Getting the Most out of Your Processor

This white paper tells image-providers—ISVs or product developers—how they can take advantage of some of the performance-enhancing features of newer Alpha processors.

The paper contains the following major sections:

1 Taking Advantage of Performance-Enhancing Features

Each generation of Alpha processors has different characteristics. Applications built to run on one generation of processors continue to run on newer generations. However, newer generations of processors are usually more capable and perform better than previous generations. By rebuilding applications for newer processors, you can take advantage of their new performance-enhancing capabilities and features.

The following two sections contain general descriptions of some of these new features. For more details, refer to the *Alpha Architecture Handbook*, which is available at the following web site:

```
http://www.support.compaq.com/alpha-tools/documentation/current/alpha-archt/alpha-architecture.pdf
```

Extended Instructions

Starting with the Alpha EV56 CPU, Alpha processors have added new groups of instructions, called **extended instructions**. The first group of extended instructions was implemented on EV56. EV6 and EV67 have additional groups of extended instructions. This paper will refer to processors with extended instructions as being “more capable” than processors with fewer or no extended instructions.

Extended instructions can improve performance for different types of work loads. For example:

Type of Extended Instruction	First Implemented in	Range of Use
Byte/word (BWX)	EV56[1]	Useful over a wide range of applications that do byte and word manipulation
Square root and Float/Integer Conversions (FIX)	EV6	Useful in many math applications
Bit Count (CIX)	EV67	Bit manipulations, including some data compression algorithms

[1]The common name and chip type for the compilers referred to in this document are as follows: EV4–21064, EV5–21164, EV56–21164A, EV6–21264, EV67–21264A

If you execute an extended instruction on a processor that does not implement it, the processor generates an exception. This exception will either cause the program to fail or cause the software to emulate the instruction.

Instruction Scheduling

Different generations of Alpha processors operate more efficiently with different sequences of instructions. For example, EV5 and EV6 processors operate simultaneously on different numbers of independent instructions.

When you rebuild an application for a specific processor, the compiler automatically optimizes the instruction sequences for that processor. An instruction sequence optimized for an EV6 processor, for example, gathers and orders independent instructions in larger groups than for an EV5 processor. Ordering of instructions is called **instruction scheduling**. Running a program with instruction scheduling for a particular processor on a different processor will cause the program to operate more slowly, but the program will not fail.

1.1 Improving Performance by Rebuilding Applications

To take advantage of the performance gains possible with extended instructions and instruction scheduling, image-providers can tell Alpha compilers to generate code using extended instructions or instruction scheduling, or both. Thus, they can ship images compiled for specific processors.

The following descriptions are very general. For more detailed explanations, refer to specific compiler manuals.

When compiling an application, you have three choices:

Method of Compiling	Results
Compile the application without considering the processor.	The compiler makes code generation choices to let the application run well on all processors.
Compile the application tuned for a particular processor.	Your application performs better on the processor for which it was built, but it still performs well on other processors.
Compile the application specifically for each processor on which you intend to run it.	Each application has the best possible performance on the particular processor for which it was built but might run poorly on some other processors.

You specify the method with one or another switch on the compile command. These switches are explained in the next section.

1.2 Using Compiler Switches to Specify Processor Optimizations

Compilers make all the changes to optimize an image for a particular processor. Most Alpha compilers—including Compaq C, C++, COBOL, and Fortran—have switches that you can use to determine how the compiler optimizes an image for a processor. The following table summarizes the performance of applications compiled with these switches.

Switch	Performance
/ARCHITECTURE=GENERIC	Good performance on all processors
/OPTIMIZE=TUNE=<processor>[1]	Better performance on specified processor; good performance on other processors.
/ARCHITECTURE=<processor>[1]	Specifies the minimum processor required for the image to perform well. Used with /OPTIMIZE=TUNE=, provides the best performance on specified processor; possibly poor performance on some other processors.[2]

[1]Acceptable values for *processor* include **GENERIC, EV4, EV5, EV56, EV6, EV67,** and so on. You can also specify HOST to indicate the same processor the compiler is currently running on. For all acceptable values, refer to the documentation for specific compilers.

[2]OpenVMS supplies an instruction emulator for the most common extended instructions so that some extended instructions work on older platforms—but much more slowly. (Compaq recommends using the methods described in this document to avoid executing extended instructions on less capable processors.)

