OpenVMS Alpha Partitioning and Galaxy Guide

Order Number: AA–REZQC–TE

April 2001

This guide describes how to use hard partitions, soft partitions (OpenVMS Galaxy), and resource affinity domains (RADs) with OpenVMS Alpha on AlphaServer systems that support these features. In addition, it describes how to create, manage, and use an OpenVMS Galaxy computing environment.

Revision/Update Information:	This supersedes the OpenVMS Alpha Version 7.2–1 manual.
Software Version:	OpenVMS Alpha Version 7.3

Compaq Computer Corporation
Houston, Texas

ZK6512

The Compaq *OpenVMS* documentation set is available on CD-ROM.

This document was prepared using DECdocument, Version 3.3-1b.

Contents

A OpenVMS Galaxy CPU Load Balancer Program

B Common Values for Environment Variables

Index

Figures

Tables

Preface

The *OpenVMS Alpha Partitioning and Galaxy Guide* describes how customers can take advantage of the partitioning and OpenVMS Galaxy capabilities available in OpenVMS Alpha Version 7.3.

The information in this document applies to OpenVMS Alpha systems only; it does not apply to OpenVMS VAX systems.

Intended Audience

This guide is intended for system managers, application programmers, technical consultants, data center managers, and anyone else who wants to learn about OpenVMS Galaxy and the partitioning capabilities of OpenVMS Alpha.

Document Structure

The *OpenVMS Alpha Partitioning and Galaxy Guide* introduces OpenVMS partitioning concepts and features on the hardware platforms that support them. It also explains how to use the OpenVMS Galaxy capabilities available in OpenVMS Alpha Version 7.3.

This guide covers the following OpenVMS Galaxy and partitioning topics:

- Basic concepts
- Hardware and software requirements
- Configuration alternatives
- Licensing information
- Installation procedures
- System management options
- Descriptions of the application programming interfaces (APIs) available to develop programs that take advantage of OpenVMS Galaxy features

The *OpenVMS Alpha Partitioning and Galaxy Guide* assumes that readers are familiar with OpenVMS concepts and operation, and it does not cover basic OpenVMS information.

Related Documents

The following manuals contain OpenVMS information that might be useful for partitioned computing environments:

- *OpenVMS Alpha Upgrade and Installation*
- *OpenVMS Cluster Systems*
- *OpenVMS Alpha System Analysis Tools Manual*

- *OpenVMS License Management Utility Manual*

For additional information about the OpenVMS products and services, access the Compaq website at the following address:

`http://www.openvms.compaq.com/`

Reader's Comments

Compaq welcomes your comments on this manual.

Print or edit the online form SYS$HELP:OPENVMSDOC_COMMENTS.TXT and send us your comments by:

Internet	**openvmsdoc@compaq.com**
Mail	Compaq Computer Corporation OSSG Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following World Wide Web address for information about how to order additional documentation:

`http://www.openvms.compaq.com/`

If you need help deciding which documentation best meets your needs, call 1-800-282-6672.

Conventions

In this manual:

- **Instance** refers to a copy of the OpenVMS Alpha operating system.
- **OpenVMS Galaxy** and **Galaxy** refer to the Compaq Galaxy Software Architecture on OpenVMS.
- **DECwindows** and **DECwindows Motif** refer to DECwindows Motif for OpenVMS software.

The following conventions are used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
$\boxed{\text{Return}}$	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.) In the HTML version of this document, this convention appears as brackets, rather than a box.

. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
[ ]	In command format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification or in the syntax of a substring specification in an assignment statement.)
|	In command format descriptions, vertical bars separating items inside brackets indicate that you choose one, none, or more than one of the options.
{ }	In command format descriptions, braces indicate required elements; you must choose one of the options listed.
bold text	This text style represents the introduction of a new term or the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace text`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1 Managing Workloads With Partitions and Resource Managemement

OpenVMS customers use systems that support hard and soft partitions in many different ways. To most effectively use these systems, customers can decide which configuration options best meet their computing and application needs.

This chapter describes how to use hard and soft partitions and the new OpenVMS support for resource affinity domains (RADs) to ensure that applications run as efficiently as possible on the new AlphaServer systems.

1.1 Using Hard and Soft Partitions on OpenVMS Systems

Hard partitioning is a physical separation of computing resources by hardware-enforced access barriers. It is impossible to read or write across a hard partition boundary. There is no resource sharing between hard partitions.

Soft partitioning is a separation of computing resources by software-controlled access barriers. Read and write access across a soft partition boundary is controlled by the operating system. OpenVMS Galaxy is an implementation of soft partitioning.

The way that customers choose to partition their new AlphaServer GS series systems depends on their computing environments and application requirements. When deciding how to configure an OpenVMS system that supports partitioning, customers need to consider the following questions:

- How many hard partitions do I need?
- How many soft partitions do I need?
- How small can I make the partitions?

1.2 OpenVMS Partitioning Guidelines

When deciding whether to use hard or soft partitons on the new AlphaServer GS series systems, you should be aware of the following:

- Hard partitions must be on Quad Building Block (QBB) boundaries. (Note that a hard partition can contain more than one QBB.)
- Soft partitions (Galaxy instances) do not have to be on QBB boundaries, but your configuration will be easier to understand and maintain if they are.
- Each partition (hard or soft) must have its own console line. Note that each QBB in the system can have one console line in it. Therefore, the maximum number of partitions (hard or soft) in the system equals the number of QBBs in the system.
- You can have multiple soft partitions within a hard partition.

- You only need one cluster license for the entire AlphaServer GS series system. It does not matter how many instances exist or how they are clustered, internally or externally. (Note that this is different from the AlphaServer 8400 system.)

1.3 Creating Hard Partitions

Each hard partition requires the following:

- Standard COM1 UART console line for each partition
- PCI drawer for each partition
- An I/O module per partition
- At least one CPU module per partition
- At least one memory module per partition

  Memory options should be selected in the context of an application's sensitivity to memory bandwidth and memory capacity, and the number of hardware partitions. This will determine the number of memory base modules and upgrades needed. The total capacity required will determine the size of the arrays to be chosen.

  Memory modules should be configured in powers of 2: That is, 0, 1, 2, or 4 base modules in a QBB. Upgrades should also be installed in powers of 2: 0, 1, 2, or 4 base modules in a QBB.

The following sections describe three hard partition configuration examples:

- Example configuration 1 has four QBBs and four hard partitions. Each hard partition contains one QBB.
- Example configuration 2 has four QBBs and two hard partitions. Each hard partition contains two QBBs.
- Example configuration 3 has four QBBs and two hard partitions. One partition contains one QBB, and the other contains three QBBs.

For more information about the partitioning procedures described in this section, see the *AlphaServer GS80/160/320 Firmware Reference Manual*.

1.3.1 Hard Partition Configuration Example 1

This example configuration uses four QBBs to configure four hard partitions. Each hard partition contains one QBB.

```
Configuration Example 1
   -------------------
  | NODE   | HP | QBB |
  | -------|----|-----|
  | WILD7  |  0 |   0 |
  | WILD8  |  1 |   1 |
  | WILD9  |  2 |   2 |
  | WILD10 |  3 |   3 |
   -------------------
```

To configure an AlphaServer GS160 system with four hard partitions, perform the following sequence of SCM commands:

From the SCM console enter the following settings for the *hp* NVRAM variables. Note that the values are bit masks.

```
SCM_E0> power off -all

SCM_E0> set hp_count 4
SCM_E0> set hp_qbb_mask0 1
SCM_E0> set hp_qbb_mask1 2
SCM_E0> set hp_qbb_mask2 4
SCM_E0> set hp_qbb_mask3 8
SCM_E0> set hp_qbb_mask4 0
SCM_E0> set hp_qbb_mask5 0
SCM_E0> set hp_qbb_mask6 0
SCM_E0> set hp_qbb_mask7 0

SCM_E0> power on -all
```

You can also power off/on individual hard partitions. For example, using this configuration, you would:

```
SCM_E0> power off -all
SCM_E0> set hp_count 4
SCM_E0> set hp_qbb_mask0 1
SCM_E0> set hp_qbb_mask1 2
SCM_E0> set hp_qbb_mask2 4
SCM_E0> set hp_qbb_mask3 8
SCM_E0> set hp_qbb_mask4 0
SCM_E0> set hp_qbb_mask5 0
SCM_E0> set hp_qbb_mask6 0
SCM_E0> set hp_qbb_mask7 0
SCM_E0> power on -partition 0
SCM_E0> power on -partition 1
SCM_E0> power on -partition 2
SCM_E0> power on -partition 3
```

During the powering up phases of each hard partition, status information is displayed showing how the partitions are coming online. Pay close attention to this information and confirm that there are no failures during this process.

As each hard partition comes online, you will be able to start working with that hard partition's console device. Also note, that depending on the setting of the NVRAM variable AUTO_QUIT_SCM, each hard partition's console will come on line in either the SCM or SRM console mode.

From each hard partition's console, you would then enter into the SRM console and configure any console variables specific to that hard partition. After that, you boot OpenVMS in each hard partition according to standard OpenVMS procedures. For example:

Hard partition 0 typical SRM console settings for OpenVMS:

```
P00>>>show bootdef_dev
bootdef_dev            dkb0.0.0.3.0
P00>>>show boot_osflags
boot_osflags           0,0
P00>>>show os_type
os_type                OpenVMS
```

Hard Partition 1 typical SRM console settings for OpenVMS:

```
P00>>>show bootdef_dev
bootdef_dev             dkb0.0.0.3.0
P00>>>show boot_osflags
boot_osflags            1,0
P00>>>show os_type
os_type                 OpenVMS
```

Hard Partition 2 typical SRM console settings for OpenVMS:

```
P00>>>show bootdef_dev
bootdef_dev             dkb0.0.0.3.0
P00>>>show boot_osflags
boot_osflags            2,1
P00>>>show os_type
os_type                 OpenVMS
```

Hard Partition 3 typical SRM console settings for Compaq *Tru64* UNIX:

```
P00>>>show bootdef_dev
bootdef_dev             dka0.0.0.1.16
P00>>>show boot_osflags
boot_osflags            A
P00>>>show os_type
os_type                 UNIX
```

1.3.2 Hard Partition Configuration Example 2

This example configuration uses four QBBs to configure two hard partitions. Each hard partition contains two QBBs.

```
Configuration Example 2
   -------------------
   |NODE   | HP | QBB |
   |-------|----|-----|
   |WILD7  |  0 |  0,1|
   |WILD8  |  1 |  2,3|
   -------------------
```

To configure an AlphaServer GS160 with two hard partitions, perform the following sequence of SCM commands:

From the SCM console enter the following settings for the *hp* NVRAM variables:

```
SCM_E0> power off -all

SCM_E0> set hp_count 2
SCM_E0> set hp_qbb_mask0 3
SCM_E0> set hp_qbb_mask1 c
SCM_E0> set hp_qbb_mask2 0
SCM_E0> set hp_qbb_mask3 0
SCM_E0> set hp_qbb_mask4 0
SCM_E0> set hp_qbb_mask5 0
SCM_E0> set hp_qbb_mask6 0
SCM_E0> set hp_qbb_mask7 0

SCM_E0> power on -all
```

As each hard partition comes on line, you'll be able to start working with that hard partition's console device.

From each hard partition's console, you would then, as in configuration example 1, enter into the SRM console, and configure any console variables specific to that hard partition, and then boot OpenVMS in each hard partition.

1.3.3 Hard Partition Configuration Example 3

Like Configuration 2, this configuration uses four QBBs to configure two hard partitions; the only difference is the changing of the number of QBBs per hard partition.

```
Configuration Example 3
    -------------------
   |NODE   | HP | QBB |
   |-------|----|-----|
   |WILD7  |  0 |0,1,2|
   |WILD8  |  1 |    3|
    -------------------
```

To configure an AlphaServer GS160 system, perform the following sequence of SCM commands:

From the SCM console enter the following settings for the *hp* NVRAM variables:

```
SCM_E0> power off -all

SCM_E0> set hp_count 2
SCM_E0> set hp_qbb_mask0 7
SCM_E0> set hp_qbb_mask1 8
SCM_E0> set hp_qbb_mask2 0
SCM_E0> set hp_qbb_mask3 0
SCM_E0> set hp_qbb_mask4 0
SCM_E0> set hp_qbb_mask5 0
SCM_E0> set hp_qbb_mask6 0
SCM_E0> set hp_qbb_mask7 0

SCM_E0> power on -all
```

As in the other examples, as each hard partition comes online, you will be able to start working with that hard partition’s console device.

From each hard partition’s console, you would then, as in configuration example 1, enter into the SRM console, and configure any console variables specific to that hard partition, and then boot OpenVMS in each hard partition.

1.3.4 Updating Console Firmware on AlphaServer GS80/160/320 Systems

To update the SRM console firmware on a system that is hard partitioned, you must do it separately for each hard partition. There is no way to update all of the firmware on each partition at one time.

1.4 OpenVMS Galaxy Support

OpenVMS Galaxy is an implementation of soft partitioning.

For information about OpenVMS Galaxy concepts, see Chapter 2.

1.4.1 Using Galaxy in Hard Partitions

You can create multiple soft partitions within a single hard partition by using the standard Galaxy procedures as described in Chapter 9.

Note that the Galaxy ID is within the hard partition. That is, if you have two hard partitions and you run Galaxy in both, each Galaxy will have its own unique Galaxy ID. Keep this in mind when you use network management tools; they will identify two Galaxy environments in this case.

1.5 OpenVMS Application Support for Resource Affinity Domains (RADs)

The large amount of physical memory in the new AlphaServer GS series systems provides opportunities for extremely large databases to be completely in memory. The AlphaServer nonuniform memory access (NUMA) system architecture provides the bandwidth to efficiently access this large amount of memory. NUMA is an attribute of a system in which the access time to any given physical memory is not the same for all CPUs.

In OpenVMS Alpha Version 7.2–1H1, OpenVMS engineering added NUMA awareness to OpenVMS memory management and process scheduling. This capability (application support for resource affinity domains) ensures that applications running in a single instance of OpenVMS on multiple QBBs can execute as efficiently as possible in a NUMA environment.

The operating system treats the hardware as a set of **resource affinity domains (RADs)**. A RAD is a set of hardware components (CPUs, memory, and I/O) with common access characteristics. On AlphaServer GS80/160/320 systems, a RAD corresponds to a Quad Building Block (QBB). A CPU references memory in the same RAD approximately three times faster than it references memory in another RAD. Therefore, it is important to keep the code being executed and the memory being referenced in the same RAD as much as possible while not giving some processes a consistently unfair advantage. Good location is the key to good performance, but it must be as fair as possible when fairness is important.

The OpenVMS scheduler and the memory management subsystem work together to achieve the best possible location by:

- Assigning each process a preferred or “home” RAD.
- Assigning process-private pages from the home RAD’s memory.
- Usually scheduling a process on a CPU in its home RAD.
- Replicating operating system read-only code on each RAD.
- Distributing global pages over RADs.
- Striping reserved memory over RADs.

For more information about using the OpenVMS RAD application programming interfaces, see Chapter 3.

2
OpenVMS Galaxy Concepts

The Compaq Galaxy Software Architecture on OpenVMS Alpha lets you run multiple instances of OpenVMS in a single computer. You can dynamically reassign system resources, mapping compute power to applications on an as-needed basis—without having to reboot the computer.

This chapter describes OpenVMS Galaxy concepts and highlights the features available in OpenVMS Alpha Version 7.3.

2.1 OpenVMS Galaxy Concepts and Components

With OpenVMS Galaxy, software logically **partitions** CPUs, memory, and I/O ports by assigning them to individual instances of the OpenVMS operating system. This partitioning, which a system manager directs, is a software function; no hardware boundaries are required. Each individual instance has the resources it needs to execute independently. An OpenVMS Galaxy environment is **adaptive** in that resources such as CPUs can be dynamically reassigned to different instances of OpenVMS.

The Galaxy Software Architecture on OpenVMS includes the following hardware and software components:

Console

The **console** on an OpenVMS system is comprised of an attached terminal and a firmware program that performs power-up self-tests, initializes hardware, initiates system booting, and performs I/O services during system booting and shutdown. The console program also provides run-time services to the operating system for console terminal I/O, environment variable retrieval, NVRAM (nonvolatile random access memory) saving, and other miscellaneous services.

In an OpenVMS Galaxy computing environment, the console plays a critical role in partitioning hardware resources. It maintains the permanent configuration in NVRAM and the running configuration in memory. The console provides each instance of the OpenVMS operating system with a pointer to the running configuration data.

Shared memory

Memory is logically partitioned into private and shared sections. Each operating system instance has its own private memory; that is, no other instance maps those physical pages. Some of the shared memory is available for instances of OpenVMS to communicate with one another, and the rest of the shared memory is available for applications.

The Galaxy Software Architecture is prepared for a nonuniform memory access (NUMA) environment and, if necessary, will provide special services for such systems to achieve maximum application performance.

CPUs

In an OpenVMS Galaxy computing environment, CPUs can be reassigned between instances.

I/O

An OpenVMS Galaxy has a highly scalable I/O subsystem because there are multiple, primary CPUs in the system—one for each instance. Also, OpenVMS currently has features for distributing some I/O to secondary CPUs in an SMP system.

Independent instances

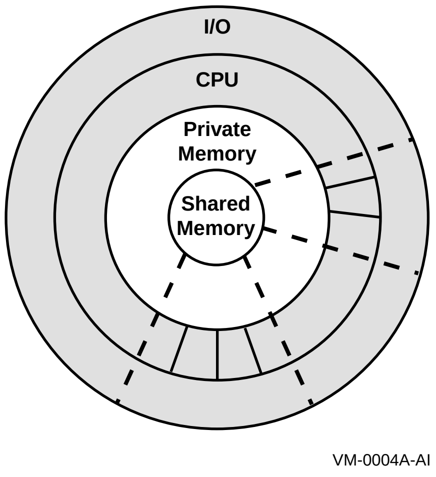
One or more OpenVMS instances can execute without sharing any resources in an OpenVMS Galaxy. An OpenVMS instance that does not share resources is called an **independent instance**.

An independent instance of OpenVMS does not participate in shared memory use. Neither the base operating system nor its applications access shared memory.

An OpenVMS Galaxy can consist solely of independent instances; such a system would resemble traditional mainframe-style partitioning. Architecturally, OpenVMS Galaxy is based on an SMP hardware architecture. It assumes that CPUs, memory, and I/O have full connectivity within the machine and that the memory is cache coherent. Each subsystem has full access to all other subsystems.

As shown in Figure 2–1, Galaxy software looks at the resources as if they were a pie. The various resources (CPUs, private memory, shared memory, and I/O) are arranged as concentric bands within the pie in a specific hierarchy. Shared memory is at the center.

Figure 2–1 OpenVMS Galaxy Architecture Diagram



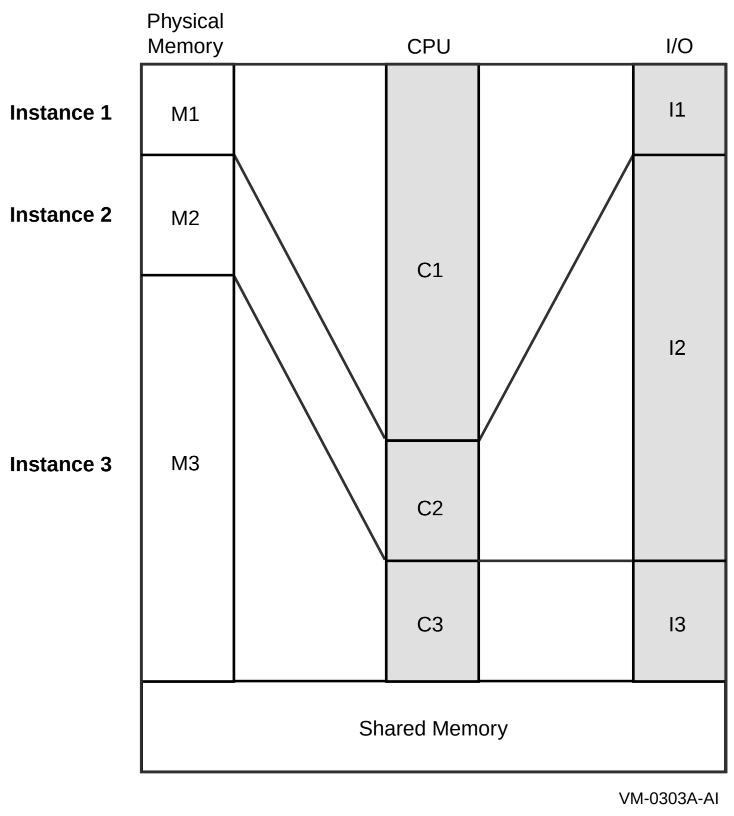
Galaxy supports the ability to divide the pie into multiple slices, each of disparate size. Each slice, regardless of size, has access to all of shared memory. Furthermore, because software partitions the pie, you can vary the number and size of slices dynamically.

In summary, each slice of the pie is a separate and complete instance of the operating system. Each instance has some amount of dedicated private memory, a number of CPUs, and the necessary I/O. Each instance can see all of shared memory, which is where the application data resides. System resources can be reassigned between the instances of the operating system without rebooting.

Another possible way to look at the Galaxy computing model is to think about how a system's resources could be divided.

For example, the overall sense of Figure 2–2 is that the proportion by which one resource is divided between instances is the proportion by which each of the other resources must be divided.

Figure 2–2 Another Galaxy Architecture Diagram



2.2 OpenVMS Galaxy Features

An evolution in OpenVMS functionality, OpenVMS Galaxy leverages proven OpenVMS Cluster, symmetric multiprocessing, and performance capabilities to offer greater levels of performance, scalability, and availability with extremely flexible operational capabilities.

Clustering

Fifteen years of proven OpenVMS Cluster technology facilitates communication among clustered instances within an OpenVMS Galaxy.

An OpenVMS Cluster is a software concept. It is a set of coordinated OpenVMS operating systems, *one per computer*, communicating over various communications media to combine the processing power and storage capacity of multiple computers into a single, shared-everything environment.

An OpenVMS Galaxy is also a software concept. However, it is a set of coordinated OpenVMS operating systems, *in a single computer*, communicating through shared memory. An instance of the operating system in an OpenVMS Galaxy can be clustered with other instances within the Galaxy or with instances in other systems.

An OpenVMS Galaxy is a complete system in and of itself. Although an OpenVMS Galaxy can be added to an existing cluster as multiple cluster nodes can be added today, the single system is the OpenVMS Galaxy architecture focus. An application running totally within an OpenVMS Galaxy can take advantage of performance opportunities not present in multisystem clusters.

SMP

Any instance in an OpenVMS Galaxy can be an SMP configuration. The number of CPUs is part of the definition of an instance. Because an instance in the OpenVMS Galaxy is a complete OpenVMS operating system, all applications behave the same as they would on a traditional, single-instance computer.

CPU reassignment

A CPU can be dynamically reassigned from one instance to another while all applications on both instances continue to run. Reassignment is realized by three separate functions: stopping, reassigning, and starting the CPU in question. As resource needs of applications change, the CPUs can be reassigned to the appropriate instances. There are some restrictions; for example, the primary CPU in an instance cannot be reassigned, and a CPU cannot specifically be designated to handle certain interrupts.

Dynamic reconfiguration

Multiple instances of the OpenVMS operating system allow system managers to reassign processing power to the instances whose applications most need it. As that need varies over time, so can the configuration. OpenVMS allows dynamic reconfiguration while all instances and their applications continue to run.

