A Cookbook Approach To Creating an OpenVMS Shareable Image

Publication Date: August 2024

Operating System: VSI OpenVMS x86-64 E9.2-3

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

A Cookbook Approach To Creating an OpenVMS Shareable Image

Legal Notice

Table of Contents

1. Introduction

This document provides a cookbook approach to shareable image creation, and includes assorted additional associated aphorisms.

Disclaimer

This legacy document is provided by VSI as-is. All questions about OpenVMS should be directed to VSI; however, please note that VSI does not own any VAX versions of distribute VAX licenses.

2. Official Documentation

For detailed information about working with shareable images, refer to the following documents:

- *VSI OpenVMS Linker Utility Manual* [https://docs.vmssoftware.com/vsi-openvms-linker-utility-manual/]
- *VSI OpenVMS DCL Dictionary: A–M* [https://docs.vmssoftware.com/vsi-openvms-dcl-dictionary-a-m/]
- *Command Definition, Librarian, and Message Utilities* [https://docs.vmssoftware.com/vsi-openvms-command-definition-librarian-and-message-utilities/]
- *VSI OpenVMS Programming Concepts Manual, Volume II* [https://docs.vmssoftware.com/vsi-openvms-programming-concepts-manual-volume-ii/]

3. Image Name Terminology

Shareable Image

This is essentially a library of routines and data that have already been linked. Shareable images do not and cannot grant additional privileges.

Shareable image primary purposes and benefits:

- Reduction in memory usage when multiple copies of the code are active;
- Reduction of the amount of time required to link images;
- Easy and selective replacement of libraries of code;
- Development of chunks of code within an entire system;
- Easy code upgrade.

Shareable images, including the OpenVMS Alpha SYS$BASE_IMAGE image, are the only images supported by the LIB$FIND_IMAGE_SYMBOL routine – this routine allows a process to perform a variant of **dynamic linking**, this call allows an image to access code in a shareable image on-the-fly. You can create shareable images from ordinary objects and object libraries by following the steps outlined in this cookbook.

Executable image

This is the parent or top-level image, and is the only image that can be **RUN**. This image is the only type of image that has a transfer address. Executable images are the only type of image that should

be used with, and the only type that can be granted extra privileges by, the **/PRIVILEGED** qualifier on an **INSTALL CREATE** or **REPLACE** command. Executable images are also the only type of images that can grant an identifier via a subsystem identifier access control list entry (ACE).

See the comments on INSTALL, below.

User-written system service (UWSS)

This image is sometimes confusingly called a *privileged shareable image*. This image is quite different from the typical shareable image and involves a completely different scheme of dispatching control within the image. The operations used to build a UWSS, and the associated UWSS data structures – called a Privileged Library Vector or PLV – are different from those used for a standard shareable image. A UWSS can grant privileges to unprivileged callers using the implicit SETPRV privilege granted to code that executes in KERNEL or EXECUTIVE mode (two highly-privileged processor execution modes), and by the **INSTALL /PROTECT** command. A UWSS image is not installed with **/PRIVILEGE**, nor can it be installed with a subsystem identifier.

4. Creating Shareable Images

Here is a cookbook approach to (ordinary) shareable image creation.

Differences between OpenVMS VAX and OpenVMS Alpha are explicitly called out.

1. Determine the names of all symbols that are to be visible outside the shareable image.

   Symbols that should be externally visible typically include the symbolic names of the various entry points that you want callers of the shareable image to be able to see and use. These externally visible symbols do not necessarily have to be subroutines, they could also be constants or variables that are intended to be externally visible.

   VSI recommends that all externally-visible symbols should follow the OpenVMS symbol naming conventions as documented in the *Guide to Creating OpenVMS Modular Procedures* manual.

   Also, determine what external symbols (the addresses of data and/or subroutines) that the shareable image will itself reference. This is code and/or data that the shareable itself is based on. These are known as external references.

   ALL external references must be resolved when an image – be it a shareable or an executable image – is linked. In the case of a link of a shareable image, all external symbols referenced by code in the shareable must be resolved when the shareable is linked.