The following sections describe the three switch options in more detail.

1.2.1 The /ARCHITECTURE=GENERIC Switch

If you use the /ARCHITECTURE=GENERIC switch when you rebuild an application, the compiler creates a generic object module. When linked into an image, this module runs well on any processor. Because you have only a single image, applications are less complex and kits are smaller. You will often see performance improvements when you compile generically simply because you are compiling with a new version of the compiler.

1.2.2 The /OPTIMIZE=TUNE= Switch

When you use the /OPTIMIZE=TUNE= switch, the compiler optimizes the generated code for the specified processor, while still allowing for acceptable performance on less capable processors. If you tune an application for an EV6 processor, for example, the application also performs well on an EV5 processor. If the compiler generates any extended instructions, it provides alternative code for processors that do not have the instructions.

You still produce only one set of images, which results in less complexity and a smaller kit size. However, in some cases different processors execute different code paths within the image. Testing on only one processor does not execute all possible code paths.

1.2.3 The /ARCHITECTURE=<processor> Switch

By using the /ARCHITECTURE=<processor> switch, you specify the least capable processor that the compiler will consider when it generates instructions. Less capable processors running code compiled with this switch might run poorly or not at all. Both the specified processor and more capable processors will perform the same as or better than if you used the /ARCHITECTURE=GENERIC switch.

You can consider that using the /ARCHITECTURE=<processor> switch is a way of telling the compiler that it can use new instructions provided by the specified processor. If you specify this switch either without /OPTIMIZE=TUNE= or with /OPTIMIZE=TUNE=<processor>, you produce the fastest possible code for the processor; however, the tradeoff is that the code will run poorly on less capable processors.

If you decide to use the /ARCHITECTURE=<processor> switch, you usually need to supply several different images, and the kit you supply will be larger. You also have more images and more combinations of images to test.

You do not need to supply separate images for every processor on which the application might run. If the application uses many byte instructions, for example, you could build images for the EV56 processor but also run them on an EV6 processor.

1.2.4 Summary of Tradeoffs of Using Different Types of Switches

The following table summarizes the tradeoffs of using different switch options.

Switch	Positive	Negative
/ARCHITECTURE= GENERIC	Application runs well on all processors. Application production is simpler.	Application code takes less advantage of features of newer processors.
/OPTIMIZE=TUNE= <processor>	Application runs better on specific processor it is tuned for. You need to supply only one image, so kits are smaller. Application production is simpler.	Application code does not produce peak performance on any processor.
/ARCHITECTURE= <processor>	Application runs with peak performance on a particular processor.	Because you must supply images for several processors, kits are larger. Application production is more complex.

1.2.5 Combining the /ARCHITECTURE=<processor> and /OPTIMIZE=TUNE=<processor> Switches

So far, this paper has implied that you would specify either the /ARCHITECTURE= or the /OPTIMIZE=TUNE= switch. In fact, you can use these two switches together. When you use both switches, the /OPTIMIZE=TUNE=<processor1> switch specifies that the compiled code will perform best on <processor1>, and /ARCHITECTURE=<processor2> acts as a qualifier to /OPTIMIZE=TUNE=, saying that the compiler does not need to consider a processor less capable than <processor2>.

If you do not specify /ARCHITECTURE=, the switch value defaults to GENERIC. If you do not specify /OPTIMIZE=TUNE= and do specify /ARCHITECTURE=<processor>, the processor for /OPTIMIZE=TUNE= defaults to the processor specified for /ARCHITECTURE=. The following table illustrates these defaults:

If you specify only....	The other switch defaults to....
/ARCHITECTURE=GENERIC	/OPTIMIZE=TUNE=GENERIC
/ARCHITECTURE=<processor>	/OPTIMIZE=TUNE=<processor>

If you specify only....	The other switch defaults to....
/OPTIMIZE=TUNE=<processor>	/ARCHITECTURE=GENERIC

Combining switches provides additional options if you know—or are willing to assume—the target processors on which your application will run. Two examples follow.

- Image built with /ARCHITECTURE=EV56/OPTIMIZE=TUNE=EV6

  This image might be a good compromise if you know that it will not run on a processor less capable than EV56, and you expect that many target processors will be EV6 or greater, or if you care most about the performance of EV6 and greater processors.