2.3 OpenVMS Galaxy Benefits

Many of the benefits of OpenVMS Galaxy technology result directly from running multiple instances of the OpenVMS operating system in a single computer.

With several instances of OpenVMS in memory at the same time, an OpenVMS Galaxy computing environment gives you quantum improvements in:

- Compatibility—Existing applications run without changes.
- Availability—Presents opportunities to upgrade software and expand system capacity without down time.
- Scalability—Offers scaling alternatives that improve performance of SMP and cluster environments.
- Adaptability—Physical resources can be dynamically reassigned to meet changing workload demands.
- Cost of ownership—Fewer computer systems reduce system management requirements, floor space, and more.

- Performance—Eliminates many bottlenecks and provides more I/O configuration possibilities.

The following descriptions provide more details about these benefits.

Compatibility

Existing single-system applications will run without changes on instances in an OpenVMS Galaxy. Existing OpenVMS Cluster applications will also run without changes on clustered instances in an OpenVMS Galaxy.

Availability

An OpenVMS Galaxy system is more available than a traditional, single-system-view, SMP system because multiple instances of the operating system control hardware resources.

OpenVMS Galaxy allows you to run different versions of OpenVMS (Version 7.2 and later) simultaneously. For example, you can test a new version of the operating system or an application in one instance while continuing to run the current version in the other instances. You can then upgrade your entire system, one instance at a time.

Scalability

System managers can assign resources to match application requirements as business needs grow or change. When a CPU is added to a Galaxy configuration, it can be assigned to any instance of OpenVMS. This means that applications can realize 100% of a CPU's power.

Typical SMP scaling issues do not restrict an OpenVMS Galaxy. System managers can define the number of OpenVMS instances, assign the number of CPUs in each instance, and control how they are used.

Additionally, a trial-and-error method of evaluating resources is a viable strategy. System managers can reassign CPUs among instances of OpenVMS until the most effective combination of resources is found. All instances of OpenVMS and their applications continue to run while CPUs are reassigned.

Adaptability

An OpenVMS Galaxy is highly adaptable because computing resources can be dynamically reassigned to other instances of the operating system while all applications continue to run.

Reassigning CPUs best demonstrates the adaptive capability of an OpenVMS Galaxy computing environment. For example, if a system manager knows that resource demands change at certain times, the system manager can write a command procedure to reassign CPUs to other instances of OpenVMS and submit the procedure to a batch queue. The same could be done to manage system load characteristics.

In an OpenVMS Galaxy environment, software is in total control of assigning and dynamically reassigning hardware resources. As additional hardware is added to an OpenVMS Galaxy system, resources can be added to existing instances; or new instances can be defined without affecting running applications.

Cost of ownership

An OpenVMS Galaxy presents opportunities to upgrade existing computers and expand their capacity, or to replace some number of computers, whether they are cluster members or independent systems, with a single computer running multiple instances of the operating system. Fewer computers greatly reduces system management requirements as well as floor space.

Performance

An OpenVMS Galaxy can provide high commercial application performance by eliminating many SMP and cluster-scaling bottlenecks. Also, the distribution of interrupts across instances provides many I/O configuration possibilities; for example, a system's I/O workload can be partitioned so that certain I/O traffic is done on specific instances.

2.4 OpenVMS Galaxy Version 7.3 Features

With OpenVMS Alpha Version 7.3, you can create an OpenVMS Galaxy environment that allows you to:

- Run eight instances on an AlphaServer GS320
- Run four instances on an AlphaServer GS160
- Run two instances on an AlphaServer GS160
- Run three instances of OpenVMS on AlphaServer GS140 or 8400 systems
- Run two instances of OpenVMS on AlphaServer GS60, GS60E, GS80, 8200, 4100, or ES40 systems
- Reassign CPUs between instances
- Perform independent booting and shutdown of instances
- Use shared memory for communication between instances
- Cluster instances within an OpenVMS Galaxy using the shared memory cluster interconnect
- Cluster instances with non-Galaxy systems
- Create applications using OpenVMS Galaxy APIs for resource management, event notification, locking for synchronization, and shared memory for global sections
- Use the Galaxy Configuration Utility to view and control the OpenVMS Galaxy environment
- Run a single-instance OpenVMS Galaxy on any Alpha system for application development

2.5 Is an OpenVMS Galaxy for You?

For companies looking to improve their ability to manage unpredictable, variable, or growing IT workloads, OpenVMS Galaxy technology provides the most flexible way to dynamically reconfigure and manage system resources. An integrated hardware and software solution, OpenVMS Galaxy allows system managers to perform tasks such as reassigning individual CPUs through a simple drag and drop procedure.

An OpenVMS Galaxy computing environment is ideal for high-availability applications, such as:

- Database servers
- Transaction processing systems
- Data Warehousing
- Data Mining

- Internet servers
- NonStop eBusiness solutions

2.6 Why a Galaxy is a Good Business Choice

An OpenVMS Galaxy computing environment is a natural evolution for current OpenVMS users with clusters or multiple sparsely configured systems.

An OpenVMS Galaxy is attractive for growing organizations with varying workloads—predictable or unpredictable.

2.7 Possible OpenVMS Galaxy Configurations

An OpenVMS Galaxy computing environment lets customers decide how much cooperation exists between instances in a single computer system.

In a **shared-nothing** computing model, the instances do not share any resources; operations are isolated from one another.

In a **shared-partial** computing model, the instances share some resources and cooperate in a limited way.

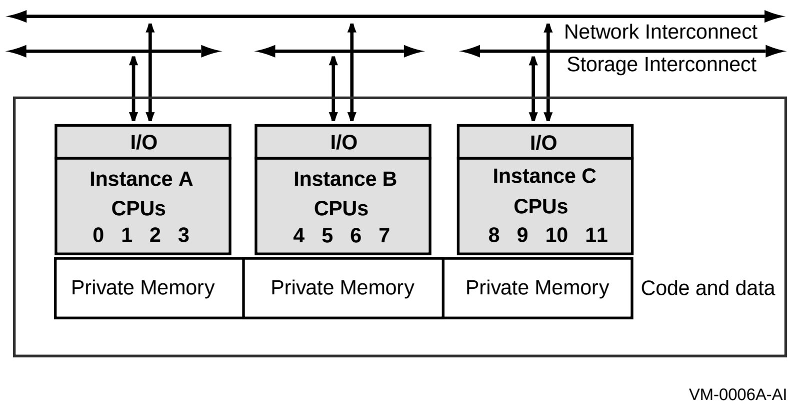
In a **shared-everything** model, the instances cooperate fully and share all available resources, to the point where the operating system presents a single cohesive entity to the network.

2.7.1 Shared-Nothing Computing Model

In a shared-nothing configuration (shown in Figure 2–3), the instances of OpenVMS are completely independent of each other and are connected through external interconnects, as though they were separate computers.

With Galaxy, all available memory is allocated into private memory for each instance of OpenVMS. Each instance has its own set of CPUs and an appropriate amount of I/O resources assigned to it.

Figure 2–3 Shared-Nothing Computing Model

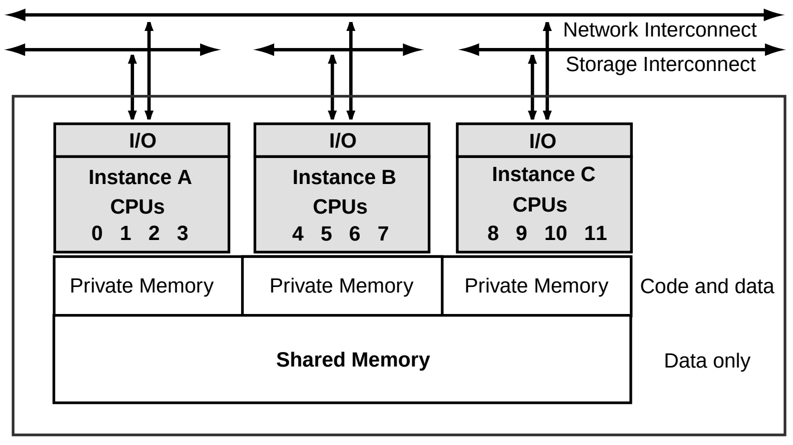


2.7.2 Shared-Partial Computing Model

In a shared-partial configuration (shown in Figure 2–4), a portion of system memory is designated as shared memory, which each instance can access. Code and data for each instance are contained in private memory. Data that is shared by applications in several instances is stored in shared memory.

The instances are not clustered.

Figure 2–4 Shared-Partial Computing Model

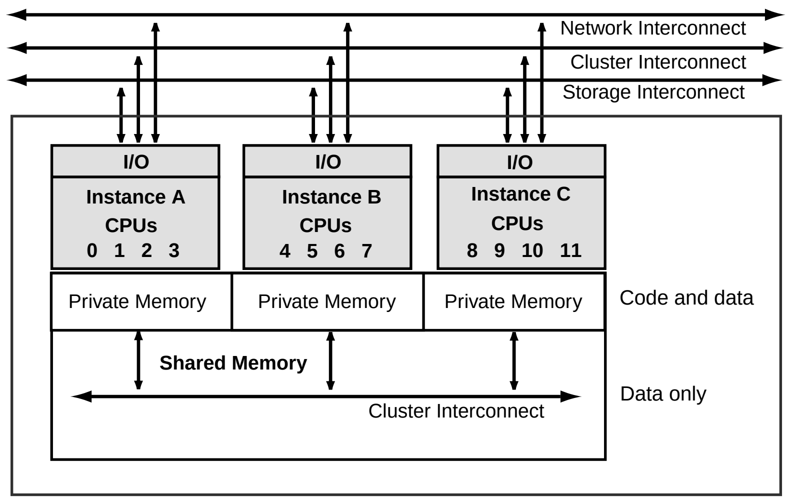


VM-0007A-AI

2.7.3 Shared-Everything Computing Model

In a shared-everything configuration (shown in Figure 2–5), the instances share memory and are clustered with one another.

Figure 2–5 Shared-Everything Computing Model



VM-0008A-AI

2.8 What Is a Single-Instance Galaxy?

A **single-instance Galaxy** is for non-Galaxy platforms, that is, those without a Galaxy console. Galaxy configuration data, which is normally provided by console firmware, is instead created in a file. By setting the system parameter GALAXY to 1, SYSBOOT reads the file into memory and the system boots as a single-instance Galaxy, complete with shared memory, Galaxy system services, and even self-migration of CPUs. This can be done on any Alpha platform.

Single-instance Galaxy configurations will run on everything from laptops to mainframes. This capability allows early adopters to evaluate OpenVMS Galaxy features, and most importantly, to develop and test Galaxy-aware applications without incurring the expense of setting up a full-scale Galaxy platform.

Because the single-instance Galaxy is not an emulator—it is real Galaxy code—applications will run on multiple-instance configurations.

For more information about running a single-instance Galaxy, see Chapter 10.

2.9 OpenVMS Galaxy Configuration Considerations

When you plan to create an OpenVMS Galaxy computing environment, you need to make sure that you have the appropriate hardware for your configuration. General OpenVMS Galaxy configuration rules include:

- One or more CPUs per instance
- One or more I/O modules per instance
- Dedicated serial console port per instance
- Memory
  - Enough private memory for OpenVMS and applications
  - Enough shared memory for the shared memory cluster interconnect, global sections, and so on.
- Display for configuration management with either an Alpha or VAX workstation running DECwindows or a Windows NT workstation with an X terminal emulator.

For more information about hardware-specific configuration requirements, see the chapter in this book specific to your hardware.

2.9.1 XMI Bus Support

The XMI bus is supported only on the first instance (Instance 0) of a Galaxy configuration in an AlphaServer 8400 system.

Only one DWLM-AA XMI plug-in-unit subsystem cage for all XMI devices is supported on an AlphaServer 8400 system. Note that the DWLM-AA takes up quite a bit of space in the system because an I/O bulkhead is required on the back of the system to connect all XMI devices to the system. This allows only two additional DWLPB PCI plug-in units in the system.

2.9.2 Memory Granularity Restrictions

Private memory must start on a 64 MB boundary.

Shared memory must start on an 8 MB boundary.

All instances except the last must have a multiple of 64MB.

2.9.3 EISA Bus Support

The EISA bus is supported only on the first instance (instance 0) of a Galaxy configuration. Due to the design of all EISA options, they must always be on instance 0 of the system. A KFE70 must be used in the first instance for any EISA devices in the Galaxy system.

All EISA devices must be on instance 0. No EISA devices are supported on any other instance in a Galaxy system.

A KFE72-DA installed in other instances provides console connection only and cannot be used for other EISA devices.

2.10 CD Drive Recommendation

Compaq recommends that a CD drive be available for each instance in an OpenVMS Galaxy computing environment. If you plan to use multiple system disks in your OpenVMS Galaxy, a CD drive per instance will be very helpful for upgrades and software installations.

If your OpenVMS Galaxy instances are clustered together and use a single common system disk, a single CD drive may be sufficient because the CD drive can be served to the other clustered instances. For operating system upgrades, the instance with the attached CD drive can be used to perform the upgrade.

2.11 Important Cluster Information

This section contains information that will be important to you if you are clustering instances with other instances in an OpenVMS Galaxy computing environment or with non-Galaxy OpenVMS Clusters.

For information about OpenVMS Galaxy licensing requirements that apply to clustering instances, see the *OpenVMS License Management Utility Manual*.

2.11.1 Becoming an OpenVMS Galaxy Instance

When you are installing OpenVMS Alpha Version 7.2–1, the OpenVMS installation dialog asks questions about OpenVMS Cluster and OpenVMS Galaxy instances.

If you answered "Yes" to the question

```
       Will this system be a member of a VMScluster? (Yes/No)
```

and you answered "Yes" to the question

```
Will this system be an instance in an OpenVMS Galaxy? (Yes/No)
```

the following information is displayed:

```
   For compatibility with an OpenVMS Galaxy, any systems in the VMScluster
   which are running versions of OpenVMS prior to V7.1-2 must have a
   remedial kit installed.  The appropriate kit from the following list
   must be installed on all system disks used by these systems.
   (Later versions of these remedial kits may be used if available.)

      Alpha V7.1 and V7.1-1xx        ALPSYSB02_071
      Alpha V6.2 and V6.2-1xx        ALPSYSB02_062

      VAX V7.1                       VAXSYSB01_071
      VAX V6.2                       VAXSYSB01_062
```

For more information, see *OpenVMS Alpha Installation and Upgrade Manual*.

2.11.2 SCSI Cluster Considerations

This section summarizes information about SCSI device naming for OpenVMS Galaxy computing environments. For more complete information about OpenVMS Cluster device naming, see the *OpenVMS Cluster Systems* manual.

If you are creating an OpenVMS Galaxy with shared SCSI buses, you must note the following:

For OpenVMS to give the SCSI devices the same name on each instance correctly, you will likely need to use the device-naming feature of OpenVMS.

For example, assume that you have the following adapters on your system when you enter the SHOW CONFIG command:

```
PKA0 (embedded SCSI for CDROM)
PKB0 (UltraSCSI controller KZPxxx)
PKC0 (UltraSCSI controller)
```

When you make this system a two-instance Galaxy, your hardware looks like the following:

```
Instance 0
PKA0  (UltraSCSI controller)

Instance 1
PKA0  (embedded SCSI for CDROM)
PKB0  (UltraSCSI controller)
```

Your shared SCSI will be connected from PKA0 on instance 0 to PKB0 on instance 1.

If you initialize the system with the LP_COUNT environment variable set to 0, you will not be able to boot OpenVMS on the system unless the SYSGEN parameter STARTUP_P1 is set to MINIMUM.

This is because, with the LP_COUNT variable set to 0, you will now have PKB connected to PKC, and the SCSI device-naming that was set up for initializing with multiple partitions is not correct for initializing with the LP_COUNT variable set to 0.

During the device configuration that occurs during boot, OpenVMS will notice that PKA0 and PKB0 are connected together. OpenVMS expects that each device has the same allocation class and names, but in this case, they will not.

The device naming that was set up for the two-instance Galaxy will not function correctly because the console naming of the controllers has changed.

2.12 Security Considerations in an OpenVMS Galaxy Computing Environment

OpenVMS Galaxy instances executing in a shared-everything cluster environment, in which all security database files are shared among all instances, automatically provide a consistent view of all Galaxy-related security profiles.

If you choose not to share all security database files throughout all Galaxy instances, a consistent security profile can only be achieved manually. Changes to an object's security profile must be followed by similar changes on all instances where this object can be accessed.

Because of the need to propagate changes manually, it is unlikely that such a configuration would ever be covered by a US C2 evaluation or by similar evaluations from other authorities. Organizations that require operating systems to have security evaluations should ensure that all instances in a single OpenVMS Galaxy belong to the same cluster.

2.13 Configuring OpenVMS Galaxy Instances in Time Zones

OpenVMS Galaxy instances do not have to be in the same time zone unless they are in the same cluster. For example, each instance in a three-instance Galaxy configuration could be in a different time zone.

2.14 Developing OpenVMS Galaxy Programs

The following sections describes OpenVMS programming interfaces that are useful in developing OpenVMS Galaxy application programs. Many of the concepts are extensions of the traditional single-instance OpenVMS system.

To see the C function prototypes for the services described in these chapters, enter the following command:

```
$ library/extract=starlet sys$library:sys$starlet_c.tlb/output=filename
```

Then search the output file for the service you want to see.

2.14.1 Locking Programming Interfaces

One of the major features of the Galaxy platform is the ability to share resources across multiple instances of the operating system. As with any shared resource, the need arises to synchronize access to that resource. The services described in this chapter provide primitives upon which a cooperative scheme can be created to synchronize access to shared resources within a Galaxy.

A **Galaxy lock** is a combination of a spinlock and a mutex. While attempting to acquire an owned galaxy lock, the thread will spin for a short period. If the lock does not become available during the spin, the thread will put itself into a wait state. This is different from SMP spinlocks in which the system crashes if the spin times out, behavior that is not acceptable in a Galaxy.

Given the nature of Galaxy locks, they will reside somewhere in shared memory. That shared memory can be allocated either by the user or by the galaxy locking services. If the user allocates the memory, the locking services track only the location of the locks. If the locking services allocate the memory, it is managed on behalf of the user.

Unlike other monitoring code which is only part of the MON version of execlets, the Galaxy lock monitoring code is always loaded.

There are several routines provided to manipulate Galaxy locks. The routines do not provide anything but the basics when it comes to locking. They are a little richer than the spinlocks used to support SMP, but far less than what the lock manager provides. Table 2–1 summarizes the OpenVMS system services for lock programming.

Table 2–1 Galaxy System Services for Lock Programming

System Service	Description
$ACQUIRE_GALAXY_LOCK	Acquires ownership of an OpenVMS Galaxy lock.
$CREATE_GALAXY_LOCK	Allocates an OpenVMS Galaxy lock block from a lock table created with the $CREATE_GALAXY_LOCK service.
$CREATE_GALAXY_LOCK_TABLE	Allocates an OpenVMS Galaxy lock table.
$DELETE_GALAXY_LOCK	Invalidates an OpenVMS Galaxy lock and deletes it.
$DELETE_GALAXY_LOCK_TABLE	Deletes an OpenVMS Galaxy lock table.
$GET_GALAXY_LOCK_INFO	Returns "interesting" fields from the specified lock.

(continued on next page)

Table 2–1 (Cont.) Galaxy System Services for Lock Programming

System Service	Description
$GET_GALAXY_LOCK_SIZE	Returns the minimum and maximum size of an OpenVMS Galaxy lock.
$RELEASE_GALAXY_LOCK	Releases ownership of an OpenVMS Galaxy lock.

2.14.2 System Events Programming Interfaces

Applications can register to be notified when certain system events occur; for example, when an instance joins the Galaxy or if a CPU joins a configure set. If events are registered, an application can decide how to respond when the registered events occur.

Table 2–2 summarizes the OpenVMS system services available for events programming.

Table 2–2 Galaxy System Services for Events Programming

System Service	Description
$CLEAR_SYSTEM_EVENT	Removes one or more notification requests previously established by a call to $SET_SYSTEM_EVENT.
$SET_SYSTEM_EVENT	Establishes a request for notification when an OpenVMS system event occurs.

2.14.3 Using SDA in an OpenVMS Galaxy

This section describes SDA information that is specific to an OpenVMS Galaxy computing environment.

For more information about using SDA, refer to the *OpenVMS Alpha System Analysis Tools Manual*.

2.14.3.1 Dumping Shared Memory

When a system crash occurs in a Galaxy instance, the default behavior of OpenVMS is to dump the contents of private memory of the failed instance and the contents of shared memory. In a full dump, every page of both shared and private memory is dumped; in a selective dump, only those pages in use at the time of the system crash are dumped.

Dumping of shared memory can be disabled by setting bit 4 the dynamic SYSGEN parameter DUMPSTYLE. This bit should only be set on the advice of “your Compaq support,” as the resulting system dump may not contain the data required to determine the cause of the system crash.

Table 2–3 shows the definitions of all the bits in DUMPSTYLE and their meanings in OpenVMS Alpha. Bits can be combined in any combination.

Table 2–3 Definitions of Bits in DUMPSTYLE

Bit	Value	Description
0	1	0= Full dump. The entire contents of physical memory will be written to the dump file. 1= Selective dump. The contents of memory will be written to the dump file selectively to maximize the usefulness of the dump file while conserving disk space. (Only pages that are in use are written).
1	2	0= Minimal console output. This consists of the bugcheck code; the identity of the CPU, process, and image where the crash occurred; the system date and time; plus a series of dots indicating progress writing the dump. 1= Full console output. This includes the minimal output described above plus stack and register contents, system layout, and additional progress information such as the names of processes as they are dumped.
2	4	0= Dump to system disk. The dump will be written to SYS$SYSDEVICE:[SYSn.SYSEXE]SYSDUMP.DMP, or in its absence, SYS$SYSDEVICE:[SYSn.SYSEXE]PAGEFILE.SYS. 1= Dump to alternate disk. The dump will be written to dump_dev:[SYSn.SYSEXE]SYSDUMP.DMP, where dump_dev is the value of the console environment variable DUMP_DEV.
3	8	0= Uncompressed dump. Pages are written directly to the dump file. 1= Compressed dump. Each page is compressed before it is written, providing a saving in space and in the time taken to write the dump, at the expense of a slight increase in time taken to access the dump.
4	16	0= Dump shared memory. 1= Do not dump shared memory.

The default setting for DUMPSTYLE is 0 (an uncompressed full dump, including shared memory, written to the system disk). Unless a value for DUMPSTYLE is specified in MODPARAMS.DAT, AUTOGEN.COM will set DUMPSTYLE to 1 (an uncompressed selective dump, including shared memory, written to the system disk) if there is less than 128 megabytes of memory on the system, or to 9 (a compressed selective dump, including shared memory, written to the system disk) otherwise.

2.14.3.2 Summary of SDA Command Interface Changes or Additions

The following list summarizes how the System Dump Analyzer (SDA) has been enhanced to view shared memory and OpenVMS Galaxy data structures. For more details, see the appropriate commands.

1. Added SHOW SHM_CPP. Default is a brief display of all SHM_CPPs.
2. Added VALIDATE SHM_CPP. Default action is to validate all SHM_CPPs and the counts and ranges of attached PFNs, but not the contents of the database for each PFN.
3. Added SHOW SHM_REG. Default is a brief display of all SHM_REGs.
4. Added /GLXSYS and /GLXGRP to SHOW GSD.
5. Added SHOW GMDB to display the contents of the GMDB and NODEB blocks. Default is detailed display of GMDB.

