P0 space, you can use the `/SEGMENT_ATTRIBUTE=CODE=P0` option. Be aware that if you use clusters in the same link command (with linker options) and if `EXE` sections of the image are put into specific clusters, setting `ALLOC_64BIT` does not change the per-cluster segment creation and you will then see more than one executable segment with base addresses in P2 space.

- Some segments created by the respective linkers on Integrity and x86-64 are different or may or may not exist. For example, global offset table segments are x86-64 specific, while unwind segments, short data segments, and signature segments are created by the linker on Integrity

only. For additional information on this topic, refer to Chapter 3 of the VSI OpenVMS Linker Utility Manual.

- On OpenVMS x86-64 systems, when an executable image calls a function in a shareable image, the call goes through one or two linker-generated code stubs (see previous comments). On OpenVMS Integrity, at run-time when the image activator maps a shareable image into memory, it calculates the actual locations of the routines and relocatable data within the image and stores these values in its symbol vector. The image activator then fixes up the references to these symbols in the executable image.
- Be sure to check the format of any `LINK` commands used by build procedures. Not all qualifiers provided for the linker on OpenVMS Integrity are available for OpenVMS x86-64, and some default values are different. For example, the default value of the page size qualifier is 8KB for OpenVMS x86-64 and 64KB for OpenVMS Integrity, and the default when linking on OpenVMS x86-64 is for multiple kernel threads support and the ability of the image to receive upcalls to both be enabled, which may not be appropriate for many older programs. Some linker options files may also require modification, especially for shareable images (see previous comments).
- When porting an application from Integrity to x86-64, be aware that the image layout may change in an incompatible way even though the commands used to build the application may be no different. This is a fundamental architectural difference.
- On OpenVMS Integrity, compilers may generate "short data", which is accessed in an efficient way. The OpenVMS Integrity linker always collects short data to the default cluster, no matter where the object module that defines this short data is collected. That is, in a partially protected shareable image, an object module may be collected into a protected cluster, but its short data may be collected into an unprotected cluster, and so it is not protected, such that user-mode code in the shareable image can write to it. On OpenVMS x86-64, there is no short data. All data defined in an object module will go where the module goes (except for the defining PSECT, which is moved with an explicit `COLLECT` option). This means that on OpenVMS x86-64, for partially protected shareable images, all data defined by an object module that is collected into a protected linker cluster will be protected such that user-mode code in the shareable image cannot write to it.
- User-written x86-assembler code must follow the VSI OpenVMS x86-64 calling standard to provide exception handling information that is used by the exception handling mechanism to find an exception handler in one of the callers of the assembler module. Without this information, exception handling can only ever call the last-chance exception handler, which generally means that the application will likely not be able to properly handle the exception.
- The implementation of variable length argument lists on VSI OpenVMS x86-64 is different from OpenVMS Integrity and OpenVMS Alpha and may require source code changes, depending on how any such variable length argument lists are used by your application code. The C and C++ compilers will generally identify any issues in this regard.

- When working with C and C++ in particular, be sure to review the CRTL manual for CRTL changes that could potentially impact the operation of your programs. For example, the interface for the function `isatty()` has been modified to ensure compatibility with the POSIX 1003.1 standard such that were previously this function would return -1 in case of an error, on OpenVMS x86-64 it returns 0 and stores any error code in `errno`. Such changes can lead to subtle program errors that are difficult to find, and a review of the changes documented in the CRTL manual may save you considerable such debugging time.

For additional information on many of the above and other related matters, refer to the following documents:

- https://docs.vmssoftware.com/vsi-openvms-linker-utility-manual/
- https://docs.vmssoftware.com/vsi-openvms-calling-standard/

Development tools

With very few exceptions, VSI have at the time of writing already ported most existing OpenVMS development tools and utilities available for OpenVMS Alpha and Integrity to the x86-64 architecture. This means that developers should be able to use their existing processes and procedures for developing, debugging, testing, and deploying their applications on the OpenVMS x86-64 platform. At the time of writing, some tools such as SCA (Source Code Analyzer) and PCA (Performance Coverage Analyzer) have not yet been ported to OpenVMS x86-64, however these tools are generally not required in order to port applications (although PCA might be useful for diagnosis of some application performance issues), and they will be made available for OpenVMS x86-64 in due course. Open source and freeware development tools such as MMK and `make` are also readily available for OpenVMS x86-64, and VSI's VMS IDE can be used with OpenVMS x86-64 so long as SSH is enabled on the OpenVMS x86-64 system.