- Image built with /ARCHITECTURE=GENERIC/OPTIMIZE=TUNE=EV6

  This image might be a good compromize if you want to supply only a single image to run on all processors, but you expect that many target processors will be EV6 or greater, or if you care most about the performance of EV6 and greater processors.

The discussions that follow seldom mention combining switches. The discussion of /OPTIMIZE=TUNE= or tuned images assumes that the /ARCHITECTURE= switch is not specified or is specified as /ARCHITECTURE=GENERIC. The discussion of the /ARCHITECTURE= switch or of Extension-Specific Images assumes that /OPTIMIZE=TUNE= is not specified or is specified as the same processor. However, it is possible—and sometimes helpful—to tune for a processor more capable than the one specified in /ARCHITECTURE=. It is important to remember that whenever you specify an architecture other than GENERIC, you create an Extension-Specific Image even if you also specify /OPTIMIZE=TUNE=.

1.3 Choosing the Types of Images to Supply

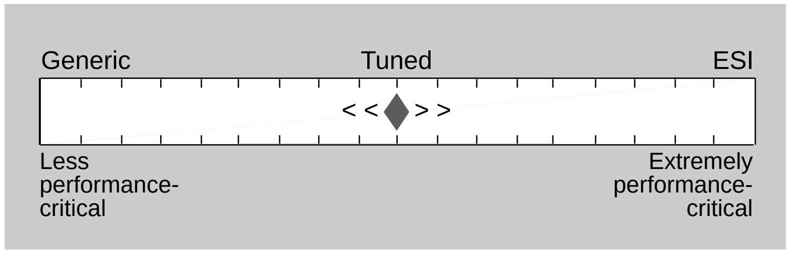
The following table describes the three types of images produced by using the switches described in the last section.

Switch Used	Image Type	Image Description
/ARCHITECTURE= <processor>	**Extension-specific image (ESI)**	Is optimized for a specific Alpha processor
/OPTIMIZE=TUNE= <processor>	**Tuned image**	Is optimized for a specific Alpha processor but also runs well on other processors
/ARCHITECTURE= GENERIC	**Generic image**	Runs on any processor (the default: the same result as using no switch)

Questions to Ask Yourself

To help you decide the types of images you need to supply to your customers, ask yourself these questions:

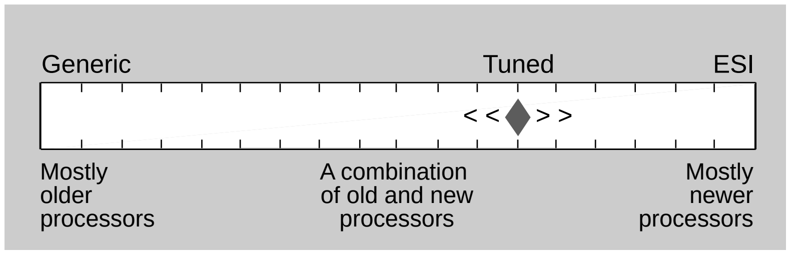
- How performance-critical are the applications you supply?



VM-0726A-AI

As the diagram indicates, the more performance-critical applications are, the more biased you should be toward creating ESIs for your customers. If your applications are mid-way between extremely and less performance-critical, consider compiling them using the /OPTIMIZE=TUNE= switch.

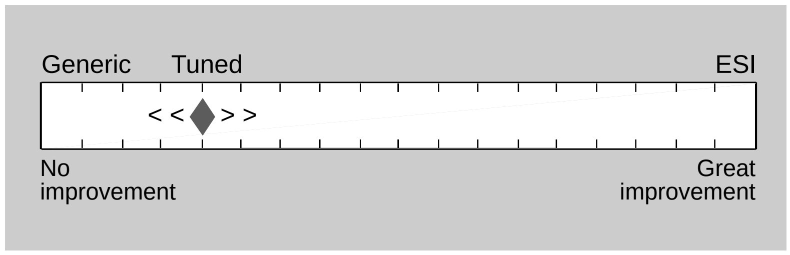
- Does your customer base have a high proportion of more capable processors?



VM-0727A-AI

If your customers have or soon will have newer processors, you should be biased toward providing either tuned applications or ESIs compiled for newer processors. Note in the diagram that a bias toward tuned images should continue until a large number of more capable processors in your customer base makes ESIs worth the extra work they require.