6. SHOW GALAXY shows a brief display of GMDB and all node blocks.
7. SHOW GLOCK displays Galaxy lock structures. Default is display of base GLOCK structures.
8. SHOW GCT displays Galaxy configuration tree. Default is /SUMMARY.
9. SHOW PAGE_TABLE and SHOW PROCESS/PAGE_TABLE.

3 NUMA Implications on OpenVMS Applications

NUMA is an attribute of a system in which access time to any given physical memory location is not the same for all CPUs. Given this architecture, you must have consistently good location (but not necessarily 100% of the time) for high performance. In the new AlphaServer GS series, CPUs will access memory in their own QBB faster than they will access memory in another QBB.

If Open VMS is running on the resources of a single QBB, then there is no NUMA effect and this discussion does not apply. Whenever possible and practical, you can benefit by running in a single QBB, thereby eliminating the complexities NUMA may present.

The most common question for overall system performance in a NUMA environment is, "uniform for all?" or "optimal for a few?" In other words, do you want all processes to have roughly equivalent performance, or do you want to focus on some specific processes and make them as efficient as possible? Whenever a single instance of OpenVMS runs on multiple QBBs (whether it is the entire machine, a hard partition, or a Galaxy instance), then you must answer this question, because the answer dictates a number of configuration and management decisions you need to understand.

The OpenVMS default NUMA mode of operation is "uniform for all". Resources are assigned so that over time each process on the system has, on average, roughly the same performance potential.

If "uniform for all" is not what you want, you must understand the interfaces available to you in order to achieve the more specialized "optimal for a few" or "dedicated" environment. Processes and data can be assigned to specific resources to give them the highest performance potential possible.

To further enhance your understanding of the NUMA environment, this chapter discusses the following:

- Base operating system NUMA actions
- Application resource considerations
- APIs

3.1 OpenVMS NUMA Awareness

OpenVMS memory management and process scheduling have been enhanced to work more efficiently on the new AlphaServer GS Series systems hardware.

The operating system treats the hardware as a set of Resource Affinity Domains (RADs). A RAD is the software grouping of physical resources (CPUs, memory, and I/O) with common access characteristics. On the new AlphaServer GS Series systems, a RAD corresponds to a Quad Building Block (QBB). When a single instance of OpenVMS runs on multiple QBBs, a QBB is seen as a RAD by OpenVMS.

Each of the following areas of enhancement adds a new capability to the system. Individually each brings increased performance potential for certain application needs. Collectively they provide the environment necessary for a diverse application mix. The areas being addressed are:

- Assignment of process private pages
- Assignment of reserved memory pages
- Process scheduling
- Replication of read-only system space pages
- Allocation of nonpaged pool
- Tools for observing page assignment

A CPU references memory in the same RAD three times faster than it references memory in another RAD. Therefore, it is important to keep the code being executed and the memory being referenced in the same RAD as much as possible. Consistently good location is the key to good performance. In assessing performance the following questions illustrate the types of things a programmer needs to consider.

- Where is the code you are executing?
- Where is the data you are accessing?
- Where is the I/O device you are using?

The OpenVMS scheduler and the memory management subsystem work together to achieve the best possible location by:

- Assigning each process a preferred or "home" RAD.
- Usually scheduling a process on a CPU in its home RAD.
- Replicating operating system read-only code and some data in each RAD.
- Distributing global pages over RADs.
- Striping reserved memory over RADs.

3.1.1 Home RAD

The OpenVMS operating system assigns a home RAD to each process during process creation. This has two major implications. First, with rare exception, one of the CPUs in the process's home RAD will run the process. Second, all process private pages required by the process will come from memory in the home RAD. This combination aids in maximizing local memory references.

When assigning home RADs, the default action of OpenVMS is to distribute the processes over the RADs.

3.1.2 System Code Replication

During system startup the operating system code is replicated in the memory of each RAD so that each process in the system will be accessing local memory whenever it requires system functions. This replication is of both the executive code and the installed resident image code granularity hint regions.

3.1.3 Distributing Global Pages

The default action of OpenVMS is to distribute global pages (the pages of a global section) over the RADs. This approach is also taken with the assignment of global pages that have been declared as reserved memory during system startup.

3.2 Application Resource Considerations

Each application environment is different. An application's structure may dictate which options are best for achieving the desired goals. Some of the deciding factors include:

- Number of processes
- Amount of memory needed
- Amount of sharing between processes
- Use of certain base operating system features
- Use of locks and their location

There are few absolute rules, but the following sections present some basic concepts and examples that will usually lead to the best outcome. Localizing (on-QBB) memory access is always the goal, but it is not always achievable and that is where tradeoffs are most likely to be made.

3.2.1 Processes and Shared Data

If you have hundreds, or maybe thousands, of processes that access a single global section, then you most likely want the default behavior of the operating system. The pages of the global section will be equally distributed in the memory of all RADs, and the processes' home RAD assignments will be equally distributed over the CPUs. This is the distributed, or "uniform", effect where over time all processes have similar performance potential given random accesses to the global section. None will be optimal but none will be at a severe disadvantage compared to the others.

On the other hand, a small number of processes accessing a global section can be "located" in a single RAD as long as 4 CPUs can handle the processing load and a single RAD contains sufficient memory for the entire global section. This will localize most memory access and therefore enhance performance of those specifically located processes. This strategy can be employed multiple times on the same system by locating one set of processes and their data in one RAD and a second set of processes and their data in another RAD.

3.2.2 Memory

A single QBB can have up to 32 GB of memory; two can have up to 64 GB, and so on. Take advantage of the large memory capacity whenever possible. For example, consider duplicating code or data in multiple RADs. It will take some analysis, may seem wasteful of space, and will require coordination. However, it may be worthwhile if it ultimately makes significantly more memory references local.

Consider the use of a RAM disk product. Even if NUMA is involved, in-memory references will outperform real device I/O.

3.2.3 Sharing and Synchronization

Sharing data usually requires synchronization. If the coordination mechanism is a single memory location (sometimes called a latch, a lock, or a semaphore), then it may be the cause of many remote accesses and therefore degrade performance if the contention is high enough. Multiple levels of such locks distributed throughout the data may reduce the amount of remote access.

3.2.4 Use of OpenVMS Features

Heavy use of certain base operating system features will result in much remote access because the data to support these functions resides in the memory of QBB0. Some data cannot be duplicated and some can be but has not been yet.

3.3 RAD Application Programming Interfaces

A number of interfaces specific to RADs are available to application programmers and system managers for controlling the location of processes and memory if the system defaults do not meet the needs of the operating environment. The following subsections are brief descriptions; the details can be found in the appropriate *OpenVMS System Services Reference Manual*.

3.3.1 Creating a Process

If you want a process to have a specific home RAD, then use the new HOME_RAD argument in the SYS$CREPRC system service. This allows the application to control the location.

3.3.2 Moving a Process

If a process has already been created and you want to relocate it, use the HOME_RAD argument to the SYS$SET_PROCESS_PROPERTIES system service. The process's working set will be purged and, as it runs on the CPUs in its new home RAD, its private pages will be reassigned from memory in the new home RAD.

3.3.3 Getting Information About a Process

The SYS$GETJPI system service returns the home RAD of a process.

3.3.4 Creating a Global Section

The SYS$CRMPSC_GDZRO_64 and SYS$CREATE_GDZRO system services accept a RAD argument mask. This indicates in which RADs OpenVMS should attempt to assign the pages of the global section.

3.3.5 Assigning Reserved Memory

The SYSMAN interface for assigning reserved memory has a RAD qualifier, so a system manager can declare that the memory being reserved should come from specific RADs.

3.3.6 Getting Information About the System

The SYS$GETSYI system service defines the following item codes for obtaining RAD information.

- RAD_MAX_RADS shows the maximum number of RADs possible on a platform.
- RAD_CPUS shows a longword array of RAD/CPU pairs.
- RAD_MEMSIZE shows a longword array of RAD/page_count pairs.

- RAD_SHMEMSIZE shows a longword array of RAD/page_count pairs.

3.3.7 RAD_SUPPORT System Parameter

The RAD_SUPPORT system parameter has numerous bits and fields defined for customizing individual RAD-related actions.

3.4 RAD System Services Summary Table

The following table describes RAD system service information for OpenVMS Version 7.3.

For additional information, refer to the *OpenVMS System Services Reference Manual*.

System Service	RAD Information
$CREATE_GDZRO	Argument: **rad_mask** Flag: **SEC$M_RAD_HINT** Error status: **SS$_BADRAD**
$CREPRC	Argument: **home_rad** Status flag bit: **stsflg** Symbolic name: **PRC$M_HOME_RAD** Error status: **SS$_BADRAD**
$CRMPSC_GDZRO_64	Argument: **rad_mask** Flag: **SEC$M_RAD_MASK** Error status: **SS$_BADRAD**
$GETJPI	Item code: **JPI$_HOME_RAD**
$GETSYI	Item codes: **RAD_MAX_RADS**, **RAD_CPUS**, **RAD_MEMSIZE**, **RAD_SHMEMSIZE**, **GALAXY_SHMEMSIZE**
$SET_PROCESS_PROPERTIESW	Item code: **PPROP$C_HOME_RAD**

3.5 RAD DCL Command Summary Table

The following table summarizes OpenVMS RAD DCL commands. For additional information, refer to the *OpenVMS DCL Dictionary*.

DCL Command/Lexical	RAD Information
SET PROCESS	Qualifier: **/RAD=HOME=*n***
SHOW PROCESS	Qualifier: **/RAD**
F$GETJPI	Item code: **HOME_RAD**
F$GETSYI	Item codes: **RAD_MAX_RADS**, **RAD_CPUS**, **RAD_MEMSIZE**, **RAD_SHMEMSIZE**

3.6 System Dump Analyzer (SDA) Support for RADs

The following System Dump Analyzer (SDA) commands have been enhanced to include RAD support.

- SHOW RAD
- SHOW RMD (reserved memory descriptor)

- SHOW PFN

3.6.1 SHOW RAD

The SDA command SHOW RAD displays:

- Settings and explanations of the RAD_SUPPORT system parameter fields
- Assignment of CPUs and memory to the RADs

This command is useful only on hardware platforms that support RADs (for example, AlphaServer GS160 systems). By default, the SHOW RAD command displays the settings of the RAD_SUPPORT system parameter fields.

Format:

```
SHOW RAD [number|/ALL]
```

Parameter:

number

Displays information on CPUs and memory for the specified RAD.

Qualifier:

/ALL

Displays settings of the RAD_SUPPORT parameter fields and all CPU/memory assignments.

3.6.2 SHOW RAD Examples

Example 1 shows the settings of the RAD_SUPPORT system parameter fields.

```
1.  SDA> SHOW RAD

    Resource Affinity Domains
    -------------------------

    RAD information header address: FFFFFFFF.82C2F940
    Maximum RAD count:                       00000008
    RAD containing SYS$BASE_IMAGE:           00000000
    RAD support flags:                       0000000F

    3         2 2         1 1
    1         4 3         6 5         8 7         0
    +-----------+-----------+-----------+-----------+
    |..|..| skip|ss|gg|ww|pp|..|..|..|..|..|fs|cr|ae|
    +-----------+-----------+-----------+-----------+
    |..|..|    0| 0| 0| 0| 0|..|..|..|..|..|00|11|11|
    +-----------+-----------+-----------+-----------+

    Bit 0 = 1:          RAD support is enabled

    Bit 1 = 1:          Soft RAD affinity support is enabled
                        (Default scheduler skip count of 16 attempts)

    Bit 2 = 1:          System-space replication support is enabled

    Bit 3 = 1:          Copy on soft fault is enabled

    Bit 4 = 0:          Default RAD-based page allocation in use

                        Allocation Type              RAD choice
                        ---------------              ----------
                        Process-private pagefault    Home
                        Process creation or inswap   Random
                        Global pagefault             Random
                        System-space page allocation  Current

    Bit 5 = 0:          RAD debug feature is disabled)
```

Example 2 shows information about the CPUs and memory for RAD 2.

```
2.  SDA> SHOW RAD 2

    Resource Affinity Domain 0002
    -----------------------------

    CPU sets:

      Active      08 09 10 11
      Configure   08 09 10 11
      Potential   08 09 10 11

    PFN ranges:

      Start PFN   End PFN     PFN count   Flags
      ---------   --------    ---------   -----
      01000000    0101FFFF    00020000    000A  OpenVMS Base
      01020000    0103FFFF    00020000    0010  Galaxy_Shared

    SYSPTBR:      01003C00)
```

3.6.3 SHOW RMD (Reserved Memory Descriptor)

The SDA command SHOW RMD has been enhanced to indicate the RAD from which reserved memory has been allocated. If a RAD was not specified when the reserved memory was allocated, then SDA displays ANY.

3.6.4 SHOW PFD

The SDA command SHOW PFD has been enhanced to include the /RAD qualifier. It is analogous to the existing /COLOR qualifier.

3.6.5 RAD Support for Hard Affinity

```
$ SET PROCESS /AFFINITY /PERMANENT /SET=a,b,c,d,...
```

On a system that supports RADs, the set of CPUs to which you affinitize a process should be in the same RAD. For example, on an AlphaServer GS160 with CPUs 0,1,2,3 in RAD 0 and with CPUs 4,5,6,7 in RAD 1, SET=2,3,4,5 would *not* be a good choice because half of the time you could be executing off your home RAD.

4 Creating an OpenVMS Galaxy on AlphaServer GS140/GS60/GS60E Systems

OpenVMS Alpha Version 7.3 provides support for OpenVMS Galaxy configurations on AlphaServer GS60, GS60E, and GS140 systems. You can run three instances of OpenVMS on AlphaServer GS140 systems or two instances on AlphaServer GS60/GS60E systems.

To create OpenVMS Galaxy environments on AlphaServer GS60, GS60E, and GS140 systems, you must download the latest version of the V5.4-xx console firmware from the following location:

`http://ftp.digital.com/pub/DEC/Alpha/firmware/`

When you have the firmware, you can:

- Create an OpenVMS Galaxy computing environment on an AlphaServer GS140 by following the procedures in Chapter 5.
- Create an OpenVMS Galaxy computing environment on AlphaServer GS60 or GS60E systems by following the procedures in Chapter 6.

5

Creating an OpenVMS Galaxy on an AlphaServer 8400 System

This chapter describes how to create an OpenVMS Galaxy computing environment on an AlphaServer 8400.

5.1 Step 1: Choose a Configuration and Determine Hardware Requirements

Quick Summary of an AlphaServer 8400 Galaxy Configuration

9 slots for:

I/O modules (of type KFTIA or KFTHA)
Memory modules
Processor modules (2 CPUs per module)

Console line for each partition:

Standard UART for first partition
KFE72-DA for each additional partition

Rules:

Must have an I/O module per partition.
Maximum of 3 I/O modules.
Must have at least one CPU module per partition.

Example Configuration 1

2 partitions, 8 CPUs, 12 GB memory

9 slots allocated as follows:

2 I/O modules
4 Processor modules (2 CPUs each)
3 Memory modules (4 GB each)

Example Configuration 2

3 partitions, 8 CPUs, 8 GB memory

9 slots allocated as follows:

3 I/O modules
4 Processor modules (2 CPUs each)
2 Memory modules (4 GB each)

5.2 Step 2: Set Up Hardware

When you have acquired the necessary hardware for your configuration, follow the procedures in this section to assemble it.

5.2.1 Overview of KFE72-DA Console Subsystem Hardware

The AlphaServer 8400 provides a standard built-in UART, which is used as the console line for the primary Galaxy instance. The console for each additional instance requires a KFE72-DA console subsystem, which is the set of EISA-bus modules that establishes an additional console port.

Note that the AlphaServer 8400 supports a maximum of three I/O modules. Attempting to configure more than three is unsupported.

Each separate KFE72-DA subsystem must be installed in a separate DWLPB card cage with a hose connecting it to a separate I/O module of type KFTIA or KFTHA.

All KFTIA I/O modules must be installed first, starting at slot 8. Any KFTHA I/O modules must follow the KFTIA modules, using the consecutively lower-numbered slots.

You can use any combination of these two I/O modules as long as you follow this slot assignment rule.

When configuring a console subsystem, the I/O hose connecting the I/O module and DWLPB card cage must be plugged into the lowest hose port. Not just the lowest *available* hose port, but the absolute first hose port; the one closest to the top of the module.

The KFE72-DA contains three EISA modules that provide:

- Two COM ports
- An Ethernet port
- A small speaker and other ports (such as a keyboard and a mouse, which are not used for Galaxy configurations)

5.2.2 Installing the KFE72-DA Modules

For each instance of the OpenVMS operating system after instance zero, you must install the following three modules in the PCI card cage:

- Standard I/O module
- Serial port module
- Connector module

To install these modules, follow the procedures in Section 5.2.2.1 to Section 5.2.2.3, which supplement the installation procedures for KFE72-DA modules in Chapter 5 of the *KFE72 Installation Guide*.

5.2.2.1 Slide the PCI Card Cage Out

Follow procedures in Section 5.2.1 in the *KFE72 Installation Guide*.

5.2.2.2 Insert Modules and Connect Ribbon Cables

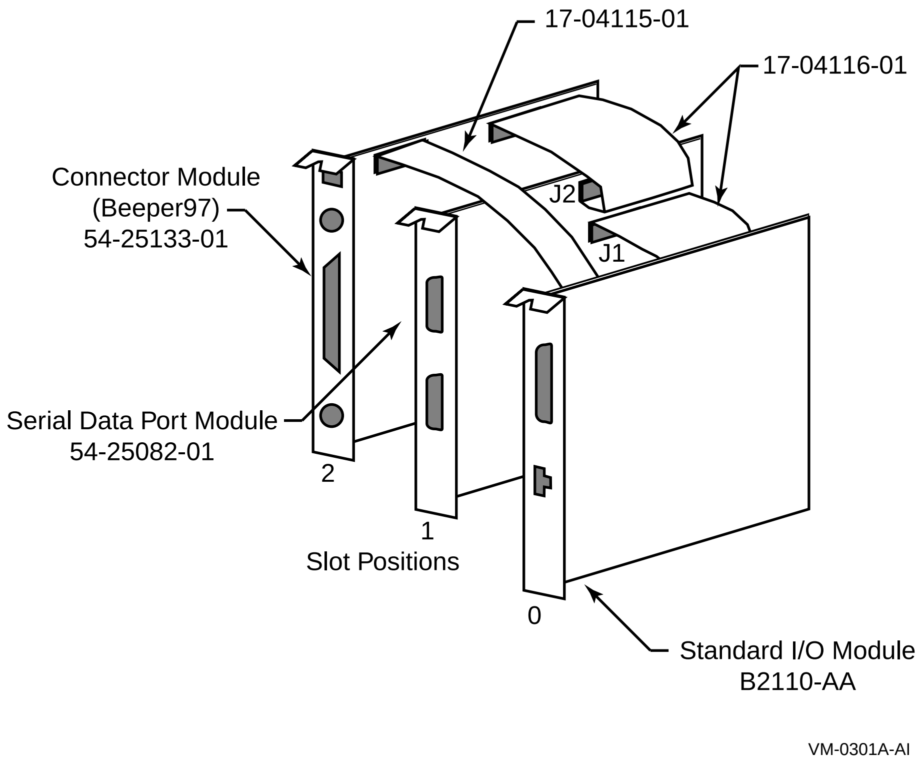
Note

When installing PCI modules, be sure the option bulkheads mate with the EMI gasket on the PCI card cage.

KFE72-DA modules must occupy slots 0, 1, and 2 of the DWLPB card cage.

To insert the modules in the PCI card cages and connect the appropriate ribbon cables, refer to Figure 5–1 and perform the following steps:

Figure 5–1 Attaching Ribbon Cables



1. Insert the standard I/O module (B2110-AA) in slot 0. Secure the module to the card cage with a screw.
2. Attach the 60-pin ribbon cables (17-04116-01) on the serial port module (54-25082-01) at connectors J1 and J2.
3. Insert the serial port module (54-25082-01) in slot 1. Secure the module to the card cage with a screw.
4. Attach the 60-pin ribbon cable (17-04116-01) from the serial port module at connector J1 to the standard I/O module.
5. Insert the connector module in slot 2.
6. Attach the 34-pin ribbon cable (17-04115-01) between the standard I/O module (B2110-AA) in slot 0 and the connector module (54-25133-01) in slot 2.

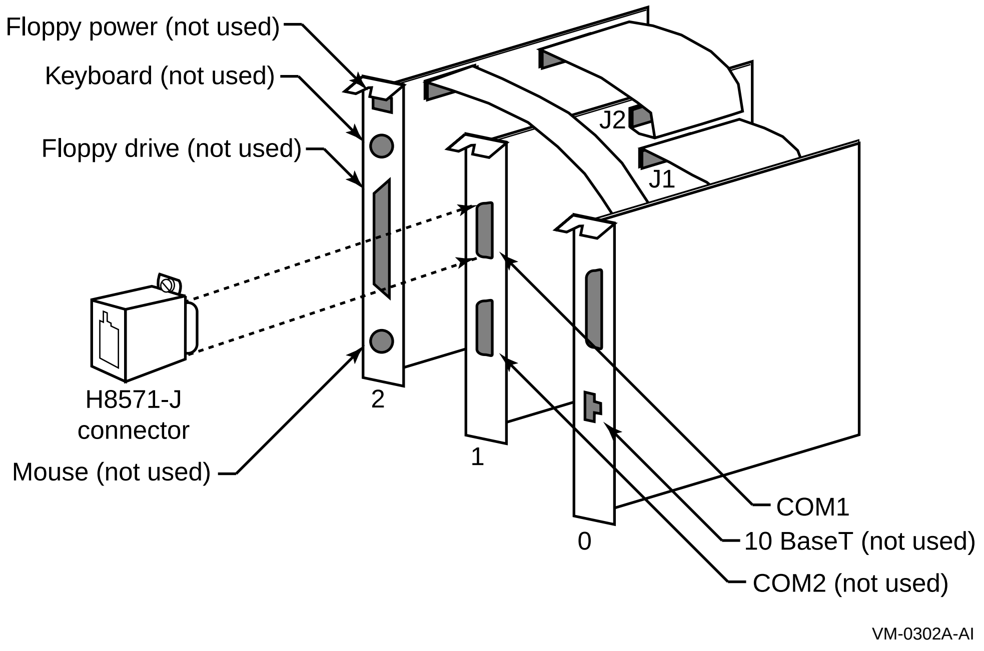
7. Attach the 60-pin ribbon cable (17-04116-01) from the serial port module at connector J2 to the connector module.

5.2.2.3 Attaching Connectors

To connect the console terminal and additional devices, refer to Figure 5–2 and connect the console serial line (H8571-J connector) to COM1.

Note that the pair of arrows between the numbers 1 and 2 on the serial port module is an industry standard symbol for a serial port and does not indicate port numbers.

Figure 5–2 Connectors



5.2.3 Slide Shelf Back Into System

To return the card cage, follow steps 2 through 9 in the procedure in Section 5.2.3 of the *KFE72 Installation Guide.*

5.2.4 Using a Terminal Server

You may want to bring your console lines together using a terminal server. For example, use a DECserver200 to allow reverse-LAT access to each console over the network. While this is not strictly required, it greatly simplifies OpenVMS Galaxy configuration management. Refer to the appropriate product documentation for details about configuring a LAT Server or other terminal concentrator.