In terms of the OpenVMS symbolic debugger, at this time there is still some work required to bring this up to the same level as it is for Alpha and Integrity, however this work is well in hand, and while a fully functional debugger can without question be a valuable asset when it comes to identifying issues in your ported code, it is generally not a firm requirement from a porting perspective, and there are often other methods that can be used to isolate and identify code-related problems.

Taking advantage of OpenVMS clustering

Within some constraints, OpenVMS supports mixed-architecture clusters comprised of OpenVMS Alpha, Integrity, and x86-64 systems, and the unique clustering capabilities of the operating system can often be leveraged to facilitate the assimilation of VSI OpenVMS x86-64 in a phased, structured, and extremely low-risk manner by providing a means to gradually introduce application components running on OpenVMS x86-64 into the production environment, and a way to quickly and easily disable such components and revert back if problems are encountered. Once the new

OpenVMS x86-64 environment has been suitably proven in the production environment, other non-x86-64 systems can be incrementally removed from the cluster as and when appropriate. This strategy decouples various aspects of the migration process to the maximum extent possible, permitting the assimilation process to proceed largely transparently to users, with minimal disruption to operations, and generally with the least possible business risk.

Aside from deployment into production, mixed-architecture clusters will generally also be useful from a porting perspective and for testing, as this will often eliminate the need to replicate all aspects of the build and test environments.

Keep in mind that for mixed-architecture clusters, each architecture must use separate system disks, and you must ensure that your applications where necessary refer to the appropriate system disk. It should also be noted that for OpenVMS x86-64 each x86-64 node currently requires its own local system disk.

Dealing with common problems

As with any type of software project, common challenges and problems can be both technical and non-technical, with the more common technical problems relating to matters such as missing code, unsupported or unavailable third-party products, old code that is not compliant with newer compiler versions, and so on, while from a non-technical perspective the problems encountered on migration projects tend to be largely the same as those for any other type of project, including (but not limited to) a loss of business knowledge, poor documentation, poor executive sponsorship for the project, insufficient budget, poor test coverage, inexperienced team, and poor planning. A detailed discussion of all of these matters is beyond the scope of this document, however the following text considers several of the more common technical challenges and how they might be addressed. It is important to appreciate that every migration project will be different, and accordingly the following comments may not be directly applicable to your particular situation, however the key point to note is that with a few exceptions it is usually possible to identify and implement acceptable solutions to most technical problems.

Missing code

When looking to port your applications to OpenVMS x86-64, identifying that you have missing code can be a major problem, depending on the amount of code that is missing, the nature of that code, and whether the functionality it implements is documented or not. In some cases, missing code will equate to a third-party library component that implements some sort of standard API, such as a simple linked list or hash table implementation, or possibly some kind of graphics library, and in such cases, it will often be straightforward to devise and implement a replacement solution. However, in other cases the missing code will relate to important business logic, and if this business logic is not well understood or documented in any way, this can present a major problem and present a significant impediment to migration, particularly given that there is no binary translator for OpenVMS Alpha or Integrity to OpenVMS x86-64. If the volume of such missing code is small (perhaps a handful of small functions with well-defined interfaces), it may

be possible to analyse the binary image to get sufficient understanding of the logic flow, however it is generally not practical via this method to reverse-engineer anything of significant scale and complexity. This being said, just about every situation will be different, and if you do find yourself in this situation, do not simply assume that nothing is possible; VSI may be able to help.

Unsupported third-party products

VSI recognizes that not all third-party products currently available for OpenVMS Alpha and/or Integrity will be ported by ISVs to VSI OpenVMS x86-64 or may not be ported in an appropriate timeframe for your desired migration timeline. In some situations, this can be a significant impediment to porting to OpenVMS x86-64, however in other situations it can in fact present an opportunity to introduce new technologies that will present opportunities to modernize and enhance your OpenVMS application environment and/or provide new and potentially better and more cost-effective ways to integrate it with other systems. Some of the more common areas where availability of such third-party products can be an issue include middleware products, databases (or database clients), job scheduling, monitoring and management solutions, and backup solutions, some of which are easier to address than others. For example, it is often possible to replace existing proprietary middleware solutions with a modern open-source alternative, and it is generally possible to identify alternative monitoring solutions, however identifying the best possible approach to dealing with such situations is often non-trivial, and we would recommend engaging VSI to assist you in such situations.