   ### Note

   The routines located in a shareable image can reference data and/or subroutines in a calling image – but the shareable image can only access this code (or data) by virtual address. Otherwise, the linker would complain long and loud about unresolved symbols during the link of the shareable image. The address of the external code and/or data must be passed into the shareable image routine(s) via some means other than the linker. One method typically used is passing the address into the shareable image via an argument list somewhere – in some call to a function in the shareable image – but the address of the code/data could be passed in to the code in the shareable image by method as esoteric as a mailbox. Typical shareable image applications will not reference data and/or code in a calling image. In most cases, the code needed by the shareable and by the parent image will be relocated into the shareable image, or into a second *lower-level* shareable image located *beneath* both the shareable image under development, and obviously underneath the parent executable image.)

Shareable images are sensitive to the external models used by the particular compiler(s) in use. This document assumes the VAX C external model, and will cover the use of OpenVMS VAX transfer vectors or OpenVMS Alpha SYMBOL_VECTOR linker options.

Note

Building shareable images with the VSI C compiler may require the specification of the (non-default) **/SHARE_GLOBAL** qualifier. This qualifier causes externs to be promoted and to thus become visible outside the image being built.

If using OpenVMS VAX transfer vectors or the OpenVMS Alpha SYMBOL_VECTOR linker option, **/SHARE_GLOBAL** need not be specified.

The **/SHARE_GLOBAL** qualifier is not required under VAX C.

Further information on transfer vectors and on SYMBOL_VECTOR linker options is included below.

---

2. Create a set of transfer vectors for the symbols (OpenVMS VAX):

   The OpenVMS VAX shareable image transfer vectors are located at a known position within the shareable image – typically at the front – and the various offsets are used to dispatch the run-time calls into the actual code position(s) within the shareable image. Use of consistent offsets means callers do not need to relink the calling image(s) when the shareable image is modified.

   An example transfer vector – XFRVEC.MAR – looks like this:

```
.title$$$XFRVEC facnam_SHR Transfer Vectors
.ident/facnam_SHR V1.0-0/
; create a program section to hold just the vectors
; keep all transfer vectors quadword aligned.
.psect$$$XFRVEC,exe,shr,nowrt,rd,pic,quad

; For routines entered via a CALLS or CALLG instruction:
.macroXFRVEC_CALL entry_point,entry_point_int
.alignQUAD
.transfer entry_point
.External entry_point_int
.mask entry_point_int
JMP l^entry_point_int+2
entry_point:: ; allows .transfer to work.
.endm

; define a macro to simplify creating a transfer vector
; to a routine located in the shareable image.
.macroXFRVECentry_point
.alignQUAD
.transfer entry_point
.mask entry_point
JMP l^entry_point+2
.endm

; define a macro that places a longword constant directly
; into the transfer vector array.
.macroXFRCONconstant
.alignQUAD
.long constant
.endm
```

```
; and start defining the transfer vectors...
XFRCON42
XFRVECentry_point_name1,entry__point_name1
XFRVECentry_point_name2,entry__point_name2

.End
```

Use the template above for building a set of transfer vectors. You will need to substitute the externally visible symbols – those symbols that you determined should be visible outside the shareable image – in place of the example entry point names (entry_point_name1 or entry_point_name2) used in the template.

You do not need to change the symbol names of any externally visible symbols. You can (and probably should) use the same name in the transfer vectors (the transfer vectors are the .TRANSFER-created stuff in the MACRO32 file) as you have used in the code or data in the object file(s).The symbol used in the transfer vector will be the one visible to the callers of the shareable image. The linker knows that a TRANSFER may duplicate another internal symbol. The linker uses the value from the transfer directive when it builds the externally-visible shareable image symbol table – shareable and executable images that link against the shareable image never realize there is a second (internal) symbol of the same name, since it is not included in the externally-visible shareable image symbol table. Note that this means all calls to an external entry point that are made from within a shareable image *do not* use the shareable image transfer vector, these calls directly call the internal entry point of the same name.