5.2.5 Installing EISA Devices

Plug-in EISA devices can only be configured in partition 0. After installing EISA devices, the console will issue a message requesting that you run the EISA Configuration Utility (ECU).

Run the ECU as follows:

1. Shut down all OpenVMS Galaxy instances.

2. Be sure your floppy disk drive is properly connected to the primary partitions hardware. Typically the drive can be cabled into the Connector Module ("Beeper" part number 54-25133-01) in PCI slot 2.
3. Insert the diskette containing the ECU image.
4. Issue the following commands from the primary console:

```
P00>>> SET ARC_ENABLE ON
P00>>> INITIALIZE
P00>>> RUN ECU
```

5. Follow the procedures outlined by the ECU and exit when done.
6. P00>>> boot
7. $ @SYS$SYSTEM:SHUTDOWN
8. P00>>> SET ARC_ENABLE OFF
9. P00>>> INITIALIZE
10. P00>>> LPINIT
11. Reboot the OpenVMS Galaxy.

There are two versions of the ECU, one that runs on a graphics terminal and another that runs on character-cell terminals. Both versions are on the diskette, and the console determines which one to run. For OpenVMS Galaxy systems, the primary console will always be a serial device with a character cell terminal.

If the ECU is not run, OpenVMS will display the following message:

```
    %SYSTEM-I-NOCONFIGDATA, IRQ Configuration data for EISA
slot xxx was not found, please run the ECU and reboot.
```

If you ignore this message, the system will boot, but the plug-in EISA devices will be ignored.

Once you have configured and set up the OpenVMS Galaxy hardware as described in the previous sections, perform the following steps to install and boot OpenVMS Galaxy instances.

5.3 Step 3: Create System Disk

Decide whether to use a system disk per instance *or* whether to use a cluster common disk.

A new SECURITY.EXE is required for all cluster members running a version prior to OpenVMS Version 7.1-2 that share the same VMS$OBJECTS.DAT file with Galaxy instances.

5.4 Step 4: Install OpenVMS Alpha Version 7.3

No special installation procedures are required to run OpenVMS Galaxy software. Galaxy functionality is included in the base operating system and can be enabled or disabled using the console command and system parameter values described later in this chapter.

For more information about installing the OpenVMS Alpha operating system, see the *OpenVMS Alpha Version 7.3 Upgrade and Installation Manual*.

5.4.1 OpenVMS Galaxy Licensing Information

See *OpenVMS License Management Utility Manual*.

5.5 Step 5: Upgrade the Firmware

Creating an OpenVMS Galaxy environment on an AlphaServer 8400 requires a firmware upgrade to each processor module. If you use these modules again in a non-Galaxy configuration, you will need to reinstall the previous firmware. It is a good practice to have a current firmware CD on hand.

It saves some time if you install ALL processor modules you intend to use and update them at the same time. The AlphaServer 8400 requires that you use the same firmware on all processor boards. If you need to upgrade a board at a later time, you must:

1. Remove all boards that are not at the same firmware revision level.
2. Update the older boards.
3. Reinstall the remaining boards.

To upgrade your firmware, the system must be powered on, running in non-Galaxy mode (that is, the **lp_count** console environment variable—if you have established it—must be set to zero).

To set the console environment variable, use the following commands:

```
P00>>> SET LP_COUNT 0
P00>>> INIT
```

To upgrade the firmware, use the standard console firmware update available from AlphaServers Engineering.

5.6 Step 6: Set Environment Variables

When you have upgraded the firmware on all of your processor modules, you can create the Galaxy-specific environment variables as shown in the following example. This example assumes you are configuring a 2-instance, 8 CPU, 1 GB OpenVMS Galaxy computing environment.

```
P00>>> create -nv lp_count        2
P00>>> create -nv lp_cpu_mask0     1
P00>>> create -nv lp_cpu_mask1     fe
P00>>> create -nv lp_io_mask0      100
P00>>> create -nv lp_io_mask1      80
P00>>> create -nv lp_mem_size0     10000000
P00>>> create -nv lp_mem_size1     10000000
P00>>> create -nv lp_shared_mem_size  20000000
P00>>> init
```

Once you create these variables, you can use console SET commands to manipulate them. These variables need only be created on processor 0.

The following descriptions give detailed information about each environment variable.

LP_COUNT

If set to zero, the system will boot a traditional SMP configuration only. Galaxy console mode is OFF.

If set to a nonzero value, the Galaxy features will be used, and the Galaxy variables will be interpreted. The exact value of LP_COUNT represents the number of Galaxy partitions the console should expect. Currently, this number must be 0, 2, or 3.

Note that if you assign resources for three partitions and set this variable to two, the remaining resources will be left unassigned. Unassigned CPUs will be assigned to partition 0. You may also create the variables for the maximum number of partitions ahead of time and simply not assign resources to them (set them to nonzero values) until needed.

LP_CPU_MASK partition number

This bit-mask determines which CPUs are to be initially assigned to the specified Galaxy partition number. The AlphaServer 8400 console chooses the first even-numbered CPU in a partition as its primary CPU, beginning with CPU 0 for the initial instance. Keep this in mind when assigning the resources. (In other words, do not assign only an odd-numbered CPU to a partition.)

LP_IO_MASK partition number

These variables assign I/O modules by slot number to each instance:

- 100 represents the I/O module in slot 8.
- 80 represents the I/O module in slot 7.
- 40 represents the I/O module in slot 6.

These are the only valid assignments for the AlphaServer 8400.

You can assign more than one I/O module to an instance using these masks, but each Galaxy instance requires at least one I/O module.

LP_MEM_SIZE partition number

These variables allocate a specific amount of private memory for the specified instance. It is imperative that you create these variables using proper values for the amount of memory in your system and the desired assignments for each instance. Refer to Table B–1 for common values.

See also the shared memory variable on the following line.

LP_SHARED_MEM_SIZE

This variable allocates memory for use as shared memory. Refer to Appendix B for common values.

Tips

Shared memory must be assigned in multiples of 8 MB and all values are expressed in hexadecimal bytes.

You can define only the amount of shared memory to use, and leave the other LP_MEM_SIZE variables undefined. This will cause the console to allocate the shared memory from the high address space, and to split the remaining memory equally among the number of partitions specified by the LP_COUNT variable. If you also explicitly assign memory to a specific partition using a LP_MEM_SIZE variable, but you leave other partition memory assignments undefined, the console will again assign the memory fragments for shared memory and any partitions with explicit assignments, and will then split and assign the remaining memory to any remaining partitions not having explicit memory assignments.

BOOTDEF_DEV and BOOT_OSFLAGS variables

You should set these variables on each of your Galaxy consoles prior to booting to ensure that AUTOGEN reboots correctly when it needs to reboot the system after an initial installation and after a system failure or operator-requested reboot.

Galaxy Environment Variables Example

```
P00>>> SHOW LP*

lp_count 2
lp_shared_mem_size 20000000   (512 MB)
lp_mem_size0 10000000 (256 MB)
lp_mem_size1 10000000 (256 MB)
lp_cpu_mask0 1 (CPU 0)
lp_cpu_mask1 fe (CPUs 1-7)
lp_io_mask0 100 (I/O module in slot 8)
lp_io_mask1 80 (I/O module in slot 7)

P00>>
```

5.7 Step 7: Start the Secondary Console Devices

If the KFE72-DA was ever configured for Windows NT, it probably expects to find the video board and will hang if one is not present. This is a common occurrence when configuring an OpenVMS Galaxy. A console command can be used to set the mode of operation as follows:

```
P00>>> SET CONSOLE SERIAL
```

When you issue this command to the primary console prior to initializing the secondary consoles, the setting will be propagated to the secondary console hardware.

If you decide to use the Ethernet port, you may need to inform the console of which media type and connection you intend to use: AUI, UDP, or Twisted Pair. The console and operating system will determine which to use, but you can assign a specific media type using the following commands:

```
P00>>> SHOW NETWORK
```

```
P00>>> SET EWA0_MODE TWISTED
```

The first command displays a list of available network devices. The second command establishes the default media type for the specified device (EWA0 in this example). This should be done for all Ethernet devices prior to initializing the secondary consoles.

Once you have set your console mode and network media types (if used) you should reinitialize the system to ensure that the current settings are saved. If you have already defined your Galaxy partitions, you can initialize now. If you have not defined your Galaxy partitions, you should defer initialization until later.

If you are ready to initialize the system, enter:

```
P00>>> INIT
```

You should see the primary console respond with its usual power-up self-test (POST) report. This could take up to 2 minutes. If you have properly defined the Galaxy partitions, only the I/O devices associated with the primary partition will be visible.

To verify that partitioning has occurred, enter:

```
P00>>> SHOW DEVICE
```

or

```
P00>>> SHOW NETWORK
```

To initialize the secondary console, enter:

```
P00>>> LPINIT
```

The console displays the following:

```
Partition 0: Primary CPU = 0
Partition 1: Primary CPU = 2
Partition 0: Memory Base = 000000000   Size = 010000000
Partition 1: Memory Base = 010000000   Size = 010000000
Shared Memory Base = 020000000   Size = 010000000
LP Configuration Tree = 12c000
starting cpu 1 in Partition 1 at address 01000c001
starting cpu 2 in Partition 1 at address 01000c001
starting cpu 3 in Partition 1 at address 01000c001
starting cpu 4 in Partition 1 at address 01000c001
starting cpu 5 in Partition 1 at address 01000c001
starting cpu 6 in Partition 1 at address 01000c001
starting cpu 7 in Partition 1 at address 01000c001

P00>>>
```

This command must be entered from the *primary Galaxy console*. If the Galaxy partitions have been properly defined, and hardware resources have been properly configured, you should see the primary console start the processors assigned to each secondary partition. Each of the secondary consoles should initialize within about 2 minutes.

If one or more consoles fails to initialize, you should double-check your hardware installation, Galaxy partition definitions, and hardware assignments.

For more information about OpenVMS console restrictions and hints, see Chapter 11.

5.8 Step 8: Boot the OpenVMS Galaxy

When you have correctly installed the Galaxy firmware and configured the consoles, you can boot the initial Galaxy environment as follows:

For each Galaxy instance:

```
P00>>> B -FL 0,1 DKA100 // or whatever your boot device is.
```

```
SYSBOOT> SET GALAXY 1
```

```
SYSBOOT> CONTINUE
```

Congratulations! You have created an OpenVMS Galaxy.

6

Creating an OpenVMS Galaxy on an AlphaServer 8200 System

This chapter describes how to create an OpenVMS Galaxy computing environment on an AlphaServer 8200. It focuses on procedures that differ from the AlphaServer 8400 procedures in Chapter 5.

6.1 Step 1: Choose a Configuration and Determine Hardware Requirements

Quick Summary of an AlphaServer 8200 Galaxy Configuration

Only one possible configuration.

- Two instances only
- 5 slots for:
  - Two processor modules (two CPUs each)
  - Two I/O modules
  - One memory module

6.2 Step 2: Set Up Galaxy Hardware

When you have acquired the necessary hardware for your configuration, follow the procedures in Section 5.2.1 through Section 5.2.4 in Chapter 5 and then in this section.

6.2.1 Installing EISA Devices

Plug-in EISA devices can only be configured in partition 0. After installing EISA devices, the console will issue a message requesting that you run the EISA Configuration Utility (ECU).

Run the ECU as follows:

1. Shut down all OpenVMS Galaxy instances.
2. Be sure your floppy disk drive is properly connected to the primary partitions hardware. Typically the drive can be cabled into the Connector Module ("Beeper" part number 54-25133-01) in PCI slot 2.
3. Insert the diskette containing the ECU image.
4. Issue the following commands from the primary console:

```
P08>>> SET ARC_ENABLE ON
P08>>> INITIALIZE
P08>>> RUNECU
```

5. Follow the procedures outlined by the ECU and exit when done.

6. P08>>> boot
7. $ @SYS$SYSTEM:SHUTDOWN
8. P08>>> SET ARC_ENABLE OFF
9. P08>>> INITIALIZE
10. P08>>> LPINIT
11. Reboot the OpenVMS Galaxy.

There are two versions of the ECU, one that runs on a graphics terminal and another that runs on character-cell terminals. Both versions are on the diskette, and the console determines which one to run. For OpenVMS Galaxy systems, the primary console will always be a serial device with a character cell terminal.

If the ECU is not run, OpenVMS will display the following message:

```
   %SYSTEM-I-NOCONFIGDATA, IRQ Configuration data for EISA
slot xxx was not found, please run the ECU and reboot.
```

If you ignore this message, the system will boot, but the plug-in EISA devices will be ignored.

Once you have configured and set up the OpenVMS Galaxy hardware as described in the previous sections, perform the following steps to install and boot OpenVMS Galaxy instances.

6.3 Step 3: Create a System Disk

Decide whether to use a system disk per instance *or* whether to use a cluster common disk.

A new SECURITY.EXE is required for all cluster members running a version prior to OpenVMS Version 7.1-2 that share the same VMS$OBJECTS.DAT file with Galaxy instances.

6.4 Step 4: Install OpenVMS Alpha Version 7.3

No special installation procedures are required to run OpenVMS Galaxy software. Galaxy functionality is included in the base operating system and can be enabled or disabled using the console command and system parameter values described later in this chapter.

For more information about installing the OpenVMS Alpha operating system, see the *OpenVMS Alpha Version 7.2 Upgrade and Installation Manual*.

6.4.1 OpenVMS Galaxy Licensing Information

See the *OpenVMS License Management Utility Manual*.

6.5 Step 5: Upgrade the Firmware

Creating an OpenVMS Galaxy environment on an AlphaServer 8200 requires a firmware upgrade to each processor module. If you use these modules again in a non-Galaxy configuration, you will need to reinstall the previous firmware. It is a good practice to have a current firmware CD on hand.

It saves some time if you install all processor modules you intend to use and update them at the same time. The AlphaServer 8200 requires that you use the same firmware on all processor boards. If you need to upgrade a board at a later time, you must:

1. Remove all boards that are not at the same firmware revision level.
2. Update the older boards.
3. Reinstall the remaining boards.

To upgrade your firmware, the system must be powered on, running in non-Galaxy mode (that is, the LP_COUNT console environment variable—if you have established it—must be set to zero).

To set the console environment variable, use the following commands:

```
P08>>> SET LP_COUNT 0
P08>>> INIT
```

To upgrade the firmware, use the standard console firmware update available from AlphaServers Engineering.

6.6 Step 6: Set Environment Variables

When you have upgraded the firmware on all of your processor modules, you can create the Galaxy-specific environment variables as shown in the following example. This example assumes you are configuring a 2-instance, 4 CPU, 1 GB OpenVMS Galaxy computing environment.

```
P08>>> create -nv lp_count        2
P08>>> create -nv lp_cpu_mask0     100
P08>>> create -nv lp_cpu_mask1     e00
P08>>> create -nv lp_io_mask0      100
P08>>> create -nv lp_io_mask1      80
P08>>> create -nv lp_mem_size0     10000000
P08>>> create -nv lp_mem_size1     10000000
P08>>> create -nv lp_shared_mem_size  20000000
P08>>> init
```

Once these variables have been created, you can use console SET commands to manipulate them. These variables need only be created on processor 0.

The following descriptions give detailed information about each environment variable.

LP_COUNT

If set to zero, the system will boot a traditional SMP configuration only. Galaxy console mode is OFF.

If set to a nonzero value, the Galaxy features will be used, and the Galaxy variables will be interpreted. The exact value of LP_COUNT represents the number of Galaxy partitions the console should expect.

LP_CPU_MASK partition number

This bit-mask determines which CPUs are to be initially assigned to the specified Galaxy partition number. The AlphaServer 8200 console chooses the first even-numbered CPU as its primary CPU, beginning with CPU 08 for the initial instance. Keep this in mind when assigning the resources. (In other words, do not assign only an odd-numbered CPU to a partition.)

LP_IO_MASK partition number

These variables assign IO processors by slot number to each instance:

- 100 represents the I/O module in slot 8.
- 80 represents the I/O module in slot 7.

These are the only valid assignments for the AlphaServer 8200.

LP_MEM_SIZE partition number

These variables allocate a specific amount of private memory for the specified instance. It is imperative that you create these variables using proper values for the amount of memory in your system and the desired assignments for each instance. Refer to Table B–1 for common values.

See also the shared memory variable on the following line.

LP_SHARED_MEM_SIZE

This variable allocates memory for use as shared memory. Refer to Appendix B for common values.

Tips

Shared memory must be assigned in multiples of 8 MB and all values are expressed in hexadecimal bytes.

You can define only the amount of shared memory to use, and leave the other LP_MEM_SIZE variables undefined. This will cause the console to allocate the shared memory from the high address space, and to split the remaining memory equally among the number of partitions specified by the LP_COUNT variable. If you also explicitly assign memory to a specific partition using a LP_MEM_SIZE variable, but leave the other partition memory assignments undefined, the console will again assign the memory fragments for shared memory and any partitions with explicit assignments, then split and assign the remaining memory to any remaining partitions not having explicit memory assignments.

BOOTDEF_DEV and BOOT_OSFLAGS variables

You should set these variables on each of your Galaxy consoles prior to booting to ensure that AUTOGEN reboots correctly when it needs to reboot the system after an initial installation and after a system crash or operator requested reboot.

Galaxy Environment Variables Example

```
P08>>> SHOW LP*

lp_count 2
lp_shared_mem_size 20000000   (512 MB)
lp_mem_size0 10000000 (256 MB)
lp_mem_size1 10000000 (256 MB)
lp_cpu_mask0 100 (CPU 0)
lp_cpu_mask1 e00 (CPUs 1-3)
lp_io_mask0 100 (I/O module in slot 8)
lp_io_mask1 80 (I/O module in slot 7)

P08>>>
```

6.7 Step 7: Start the Secondary Console Device

If the KFE72-DA was ever configured for Windows NT, it probably expects to find the video board and will hang if one is not present. This is a common occurrence when configuring an OpenVMS Galaxy. A console command can be used to set the mode of operation as follows:

```
P08>>> SET CONSOLE SERIAL
```

When you issue this command to the primary console prior to initializing the secondary console, the setting will be propagated to the secondary console hardware.

If you decide to use the Ethernet port, you may need to inform the console of which media type and connection you intend to use: AUI, UDP, or Twisted-Pair. The console and operating system will determine which to use, but you can assign a specific media type using the following commands:

```
P08>>> SHOW NETWORK
```

```
P08>>> SET EWA0_MODE TWISTED
```

The first command displays a list of available network devices. The second command establishes the default media type for the specified device (EWA0 in this example). This should be done for all Ethernet devices prior to initializing the secondary console.

Once you have set your console mode and network media types (if used) you should reinitialize the system to ensure that the current settings are saved. If you have already defined your Galaxy partitions, you can initialize now. If you have not defined your Galaxy partitions, you should defer initialization until later.

If you are ready to initialize the system, enter:

```
P08>>> INIT
```

You should see the primary console respond with its usual power-up self-test (POST) report. This could take up to 2 minutes. If you have properly defined the Galaxy partitions, only the I/O devices associated with the primary partition will be visible.

To verify that partitioning has occurred, enter:

```
P08>>> SHOW DEVICE
```

```
or
```

```
P08>>> SHOW NETWORK
```

To initialize the secondary console, enter:

```
P08>>> LPINIT
```

The console displays the following:

```
Partition 0: Primary CPU = 0
Partition 1: Primary CPU = 2
Partition 0: Memory Base = 000000000   Size = 010000000
Partition 1: Memory Base = 010000000   Size = 010000000
Shared Memory Base = 020000000   Size = 010000000
LP Configuration Tree = 12c000
starting cpu 1 in Partition 1 at address 01000c001
starting cpu 2 in Partition 1 at address 01000c001
starting cpu 3 in Partition 1 at address 01000c001
```

```
P08>>>
```

This command must be entered from the *primary Galaxy console*. If the Galaxy partitions have been properly defined, and hardware resources have been properly configured, you should see the primary console start the processors assigned to the secondary partition. The secondary console should initialize within about 2 minutes.

If one or more consoles fails to initialize, you should double-check your hardware installation, Galaxy partition definitions, and hardware assignments.

For more information about OpenVMS console restrictions and hints, see Chapter 11.

6.8 Step 8: Boot the OpenVMS Galaxy

When you have correctly installed the Galaxy firmware and configured the consoles, you can boot the initial Galaxy environment as follows:

For each Galaxy instance:

```
P08>>> B -FL 0,1 DKA100 // or whatever your boot device is.
SYSBOOT> SET GALAXY 1
SYSBOOT> CONTINUE
```

Congratulations! You have created an OpenVMS Galaxy.

7 Creating an OpenVMS Galaxy on an AlphaServer 4100 System

This chapter describes the requirements and procedures for creating an OpenVMS Galaxy computing environment on an AlphaServer 4100.

7.1 Before You Start

To create an OpenVMS Galaxy on an AlphaServer 4100, you must also be familiar with the following configuration and hardware requirements:

Two-instance maximum

You can run a maximum of two instances of OpenVMS on an AlphaServer 4100.

Console firmware

You must have AlphaServer 4100 console firmware that is available in the OpenVMS Version 7.3 CD-ROM.

Console commands

In addition to the console hints in Chapter 5, you should be aware of the following:

- Enter console commands on one instance at a time.
- Do not enter console commands at another console until the command entered at the first console has completed.

AlphaServer 4100 clock

An AlphaServer 4100 has one clock. For an OpenVMS Galaxy, this means that you cannot run the two instances at different times. Also, the SET TIME command affects both instances. Note that this may not become evident until a number of hours have passed.

Console ports

COM1 (upper) is the console port for instance 0.
COM2 (lower) is the console port for instance 1.

Unlike creating an OpenVMS Galaxy on an AlphaServer 8400, you do not need additional hardware for the second console. COM2 is used for this purpose.

CPUs

CPU0 must be the primary for instance 0.
CPU1 must be the primary for instance 1.
CPUs 2 and 3 are optional secondary CPUs that can be migrated.

I/O adapters

The four lower PCI slots belong to IOD0, which is the I/O adapter for instance 0.
The four upper PCI slots belong to IOD1, which is the I/O adapter for instance 1.

Storage controllers

You will need two storage controllers, such as KZPSAs. These can go to separate Storageworks boxes or to the same box for running as a SCSI cluster. One controller each goes in IOD0 and IOD1.

Network cards

If each instance needs network access, a network card (such as a DE500) is required for each instance.

One card each goes in IOD0 and IOD1.

Physical memory

Because OpenVMS Galaxy on an AlphaServer 4100 does not support memory holes, physical memory for an OpenVMS Galaxy environment must be contiguous. To achieve this on an AlphaServer 4100, one of the following must be true:

- All memory modules must be the same size (for example, 1GB).
- If two sizes are present, only one module can be a smaller size. You must put the larger modules into the lower numbered slots.

To create an OpenVMS Galaxy on an AlphaServer 4100 system, perform the steps in the following sections.

7.2 Step 1: Confirm the AlphaServer 4100 Configuration

Use the SHOW CONFIG command to make sure that the AlphaServer 4100 you are using to create an OpenVMS Galaxy environment meets the requirements described in Section 7.1.