It should be noted that in general it will not be possible to identify a fully 100% drop-in replacement product and it will often be necessary to make changes to your application code in order for it to work with any replacement solution, however with careful design it is often possible to minimize the scope and impact of any such changes. That being said, it is also important to keep in mind that it is generally a good idea to limit as much as possible the amount of change to the application environment during the porting project, and it will often be better to introduce new technologies within the existing OpenVMS Alpha or Integrity environment and validate their operation before porting, rather than to introduce them as part of the porting project, as if problems are found during testing of the ported application environment it will often be more difficult to determine whether such problems are related to the introduction of the new product or to the port itself.

Latent bugs

Despite all best efforts, you may when porting encounter bugs that have existed in your code for a long time but that never caused a problem on OpenVMS Alpha and/or OpenVMS Integrity. For example, failure to initialize a variable in your code or to correctly null-terminate a C string might have been benign on Alpha or Integrity but could produce an arithmetic exception or an access violation on OpenVMS x86-64. The same could be true of moving between any other two OpenVMS architectures, because the available instructions and the way the compilers optimize them is subject to change. For any application of appreciable size and complexity, there is generally no simple way to readily identify these and similar such problems (unless it is something

that the compiler in question is able to detect and warn you about), and it is therefore vitally important to ensure that your ported applications are thoroughly tested before being deployed into production. And do not simply ignore (or disable) any warnings that are emitted by the compiler when porting your code. Many such warnings may indeed be innocuous, but some may not be, and it is much easier (and preferable) to find and fix problems when porting the code than it is to be trying to find them later.

Still running on VAX

Although this document primarily addresses porting OpenVMS Alpha and Integrity applications to OpenVMS x86-64, for users who have much older VAX applications and have all of the source code for those applications, it may still be possible to port those applications to OpenVMS x86-64, with the level of effort likely to be comparable to the work that would have been required to port those applications to Alpha, however there may be complications related to availability of third-party products, changes to language compiler adherence to standards, changes to language runtime libraries, and so on, and if you are in this situation we would recommend that you contact VSI for advice and assistance.

Top 10 porting considerations

The following list identifies what will often be the most important porting considerations from a technical perspective. As discussed elsewhere in this document, there are many other factors that may also need to be considered, and technical considerations form only one aspect of the overall porting project, however taking these 10 points into consideration coupled with the project methodology outlined above will help to ensure that you are heading in the right direction and will help to ensure a successful outcome for your porting project.

1. Do a complete inventory of all third-party software products and HPE/VSI OpenVMS layered products before you start your port. These may be required for development, testing, or for production deployment. Ensure you know the status of each of these products on VSI OpenVMS x86-64 before you go too far with your port.
2. Make sure your application builds cleanly and runs on the latest version of OpenVMS Alpha or Integrity (whichever you are currently using) built using the latest released compilers and other relevant development tools.
3. Check for hardware architecture dependencies in all source code and DCL command procedures.
4. Use automated regression testing as much as possible and ensure that any manual tests are properly documented (and are up to date).
5. Document your build processes.
6. Read all relevant OpenVMS x86-64 product release notes.

7. Reduce/re-code/eliminate as much MACRO code as possible (if at all possible).
8. Where possible, use IEEE floating point.
9. Ensure that you have working OpenVMS Alpha or Integrity development and test environments available so that you can compare results easily between Alpha or Integrity and x86-64.
10. The compiler is your friend. Enable (and heed) any and all warnings emitted by the compiler in both the source (Alpha or Integrity) and target (x86-64) environments.

How VSI can help

The VSI Application Services and Professional Services teams provide a comprehensive set of services to help customers get the most from their OpenVMS application environments, including migration services to help customers migrate their Alpha and Integrity application environments to OpenVMS x86-64. VSI understands that some customers will require more help than others, and the migration service can be tailored accordingly, to meet the specific requirements of each project, ranging from the provision of a few hours consulting to help customers get going on their migration project, through to taking total responsibility for the migration of the application code and configuration of the x86-64 OpenVMS environment.