Put in as many occurrences of XFRVEC as you need. Always add new transfer vectors to the end of the list. Never reorder the list. Do not remove any existing vectors. Jump to a dummy routine that returns a failure status, if you must. Reordering the list of vectors or removing individual vectors will tend to break existing applications that are linked against the shareable image.

If you really must change the order or remove a vector – you have decided you must make an incompatible change – further information on the GSMATCH option is available.

3. Create a set of SYMBOL_VECTOR transfer vectors in the LINKER options file for the symbols (OpenVMS Alpha):

```
SYMBOL_VECTOR = ( entry_point_name1 = PROCEDURE,-
entry_point_name2 = PROCEDURE )
```

The OpenVMS Alpha LINKER implements the transfer vectors using the LINKER options file and the image symbol table. The design of the Alpha Alpha pipelining and branch prediction makes the OpenVMS VAX transfer vector scheme rather expensive. The shareable image support is otherwise the same. Rephrased: a Macro (the language) file containing transfer vectors is *not* required on OpenVMS Alpha. It's all done in the LINKER options file...

It is possible that a large or complex symbol vector may exceed the capacity of the LINKER to process a single line in the options file. If this occurs, multiple SYMBOL_VECTOR declarations can be used.

Always add new SYMBOL_VECTOR entries to the end of the list. Never reorder the list. Do not remove any existing vectors. Jump to a dummy routine that returns a failure status, if you must. Reordering the list of vectors or removing individual vectors will tend to break existing applications that are linked against the shareable image.

For declaring a transfer vector for data, use the following lines in the OpenVMS Alpha LINKER options file:

```
SYMBOL_VECTOR = ( variable1 = DATA,-
variable2 = DATA )
```

4. Procedure to create common entry points (OpenVMS VAX and Alpha):

   It is quite possible to create a small command procedure that produces either a linker options file or a transfer vector array, based on the contents of a list of entry points – this procedure would allow maintenance of one source file for both varieties (OpenVMS VAX and Alpha) of transfer vector.

5. Adapt the following example procedure for your environment and files (OpenVMS VAX):

   The following command procedure assumes that the objects, object libraries, and/or shareable images specific to the application have already been created, and that the necessary Macro32 module containing the transfer vectors has already been created, and thus that this procedure need only specify the components and commands specifically necessary for the shareable image LINK operation. This particular example assumes that all application object modules are included in the object library named facnam_SHR.OLB.

```
$!LINK.COM
$!
$!This is an example of how to convert a user-created,
$!application-specific object library, facnam_SHR.OLB,
$!into an OpenVMS shareable image...
$
$!Assemble the transfer vectors...
$!
$ Macro xfrvec.mar /Object=Sys$Scratch:$$$XFRVEC
$
$!Now create the link options file on the fly...
$
$!Force any existing LNKOUT output file closed...
$!
$ Close/NoLog LNKOPT
$
$!Now open and populate the LNKOPT linker options
$!output file...
$!
$ Open/Write LNKOPT Sys$Scratch:$$$xfrvec.opt
$!The (CLUSTER and COLLECT directives are present
$!to pull the transfer vectors to the very front
$!of the shareable image.)
$ Write LNKOPT"Cluster=$$$XFRVEC"
$ Write LNKOPT"Collect=$$$XFRVEC,$$$XFRVEC"
$ Write LNKOPT"GSMATCH=LEQUAL,19,88"
$ Write LNKOPT"IDENTIFICATION=""facnam V1.0"" "
$ Close LNKOPT
$
$!Pull everything together...
$!
$ Link -
/Shareable=facnam_SHR.EXE -
/Symbol=facnam_SHR.Stb -
/Map=facnam_SHR.Map -
/NODEBUG /NOTRACEBACK -
Sys$Scratch:$$$XFRVEC.Obj, -
Sys$Scratch:$$$XFRVEC.Opt/Option, -
facnam_SHR.OLB/Lib
```

```
$
$!And clean up
$!
$ Delete Sys$Scratch:$$$XFRVEC.Obj.*
$ Delete Sys$Scratch:$$$XFRVEC.Opt.*
$
$ Exit
$
```

6. OpenVMS Alpha shareable images Macro32 transfer vector modules:

   OpenVMS Alpha shareable images do not require a Macro32 transfer vector module, the linker options file replaces this module.