At the console prompt, enter the following command:

```
P00>>>show config
```

The console displays the following information:

```
 Console G53_75  OpenVMS PALcode V1.19-16, Compaq UNIX PALcode V1.21-24

 Module                         Type     Rev    Name
 System Motherboard             0        0000   mthrbrd0
 Memory  512 MB EDO             0        0000   mem0
 Memory  256 MB EDO             0        0000   mem1
 CPU (Uncached)                 0        0000   cpu0
 CPU (Uncached)                 0        0000   cpu1
 Bridge (IOD0/IOD1)             600      0021   iod0/iod1
 PCI Motherboard                8        0000   saddle0
 CPU (Uncached)                 0        0000   cpu2
 CPU (Uncached)                 0        0001   cpu3

 Bus 0  iod0 (PCI0)
 Slot   Option Name             Type     Rev    Name
 1      PCEB                    4828086  0005   pceb0
 4      DEC KZPSA               81011    0000   pks1
 5      DECchip 21040-AA        21011    0023   tulip1

 Bus 1  pceb0 (EISA Bridge connected to iod0, slot 1)
 Slot   Option Name             Type     Rev    Name

 Bus 0  iod1 (PCI1)
 Slot   Option Name             Type     Rev    Name
 1      NCR 53C810              11000    0002   ncr0
 2      DECchip 21040-AA        21011    0024   tulip0
 3      DEC KZPSA               81011    0000   pks0
```

7.3 Step 2: Install OpenVMS Alpha Version 7.3

No special installation procedures are required to run OpenVMS Galaxy software. Galaxy functionality is included in the base operating system and can be enabled or disabled using the console command and system parameter values described later in this chapter.

If your AlphaServer 4100 is not part of a SCSI cluster, you must install OpenVMS Version 7.3 on two system disks—one disk for each instance.

If your AlphaServer 4100 is part of a SCSI cluster with a cluster-common system disk, install OpenVMS Version 7.3 on one system disk.

For more information about installing the OpenVMS Alpha operating system, see the *OpenVMS Alpha Version 7.3 Upgrade and Installation Guide*.

7.4 Step 3: Upgrade the Firmware

To upgrade the firmware, use the Alpha Systems Firmware Update Version 5.4 CD–ROM that is included in the OpenVMS Version 7.3 CD–ROM package. Be sure to read the release notes that are included in the package before installing the firmware.

7.5 Step 4: Set Environment Variables

Configure the primary console for instance 0.

CPU0 is the primary for instance 0.

Create the Galaxy environment variables. For descriptions of the Galaxy environment variables and common values for them, refer to Chapter 5.

The following example is for an AlphaServer 4100 with three CPUs and 512 MB of memory divided into 256 MB + 192 MB + 64 MB.

```
P00>>> create -nv lp_count           2
P00>>> create -nv lp_cpu_mask0       1
P00>>> create -nv lp_cpu_mask1       6
P00>>> create -nv lp_io_mask0        10
P00>>> create -nv lp_io_mask1        20
P00>>> create -nv lp_mem_size0       10000000
P00>>> create -nv lp_mem_size1       c000000
P00>>> create -nv lp_shared_mem_size  4000000
P00>>> set auto_action halt
```

If you have four CPUs and you want to assign all secondary CPUs to instance 1, the LP_CPU_MASK1 variable will be E. If you split the CPUs between both instances, CPU 0 must be the primary for instance 0, and CPU 1 must be the primary CPU for instance 1.

The MEM_SIZE variables depend on your configuration and how you want to split it up.

galaxy_io_mask0 must be set to 10.
galaxy_io_mask1 must be set to 20.

You must set the console environment variable AUTO_ACTION to HALT. This will ensure that the system does not boot and that you will be able to enter the Galaxy command.

7.6 Step 5: Initialize the System and Start the Console Devices

1. Initialize the system and start the Galaxy firmware by entering the following commands:

   ```
   P00>>> init
   P00>>> galaxy
   ```

   After the self-test completes, the Galaxy command will start the console on instance 1.

   The first time that the Galaxy starts, it might display several messages like the following:

   ```
   CPU0 would not join
   ```

   ```
   IOD0 and IOD1 did not pass the power-up self-test
   ```

   This happens because there are two sets of environment variables, and the **galaxy** variables are not present initially on instance 1.

   Note that when the I/O bus is divided between the two Galaxy partitions, the port letter of a device might change. For example, a disk designated as DKC300 when the AlphaServer 4100 is a single system could become DKA300 when it is configured as partition 0 of the OpenVMS Galaxy.

2. Configure the console for instance 1.

   ```
   P01>>> create -nv lp_cpu_mask0       1
   P01>>> create -nv lp_cpu_mask1       6
   P01>>> create -nv lp_io_mask0        10
   P01>>> create -nv lp_io_mask1        20
   P01>>> create -nv lp_mem_size0       10000000
   P01>>> create -nv lp_mem_size1       c000000
   P01>>> create -nv lp_count      2
   P01>>> create -nv lp_shared_mem_size  4000000
   P01>>> set auto_action halt
   ```

3. Initialize the system and restart the Galaxy firmware by entering the following command:

   ```
   P00>>> init
   ```

   When the console displays the following confirmation prompt, type Y:

   ```
   Do you REALLY want to reset the Galaxy (Y/N)
   ```

4. Configure the system root, boot device, and other related variables.

   The following example settings are from an OpenVMS Engineering system. Change these variables to meet the needs of your own environment.

   ```
   P00>>> set boot_osflags 12,0
   P00>>> set bootdef_dev dka0
   P00>>> set boot_reset off            !!! must be OFF !!!
   P00>>> set ewa0_mode twisted

   P01>>> set boot_osflags 11,0
   P01>>> set bootdef_dev dkb200
   P01>>> set boot_reset off            !!! must be OFF !!!
   P01>>> set ewa0_mode twisted
   ```

5. Boot instance 1 as follows:

   ```
   P01>>> boot
   ```

Once instance 1 is booted, log in to the system account and edit the SYS$SYSTEM:MODPARAMS.DAT file to include the following line:

```
GALAXY=1
```

Confirm that the lines for the SCS node and SCS system ID are correct. Run AUTOGEN as follows to configure instance 1 as a Galaxy member, and leave the system halted:

```
$ @SYS$UPDATE:AUTOGEN GETDATA SHUTDOWN INITIAL
```

6. Boot instance 0 as follows:

   ```
   P00>>> boot
   ```

   Once instance 0 is booted, log in to the system account and edit the SYS$SYSTEM:MODPARAMS.DAT file to include the following line:

   ```
   Add the line GALAXY=1
   ```

   Confirm that the lines for the SCS node and SCS system ID are correct. Run AUTOGEN as follows to configure instance 0 as a Galaxy member, and leave the system halted:

   ```
   $ @SYS$UPDATE:AUTOGEN GETDATA SHUTDOWN INITIAL
   ```

7. Prepare the Galaxy to come up automatically upon initialization or power cycle of the system. Set the AUTO_ACTION environment variable on both instances to RESTART.

   ```
   P00>>> set auto_action restart
   ```

   ```
   P01>>> set auto_action restart
   ```

8. Initialize the Galaxy again by entering the following commands at the primary console:

   ```
   P00>>> init
   ```

   When the console displays the following confirmation prompt, type Y:

   ```
   Do you REALLY want to reset the Galaxy (Y/N)
   ```

   Alternatively, you could power-cycle your system, and the Galaxy with both instances should bootstrap automatically.

Congratulations! You have created an OpenVMS Galaxy.

8 Creating an OpenVMS Galaxy on an AlphaServer ES40 System

This chapter describes the requirements and procedures for creating an OpenVMS Galaxy computing environment on an AlphaServer ES40 system.

Note that this chapter contains revised procedures that were originally published in the OpenVMS Alpha VMS721_DS20E_ES40 remedial kit.

To create an OpenVMS Galaxy on an AlphaServer ES40 system:

1. Read the configuration and hardware requirements in Section 8.1.
2. Perform the steps in Section 8.2 through Section 8.6.

8.1 Before You Start

You must be familiar with the following AlphaServer ES40 configuration and hardware requirements:

Two-instance maximum

You can run a maximum of two instances of OpenVMS on an AlphaServer ES40.

Console firmware

To create an OpenVMS Galaxy environment on AlphaServer ES40 systems, you must download the latest version of the V5.6-xx console firmware from the following location:

```
http://ftp.digital.com/pub/DEC/Alpha/firmware/
```

AlphaServer ES40 clock

An AlphaServer ES40 has one clock. For an OpenVMS Galaxy, this means that you cannot run the two instances at different times. Also, the SET TIME command affects both instances. Note that this may not become evident until a number of hours have passed.

Console ports

On a rack-mounted system:
COM1 (lower) is the console port for instance 0.
COM2 (upper) is the console port for instance 1.

On a pedestal system:
COM1 (left) is the console port for instance 0.
COM2 (right) is the console port for instance 1.

Unlike creating an OpenVMS Galaxy on an AlphaServer 8400, you do not need additional hardware for the second console. COM2 is used for this purpose.

CPUs

CPU0 must be the primary for instance 0.
CPU1 must be the primary for instance 1.
CPUs 2 and 3 are optional secondary CPUs that can be migrated.

For an example of the CPU environment variable settings on an AlphaServer ES40, see Section 8.5.

I/O adapters

On a rack-mounted system:
PCI Hose 0 (PCI0) belongs to instance 0 (upper 4 PCI slots)
PCI Hose 1 (PCI1) belongs to instance 1 (lower 6 PCI slots)

On a pedestal system:
PCI Hose 0 (PCI0) belongs to instance 0 (right-hand slots)
PCI Hose 1 (PCI1) belongs to instance 1 (left-hand slots)

Note that PCI0 contains an embedded ISA controller.

To see an I/O adapter configuration example, refer to Section 8.2.

Storage controllers

You will need one storage controller (such as a KZPSA) per instance. For each instance, this can go to a separate StorageWorks box or to the same box for running as a SCSI cluster.

Network cards

If each instance needs network access, a network card (such as a DE600) is required for each instance.

One card each goes in PCI0 and PCI1.

Memory Granularity Restrictions

Private memory must start on a 64 MB boundary.

Shared memory must start on an 8 MB boundary.

Instance 0 must have a multiple of 64 MB.

8.2 Step 1: Confirm the AlphaServer ES40 Configuration

Use the SHOW CONFIG command to make sure that the AlphaServer ES40 you are using to create an OpenVMS Galaxy environment meets the requirements described in Section 8.1.

At the console prompt, enter the following command:

```
P00>>>show config
```

The console displays information similar to the following example:

```
Firmware
SRM Console:    X5.6-2323
ARC Console:    v5.70
PALcode:        OpenVMS PALcode V1.61-2, Tru64 UNIX PALcode V1.54-2
Serial Rom:     V2.2-F
RMC Rom:        V1.0
RMC Flash Rom:  T2.0

Processors
CPU 0           Alpha 21264-4 500 MHz  4MB Bcache
CPU 1           Alpha 21264-4 500 MHz  4MB Bcache
CPU 2           Alpha 21264-4 500 MHz  4MB Bcache
CPU 3           Alpha 21264-4 500 MHz  4MB Bcache
```

```
Core Logic
Cchip           DECchip 21272-CA Rev 9(C4)
Dchip           DECchip 21272-DA Rev 2
Pchip 0         DECchip 21272-EA Rev 2
Pchip 1         DECchip 21272-EA Rev 2
TIG             Rev 10

Memory
  Array       Size       Base Address    Intlv Mode
---------  ----------  ----------------  ----------
    0       4096Mb     0000000000000000    2-Way
    1       4096Mb     0000000100000000    2-Way
    2       1024Mb     0000000200000000    2-Way
    3       1024Mb     0000000240000000    2-Way

     10240 MB of System Memory

 Slot   Option                  Hose 0, Bus 0, PCI
   1    DAPCA-FA ATM622 MMF
   2    DECchip 21152-AA                                Bridge to Bus 2, PCI
   3    DEC PCI FDDI            fwb0.0.0.3.0            00-00-F8-BD-C6-5C
   4    DEC PowerStorm
   7    Acer Labs M1543C                                Bridge to Bus 1, ISA
  15    Acer Labs M1543C IDE    dqa.0.0.15.0
                                dqb.0.1.15.0
                                dqa0.0.0.15.0           TOSHIBA CD-ROM XM-6302B
  19    Acer Labs M1543C USB

        Option                  Hose 0, Bus 1, ISA
        Floppy                  dva0.0.0.1000.0

 Slot   Option                  Hose 0, Bus 2, PCI
   0    NCR 53C875              pkd0.7.0.2000.0         SCSI Bus ID 7
   1    NCR 53C875              pke0.7.0.2001.0         SCSI Bus ID 7
                                dke100.1.0.2001.0       RZ1BB-CS
                                dke200.2.0.2001.0       RZ1BB-CS
                                dke300.3.0.2001.0       RZ1CB-CS
                                dke400.4.0.2001.0       RZ1CB-CS
   2    DE500-AA Network Con    ewa0.0.0.2002.0         00-06-2B-00-0A-58

 Slot   Option                  Hose 1, Bus 0, PCI
   1    NCR 53C895              pka0.7.0.1.1            SCSI Bus ID 7
                                dka100.1.0.1.1          RZ2CA-LA
                                dka300.3.0.1.1          RZ2CA-LA
   2    Fore ATM 155/622 Ada
   3    DEC PCI FDDI            fwa0.0.0.3.1            00-00-F8-45-B2-CE
   4    QLogic ISP10x0          pkb0.7.0.4.1            SCSI Bus ID 7
                                dkb100.1.0.4.1          HSZ50-AX
                                dkb101.1.0.4.1          HSZ50-AX
                                dkb200.2.0.4.1          HSZ50-AX
                                dkb201.2.0.4.1          HSZ50-AX
                                dkb202.2.0.4.1          HSZ50-AX
   5    QLogic ISP10x0          pkc0.7.0.5.1            SCSI Bus ID 7
                                dkc100.1.0.5.1          RZ1CB-CS
                                dkc200.2.0.5.1          RZ1CB-CS
                                dkc300.3.0.5.1          RZ1CB-CS
                                dkc400.4.0.5.1          RZ1CB-CS
   6    DECchip 21154-AA                                Bridge to Bus 2, PCI

 Slot   Option                  Hose 1, Bus 2, PCI
   4    DE602-AA                eia0.0.0.2004.1         00-08-C7-91-0A-AA
   5    DE602-AA                eib0.0.0.2005.1         00-08-C7-91-0A-AB
   6    DE602-TA                eic0.0.0.2006.1         00-08-C7-66-80-9E
   7    DE602-TA                eid0.0.0.2007.1         00-08-C7-66-80-5E
```

8.3 Step 2: Install OpenVMS Alpha Version 7.3

No special installation procedures are required to run OpenVMS Galaxy software. Galaxy functionality is included in the base operating system and can be enabled or disabled using the console command and system parameter values described later in this chapter.

If your AlphaServer ES40 is not part of a SCSI cluster, you must install OpenVMS Version 7.3 on two system disks—one disk for each instance.

If your AlphaServer ES40 is part of a SCSI cluster with a cluster-common system disk, install OpenVMS Version 7.3 on one system disk.

For more information about installing the OpenVMS Alpha operating system, see the *OpenVMS Alpha Version 7.3 Upgrade and Installation Guide*.

8.4 Step 3: Upgrade the Firmware

To upgrade the firmware, use one of the following procedures:

Copy the firmware file to MOM$SYSTEM on a MOP-enabled server that is accessible to the AlphaServer ES40. Enter the following commands on the console:

```
P00>>> boot -fl 0,0 ewa0 -fi {firmware filename}
UPD> update srm*
<power-cycle system>
```

Or, use the following commands:

```
P00>>> BOOT -FLAGS 0,A0 cd_device_name
.
.
.
Bootfile: {firmware filename}
.
.
.
```

8.5 Step 4: Set Environment Variables

Configure the primary console for instance 0.

CPU0 is the primary for instance 0. CPU1 is the primary for instance 1.

The following example is for an AlphaServer ES40 with three CPUs and 512 MB of memory divided into 256 MB + 192 MB + 64 MB.

```
P00>>> set  lp_count           2
P00>>> set  lp_cpu_mask0       1
P00>>> set  lp_cpu_mask1       6
P00>>> set  lp_io_mask0        1
P00>>> set  lp_io_mask1        2
P00>>> set  lp_mem_size0       10000000
P00>>> set  lp_mem_size1       c000000
P00>>> set  lp_shared_mem_size  4000000
P00>>> set  console_memory_allocation new
P00>>> set auto_action halt
```

If you have four CPUs and you want to assign all secondary CPUs to instance 1, the LP_CPU_MASK1 variable will be E. If you split the CPUs between both instances, CPU 0 must be the primary for instance 0, and CPU 1 must be the primary CPU for instance 1.

The following example shows LP_CPU_MASK values for secondary CPU assignments with primary CPUs.

```
Assign secondary CPU 2 with primary CPU 0 and secondary CPU
3 with primary CPU 1.

>>>set lp_cpu_mask0 5
>>>set lp_cpu_mask1 A

CPU Selection                       LP_CPU_MASK

0(primary partition 0)              2^0 =   1
1(primary partition 1)              2^1 =   2
2(secondary)                        2^2 =   4
3(secondary)                        2^3 =   8
```

The **mem_size** variables depend on your configuration and how you want to split it up.

lp_io_mask0 must be set to 1.
lp_io_mask1 must be set to 2.

You must set the console environment variable AUTO_ACTION to HALT. This will ensure that the system does not boot and that you will be able to enter the LPINIT command.

8.6 Step 5: Initialize the System and Start the Console Devices

1. Initialize the system and start the Galaxy firmware by entering the following commands:

   ```
   P00>>> init      ! initialize the system
   P00>>> lpinit    ! start firmware
   ```

   After the self-test completes, the Galaxy command will start the console on instance 1.

   Note that when the I/O bus is divided between the two Galaxy partitions, the port letter of a device might change. For example, a disk designated as DKC300 when the AlphaServer ES40 is a single system could become DKA300 when it is configured as partition 0 of the OpenVMS Galaxy.

2. Configure the console for instance 1.
3. Configure the system root, boot device, and other related variables.

   The following example settings are from an OpenVMS Engineering system. Change these variables to meet the needs of your own environment.

   ```
         Instance 0
   P00>>> set boot_osflags 12,0
   P00>>> set bootdef_dev dka0
   P00>>> set boot_reset off           !!! must be OFF !!!
   P00>>> set ewa0_mode twisted

         Instance 1
   P01>>> set boot_osflags 11,0
   P01>>> set bootdef_dev dkb200
   P01>>> set boot_reset off           !!! must be OFF !!!
   P01>>> set ewa0_mode twisted
   ```

4. Boot instance 1 as follows:

   ```
   P01>>> boot
   ```

Once instance 1 is booted, log in to the system account and edit the SYS$SYSTEM:MODPARAMS.DAT file to include the following line:

```
GALAXY=1
```

Confirm that the SCSNODE and SCSSYSTEMID SYSGEN parameters are correct. Run AUTOGEN as follows to configure instance 1 as a Galaxy member, and leave the system halted:

```
$ @SYS$UPDATE:AUTOGEN GETDATA SHUTDOWN INITIAL
```

5. Boot instance 0 as follows:

   ```
   P00>>> boot
   ```

   Once instance 0 is booted, log in to the system account and edit the SYS$SYSTEM:MODPARAMS.DAT file to include the following line:

   ```
   GALAXY=1
   ```

   Confirm that the SCSNODE and SCSSYSTEMID SYSGEN parameters are correct. Run AUTOGEN as follows to configure instance 0 as a Galaxy member, and leave the system halted:

   ```
   $ @SYS$UPDATE:AUTOGEN GETDATA SHUTDOWN INITIAL
   ```

6. Prepare the Galaxy to come up automatically upon initialization or power cycle of the system. Set the AUTO_ACTION environment variable on both instances to RESTART.

   ```
   P00>>> set auto_action restart
   ```

   ```
   P01>>> set auto_action restart
   ```

7. Initialize the Galaxy again by entering the following commands at the primary console:

   ```
   P00>>> init
   ```

   When the console displays the following confirmation prompt, type Y:

   ```
   Do you REALLY want to reset all partitions? (Y/N)
   ```

   Alternatively, you could power-cycle your system, and the Galaxy with both instances should bootstrap automatically.

Congratulations! You have created an OpenVMS Galaxy on an AlphaServer ES40 system.

9 Creating an OpenVMS Galaxy on AlphaServer GS80/160/320 Systems

This chapter describes how to create an OpenVMS Galaxy computing environment on AlphaServer GS80/160/320 systems.

9.1 Step 1: Choose a Configuration and Determine Hardware Requirements

OpenVMS Alpha Version 7.3 supports the following maximum configuration on AlphaServer GS160 systems:

4 instances
4 QBBs
16 CPUs
128 GB memory

Rules:

Must have standard COM1 UART console line for each partition
Must have PCI drawer for each partition
Must have an I/O module per partition
Must have at least one CPU module per partition
Must have at least one memory module per partition.

9.2 Step 2: Set Up Hardware

When you have acquired the necessary hardware for your configuration, follow the procedures in the appropriate hardware manauals to assemble it.

9.3 Step 3: Create a System Disk

Decide whether to use a system disk per instance *or* whether to use a cluster common-disk.

A new SECURITY.EXE file is required for all cluster members running a version prior to OpenVMS Version 7.1-2 that share the same VMS$OBJECTS.DAT file with Galaxy instances.

9.4 Step 4: Install OpenVMS Alpha Version 7.3

No special installation procedures are required to run OpenVMS Galaxy software. Galaxy functionality is included in the base operating system and can be enabled or disabled using the console command and system parameter values described later in this chapter.

For more information about installing the OpenVMS Alpha operating system, see the *OpenVMS Alpha Version 7.3 Upgrade and Installation Manual*.

9.4.1 OpenVMS Galaxy Licensing Information

In a Galaxy environment, the OPENVMS-GALAXY license units are checked during system startup and whenever a CPU reassignment between instances occurs.

If you attempt to start a CPU and there are insufficient OPENVMS-GALAXY license units to support it, the CPU will remain in the instance's configured set but it will be stopped. You can subsequently load the appropriate license units and start the stopped CPU while the system is running. This is true of one or more CPUs.

9.5 Step 5: Set Environment Variables

When you have installed the operating system, you can set the Galaxy-specific environment variables as shown in the examples in this section.