- Does your application perform better with extended instructions?



VM-0728A-AI

The diagram indicates that if the performance of your application does not improve from tuning and extended instructions, you might as well keep shipping generic images. However, if you start to see some improvement, you can become biased toward providing a tuned image. The more benefits you see, the more biased you can become toward ESIs because the extra work becomes more worthwhile.

Some applications benefit from tuning and ESIs more than others. If your applications do many operations that extended operations help, for example, you will benefit from tuned images and ESIs. The best way to determine whether your application will benefit is to do performance testing, but you can also get an idea of how much performance increase you can obtain by analysis.

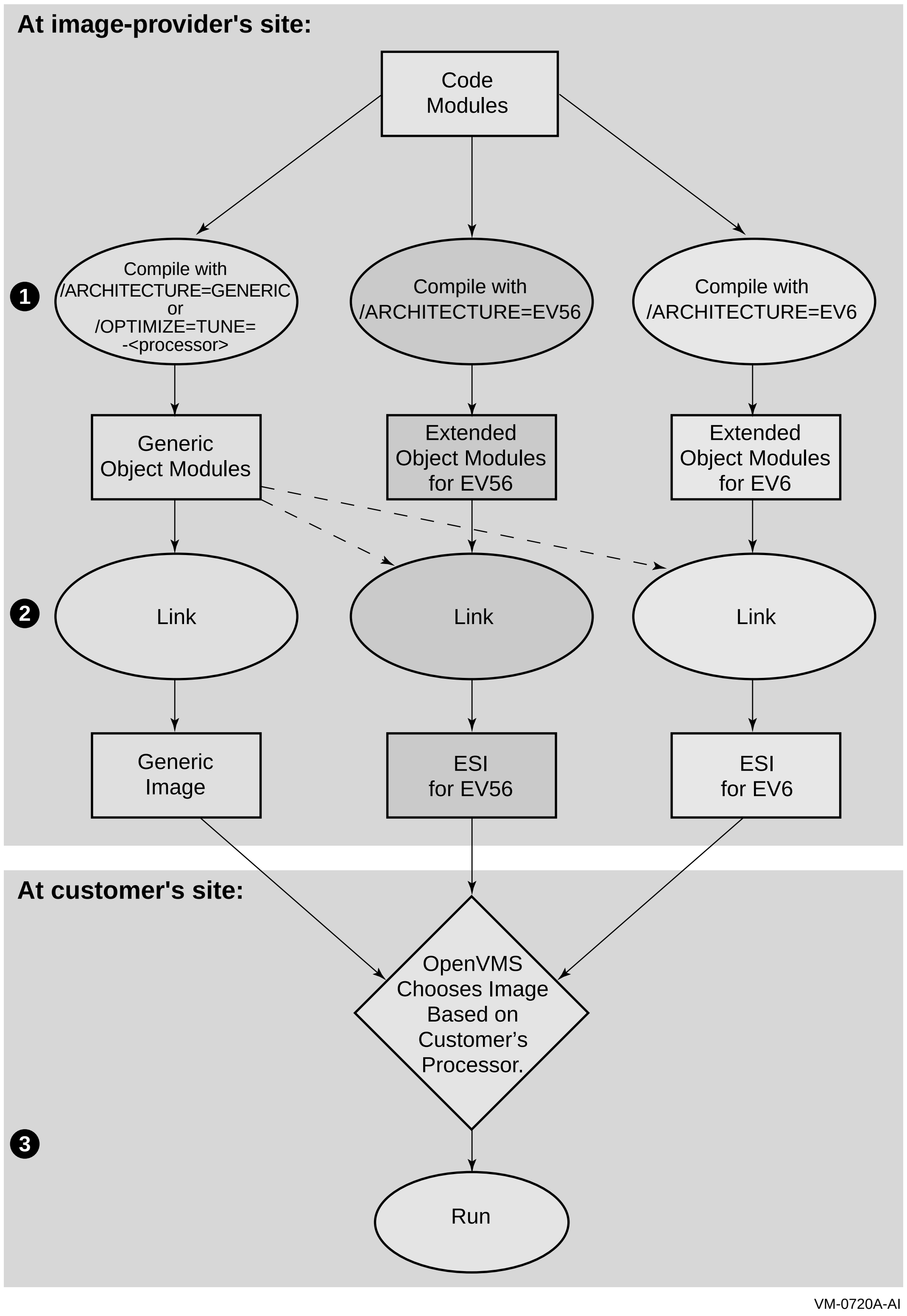
For example, if you know you do many byte operations in your application, you will probably get some benefit from images tuned for EV56 and above or from ESIs for EV56 and above. However, if you do mostly byte reads from memory, you will benefit less from tuned images or ESIs than if you do many byte stores. Also, many square root operations might bias you toward EV6 tuning or ESIs.