7. Evaluate the PSECTs for Shareability:

   Look through the image **/MAP** produced by the image LINK. In particular, locate ANY variables that have both the PSECT (program section) attribute of SHR (shareable) and WRT (writeable). Make these variables NOSHR (not shareable) or NOWRT (not writeable) – how to do this is further discussed later in this section.

   If you fail to make all shareable PSECTS non-writable, you will have writeable variables shared between processes, and this typically leads to undesirable consequences. If both WRT and SHR are desired, you will have to **INSTALL** the shareable with the **/WRITEABLE** option before anyone can use it. Using this qualifier also restricts installation to a single node in a VMScluster.

   Also make certain that all code in the shareable is position independent code (PIC). (All Digital VAX high level languages typically produce position independent code – watch out for code written in MACRO32, however. It is quite possible to write position dependent code in MACRO32.)

Note

VAX C has several documented "quirks" involving "extern", "globalref", "globaldef" and "globalvalue" declarations. Each "extern" creates a seperate PSECT.The "global*" declarations typically all get placed in the same PSECT. Also note that C extern variables are visible to callers of the shareable image – the LINKER effectively translates them into UNIVERSAL or TRANSFER type symbols.(As a matter of preference, some folks tend to avoid using extern and global* symbols as much as is possible – for reasons of personal preference, flexibility and modularity.If one cannot avoid extern or global* symbols, or that a particular situation is made easier by using them, consider using one or two extern structures – which allows as many variables as needed, without creating a volume of external symbols. And as is to be expected, one should ALWAYS use unique facility prefixes on any extern or global* symbols. This helps avoid symbol name collisions.)

VSI C supports various external model pragmas, and these provide better control of the external declarations.

Should you wind up with a PSECT marked SHR and you want it NOSHR, you can copy the other PSECT attributes and a NOSHR onto a PSECT_ATTR directive in the LINKER options file. This is a `brute force' method – fixing the problem(s) by adding a language keyword (eg: "noshare" in VAX C) in the source is a better long term solution.) The following command procedure can be used to scan the LINK map looking for the WRT, SHR attribute combination.

```
$!PSECT_SCAN.COM - This procedure uses $SEARCH to find any
$!PSECTs marked both SHR and WRT.Any such lines lines
$!lines must be redefined as NOSHR,WRT with a PSECT_ATTR
```

```
$!line in the linker or by using a language-specific
$!directive such as the VAX C "noshare" storage modifier.
$!
$!
$
$ search facnam_SHR.map " SHR,"," WRT,"/MATCH=AND -
/output=SYS$SCRATCH:facnam_SHR_SHR-WRT.tmp
$ save_status = $status
$ If save_status .eqs. "%X00000001"
$ Then
$ ! Successful status means a bad PSECT combination was detected.
$ Write Sys$Output "Invalid PSECT attributes have been detected."
$ Write Sys$Output "The PSECT(s) requiring adjustment(s) are:"
$ Type SYS$SCRATCH:facnam_SHR_SHR-WRT.tmp
$ Exit 9
$ Endif
$ If save_status .eqs. "%X08D78053"
$ Then
$ ! Here, "no strings matched" means that you have no problem(s).
$ Write Sys$Output "Successful PSECT_SCAN completion."
$ Write Sys$Output "No invalid PSECT attributes have been detected."
$ Delete SYS$SCRATCH:facnam_SHR_SHR-WRT.tmp;*
$ Exit 1
$ Endif
$ Exit 1
```

8. Debugging:

   Adapt the LINK.COM command procedure to allow **/TRACEBACK** and/or **/DEBUG** during testing and development. But by all means, do *not* ship the shareable image with those options... To facilitate debugging, you will typically need to compile with qualifiers such as **/DEBUG** and **/NOOPTIMIZE**, and **LINK** with **/DEBUG**.