9.5.1 AlphaServer GS160 Example

This example for an AlphaServer GS160 assumes you are configuring an OpenVMS Galaxy computing environment with:

4 instances
4 QBBs
16 CPUs
32 GB of memory

```
P00>>>show lp*

lp_count            4
lp_cpu_mask0        000F
lp_cpu_mask1        00F0
lp_cpu_mask2        0F00
lp_cpu_mask3        F000
lp_cpu_mask4        0
lp_cpu_mask5        0
lp_cpu_mask6        0
lp_cpu_mask7        0
lp_error_target     0
lp_io_mask0         1
lp_io_mask1         2
lp_io_mask2         4
lp_io_mask3         8
lp_io_mask4         0
lp_io_mask5         0
lp_io_mask6         0
lp_io_mask7         0
lp_mem_size0        0=4gb
lp_mem_size1        1=4gb
lp_mem_size2        2=4gb
lp_mem_size3        3=4gb
lp_mem_size4        0
lp_mem_size5        0
lp_mem_size6        0
lp_mem_size7        0
lp_shared_mem_size  16gb

P00>>>lpinit
```

9.5.2 AlphaServer GS320 Example

This example for an AlphaServer GS320 system assumes you are configuring an OpenVMS Galaxy computing environment with:

4 instances
8 QBBs
32 CPUs
32 GB memory

```
P00>>>show lp*

lp_count            4
lp_cpu_mask0        000F000F
lp_cpu_mask1        00F000F0
lp_cpu_mask2        0F000F00
lp_cpu_mask3        F000F000
lp_cpu_mask4        0
lp_cpu_mask5        0
lp_cpu_mask6        0
lp_cpu_mask7        0
lp_error_target     0
lp_io_mask0         11
lp_io_mask1         22
lp_io_mask2         44
lp_io_mask3         88
lp_io_mask4         0
lp_io_mask5         0
lp_io_mask6         0
lp_io_mask7         0
lp_mem_size0        0=2gb, 4=2gb
lp_mem_size1        1=2gb, 5=2gb
lp_mem_size2        2=2gb, 6=2gb
lp_mem_size3        3=2gb, 7=2gb
lp_mem_size4        0
lp_mem_size5        0
lp_mem_size6        0
lp_mem_size7        0
lp_shared_mem_size  16gb

P00>>>lpinit
```

9.5.3 Environment Variable Descriptions

This section describes each environment variable. For more details about using these variables, refer to the *AlphaServer GS80/160/320 Firmware Reference Manual*.

LP_COUNT *number*

If set to zero, the system will boot a traditional SMP configuration only. Galaxy console mode is OFF.

If set to a nonzero value, the Galaxy features will be used, and the Galaxy variables will be interpreted. The exact value of LP_COUNT represents the number of Galaxy partitions the console creates.

Note that if you assign resources for three partitions and set LP_COUNT to two, the remaining resources will be left unassigned.

LP_CPU_MASK*n mask*

This bitmask determines which CPUs are to be initially assigned to the specified Galaxy partition number. The AlphaServer GS160 console chooses the first CPU that passes the self test in a partition as its primary CPU.

LP_ERROR_TARGET

The new Alphaserver GS series introduces a new Galaxy environment variable called LP_ERROR_TARGET. The value of the variable is the number of the Galaxy instance that system errors will initially be reported to. Unlike other Galaxy platforms, all system correctable, uncorrectable and system event errors will go to a single instance. It is possible for the operating system to change this target, so the value of the variable represents the target when the system is first partitioned.

Every effort is made to isolate system errors to a single instance so that the error does not bring down the entire Galaxy. The error target instance will determine, on receipt of an error, if it is safe to remotely crash the single instance that incurred the error. A bugcheck code of GLXRMTMCHK is used in this case. Note that error log information pertaining to the error will be on the error target instance, not necessarily on the instance that incurred the error.

While every effort is made to keep the error target instance identical to the one the user designated with the environment variable, the software monitors the instances and will change the error target if necessary.

LP_IO_MASK*n mask*

These variables assign the I/O modules by QBB number to each instance:

Mask Value	QBB Number
1	QBB 0
2	QBB 1
4	QBB 2
8	QBB 3

For the *n*, supply the partition number (0 - 7). The value *mask* gives a binary mask indicating which QBB's (containing I/O risers) are included in the partition.

LP_MEM_SIZE*n size*

These variables allocate a specific amount of private memory for the specified instance. It is imperative that you create these variables using proper values for the amount of memory in your system and the desired assignments for each instance.

You can define only the amount of shared memory to use, and leave the other LP_MEM_SIZE variables undefined. This will cause the console to allocate the shared memory from the high address space, and split the remaining memory equally among the number of partitions specified by the LP_COUNT variable. If you also explicitly assign memory to a specific partition using a LP_MEM_SIZE variable, but leave other partition memory assignments undefined, the console will again assign the memory fragments for shared memory and any partitions with explicit assignments, then split and assign the remaining memory to any remaining partitions not having explicit memory assignments.

For example:

```
lp_mem_size0  0=2gb, 1=2gb
```

Note

Do not assign private memory to an instance from a QBB that has no CPUs in the instance.

For example, if LP_CPU_MASK0 is FF, then you should only assign private memory for instance 0 from QBBs 0 and 1.

Refer to see *AlphaServer GS80/160/320 Firmware Reference Manual* for more details about using this variable.

LP_SHARED_MEM_SIZE *size*

This variable allocates memory for use as shared memory. For example:

```
lp_shared_mem_size      16gb
```

Note that shared memory must be assigned in multiples of 8 MB.

Refer to the *AlphaServer GS80/160/320 Firmware Reference Manual* for more details about using this variable.

BOOTDEF_DEV and BOOT_OSFLAGS variables

You should set these variables on each of your Galaxy consoles prior to booting to ensure that AUTOGEN reboots correctly when it needs to reboot the system after an initial installation and after a system failure or operator-requested reboot.

9.6 Step 6: Start the Secondary Console Devices

If you decide to use the Ethernet port, you may need to inform the console which media type and connection you intend to use: AUI, UDP, or Twisted Pair. The console and operating system will determine which to use, but you can assign a specific media type using the following commands:

```
P00>>> SHOW NETWORK
P00>>> SET EWA0_MODE TWISTED
```

The first command displays a list of available network devices. The second command establishes the default media type for the specified device (EWA0 in this example). This should be done for all Ethernet devices prior to initializing the secondary consoles.

9.7 Step 7: Initialize the Secondary Consoles

Once you have established the Galaxy variables, to initialize the secondary consoles, enter:

```
P00>>> LPINIT
```

The console displays the following:

```
P00>>>lpinit
lp_count = 2
lp_mem_size0 = 1800 (6 GB)
CPU 0 chosen as primary CPU for partition 0
lp_mem_size1 = 1800 (6 GB)
CPU 4 chosen as primary CPU for partition 1
lp_shared_mem_size = 1000 (4 GB)
initializing shared memory
partitioning system
QBB 0 PCA 0 Target 0 Interrupt Count = 2
QBB 0 PCA 0 Target 0 Interrupt CPU = 0
Interrupt Enable = 000011110000d05a
Sent Interrupts = 0000100000000010
Enabled Sent Interrupts = 0000100000000010
Acknowledging Sent Interrupt 0000000000000010 for CPU 0
QBB 0 PCA 0 Target 0 Interrupt Count = 1
QBB 0 PCA 0 Target 0 Interrupt CPU = 0
Interrupt Enable = 000011110000d05a
Sent Interrupts = 0000100000000000 Enabled Sent Interrupts = 0000100000000000
Acknowledging Sent Interrupt 0000100000000000 for CPU 0

OpenVMS PALcode V1.80-1, Tru64 UNIX PALcode V1.74-1

system = QBB 0 1 2 3        + HS                            (Hard Partition 0)
 QBB 0 = CPU 0 1 2 3 + Mem 0      + Dir + IOP + PCA 0 1     + GP  (Hard QBB 0)
 QBB 1 = CPU 0 1 2 3 + Mem 0      + Dir + IOP + PCA 0 1     + GP  (Hard QBB 1)
 QBB 2 = CPU 0 1 2 3 + Mem 0      + Dir + IOP + PCA         + GP  (Hard QBB 4)
 QBB 3 = CPU 0 1 2 3 + Mem 0      + Dir + IOP + PCA         + GP  (Hard QBB 5)
partition 0
 CPU 0 1 2 3 8 9 10 11
 IOP 0 2
 private memory size is 6 GB
 shared memory size is 4 GB
micro firmware version is T5.4
shared RAM version is 1.4
hose 0 has a standard I/O module
starting console on CPU 0
QBB 0 memory, 4 GB
QBB 1 memory, 4 GB
QBB 2 memory, 4 GB
QBB 3 memory, 4 GB
total memory, 16 GB
probing hose 0, PCI
probing PCI-to-ISA bridge, bus 1
bus 1, slot 0 -- dva -- Floppy
bus 0, slot 1 -- pka -- QLogic ISP10x0
bus 0, slot 2 -- pkb -- QLogic ISP10x0
bus 0, slot 3 -- ewa -- DE500-BA Network Controller
bus 0, slot 15 -- dqa -- Acer Labs M1543C IDE
probing hose 1, PCI
probing hose 2, PCI
bus 0, slot 1 -- fwa -- DEC PCI FDDI
probing hose 3, PCI
starting console on CPU 1
starting console on CPU 2
starting console on CPU 3
starting console on CPU 8
starting console on CPU 9
starting console on CPU 10
starting console on CPU 11
initializing GCT/FRU at 1fa000
initializing pka pkb ewa fwa dqa
Testing the System
Testing the Disks (read only)
Testing the Network
AlphaServer Console X5.8-2842, built on Apr  6 2000 at 01:43:42
P00>>>
```

This command must be entered from the *primary Galaxy console*. If the Galaxy partitions have been properly defined, and hardware resources have been properly configured, you should see that the primary CPU in each instance has started.

If one or more consoles fails to initialize, you should double-check your hardware installation, Galaxy partition definitions, and hardware assignments.

9.8 Step 8: Boot the OpenVMS Galaxy

When you have correctly installed the Galaxy firmware and configured the consoles, you can boot the initial Galaxy environment as follows:

For each Galaxy instance:

```
P00>>> B -FL 0,1 DKA100 // or whatever your boot device is.
SYSBOOT> SET GALAXY 1
SYSBOOT> CONTINUE
```

Congratulations! You have created an OpenVMS Galaxy.

10 Using a Single-Instance Galaxy on Any Alpha System

With OpenVMS Alpha Version 7.3, you can run a single-instance Galaxy on any Alpha platform. This capability allows early adopters to evaluate OpenVMS Galaxy features and, most important, to develop and test Galaxy-aware applications without incurring the expense of setting up a full-scale Galaxy computing environment on a system capable of running multiple instances of OpenVMS (for example, an AlphaServer 8400).

A single-instance Galaxy running on any Alpha system is not an emulator. It is OpenVMS Galaxy code with Galaxy interfaces and underlying operating system functions. All Galaxy APIs are present in a single-instance Galaxy (for example, resource management, shared memory access, event notification, locking for synchronization, and shared memory for global sections).

Any application that is run on a single-instance Galaxy will exercise the identical operating system code on a multiple-instance Galaxy system. This is accomplished by creating the configuration file SYS$SYSTEM:GLX$GCT.BIN, which OpenVMS reads into memory. On a Galaxy platform (for example, an AlphaServer 8400), the console places configuration data in memory for OpenVMS to use. Once the configuration data is in memory, regardless of its origin, OpenVMS boots as a Galaxy instance.

To use the Galaxy Configuration Utility (GCU) to create a single-instance Galaxy on any Alpha system, use the following procedure:

Run the GCU on the OpenVMS Alpha system on which you want to use the single-instance Galaxy.

If the GCU is run on a non-Galaxy system, it will prompt as to whether you want to create a single-instance Galaxy. Click on **OK.**

The GCU next prompts for the amount of memory to designate as shared memory. Enter any value that is a multiple of 8 MB. Note that you must specify at least 8 MB of shared memory if you want to boot as a Galaxy instance.

When the GCU has displayed the configuration, it will already have written the file GLX$GCT.BIN to the current directory. You can exit the GCU at this point. If you made a mistake or want to alter the configuration, you can close the current model and repeat the process.

To reboot the system as a Galaxy instance:

1. Copy the GLX$GCT.BIN file to SYS$SYSROOT:[SYSEXE]GLX$GCT.BIN.
2. Shut down the system.
3. Reboot with a conversational boot command (>>> B -FL 0,1 device).
4. SYSBOOT> SET GALAXY 1

5. SYSBOOT> CONTINUE
6. Add GALAXY=1 to SYS$SYSTEM:MODPARAMS.DAT.

11 OpenVMS Galaxy Tips and Techniques

This chapter contains information that OpenVMS Engineering has found useful in creating and running OpenVMS Galaxy environments.

11.1 System Auto-Action

Upon system power-up, if the AUTO_ACTION console environment variable is set to BOOT or RESTART for instance 0, then the GALAXY command will automatically be issued and instance 0 will attempt to boot.

The setting of AUTO_ACTION in the console environment variables for the other instances will dictate their behavior upon the issuing of the GALAXY comamnd (whether it is issued automatically or by the user from the console).

To setup your system for this feature, you must set the console environment variable AUTO_ACTION to RESTART or BOOT on each instance, and be sure to specify appropriate values for the BOOT_OSFLAGS and BOOTDEF_DEV environment variables for each instance.

11.2 Changing Console Environment variables

Once you have established the initial set of **LP_*** environment variables for OpenVMS Galaxy operation and booted your system, changing environment variable values requires that you first reinitialize the system, change the values, and reinitialize again. Wrapping the changes between INIT commands is required to properly propagate the new values to all partitions.

Note

For AlphaServer 4100 systems no INIT command is needed to start, but you must change these variables on both instances.

11.3 Console Hints

Because AlphaServer 8400 and 8200 systems were designed prior to the Galaxy Software Architecture, OpenVMS Galaxy console firmware and system operations must handle a few restrictions.

The following list briefly describes some things you should be aware of and some things you should avoid doing:

- Do not set the **BOOT_RESET** environment variable to 1. This causes each secondary console to reset the bus before booting, thus resetting all previously booted partitions. Remember that OpenVMS Galaxy partitions share the hardware.
- Be patient. Console initialization and system rebooting can take several minutes.

- **Do not attempt to abort a firmware update process!**

  This can leave your system seriously hung.
- When updating console firmware, update *all CPUs* at the same time.

  You cannot run two different types of CPUs or two different firmware revisions. If you fail to provide consistent firmware revisions, the system will hang on power-up.
- Never issue the GALAXY command from a secondary console. This will reinitialize the system, and you will need to start over from the primary console.

11.4 Turning Off Galaxy Mode

If you want to turn off OpenVMS Galaxy software, change the lp_count environment variable as follows and enter the following commands:

```
>>> SET LP_COUNT 0   ! Return to monolithic SMP config
>>> INIT                      ! Return to single SMP console
>>> B -fl 0,1 device          ! Stop at SYSBOOT
SYSBOOT> SET GALAXY 0
SYSBOOT> CONTINUE
```

12 OpenVMS Galaxy Configuration Utility

The Galaxy Configuration Utility (GCU) is a DECwindows Motif application that allows system managers to configure and manage an OpenVMS Galaxy system from a single workstation window.

Using the GCU, system managers can:

- Display the active Galaxy configuration.
- Reassign resources among Galaxy instances.
- View resource-specific characteristics.
- Shut down or reboot one or more Galaxy instances.
- Invoke additional management tools.
- Create and engage Galaxy configuration models.
- Create a single-instance Galaxy on any Alpha system (for software development on non-Galaxy hardware platforms).
- View the online Galaxy documentation.
- Determine hot-swap characteristics of the current hardware platform.

The GCU resides in the SYS$SYSTEM directory along with a small number of files containing configuration knowledge.

The GCU consists of the following files:

SYS$SYSTEM:GCU.EXE	GCU executable image
SYS$MANAGER:GCU.DAT	Optional DECwindows resource file
SYS$MANAGER:GALAXY.GCR	Galaxy Configuration Ruleset
SYS$MANAGER:GCU$ACTIONS.COM	System management procedures
SYS$MANAGER:*xxx*.GCM	User-defined configuration models
SYS$HELP:GALAXY_GUIDE.DECW$BOOK	Online help in Bookreader form

The GCU can be run from any Galaxy instance. If the system does not directly support graphics output, then the DECwindows display can be set to an external workstation or suitably configured PC. However, the GCU application itself must always run on the Galaxy system.

When the GCU is started, it loads any customizations found in its resource file (GCU.DAT); then it loads the Galaxy Configuration Ruleset (GALAXY.GCR). The ruleset file contains statements that determine the way the GCU displays the various system components, and includes rules that govern the ways in which users can interact with the configuration display. Users do not typically alter the ruleset file unless they are well versed in its structure or are directed to do so by a Compaq Services Engineer. After the GCU display becomes visible, the GCU determines whether the system is currently configured as an OpenVMS Galaxy or as a single-instance Galaxy on a non-Galaxy platform. If the system is configured as a Galaxy, the GCU displays the active Galaxy configuration model. The main observation window displays a hierarchical view of the Galaxy. If the system has not yet been configured as a Galaxy, the GCU prompts you as to whether or not to create a single-instance Galaxy. Note that the GCU can create a single-instance Galaxy on any Alpha system, but multiple-instance OpenVMS Galaxy environments are created by using console commands and console environment variables.

Once the Galaxy configuration model is displayed, users can either interact with the active model or take the model off line and define specific configurations for later use. The following sections discuss these functions in greater detail.

12.1 GCU Tour

The GCU can perform three types of operations:

- Create Galaxy configuration models or a single-instance Galaxy.
- Observe active Galaxy resources.
- Interact with active Galaxy resource configurations.

Most GCU operations are organized around the main observation window and its hierarchical display of Galaxy components. The observation window provides a porthole into a very large space. The observation window can be panned and zoomed as needed to observe part of or all of the entire Galaxy configuration. The main toolbar contains a set of buttons that control workspace zoom operations. Workspace panning is controlled by the horizontal and vertical scrollbars; workspace sliding is achieved by holding down the middle mouse button as you drag the workspace around. This obviously assumes you have a three-button mouse.

The various GCU operations are invoked from pull-down or pop-up menu functions. General operations such as opening and closing files, and invoking external tools, are accomplished using the main menu bar entries. Operations specific to individual Galaxy components are accomplished using pop-up menus that appear whenever you click the right mouse button on a component displayed in the observation window.

In response to many operations, the GCU displays additional dialog boxes containing information, forms, editors, or prompts. Error and information responses are displayed in pop-up dialog boxes or inside the status bar along the bottom of the window, depending on the severity of the error and importance of the message.

12.1.1 Creating Galaxy Configuration Models

You can use the GCU to create Galaxy configuration models and a single-instance Galaxy on any Alpha system.

When viewing the active Galaxy configuration model, direct manipulation of display objects (components) may alter the running configuration. For example, dragging a CPU from its current location and dropping it on top of a different instance component will invoke a management action procedure that reassigns the selected CPU to the new instance. At certain times this may be a desirable operation; however, in other situations you might want to reconfigure your Galaxy all at once rather than component by component. To accomplish this, you must create an offline Galaxy configuration model.

To create a Galaxy configuration model, we must start with an existing model, typically the active one, alter it in some manner, and save it in a file.

Starting from the active Galaxy Configuration Model:

1. Press the ENGAGE button such that the model becomes DISENGAGED. The button should turn from red to white, and its appearance should be popped outward. When disengaged, all CPU components in the display will turn red as an indication that they are no longer engaged. Do not panic, they have not been shut down!
2. Alter the CPU assignments by dragging and dropping individual CPUs onto the instances on which you want to assign them.
3. When finished, you can either reengage the model, or save the model in a file for later use. Whenever you reengage a model, regardless of whether the model was derived from the active model or from a file-based model, the GCU will compare the active system configuration with the configuration proposed by the model. It will then provide a summary of management actions that would need to be performed to reassign the system to the new model. If the user approves of the actions, the GCU will commence with execution of the required management actions and the resulting model will be displayed as the active and engaged model.

The reason for creating offline models is to allow significant configuration changes to be automated. For example, you can create models representing the desired Galaxy configuration at different times and then engage the models interactively by following this procedure.

12.1.2 Observation

The GCU can display the single active Galaxy configuration model, or any number of offline Galaxy configuration models. Each loaded model appears as an item in the Model menu on the toolbar. You can switch between models by clicking the desired menu item.

The active model is always named GLX$ACTIVE.GCM. When the active model is first loaded, a file by this name will exist briefly as the system verifies the model with the system hardware.

When a model is visible, you can zoom, pan, or slide the display as needed to view Galaxy components. Use the buttons on the left side of the toolbar to control the zoom functions.

The zoom functions include:

Galactic zoom	Zoom to fit the entire component hierarchy into observation window.
Zoom 1:1	Zoom to the component normal scale.
Zoom to region	Zoom to a selected region of the display.
Zoom in	Zoom in by 10 percent.
Zoom out	Zoom out by 10 percent.

Panning is accomplished by using the vertical and horizontal scrollbars. Sliding is done by pressing and holding the middle mouse button and dragging (sliding) the cursor and the image.

12.1.2.1 Layout Management

The Automatic Layout feature manages the component layout. If you ever need to refresh the layout while in Automatic Layout mode, simply select the root (topmost) component.

To alter the current layout, select Manual Layout from the Windows menu. In Manual Layout Mode, you can freely drag and drop components however you like to generate a pleasing structure. Because each component is free from automatic layout constraints, you may need to invest some time in positioning each component, possibly on each of the charts. To make things simpler, you can click the right mouse button on any component and select Layout Subtree to provide automatic layout assistance below that point in the hierarchy.

When you are satisfied with the layout, you must save the current model in a file to retain the manual layout information. The custom layout is used when the model is open. Note that if you select Auto Layout mode, your manual layout will be lost for the in-memory model. Also, in order for CPU components to reassign in a visually effective manner, they must perform subtree layout operations below the instance level. For this reason, it is best to limit any manual layout operations to the instance and community levels of the component hierarchy.

12.1.2.2 OpenVMS Galaxy Charts

The GCU provides six distinct subsets of the model, known as charts.

The six charts include:

Chart Name	Shows
Logical Structure	Dynamic resource assignments
Physical Structure	Nonvolatile hardware relationships
CPU Assignment	Simplified view of CPU assignments
Memory Assignment	Memory subsystem components
IOP Assignment	I/O module relationships
Failover Targets	Processor failover assignments

These charts result from enabling or disabling the display of various component types to provide views of sensible subsets of components.

Specific charts may offer functionality that can be provided only for that chart type. For example, reassignment of CPUs requires that the instance components be visible. Because instances are not visible in the Physical Structure or Memory Assignment charts, you can reassign CPUs only in the Logical Structure and CPU Assignment charts.

For more information about charts, refer to Section 12.4.

12.1.3 Interaction

When viewing the active Galaxy configuration model, you can interact directly with the system components. For example, to reassign a CPU from one instance to another, you can drag and drop a CPU onto the desired instance. The GCU will validate the operation and execute an external command action to make the configuration change. Interacting with a model that is not engaged, is simply a drawing operation on the offline model, and has no impact to the running system.

While interacting with Galaxy components, the GCU applies built-in and user-defined rules that prevent misconfiguration and improper management actions. For example, you cannot reassign primary CPUs, and you cannot reassign a CPU to any component other than a Galaxy instance. Either operation would result in an error message on the status bar, and the model would return to its proper configuration. If the attempted operation violates one of the configuration rules, the error message, displayed in red on the status bar, will describe the rule that fired.

You can view details for any selected component by clicking the right mouse button and either selecting the Parameters item from the pop-up menu or by selecting Parameters from the Components menu on the main toolbar.

The GCU can shut down or reboot one or more Galaxy instances using the Shutdown or Reboot items on the Galaxy menu. The various shutdown or reboot parameters can be entered in the Shutdown dialog box. Be sure to specify the CLUSTER_SHUTDOWN option to fully shut down clustered Galaxy instances. The Shutdown dialog box allows you to select any combination of instances, or all instances. The GCU is "smart" enough to shut down its owner instance last.

12.2 Managing an OpenVMS Galaxy with the GCU

Your ability to manage a Galaxy system using the Galaxy Configuration Utility (GCU) depends on the capabilities of each instance involved in a management operation.