You must combine the "biases" that the diagrams indicate for your applications to decide whether to provide a generic image, a tuned image, ESIs, or a combination of /OPTIMIZE=TUNE= and /ARCHITECTURE= switches as described in Section 1.2.5. If you decide to create ESIs, the following sections provide guidelines for creating them.

1.4 Understanding How ESIs Are Created and Run

The following illustration shows an overview of the operations you must perform at your site and the customer site to create and run ESIs for EV56 or EV6 processors as well as generic images for less capable processors.

The example uses EV56 and EV6 processors. However, other processor types could be substituted for these two types.


At image-provider's site:
Code Modules
1
Compile with /ARCHITECTURE=GENERIC or /OPTIMIZE=TUNE= -<processor>
Compile with /ARCHITECTURE=EV56
Compile with /ARCHITECTURE=EV6
Generic Object Modules
Extended Object Modules for EV56
Extended Object Modules for EV6
2
Link
Link
Link
Generic Image
ESI for EV56
ESI for EV6
At customer's site:
OpenVMS Chooses Image Based on Customer's Processor.
3
Run
VM-0720A-AI

The numbers in the following list correspond to numbers in the figure. Actions 1 and 2 occur at the image-provider's site. Action 3 takes place at the customer's site.

In this section, any reference to *generic* also applies to *tuned*.

❶ The image-provider compiles the code modules.

- On the left side of the figure, the image-provider compiles code modules and tells the compiler to generate generic instruction sequences. Generic object modules are the result.
- In the center, the image-provider compiles code modules and tells the compiler to generate code for an EV56 processor. Extended object modules for an EV56 processor are the result.
- On the right, the image-provider compiles code modules and tells the compiler to generate code for an EV6 processor. Extended object modules for an EV6 processor are the result.

❷ The image-providers links the object modules.

- On the left, the image-provider links generic object modules to create a generic image.
- In the center, the image-provider links extended object modules for an EV56 processor to create an ESI for an EV56 processor. Dotted lines indicate that the image-provider might include some generic object modules in the ESI.
- On the right, the image-provider links extended object modules for an EV6 to create an ESI for an EV6 processor. Dotted lines indicate that the image-provider might include some generic object modules in the ESI.

You ship these images to the customer.

❸ At the customer site, when the customer begins to run the image:

- The OpenVMS operating system chooses the correct image for the processor on which it is running:
  - If the processor supports one of the ESIs supplied, OpenVMS uses it.
  - Otherwise, OpenVMS uses the generic image.
- OpenVMS then runs the selected image.

Questions You Might Have

Once you decide to use ESIs, you might have the following questions.

- **Which images should be ESIs?**

  If your product supplies multiple images, not all images need to be ESIs. Generic or tuned images can call ESIs, and ESIs can call generic or tuned images, without restriction. You might decide to supply ESI versions of only a few performance-critical images.

- **Should I try to decrease the number of ESIs in my product?**

  You might want to decrease the number of ESIs in order to reduce the size of your kit as well as the testing burden. However, be careful about moving performance-critical functions to a different module to decrease the number of ESIs. An extreme example is the following:

  1. Change all byte-store operations in your product to function calls.
  2. Put the function definition in an ESI module that could use the BWX instructions on EV56 and EV6 processors.

You will probably discover that the time the additional function calls take and the loss of cache and paging locality degrade performance more than the BWX instructions improve it.

- **How do I know it is worth using ESIs?**

  You do not. It is difficult to predict reliably whether an ESI will perform better than a tuned or generic image because so many factors affect performance. The best advice is to try building an ESI and test to make sure you are getting the hoped-for performance improvement.