   The debugger **SET IMAGE** command allows you to access and debug shareable image.

   The **INSTALL** utility, to prevent security problems, is explicitly coded to refuse installation of any images linked with **/TRACEBACK** or with **/DEBUG**. See the comments on INSTALL, below.

9. Set the Shareable Image Version and Matching:

   To update the minor version of the GSMATCH line (8 in the above LINK.COM command procedure) whenever making upward-compatible changes. Update the major version (19 in the example) when making incompatible changes.

   Updating the minor version means programs linked against a new shareable will fail when an attempt is made to run the image against an older shareable. Updating the major version forces all images linked against the shareable to be relinked.

   Programs linked against an older shareable image will work with a newer shareable image. Assuming both have the same major version number.

   A few examples of incompatible changes:

   - re-ordered or removed transfer vectors...
   - routines of the same name that no longer return the same status values... One may see routines return a NOTIMPL or other status, when the routine is no longer needed.

- routines that change the documented behaviour of the shareable image in undocumented ways.

A few examples of upward compatible changes:

- new entry points are added at the end of the transfer vectors.
- a bug is fixed in a routine
- a new argument is added in an existing routine and the routine knowns how to deal with calls with differing numbers of arguments.
- An older routine is no longer needed, and is updated to call the replacement routine directly, or is updated to do nothing of consequence other than return a valid and previously-documented status code.

10. Update the IDENTIFICATION option:

    Update the image IDENTIFICATION string. This string is included in the LINKER options file. In the above example, this file is dynamically created by the example command procedure shown above. This string should be altered whenever making releases, so that it is obvious what product the shareable is part of. It is also often useful to encode a version string and a build base level number, to indicate which version and build base level the shareable image represents.

    The VSI Software Engineering Standards requires that all VSI products place their registered product name and version in this field. (And even if one is not developing a "real" product, this is still considered polite programming practice.)

    Some shareable images deliberately preallocate a series of dummy transfer vectors within the sharable image transfer vector array, and the programmers then reassign these vectors as newer routines are added. This technique avoids having to increment the GSMATCH version whenever newer routines are implemented or updated, easing the application compatibility across multiple versions of the shareable image. This technique can be useful when there are frequent changes during debugging, and in cases where both upward and downward compatibility can be maintained. However, this technique removes any chance of catching cases of newer applications running against older and potentially incompatible shareable images, and the shareable image programmers must use particular care to maintain both upward and downward compatibility of the interface across the entire series of shareable image versions sharing the same GSMATCH value.

    When a transfer vector does not reference an actual routine, it must return an application status code that indicates the routine is not currently implemented in the library. Further, it is imperative that the shareable image identification be maintained, so that there is some indication of what is contained within a particular copy of the shareable image.

    When using shareable images that use the same GSMATCH value and the dummy transfer vector technique, all applications that reference the shareable image should be coded to expect an application-defined NOTIMPL status, should the application be invoked against an older shareable image, a version that predates the necessary support. This is one of the cases that most programmers will depend on GSMATCH to detect and prevent, and this is one of the reasons why most shareable images should increment the minor GSMATCH whenever changes are made to the shareable image.

5. Errors and Solutions

%LINK-E-OUTSIMG

Sometimes you will see `LINK-E-OUTSIMG` errors. These errors are often due to an attempt to write data into a shareable image you are trying to link against. This error can obviously occur during an attempt to initialize an external symbol.

%SYSTEM-F-VECFULL

This error can also occur as a result of a bug in the image section processing in the LINKER utility – if the address is above 64K pages (hex 02000000), this error can be generated. Work with the options file to reduce the size of the image section below 64K pages. This bug is present in the LINKER in OpenVMS V5.5-2, and possibly in later versions.

%SYSTEM-F-NOTINSTALL

When an attempt is made to execute an image that contains one or more writeable shareable program sections (PSECTs), the image activator will return the SS$_NOTINSTALL error message. Writeable shareable images must be installed using the **/WRITEABLE** qualifier on **INSTALL**, or by altering the image to mark all writeable PSECTs to be non-shareable.