The GCU can be run from any instance in the Galaxy. However, the Galaxy Software Architecture implements a push-model for resource reassignment. This means that, in order to reassign a processor, you must execute the reassign command function on the instance that currently owns the processor. The GCU is aware of this requirement, and will attempt to use one or more communications paths to send the reassignment request to the owner instance. DCL is not inherently aware of this requirement; therefore, if you use DCL to reassign resources, you will need to use SYSMAN or a separately logged-in terminal to issue the commands on the owner instance.

The GCU favors using SYSMAN, and its underlying SMI_Server processes to provide command paths to the other instances in the Galaxy. However, the SMI_Server requires that the instances be in a cluster so that the command environment falls within a common security domain. However, Galaxy instances might not be clustered.

If the system cannot provide a suitable command path for the SMI_Server to use, the GCU will attempt to use DECnet task-to-task communications. This requires that the participating instances be running DECnet, and that each participating Galaxy instance have a proxy set up for the SYSTEM account.

12.2.1 Independent Instances

You can define a Galaxy system so that one or more instances are not members of the Galaxy sharing community. These are known as **independent instances**, and they are visible to the GCU.

These independent instances can still participate in CPU reassignment. They cannot utilize shared memory or related services.

12.2.2 Isolated Instances

It is possible for an instance to not be clustered, have no proxy account established, and not have DECnet capability. These are known as **isolated instances**. They are visible to the GCU, and you can reassign CPUs to them. The only way to reassign resources from an isolated instance is from the console of the isolated instance.

12.2.3 Required PROXY Access

When the GCU needs to execute a management action, it always attempts to use the SYSMAN utility first. SYSMAN requires that the involved instances be in the same cluster. If this is not the case, the GCU will next attempt to use DECnet task-to-task communications. For this to work, the involved instances must each have an Ethernet device, DECnet capability, and suitable proxy access on the target instance.

For example, consider a two-instance configuration that is not clustered. If instance 0 were running the GCU and the user attempts to reassign a CPU from instance 1 to instance 0, the actual reassignment command must be executed on instance 1. To do this, the GCU's action procedures in the file SYS$MANAGER:GCU$ACTIONS.COM will attempt to establish a DECnet task-to-task connection to the SYSTEM account on instance 1. This requires that instance 1 has granted proxy access to the SYSTEM account of instance 0. Using the established connection, the action procedure on instance 0 will pass its parameters to the equivalent action procedure on instance 1, which now treats the operation as a local operation.

The GCU action procedures assume that they will be used by the system manager. Thus, in the action procedure file SYS$MANAGER:GCU$ACTIONS.COM, the SYSTEM account is used. To grant access to the opposite instances SYSTEM account, the proxy must be set up on instance 1.

To establish proxy access:

1. Enter the following commands at the DCL prompt:

   ```
   $ SET DEFAULT SYS$SYSTEM
   $ RUN AUTHORIZE
   ```

2. If proxy processing is not yet enabled, enable it by entering the following commands:

   ```
   UAF> CREATE/PROXY
   UAF> ADD/PROXY instance::SYSTEM SYSTEM
   UAF> EXIT
   ```

Replace *instance* with the name of the instance to which you are granting access. Perform these steps for each of the instances you want to manage from the instance on which you run the GCU. For example, in a typical two-instance Galaxy, if you run the GCU only on instance 0, then you need to add proxy access only on instance 1 for instance 0. If you intend to run the GCU on instance 1 also, then you need to add proxy access on instance 0 for instance 1. In three-instance Galaxy systems, you may need to add proxy access for each combination of instances you want to control. For this reason, a good rule of thumb is to always run the GCU from instance 0.

You are not required to use the SYSTEM account. To change the account, you need to edit SYS$MANAGER:GCU$ACTIONS.COM on each involved instance. Locate the line that establishes the task-to-task connection, and replace the SYSTEM account name with one of your choosing.

Note that the selected account must have OPER, SYSPRV, and CMKRNL privileges. You also need to add the necessary proxy access to your instances for this account.

12.3 Galaxy Configuration Models

The GCU is a fully programmable display engine. It uses a set of rules to learn the desired characteristics and interactive behaviors of the system components. Using this specialized configuration knowledge, the GCU assembles models that represent the relationships among system components. The GCU obtains information about the current system structure by parsing a configuration structure built by the console firmware. This structure, called the Galaxy Configuration File, is stored in memory and is updated as needed by firmware and by OpenVMS executive routines to ensure that it accurately reflects the current system configuration and state.

The GCU converts and extends the binary representation of the configuration file into a simple ASCII representation, which it can store in a file as an offline model. The GCU can later reload an offline model and alter the system configuration to match the model. Whether you are viewing the active model or an offline model, you are always free to save the current configuration as an offline Galaxy Configuration Model (.GCM) file.

To make an offline model drive the current system configuration, the model must be loaded and engaged. To engage a model, click the Engage button. The GCU will scan the current configuration file, compare it against the model, and create a list of any management actions that are required to engage the model. The GCU presents this list to you for final confirmation. If you approve, the GCU will execute the actions, and the model will become engaged to reflect the current system configuration and state.

When you disengage a model, the GCU immediately marks the CPUs and instances as offline. You can then freely arrange the model however you like, and either save the model, or reengage the model. In typical practice, you are likely to have a small number of models that have proved to be useful for your business operations. These can be engaged by a system manager or a suitably privileged user, or through DCL command procedures.

12.3.1 Active Model

The GCU maintains a single active model. This model is always derived from the in-memory configuration file. The configuration file can be from a Galaxy console or from a file-based, single-instance Galaxy on any Alpha system. Regardless of its source, console callbacks maintain the integrity of the file. The GCU utilizes Galaxy event services to determine when a configuration change has occurred. When a change occurs, the GCU parses the configuration file and updates its active model to reflect the current system. The active model is not saved to a file unless you choose to save it as an offline model. Typically, the active model becomes the basis for creating additional models. When creating models, it is generally best to do so online so that you are sure your offline models can engage when they are needed.

12.3.2 Offline Models

The GCU can load any number of offline Galaxy configuration models and freely switch among them, assuming they were created for the specific system hardware. The model representation is a simple ASCII data definition format.

You should never need to edit a model file in its ASCII form. The GCU models and ruleset adhere to a simple proprietary language known as the Galaxy Configuration Language (GCL). This language continues to evolve as needed to represent new Galaxy innovations. Beware of this fact if you decide to explore the model and ruleset files directly. If you accidentally corrupt a model, you can always generate another. If you corrupt the ruleset, you may need to download another from the OpenVMS Galaxy website.

12.3.2.1 Example: Creating an Offline Model

To create an offline Galaxy configuration model:

1. Boot your Galaxy system, log in to the system account, and run the GCU.
2. By default, the GCU displays the active model.
3. Disengage the active model by clicking the Engage button (it toggles).
4. Assuming your system has a few secondary CPUs, drag and drop some of the CPUs to a different Galaxy instance.
5. Save the model by selecting Save Model from the Model menu. Give the model a suitable name with a .GCM extension. It is useful to give the model a name that denotes the CPU assignments; for example, such as G1x7.GCM for a system in which instance 0 has 1 CPU and instance 1 has 7 CPUs, or G4x4.GCM for a system with 4 CPUs on each of its two instances. This naming scheme is optional, but be sure to give the file the proper .GCM extension.

   You can create and save as many variations of the model as you like.

To engage an offline model:

1. Run the GCU.
2. By default, the GCU displays the active model. You can close the active model or just leave it.
3. Load the desired model by selecting Open Model from the Model menu.
4. Locate and select the desired model and click OK. The model will be loaded and displayed in an offline, disengaged state.
5. Click the Engage button to reengage the model.
6. The GCU will display any management operations required to engage the model. If you approve of the actions, click OK. The GCU will perform the management actions, and the model will be displayed as active and engaged.

12.4 Using the GCU Charts

The Galaxy Configuration File contains a considerable amount of configuration data and can grow quite large for complex Galaxy configurations. If the GCU displayed all the information it has about the system, the display would become unreasonably complex. To avoid this problem, the GCU provides Galaxy charts. Charts are simply a set of masks that control the visibility of the various components, devices, and interconnections. The entire component hierarchy is present, but only the components specified by the selected chart are visible. Selecting a different chart alters the visibility of component subsets.

By default, the GCU provides five preconfigured charts. Each is designed to show a specific component relationship. Some GCU command operations can be performed only within specific charts. For example, you cannot reassign CPUs from within the Physical Structure chart. The Physical Structure chart does not show the Galaxy instance components, thus you would have no target to drag and drop a CPU on. Because you can modify the charts the GCU does not restrict its menus and command operations to specific chart selections. In some cases, the GCU displays an informational message to help you select an appropriate chart.

12.4.1 Component Identification and Display Properties

Each component has a unique identifier. This identifier can be a simple sequential number, such as with CPU IDs, a physical backplane slot number, as with I/O adapters, or a physical address, as with memory devices. Each component type is also assigned a shape and color by the GCU. Where possible, the GCU further distinguishes each component using supplementary information it gathers from the running system.

The display properties of each component are assigned within the Galaxy Configuration Ruleset (SYS$MANAGER:GALAXY.GCR). You should not edit this file, except to customize certain display properties, such as window color or display text style.

The text that gets displayed about each component is also customizable. Each component type has a set of statements in the ruleset that determine its appearance, data content, and interaction.

One useful feature is the ability to select which text is displayed in each component type on the screen. The device declaration in the ruleset allows you to specify the text and parameters, which make up the display text statement. A subset of this display text is displayed whenever the zoom scale factor does not allow the full text to be displayed. This subset is known as the mnemonic. The mnemonic can be altered to include any text and parameters.

12.4.2 Physical Structure Chart

The Physical Structure chart describes the physical hardware in the system. The large rectangular component at the top, or root, of the chart represents the physical system cabinet itself. Typically, below the root, you will find physical components such as modules, slots, arrays, adapters, and so on. The type of components presented and the depth of the component hierarchy is directly dependent on the level of support provided by the console firmware for each hardware platform. If you are viewing a single-instance Galaxy on any Alpha system, then only a small subset of components can be displayed. As a general rule, the console firmware presents components only down to the level of configurable devices, typically to the first-level I/O adapter or slightly beyond. It is not a goal of the GCU or of the Galaxy console firmware to map every device, but rather those that are of interest to Galaxy configuration management.

The Physical Structure chart is useful for viewing the entire collection of components in the system; however, it does not display any logical partitioning of the components.

In the Physical Structure chart you can:

- Examine the parameters of any system component.

- Perform a hot-swap inquiry to determine how to isolate a component for repairs.
- Apply an Optimization Overlay to determine whether the hardware platform has specific optimizations that will ensure the best performance. For example, multiple-CPU modules may run best if all CPUs residing on a common module are assigned to the same Galaxy instance.
- Shut down or reboot the Galaxy or specific Galaxy instances.

12.4.2.1 Hardware Root

The topmost component in the Physical Structure chart is known as the hardware root (HW_Root). Every Galaxy system has a single hardware root. It is useful to think of this as the physical floorplan of the machine. If a physical device has no specific lower place in the component hierarchy, it will appear as a child of the hardware root. A component that is a child can be assigned to other devices in the hierarchy when the machine is partitioned or logically defined.

Tip

Clicking the root instance of any chart will perform an auto-layout operation if the Auto Layout mode is set.

12.4.2.2 Ownership Overlay

Choose Ownership Overlay from the Windows menu to display the initial owner relationships for the various components. These relationships indicate the instance that will own the component after a power cycle. Once a system has been booted, migratable components may change owners dynamically. To alter the initial ownership, the console environment variables must be changed.

The ownership overlay has no effect on the Physical Structure chart or the Failover Target chart.

12.4.3 Logical Structure Chart

The Logical Structure chart displays Galaxy communities and instances and is the best illustration of the relationships that form the Galaxy. Below these components are the various devices they currently own. Ownership is an important distinction between the Logical Structure chart and Physical Structure chart. In a Galaxy, resources that can be partitioned or dynamically reconfigured have two distinct “owners”.

The owner describes where the device will turn up after a system power up. This value is determined by the console firmware during bus-probing procedures and through interpretation of the Galaxy environment variables. The owner values are stored in console nonvolatile memory so that they can be restored after a power cycle.

The current_owner describes the owner of a device at a particular moment in time. For example, a CPU is free to reassign among instances. As it does, its current_owner value is modified, but its owner value remains whatever it was set to by the lp_cpu_mask# environment variables.

The Logical Structure chart illustrates the current_owner relationships. To view the nonvolatile owner relationships, select Ownership Overlay from the Window menu.

12.4.3.1 Software Root

The topmost component in the Logical Structure chart is known as the software root (SW_Root). Every Galaxy system has a single software root. If a physical device has no specific owner, it will appear as a child of the software root. A component that has a child can be assigned to other devices in the hierarchy when the machine is logically defined.

Tip

Clicking the root instance of any chart will perform an auto layout operation if the Auto Layout mode is set.

12.4.3.2 Unassigned Resources

You can configure Galaxy partitions without assigning all devices to a partition, or you can define but not initialize one or more partitions. In either case, some hardware may be unassigned when the system boots.

The console firmware handles unassigned resources in the following manner:

- Unassigned CPUs will be assigned to partition 0.
- Unassigned memory will be ignored.

Devices that remain unassigned after the system boots will appear assigned to the software root component and may not be accessible.

12.4.3.3 Community Resources

Resources such as shared memory can be accessed by all instances within a sharing community. Therefore, for shared memory, the community itself is considered the owner.

12.4.3.4 Instance Resources

Resources that are currently or permanently owned by a specific instance are displayed as children of the instance component.

12.4.4 Memory Assignment Chart

The Memory Assignment chart illustrates the partitioning and assignment of memory fragments among the Galaxy instances. This chart displays both hardware components (arrays, controllers, and so on) and software components (memory fragments).

Current Galaxy firmware and operating system software does not support dynamic reconfiguration of memory. Therefore, the Memory Assignment chart reflects the way the memory address space has been partitioned by the console among the Galaxy instances. This information can be useful for debugging system applications or for studying possible configuration changes.

12.4.4.1 Console Fragments

The console requires one or more small fragments of memory. Typically, a console allocates approximately 2MB of memory in the low address range of each partition. This varies by hardware platform and firmware revision. Additionally, some consoles allocate a small fragment in high address space for each partition to store memory bitmaps. The console firmware may need to create additional fragments to enforce proper memory alignment.

12.4.4.2 Private Fragments

Each Galaxy instance is required to have at least 64MB of private memory (includes the console fragments) to boot OpenVMS. This memory can consist of a single fragment, or the console firmware may need to create additional private fragments to enforce proper memory alignment.

12.4.4.3 Shared Memory Fragments

To create an OpenVMS Galaxy, a minimum of 8MB of shared memory must be allocated. This means the minimum memory requirement for an OpenVMS Galaxy is actually 72MB (64MB for a single instance, and 8MB for shared memory).

12.4.5 CPU Assignment Chart

The CPU Assignment chart displays the minimal number of components required to reassign CPUs among the Galaxy instances. This chart can be useful for working with very large Galaxy configurations.

12.4.5.1 Primary CPU

Each primary CPU is displayed as an oval rather than a hexagon. This is a reminder that primary CPUs cannot be reassigned or stopped. If you attempt to drag and drop a primary CPU, the GCU displays an error message in its status bar and does not allow the operation to occur.

12.4.5.2 Secondary CPUs

Secondary CPUs are displayed as hexagons. Secondary CPUs can be reassigned among instances in either the Logical Structure chart or the CPU Assignment chart. Simply drag and drop the CPU on the desired instance. If you drop a CPU on the same instance that currently owns it, the CPU will be stopped and restarted.

12.4.5.3 Fast Path and Affinitized CPUs

If you reassign a CPU that has a Fast Path device currently affinitized to the CPU, the affinity device will move to another CPU and the CPU reassignment will suceed. If a CPU has current process affinity assignment, the CPU cannot be reassigned.

For more information about using OpenVMS Fast Path features, see the *OpenVMS I/O User's Reference Manual*.

12.4.5.4 Lost CPUs

You can reassign secondary CPUs to instances that are not yet booted (partitions).

Similarly, you can reassign a CPU to an instance that is not configured as a member of the Galaxy sharing community. In this case, you can push the CPU away from its current owner instance, but you cannot get it back unless you log in to the independent instance (a separate security domain) and reassign the CPU back to the current owner.

Regardless of whether an instance is part of the Galaxy sharing community or is an independent instance, it will still be present in the Galaxy configuration file; therefore, the GCU will still be able to display it.

12.4.6 IOP Assignment Chart

The IOP Assignment chart displays the current relationship between I/O modules and the Galaxy instances. Note that, depending on what type of hardware platform is being used, a single-instance Galaxy on any Alpha system may not show any I/O modules in this display.

12.4.7 Failover Target Chart

The Failover Target chart shows how each processor will automatically fail over to other instances in the event of a shutdown or failure. Additionally, this chart illustrates the state of each CPU's autostart flag.

For each instance, a set of failover objects are shown, representing the full set of potential CPUs. By default, no failover relationships are established and all autostart flags are set.

To establish automatic failover of specific CPUs, drag and drop the desired failover object to the instance you want the associated CPU to target. To set failover relationships for all CPUs owned by an instance, drag and drop the instance object on top of the instance you want the CPUs to target.

To clear individual failover targets, drag and drop a failover object back to its owner instance. To clear all failover relationships, right-click on the instance object to display the Parameters & Commands dialog box, click on the Commands button, click the "Clear ALL failover targets?", button and then click OK.

By default, whenever a failover operation occurs, the CPUs will automatically start once they arrive in the target instance. You can control this autostart function using the autostart commands found in the Parameters & Commands dialog box for each failover object, or each instance object. The Failover Target chart displays the state of the autostart flag by displaying the failover objects in green if autostart is set, and red if autostart is clear.

Please note the following restrictions in the current implementation of failover and autostart management:

- The failover and autostart settings are not preserved across system boots. Thus, you will need to reestablish the model whenever the system reboots. To do this, invoke a previously saved configuration model, either by manually restoring the desired model or by using a command procedure during system startup.
- The GCU currently is not capable of determining the autostart and failover relationships of instances other than the one the GCU is running on, unless the instances are clustered.
- The GCU currently does not respond to changes in failover or autostart state that are made from another executing copy of the GCU or from DCL commands. If this state is altered, the GCU refreshes its display only if the active model is closed and then reopened.

12.5 Viewing Component Parameters

Each component has a set of parameters that can be displayed and, in some cases, altered. To display a component's parameters, position the cursor on the desired component, click the right mouse button, and select the Parameters item from the pop-up menu entry. Alternately, you can select a component, then select the Parameters item from the Components menu.

Where parameters are subject to unit conversion, changing the display unit will update the display and any currently visible parameter dialog boxes. Other parameters represent a snapshot of the system component and are not dynamically updated. If these parameters change, you must close and then reopen the Parameters dialog box to see the updated values.

12.6 Executing Component Commands

A component's Parameters dialog box can also contain a command page. If so, you can access the commands by clicking on the Commands button at the top of the dialog box. Most of the commands are executed by clicking on their toggle buttons and then clicking the OK or Apply buttons. Other commands may require that you enter information, or select values from a list or option menu. Note that if you select several commands, they will be executed in a top-down order. Be sure to choose command sequences that are logical.

12.7 Customizing GCU Menus

System Managers can extend and customize the GCU menus and menu entries by creating a file named SYS$MANAGER:GCU$CUSTOM.GCR. The file must contain only menu statements formatted as illustrated in the following examples. The GCU$CUSTOM.GCR file is optional. It will be preserved during operating system upgrades.

```
FORMAT EXAMPLE:

  MENU "Menu-Name" "Entry-Name" Procedure-type "DCL-command"

      * Menu-Name - A quoted string representing the name of the
                    pulldown menu to add or extend.

      * Entry-Name - A quoted string representing the name of the
                    menu entry to add.

      * Procedure-type - A keyword describing the type of procedure
                    to invoke when the menu entry is selected.

        Valid Procedure-type keywords include:

        COMMAND_PROCEDURE - Executes a DCL command or command file.
        SUBPROC_PROCEDURE - Executes a DCL command in subprocess context.

      * DCL-command - A quoted string containing a DCL command statement
                     consisting of an individual command or invokation
                     of a command procedure.
```

To create a procedure to run on other instances, create a command procedure that uses SYSMAN or task-to-task methods similar to what the GCU uses in SYS$MANAGER:GCU$ACTIONS.COM. You can extend GCU$ACTIONS.COM, but this file will be replaced during operating system upgrades and is subject to change.

```
  EXAMPLE MENU STATEMENTS (place in SYS$MANAGER:GCU$CUSTOM.GCR):

  // GCU$CUSTOM.GCR - GCU menu customizations
  // Note that the file must end with the END-OF-FILE statement.
  //
  MENU "Tools" "Availability Manager" SUBPROC_PROCEDURE "AVAIL/GROUP=DECamds"
  MENU "Tools" "Create DECterm" COMMAND_PROCEDURE  "CREATE/TERM/DETACH"
  MENU "DCL"   "Show CPU"       COMMAND_PROCEDURE  "SHOW CPU"
  MENU "DCL"   "Show Memory"    COMMAND_PROCEDURE  "SHOW MEMORY"
  MENU "DCL"   "Show System"    COMMAND_PROCEDURE  "SHOW SYSTEM"
  MENU "DCL"   "Show Cluster"   COMMAND_PROCEDURE  "SHOW CLUSTER"
  END-OF-FILE
```

12.8 Monitoring an OpenVMS Galaxy with DECamds

The DECamds availability manager software provides a valuable real-time view of the Galaxy system. DECamds can monitor all Galaxy instances from a single workstation or PC anywhere on the local area network. DECamds utilizes a custom OpenVMS driver (RMDRIVER) that periodically gathers availability data from the system. This information is returned to the DECamds client application using a low-level Ethernet protocol. The client application provides numerous views and graphs of the system's availability characteristics. Additionally, when DECamds detects one of numerous known conditions, it notifies the user and offers a set of solutions (called fixes) that can be applied to resolve the condition.

Every OpenVMS system comes with the DECamds Data Collector (RMDRIVER) installed. To enable the collector, you must execute its startup procedure inside SYSTARTUP_VMS.COM or manually on each Galaxy instance you want to monitor. Use the following commands to start the data collector:

```
$ @SYS$STARTUP:AMDS$STARTUP START or STOP
```

Prior to starting the collector, you need to specify a group name for your Galaxy. Do so by editing the file SYS$COMMON:[AMDS]AMDS$LOGICALS.COM. This file includes a statement for declaring a group name. Choose any unique name, making sure this file on each Galaxy instance contains the same group name.

When using DECamds, OpenVMS Engineering finds it useful to display the System Overview window, the Event window, and a CPU Summary window for each Galaxy instance. There are a number of additional views you can monitor depending on your specific interests. For more information about DECamds, refer to the *DECamds Users Guide*.

12.9 Running the CPU Load Balancer Program

The OpenVMS Galaxy CPU Load balancer program is a privileged application that dynamically reassigns CPU resources among instances in an OpenVMS Galaxy.

For information about how to run this program from the GCU, see Appendix A.

12.10 Creating an Instance

The current implementation of the Galaxy Software Architecture for OpenVMS requires that you predefine the Galaxy instances you intend to use. You can do this by using console environment variables. Refer to the appropriate sections of this guide for more details about Galaxy environment variables.

12.11 Dissolving an Instance

The only way to effectively dissolve a Galaxy instance is to shut it down, reassign its resources using console environment variables, and, if necessary, reboot any instances that will acquire new resources.