- **What about combining tuning and ESIs as described in Section 1.2.5?**

  Combining the /ARCHITECTURE= and /OPTIMIZE=TUNE= switches is acceptable and often a good idea. The main principle to remember is that if you use /ARCHITECTURE=<processor>, you create an ESI; unless you know what other processors your image will be run on, you must also supply a generic image.

  You could consider supplying an image compiled with both switches if, for example, you know that your customers have a large number of both EV6 and EV56 processors. In this situation, /ARCHITECTURE=EV56/OPTIMIZE=TUNE=EV6 creates an ESI that runs best on EV6 but also takes advantage of the new instructions in the EV56. As usual, you must do performance testing with your own application to determine what is best for your particular situation.

2 Creating and Activating ESIs

Providers of ESIs need to determine which ESIs to supply and to define the images customers need to run on specific systems using OpenVMS tools. Image-providers need to follow the steps in the following table.

Step	Problem	Action
1.	How do you build extended object modules?	Compile code modules with the /ARCHITECTURE=<processor>, as described in Section 2.1.
2.	How do you link object modules to create ESIs?	Link generic or extended object modules, or both, as described in Section 2.2.
3.	How do you distinguish among multiple images that are specific to different processors?	Name ESIs according to the guidelines in Section 2.3.
4.	How does OpenVMS know which image is built for the current processor?	Use logical names to define the correct image, as explained in Section 2.4.

The following sections explain these steps.

2.1 Compiling Extended Instructions

Review the reasons for using compiler switches in Section 1.2. To compile code modules for specific extended instructions, enter a command similar to the following, which compiles a C module for an EV6 processor:

```
$ CC /OBJ=TEST_EV6.OBJ/ARCHITECTURE=EV6 TEST.C
```

Section 2.4.2 contains some precautions to consider when you use ESI logical names on the system you are using to build your application.

2.2 Linking Object Modules

You do not need to use a special link command in order for an image to be an ESI. You simply link the object modules that you have compiled as you usually do. Remember that even if you include only one extended object module with a large number of generic or tuned object modules, the result is still an ESI.

If you are explicitly linking against shareable images, you must link against the **generic shareable image** and not against the shareable ESI. If you do not link against the generic shareable image, the OpenVMS system will be unable to select the correct image to run on the customer's system.

Suppose, for example, that you want to link TEST.OBJ against TESTSHR.EXE, and you have the following shareable images on your (the image-provider's) system:

```
TESTSHR.EXE
TESTSHR_EV56
TESTSHR_EV6.EXE
```

Examples of correct link commands against shareable images follow.

- Example 1

  The following LINK command allows OpenVMS to choose, on the customer's system, the processor on which the image would run best:

  ```
  $ LINK/EXE=TEST.EXE TEST.OBJ/LIB,SYS$INPUT:/OPT
  SYS$SHARE:TESTSHR.EXE/SHARE
  ```

  Note that the second line in the example is input to the linker.

  If you link against TESTSHR_EV56.EXE, instead of TESTSHR.EXE, OpenVMS always uses the EV56 image regardless of the processor.

- Example 2

  The following LINK command links the EV6 ESI main image, TEST_EV6, but still allows the customer's system to choose the shareable image that runs best on its processor.

  ```
  $ LINK/EXE=TEST_EV6.EXE TEST_EV6.OBJ/LIB,SYS$INPUT:/OPT
  SYS$SHARE:TESTSHR.EXE/SHARE
  ```

Section 2.4.2 contains some precautions to take when you use ESI logical names on the system you are using to build your application.

2.3 Naming ESIs

To distinguish among multiple images for different processors, image-providers need to add a short, distinguishing string to the end of the name; for example:

Name	Description
TEST.EXE	Generic image
TEST_EV56.EXE	ESI for EV56 platforms
TEST_EV6.EXE	ESI for EV6 platforms

Here are some rules for naming images:

- Do not use a special suffix with generic images.

- Name ESIs with one of the following suffixes: EV56, EV6, EV67, and so on, matching the suffix with the value of the /ARCHITECTURE= switch used when you are compiling modules in the image.

Here are rules for supplying images:

- Generic images are the only images that you must supply.
- For each generic image, you can supply any number of ESIs. In other words, you can optionally supply any combination of EV56, EV6, EV67, and so on, or you can supply none.