In most cases, a single PSECT that is marked with both the WRT (writeable) and the SHR (shareable) attributes is undesirable, and should be avoided – the unintentional and/or uncoordinated sharing of data among multiple applications can cause various transient problems at run-time. Additional information on program sections is available.

6. Frequently Asked Questions and Notes

How does one use and debug the shareable image?

One links against an OpenVMS shareable image using a linker options file. The following example command procedure demonstrates this, using a LINKER options file read directly from the command procedure's SYS$INPUT – from the command input:

```
$! LINK against the facnam_SHR.EXE image.
$ Link/Executable=ImageName.EXE -
/NODEBUG /NOTRACEBACK -
UserObject,
Sys$Input:/Option
facnam_SHR.EXE/Share
$
```

To debug a shareable image, one can use the **/DEBUG** and **/NOOPTIMIZE** qualifiers on the object modules used to create the shareable image, and one can also use the **/DEBUG** qualifier on the **LINK** command used to **LINK** the shareable image. Within the OpenVMS Debugger, one can use the **SET IMAGE** command to access the symbol table and the source code associated with the shareable image.

What Is An OpenVMS VAX Transfer Vector?

The OpenVMS VAX transfer vector construct is little more than a small amount of memory space located at a fixed location, usually at the front of the shareable image. This memory space can contain executable instructions, data, or pointers to either.

OpenVMS VAX transfer vectors are typically aligned at longword, quadword or octaword boundaries. Correct alignment facilitates easy replacement of an individual entry without affecting the other entries in the shareable image transfer vector table, and without requiring error-prone padding offset calculations. [For example, the ".mask"/"JMP nn+2" pair might be replaced with a "MOVL SS$_NORMAL,R0"/"RET" instruction pair, should a particular routine in the transfer vector no longer be required.]To maintain upward compatibility, the size of the replacement instruction(s) must not alter the alignment (the offset from the base of the image) of other vectors in the table.

In the typical case, a vector that references a CALLS/CALLG routine entry point, each entry in the transfer vector contains the contents of the "real" (word-length) CALLS/CALLG entry mask, and a (self-relative and position independent, also known as PC-relative) jump to the first instruction of the "real" routine. The word-length entry mask [per the VAX Calling Standard] is the two bytes that are located at the entry point, thus the "JMP real_routine_offset_address+2" must skip over them.

Logically, a constant value can be embedded in the transfer vector as is demonstrated in the XFRCON macro, or as implemented in the ".mask" of the XFRVEC macro. The former value is a longword value, the latter case is obviously a word. XFRCON does not attempt to provide a symbolic name for the constant, so any images calling the shareable will have to do some pointer arithmetic to locate the constant value. Your application might want to use a symbol – use a MACRO32 external_symbol:: (the double colon notation is important), or you might be able to get the .TRANSFER directive to work for you.

OpenVMS Alpha uses a somewhat different construct.

Based Shareable Image Caveats

All VSI compilers produce position independent code. Typically, based shareable images result from non-position independent MACRO32 code, or from certain C coding constructs.

Based shareable images should be avoided ("like the plague") whenever possible. Based images result in address space conflicts when other shareable images increase in size; thus cause a requirement for frequent image relinks.

If the linker reports the shareable image will be based, review the code, locate the offending instruction(s), and rework the path. The MACRO32 manual contains instructions on position independent design.

If you are taking the time to write a shareable, and the result is a based shareable image – you've wasted considerable effort for a rather small gain.

User-Written System Service Caveats (OpenVMS VAX)

User-Written System Services, often (confusingly) referred to as "Privileged System Services", are *not* shareable images installed with **/PRIVILEGED**. User-Written system services are one of the two types of images (the other is an executable image) that can be installed to grant enhanced privileges to a caller.

The design and coding required for a user-written system service is documented in the appendix of the system services manual and in the examples in SYS$EXAMPLES:USS*.*.To grant the caller the enhanced privileges, the user-written system service must be installed with the **/PROTECT** qualifier, *not* with the **/PRIVILEGED** qualifier.(The **INSTALL /PRIVILEGED** qualifier should be specified only on an executable image, the qualifier does not have the desired effect of granting privileges on an ordinary shareable image. Specification of **/PRIVILEGED** does, however, have some potentially interesting, unintended and likely undesirable side effects; it should not be specified on an ordinary shareable image. The same prohibition on specification of the **/PRIVILEGED** qualifier holds for user-written system services.)User-written system services gain enhanced privileges by the implicit SETPRV

privilege that is granted all routines running in executive or kernel mode. See the appendix of the system services reference manual, and the examples in SYS$EXAMPLES:USS*.*, for more information.

Failure to specify the INSTALL qualifiers correctly for the specific type of image involved (executable, shareable or privileged system service) leads to confusing error messages, failure to receive enhanced privileges and overzealous treatment as a privileged image.(in the case of an ordinary shareable image installed **/PRIVILEGED**, no enhanced privileges are granted, but image rundown treatment is that of a privileged image – no **CTRL/Y DEBUG** is allowed.

User-written system services should not be installed with **/PRIVILEGED**, only with **/PROTECT**.

An example of a VSI image that uses the user-written system service interface is the OpenVMS V5.3 (and later) image SYS$SHARE:SECURESHRP.EXE

User-written system services, as the routines execute in elevated processor modes and the routines are located in program sections placed on virtual memory pages owned by elevated processor modes, should be debugged with the XDELTA debugger. [One known bug in XDELTA: program sections (psects) in the process or control regions (P0/P1 address space) may need to be marked writeable ("WRT") for the duration of the debugging. Obviously, this problem directly affects user-written system services. XDELTA currently has occasional problems inserting breakpoints in P0/P1 space marked as NOWRT. This typically manifests itself on the XDELTA "B" and "O" commands.]

Do not use the OpenVMS Debugger (DEBUG) to debug executive or kernel mode code – if you somehow manage to get the privileged-mode code going in DEBUG, the mechanism by which the debugger implements breakpoints will likely cause a system crash when the breakpoint is hit – the system will likely crash soon.

For privileged-mode code, use the XDELTA or DELTA debugger. XDELTA is used for kernel-mode code that normally executes at elevated IPL and for code that executes without a process context, while DELTA is suited for inner-mode code running in the context of a process.

Alternatively, if symbolic debugging of privileged-mode code is desired, the OpenVMS Alpha System Code Debugger can be configured and used. The System Code Debugger uses the OpenVMS Debugger to debug inner-mode software that is running on a remote host.

User-Written System Service Caveats (OpenVMS Alpha)

OpenVMS Alpha user-written system services operate in a fashion very similar to those of OpenVMS VAX, save for the differences specific to the declaration of the entry points.

OpenVMS Alpha UWSS examples are located in SYS$EXAMPLES:UWSS*.*.

UWSS System Service Alternative

As code in a UWSS gains privileges by execution in a privileged processor mode – executive or kernel mode – it can be potentially difficult to perform various operations normally performed only from user-mode code. And as code effectively becomes fully privileged – implicit SETPRV is granted to code executing in privileged processor modes – great care must be taken to determine and control the access that is granted intentionally *and potentially unintentionally* to the users and to the programmers via the UWSS code.

If the system is running OpenVMS V6.0 or later, consider using a subsystem access control list entry (ACE) to allow finely-grained control over access, and to avoid the need for executive or kernel mode code – the subsystem ACE allows the system or application manager to "grant" one or more identifiers to the user for the duration of the execution of an image, via a subsystem ACE on the image(s), and via a

standard identifier ACE on the target object.(An "object" can be a file, mailbox, global section, etc.)And unlike an OpenVMS privilege, the access granted (or denied) by a specific identifier can be directly and easily controlled.

For information on the subsystem ACE, see the *Guide to OpenVMS System Security* [https://docs.vmssoftware.com/vsi-openvms-guide-to-system-security/].