2.4 Choosing the Correct Image to Activate

No new functionality has been added to OpenVMS systems to support ESIs. Instead, image-suppliers must define logical names on the customer's system. The logical names you define depend on the customer's processor type.

Defining logical names causes OpenVMS to activate the correct ESI when a customer runs an application. (Typically, image-providers define logical names in a startup command file in SYS$STARTUP, which runs either when the system is booted or when the product is initialized.)

For example, to ensure that OpenVMS activates TEST_EV6.EXE if a customer runs an ESI on an EV6 processor, you must define logical names for the EV6 processor; for example:

```
$ DEFINE/SYSTEM TESTSHR TESTSHR_EV6
$ DEFINE/SYSTEM TEST TEST_EV6
```

2.4.1 Selection of the Correct Image

Defining system-wide ESI logical names like the ones described in the last section causes OpenVMS to select the correct ESI under most circumstances. For example, OpenVMS selects the correct ESI under the following circumstances:

- When OpenVMS activates the shareable image because an image linked against it was activated
- When an image dynamically activates another image by calling the library routine LIB$FIND_IMAGE_SYMBOL
- When a CLD file specifies an image, and a DCL command activates that image
- When a user or command file runs an image directly ($ RUN image_name)
- When a user or command file specifies an image in an INSTALL command

In the last three cases, you must specify the image alone, without a directory specification. If you need to include a directory specification, do so in the logical name definition. Then use only the image name in the RUN command, for example:

```
$ DEFINE TEST DKA300:[TEST_DIR]TEST_EV6
$ RUN TEST
```

If you define a logical name for a shareable image that might be called either from a protected shareable image or from a main image installed with privileges, you must define the logical name in executive mode.

2.4.2 Precautions About Using Logical Names on the Image-Provider's System

Be careful when you build images—or any other part of your product—on an OpenVMS system where logical names for ESIs are already defined. (Section 2.4 explains how to define logical names.) If you simply specify a name (without other components of a file specification), OpenVMS tries to perform a logical name translation on the name. Thus, when you link a generic image on a system that has an ESI logical name defined for that image, you must specify more than just the base name. This is to prevent OpenVMS from using the equivalence name. For example, the following syntax is correct:

```
    $ LINK/EXE=TEST.EXE TEST.OBJ
```

If, instead of TEST.EXE, you enter TEST, OpenVMS attempts to translate TEST and might create an image with the wrong name.

2.4.3 Example of Determining Processor Type and Defining Logical Names Based on Processor

The following DCL code tests for the processor type and defines the appropriate ESI logicals. You can place code like this in the product's startup command file in SYS$STARTUP.

```
$ cputype = f$getsyi("REAL_CPUTYPE")
$! Do defines for older processors
$ if cputype .lt. 7 then goto END
$! Do defines for EV56
$ define/system/nolog TEST TEST_EV56
$ define/system/nolog TESTSHR TESTSHR_EV56
$ if cputype .lt. 8 then goto END
$! Do defines for EV6
$ define/system/nolog TEST TEST_EV6
$ if cputype .lt. 11 then goto END
$! Do defines for EV67
$ define/system/nolog TEST TEST_EV67
$ END:
```

This code compares the current processor against a list of possible processors. The order of processors in the list is from less to more capable, such that an ESI for one processor works on subsequent processors in the list.

After the code tests for a processor on the list, it creates a logical name for each ESI that you have supplied for that processor. If you have supplied ESIs for processors earlier in the list, the code overwrites their logicals.

Thus, when the code completes, logicals for each image point to the most capable corresponding ESI that can run on your processor.

To customize this code for your application, for each processor section delimited by a "Do defines" comment, place a define command for every image you supplied for that processor. The example code assumes that you have supplied ESIs for the processors shown in the following table. (Note that only one ESI has been supplied for EV67.)

Generic	EV56	EV6	EV67
TEST	TEST_EV56	TEST_EV6	TEST_EV67
TESTSHR	TESTSHR_EV56	TESTSHR_EV6	None

When you reach the label "END" in the code, a logical will have been defined to point to the most capable image that will run well on the current processor.

For example, if you run the code on an EV67 processor, the example code defines TEST to be TEST_EV67 and TESTSHR to be TESTSHR_EV6.