Note that there exists a patch for images that use the subsystem ACE and are also installed /SHARED – this combination causes the image to erroneously exit with the "SUBTRACED" (SS$_SUBTRACED) error. This patch is needed for OpenVMS V6.0 and V6.1 systems – this problem was corrected in OpenVMS V6.2.

UWSS code cannot make arbitrary "outbound" calls to shareable images, as this could result in security problems. UWSS images can be linked against specific shareable images via specification of **/NOSYSSHR** on the **LINK** command *and* the explicit specification of the target shareable image(s) via the options file.

UWSS SECURITY NOTE

Great care should be used in the design and the implementation of user-written system services (privileged shareable images). User-Written system services are one of the `targets of choice' of system crackers. The user-written system service images are typically rather easy to locate on a system, and an incorrectly designed or coded image – an insecure image – will often grant a system cracker enhanced privileges. *Expect* that attempts to breach system security will be made against *all* user-written system service images.

Avoid an attempt to implement security checks within a UWSS image if it is possible to perform the necessary security check(s) via a call to the SYS$CHKPRO or SYS$CHECK_ACCESS system service.

INSTALL, RUN, LIB$FIND_IMAGE_SYMBOL (and the Image Activator) NOTES

The following points discuss some general features and some general restrictions of the OpenVMS image activation system services – there are a number of clients of the image activator system services.

- If an executable image is installed, it can cause the image activator to require that all component shareable images also be installed. This behaviour is intentional and prevents a nefarious user from redirecting a shareable image activation to an "evil twin" shareable image.

- Installing an image can also require that all logical names used to redirect shareable images during an image activation be defined in executive mode.

- At most, one image file can be installed **/SHARE** under a single file name.(The image activator uses only the file name – and it does not consider the device nor the directory – when it creates names for sections.)This restriction has been removed in OpenVMS V6.2 and later.

- To bypass the known image activation search logic in the image activator, specify the file version number on the **RUN** command.

- Do not base applications on the section names chosen for an INSTALLed image – these are the names are displayed by the **INSTALL LIST/GLOBAL** command. Do *not* attempt to mix sections created by **INSTALL** and any global section system services. (This erroneous programming technique is occasionally seen with the COMMON construct.)

- The LIB$FIND_IMAGE_SYMBOL routine performs a "merged image activation" – it dynamically merges a shareable image into the address space accessible to the currently executing image, and it

returns a pointer to a symbol in the shareable image. (This symbol allows the executable or shareable image that performed the merge to resolve the addresses within the merged image – these addresses are necessary in order to access and use the code and/or data present in the merged image.

Note that there is no way to "unmerge" a specific image, short of the execution of an image (or process) rundown.

- Shareable images are the only images that are supported by the LIB$FIND_IMAGE_SYMBOL routine.(Other image types might work under certain circumstances; possibly restricted to a particular operating system version or operating system hardware platforms – but LIB$FIND_IMAGE_SYMBOL merged image activations of these other image types are not supported.)

VSI C++ NOTES

VSI C++ includes a feature known as symbol name mangling, to hash the longer C++ symbol names into the traditional symbol name length limits of OpenVMS. This name mangling can have obvious implications for the creation of shareable images using C++ code.

VSI C++ versions V5.6 and later include instructions, located in SYS$COMMON:[SYSHLP.EXAMPLES.CXX], that allow you to correctly create shareable images using C++.Included in the information is how to determine what C++ and OpenVMS features you require, how to correctly deal with the mangled names, and how to create C++ shareable images on OpenVMS VAX and OpenVMS Alpha.

Unintentional Inclusion of Shareable Images

It is possible that shareable images are indirectly included into the link of a shareable or executable image by the OpenVMS VAX LINKER – where linking against one shareable image can implicitly include one or more other shareable images, and these images are also included in the shareable image list incorporated in the image headers of the image being linked. While this is often benign, there are situations where this behaviour of the OpenVMS VAX LINKER can cause problems.

Should it be necessary to avoid this behaviour, duplicate the entry points for the explicitly included shareable image and use these to create a "stub" image. One can then link any image(s) against this "stub" image and not include any other shareable images.

The OpenVMS Alpha LINKER does not have this behaviour.