12.12 Shutdown and Reboot Cycles

Resources such as CPUs can be dynamically reassigned once the involved instances are booted. To reassign statically assigned resources, such as I/O modules, you must shut down and reboot the involved instances after executing the appropriate console commands.

12.13 Online versus Offline Models

The GCU allows you to display and interact with the active (online) or inactive (offline) Galaxy configuration models. When the configuration display represents a model of the active system, the GCU displays the state of the CPUs and instances using color and text. When the configuration model is engaged in this manner, you can interact with the active system using drag-and-drop procedures. The formal description for this mode of operation is interacting with the engaged, online model.

GCU users can also interact with any number of disengaged, or offline, models. Offline models can be saved to or loaded from files. An offline model can also be derived from the active online model by clicking the Engage button to be disengaged when the active online model is displayed. In addition to the visual state of the Engage button, the GCU also indicates the online versus offline characteristic of the CPUs and instances by using color and text. Any drag-and-drop actions directed at an offline model are interpreted as simple editing functions. They change the internal structure of the model but do not affect the active system.

When an offline model is engaged, the GCU compares the structure of the model with that of the active system. If they agree, the offline model is engaged and its new online state is indicated with color and text. If they do not agree, the GCU determines what management actions would be required to alter the active system to match the proposed model. A list of the resulting management actions is presented to the user and the user is asked whether they would like to execute the action list. If the user disapproves, the model remains offline and disengaged. If the user approves, the GCU executes the management actions and the resulting model is displayed as online and engaged.

12.14 GCU System Messages

```
%GCU-E-SUBPROCHALT, Subprocess halted; See GCU.LOG.

  The GCU has launched a user-defined subprocess which has terminated
  with error status.  Details may be found in the file GCU.LOG.

%GCU-S-SUBPROCTERM, Subprocess terminated

  The GCU has launched a user-defined subprocess which has terminated.

%GCU-I-SYNCMODE, XSynchronize activated

  The GCU has been invoked with X-windows synchronous mode enabled.
  This is a development mode which is not generally used.

%GCU-W-NOCPU, Unable to locate CPU

  A migration action was initiated which involved an unknown CPU.  This
  can result from engaging a model which contains invalid CPU identifiers
  for the current system.

%GCU-E-NORULESET, Ruleset not found:

  The GCU was unable to locate the Galaxy Configuration Ruleset in
  SYS$MANAGER:GALAXY.GCR.  New versions of this file can be downloaded
  from the OpenVMS Galaxy web page.

%GCU-E-NOMODEL, Galaxy configuration model not found:

  The specified Galaxy Configuration Model was not found.  Check your
  command line model file specification.

%GCU-W-XTOOLKIT, X-Toolkit Warning:
```

The GCU has intercepted an X-Toolkit warning. You may or may not be able to continue, depending on the type of warning.

%GCU-S-ENGAGED, New Galaxy configuration model engaged

The GCU has successfully engaged a new Galaxy Configuration Model.

%GCU-E-DISENGAGED, Unable to engage Galaxy configuration model

The GCU has failed to engage a new Galaxy Configuration Model. This can happen when a specified model is invalid for the current system, or when other system activities prevent the requested resource assignments.

%GCU-E-NODECW, DECwindows is not installed.

The current system does not have the required DECwindows support.

%GCU-E-HELPERROR Help subsystem error.

The DECwindows Help system (Bookreader) encountered an error.

%GCU-E-TOPICERROR Help topic not found.

The DECwindows Help system could not locate the specified topic.

%GCU-E-INDEXERROR Help index not found.

The DECwindows Help system could not locate the specified index.

%GCU-E-UNKNOWN_COMPONENT: {name}

The current model contains reference to an unknown component. This can result from model or ruleset corruption. Search for the named component in the ruleset SYS$MANAGER:GALAXY.GCR. If it is not found, download a new one from the OpenVMS Galaxy web site. If the problem persists, delete and recreate the offending model.

%GCU-I-UNASSIGNED_HW: Found unassigned {component}"

The GCU has detected a hardware component which is not currently assigned to any Galaxy instance. This may result from intentionally leaving unassigned resources. Note the message and continue or assign the hardware component from the primary Galaxy console and reboot.

%GCU-E-UNKNOWN_KEYWORD: {word}

The GCU has parsed an unknown keyword in the current model file. This can only result from model file format corruption. Delete and recreate the offending model.

%GCU-E-NOPARAM: Display field {field name}

The GCU has parsed an incomplete component statement in the current model. This can only result from model file format corruption. Delete and recreate the offending model.

%GCU-E-NOEDITFIELD: No editable field in display.

The GCU has attempted to edit a component parameter which is undefined. This can only result from model file format corruption. Delete and recreate the offending model.

%GCU-E-UNDEFTYPE, Undefined Parameter Data Type: {type}

The GCU has parsed an unknown data type in a model component parameter. This can result from model file format corruption or incompatible ruleset for the current model. Search the ruleset SYS$MANAGER:GALAXY.GCR for the offending datatype. If not found, download a more recent ruleset from the OpenVMS Galaxy web site. If found, delete and recreate the offending model.

%GCU-E-INVALIDMODEL, Invalid model structure in: {model file}

The GCU attempted to load an invalid model file. Delete and recreate the offending model.

```
%GCU-F-TERMINATE Unexpected termination.

  The GCU encountered a fatal DECwindows event.

%GCU-E-GCTLOOP: Configuration Tree Parser Loop

  The GCU has attempted to parse a corrupt configuration tree.  This
  may be a result of console firmware or operating system fault.

%GCU-E-INVALIDNODE: Invalid node in Configuration Tree

  The GCU has parsed an invalid structure within the configuration tree.
  This can only result from configuration tree corruption or revision
  mismatch between the ruleset and console firmware.

%GCU-W-UNKNOWNBUS: Unknown BUS subtype: {type}

  The GCU has parsed an unknown bus type in the current configuration
  tree.  This can only result from revision mismatch between the
  ruleset and console firmware.

%GCU-W-UNKNOWNCTRL, Unknown Controller type: {type}

  The GCU has parsed an unknown controller type in the current configuration
  tree.  This can only result from revision mismatch between the
  ruleset and console firmware.

%GCU-W-UNKNOWNCOMP, Unknown component type: {type}

  The GCU has parsed an unknown component type in the current configuration
  tree.  This can only result from revision mismatch between the
  ruleset and console firmware.

%GCU-E-NOIFUNCTION, Unknown internal function

  The user has modified the ruleset file and specified an unknown
  internal GCU function.  Correct the ruleset or download a new one
  from the OpenVMS Galaxy web page.

%GCU-E-NOEFUNCTION, Missing external function

  The user has modified the ruleset file and specified an unknown
  external function.  Correct the ruleset or download a new one
  from the OpenVMS Galaxy web page.

%GCU-E-NOCFUNCTION, Missing command function

  The user has modified the ruleset file and specified an unknown
  command procedure.  Correct the ruleset or download a new one
  from the OpenVMS Galaxy web page.

%GCU-E-UNKNOWN_COMPONENT: {component}

  The GCU has parsed an unknown component.  This can result from
  ruleset corruption or revision mismatch between the ruleset and
  console firmware.

%GCU-E-BADPROP, Invalid ruleset DEVICE property

  The GCU has parsed an invalid ruleset component statement.  This can
  only result from ruleset corruption.  Download a new one from the
  OpenVMS Galaxy web page.

%GCU-E-BADPROP, Invalid ruleset CHART property

  The GCU has parsed an invalid chart statement.  This can
  only result from ruleset corruption.  Download a new one from the
  OpenVMS Galaxy web page.

%GCU-E-BADPROP, Invalid ruleset INTERCONNECT property

  The GCU has parsed an invalid ruleset interconnect statement.  This can
  only result from ruleset corruption.  Download a new one from the
  OpenVMS Galaxy web page.

%GCU-E-INTERNAL Slot {slot detail}
```

```
  The GCU has encountered an invalid datatype from a component parameter.
  This can result from ruleset or model corruption.  Download a new one
  from the OpenVMS Galaxy web page.  If the problem persists, delete and
  recreate the offending model.

%GCU-F-PARSERR, {detail}

  The GCU encountered a fatal error while parsing the ruleset.  Download
  a new one from the OpenVMS Galaxy web page.

%GCU-W-NOLOADFONT: Unable to load font: {font}

  The GCU could not locate the specified font on the current system.
  A default font will be used instead.

%GCU-W-NOCOLORCELL: Unable to allocate color

  The GCU is unable to access a colormap entry.  This can result from
  a system with limited color support or from having an excessive number
  of graphical applications open at the same time.

 GCU-E-NOGALAXY, This system is not configured as a Galaxy.

  Description:

     The user has issued the CONFIGURE GALAXY/ENGAGE command on a
     system which is not configured for Galaxy operation.

  User Action:

     Configure your system for Galaxy operation using the procedures
     described in the OpenVMS Galaxy Guide.  If you only want to run a
     single-instance Galaxy, enter CONFIGURE GALAXY without the
     /ENGAGE qualifier and follow the instructions provided by the
     Galaxy Configuration Utility.

%GCU-E-ACTIONNOTALPHA GCU actions require OpenVMS Alpha

  A GCU user has attempted to invoke a Galaxy configuration operation
  on an OpenVMS VAX system.

%GCU-I-ACTIONBEGIN at {time}, on {instance} {mode}

  This informational message indicates the start of a configuration
  action on the specified Galaxy instance.  Note that many actions
  require collaboration between command environments on two separate
  Galaxy instances, thus, you may encounter two of these messages, one
  per instance involved in the operation.  The mode argument indicates
  which instance is local versus remote.

%GCU-S-ACTIONEND  at {time}, on {nodename}

  This is the normal successful completion message following a Galaxy
  configuration action.  Note that many actions require collaboration
  between command environments on two separate Galaxy instances, thus, you
  may encounter two of these messages, one per instance involved in the
  operation.

%GCU-S-ACTIONEND, Exiting GCU$ACTIONS on ^Y

  Indicates that the user has aborted a Galaxy configuration action
  using Control-Y.

%GCU-S-ACTIONEND, Exiting GCU$ACTIONS on error {message}

  Indicates that a Galaxy configuration action terminated with error
  status as indicated by the message argument.

%GCU-E-ACTIONUNKNOWN no action specified

  Indicates that the GCU$ACTIONS.COM procedure was called improperly.
  It is possible that the command procedure has been corrupted or is
  out of revision for the current system.

%GCU-E-ACTIONNOSIN no source instance name specified
```

Indicates that the GCU$ACTIONS.COM procedure was called improperly. It is possible that the command procedure has been corrupted or is out of revision for the current system.

```
%GCU-E-ACTIONBAD failed to execute the specfied action
```

Indicates that a Galaxy configuration action aborted due to an indeterminate error condition. Review related screen messages and verify that the necessary proxy accounts have been established.

```
%GCU-E-INSFPRIVS, Insufficient privileges for attempted operation
```

An underprivileged user has attempted to perform a Galaxy configuration action. Typically, these actions are performed from within the system managers account. OPER and SYSPRV privileges are required.

```
%GCU-E-NCF network connect to {instance} failed
```

An error has occurred trying to open a DECnet task-to-task connection between the current and specified instances. Review related screen messages and verify that the necessary proxy accounts have been established.

13
CPU Reassignment

OpenVMS supports several methods of managing CPU resources. The console establishes the default owner instance for each CPU using the console environment variables. This allows the CPU resources to be statically assigned, providing a precise initial configuration. In the event of a cold boot (that is, power cycle or initialization), this default configuration is restored from console nonvolatile RAM.

Once a configuration has been booted, OpenVMS provides more elegant means of assigning resources for users with CMKRNL (Change Mode to Kernel) privilege. The following sections describe these methods.

13.1 DCL Reassignment

Users with CMKRNL privilege can perform CPU reassignment operations using the following DCL command:

```
$ STOP/CPU/MIGRATE=instance-or-id   cpu-id
```

The user must provide the target instance name (SCSNAME) or numeric ID (0, 1, and so on), and the numeric ID of the CPU being reassigned. The following examples show a few forms of this command.

```
$ STOP/CPU/MIGRATE=0  4      !Reassign CPU 4 to instance 0
$ STOP/CPU/MIGRATE=1  3,4,5  !Reassign CPUs 3,4,5 to instance 1
$ STOP/CPU 7/MIGRATE=BIGBNG  !ReassignCPU 7 to instance BIGBNG
$ STOP/CPU/ALL/MIGRATE=0     !Reassign all secondary CPUs to instance 0
```

These commands can be inserted into command procedures. For example, you might want to move extra CPU resources to an instance in a startup procedure of an application with known processing requirements. Similarly, you may want to reassign CPUs away from an instance that is about to perform lengthy, I/O intensive operations (such as backups) so that the CPUs are available to other instances. When the job completes, you may reassign them back. Or you may reassign CPUs away from an instance that is shutting down.

Note that you can only reassign resources away from an instance. This is the push model defined by the Galaxy Software Architecture. This model prevents resources from being “stolen” by other instances that may not be aware of their current usage. To effectively manage the entire Galaxy system using DCL, you must either log in to each of the involved instances or use the SYSMAN utility to execute the commands on the owner instance.

13.2 GCU Drag-and-Drop Reassignment

The GCU provides an interactive visual interface for managing Galaxy resources. Using the GCU, you can reassign CPUs by simply dragging and dropping them among instances. Additionally, the GCU allows you to draw charts of various configurations (known as configuration models) and save them as files. At any time, you can load and engage a configuration model and the system will reassign resources as needed to achieve the desired model.

13.3 Intermodal Reassignment

Because of the resource push model defined by the Galaxy Software Architecture, resources must be given away by the Galaxy instance that currently owns them. For a utility or user to effectively manage resource assignment in a multiple-instance Galaxy configuration, you must establish some means of executing commands on each instance.

One such means is to open a window or terminal session on each of the Galaxy instances and perform your resource management operations in each of these windows.

Another method is use the SYSMAN utility and its underlying SMI server to establish a command environment on the owner instance. Using this method, a fairly simple command procedure can be written to perform specific resource management operations. This method has some limitations, however. First, it requires that the involved Galaxy instances must be in a cluster. Also, a command procedure cannot effectively pass variable parameters to SYSMAN environment scripts, and you cannot specify a remote system password within a SYSMAN script. So it is cumbersome to generate a general-purpose command procedure interface that uses SYSMAN.

The GCU does, in fact, use SYSMAN wherever possible to accomplish its management actions. When a system is not configured to support SYSMAN, the GCU will attempt to use DECnet task-to-task comminations among proxy accounts as its management transport. If that approach also fails (that is, if the system is not running DECnet or if the necessary proxy accounts are not set up), the GCU will not be able to manage Galaxy instances other than the one on which the GCU is currently running. You could certainly run multiple copies of the GCU if you choose, one per Galaxy instance. However, you can assume that OpenVMS Galaxy systems are likely to be clustered or to use DECnet.

The GCUs management actions are based in the SYS$MANAGER:GCU$ACTIONS.COM command procedure. You can modify this file to customize actions for your own environment. For example, in a TCP/IP environment, you may choose to use REXEC or a similar utility for your management transport, or you may want to include some form of notification or logging whenever a management action is executed.

The GCU$ACTIONS.COM file is somewhat unusual in the way it operates. When using SYSMAN, the procedure builds small SYSMAN command scripts in temporary files to deal with variable parameters that SYSMAN cannot handle. When SYSMAN is not available, the procedure attempts to open a DECnet task-to-task connection to a proxy account on the owner instance. If successful, it uses this connection to shuffle command parameters to the copy of GCU$ACTIONS.COM that resides on the owner instance. The end result is execution of the command locally by the owner instance.

13.4 Software Reassignment Using Galaxy Services

Perhaps the best method for managing resource assignment is to use the Galaxy APIs to write your own resource management routines. This allows you to base your decisions for resource management on your own criteria and application environment. The same push-model restriction described in Section 13.3 still exists, however, so your routines will need to be Galaxy aware, possibly using shared memory to coordinate their operations.

Table 13–1 briefly describes the OpenVMS system services available to manage CPUs.

Table 13–1 Galaxy System Services for CPU Management

System Service	Description
$CPU_TRANSITION[W]	Changes the current processing state of a CPU in the configure set of the current system or an unassigned CPU in an OpenVMS Galaxy configuration.

13.5 Reassignment Faults

CPU reassignment can fail or be blocked, for several reasons. Because the GCU buries its management actions in SYSMAN or DCL scripts, it may not always identify and report the reasons for a reassignment fault. The GCU does perform certain checks prior to allowing reassignment actions in order, for example, to prevent attempts to reassign the primary CPU. Other reasons exist for reassignment faults that can only be detected by the operating system or console firmware. For example, if the operating system detects a fault attempting to reassign a CPU that currently has process affinity or Fast Path duties, a DCL message will be displayed on both the console and the users terminal.

The Galaxy APIs for reassignment are capable of reporting most faults to the caller. However, even using the reassignment services, the console may reject a reassignment because of hardware platform dependencies not readily visible to the operating system.

14
DCL Commands

The following DCL commands are useful for managing an OpenVMS Galaxy:

- STOP/CPU/MIGRATE
- SHOW CPU
- SET CPU
- SHOW MEMORY
- Lexical functions
- INSTALL LIST
- CONFIGURE GALAXY

14.1 CPU Commands

CPUs are assignable resources in an OpenVMS Galaxy.

14.1.1 STOP/CPU/MIGRATE

The STOP/CPU/MIGRATE command stops and removes the specified secondary processors from the active set in an OpenVMS SMP (symmetric multiprocessing) system.

For example, a user enters:

```
$ STOP/CPU/MIGRATE=GLX0  4
```

The following message is displayed at the user's terminal:

```
%SYSTEM-I-CPUSTOPPING, trying to stop CPU 4 after it reaches quiescent state
```

The source console displays:

```
%SMP-I-STOPPED, CPU #04 has been stopped.
```

The destination console displays:

```
%SMP-I-SECMSG, CPU #04 message:   P04>>>START
%SMP-I-CPUTRN, CPU #04 has joined the active set.
```

14.1.2 SHOW CPU

The SHOW CPU command displays information about the status, characteristics, and capabilities of the specified processors.

For example:

```
$ SHOW CPU

GLX0,  AlphaServer 8400 Model 5/440

Multiprocessing ENABLED. Full checking synchronization image loaded.

Minimum multiprocessing revision levels: CPU = 1

PRIMARY CPU =  00

Active CPUs:      00 01

Configured CPUs:  00 01

Potential CPUs:   00 01 03 04 05 06 07
```

14.1.3 SET CPU

- SET CPU *n*/FAILOVER=*y*

  Establishes instance-specific failover relationships for each CPU in the instances potential set.

  n is the CPU number, a comma-separated list of CPUs, or the /ALL qualifier.

  y is the instance number or name.
- SET CPU *n*/NOFAILOVER

  Removes any instance-specific failover relationship for the specified CPUs.
- SET CPU *n*/[NO]AUTOSTART

  Sets or clears the instance-specific autostart flag for the specified CPUs.

  n is the CPU number, a comma-separated list of CPUs, or the /ALL qualifier.

14.2 SHOW MEMORY

The SHOW MEMORY command displays the uses of memory by the system.

For example:

```
$ SHOW MEMORY/PHYSICAL

            System Memory Resources on 5-OCT-2001 20:50:19.03

Physical Memory Usage (pages):     Total        Free      In Use    Modified
   Main Memory (2048.00Mb)       262144        228183    31494      2467
Of the physical pages in use, 11556 pages are permanently allocated to OpenVMS.

$ SHOW MEMORY/PHYSICAL

              System Memory Resources on 5-OCT-2001 07:55:14.68

Physical Memory Usage (pages):     Total        Free      In Use    Modified
   Private Memory (512.00Mb)      65536        56146      8875         515
   Shared Memory (1024.00Mb)      131072       130344      728

Of the physical pages in use, 6421 pages are permanently allocated to OpenVMS.
$
```

14.3 Lexical Function Example

Lexical Function Example Command Procedure

```
$ write sys$output ""
$ write sys$output "Instance = ",f$getsyi("scsnode")
$ write sys$output "Platform = ",f$getsyi("galaxy_platform")
$ write sys$output "Sharing Member = ",f$getsyi("galaxy_member")
$ write sys$output "Galaxy ID = ",f$getsyi("galaxy_id")
$ write sys$output "Community ID = ",f$getsyi("community_id")
$ write sys$output "Partition ID = ",f$getsyi("partition_id")
$ write sys$output ""
$ exit
```

Lexical Function Command Procedure Output

```
$ @SHOGLX

Instance = COBRA2
Platform = 1
Sharing Member = 1
Galaxy ID = 5F5F30584C47018011D3CC8580F40383
Community ID = 0
Partition ID = 0

$
```

14.4 INSTALL LIST

The INSTALL LIST command now returns the Galaxywide sections as well as standard global sections.

14.5 CONFIGURE GALAXY

The CONFIGURE GALAXY command invokes the Galaxy Configuration Utility (GCU) to monitor, display, and interact with an OpenVMS Galaxy system. The GCU requires DECwindows Motif Version 1.2-4 or higher and OpenVMS Alpha Version 7.2 or higher.

The optional model parameter specifies the location and name of a Galaxy Configuration Model to load and display. If no model is provided and the system is running as an OpenVMS Galaxy, the current active configuration is displayed.

If the system is not running as an OpenVMS Galaxy, the GCU will assist the user in creating a single-instance OpenVMS Galaxy system.

OpenVMS Galaxy Configuration Models are created with the Galaxy Configuration Utility. Refer to the GCU online help for more information.

Format:

CONFIGURE GALAXY [model.gcm]

Parameters:

`GALAXY [model.GCM]`

Specifies the location and name of a Galaxy configuration model to load and display.

If no model is provided and the system is running as an OpenVMS Galaxy, the current active configuration is displayed.

Qualifiers:

/ENGAGE

Causes the GCU to engage (load, validate, and activate) the specified OpenVMS Galaxy Configuration Model without displaying the graphical user interface. After validation, the specified model becomes the active system configuration.

This qualifier allows system managers to restore the OpenVMS Galaxy system to a known configuration, regardless of what dynamic resource reassignments may have occurred since the system was booted. This command can be embedded in DCL command procedures to automate configuration operations.

/VIEW

When used in conjunction with /ENGAGE and a model parameter, causes the GCU to load, validate, activate, and display the specified configuration model.

Examples:

```
$ CONFIGURE GALAXY
```

Displays the GCU's graphical user interface. If the system is currently configured as an OpenVMS Galaxy, the active system configuration is displayed.

```
$ CONFIGURE GALAXY model.GCM
```

Displays the GCU's graphical user interface. The specified OpenVMS Galaxy Configuration Model is loaded and displayed, but does not become the active configuration until the user chooses to engage it.

```
$ CONFIGURE GALAXY/ENGAGE model.GCM
```

Invokes the GCU command line interface to engage the specified OpenVMS Galaxy Configuration Model without displaying the GCU's graphical user interface.

```
$ CONFIGURE GALAXY/ENGAGE/VIEW model.GCM
```

Invokes the GCU command line interface to engage the specified OpenVMS Galaxy Configuration Model and display the GCU's graphical user interface.

15
Communicating With Shared Memory

This chapter describes the following two OpenVMS internal mechanisms that use shared memory to communicate between instances in an OpenVMS Galaxy computing environment:

- Shared Memory Cluster Interconnect (SMCI)
- Local area network (LAN) shared memory device driver

15.1 Shared Memory Cluster Interconnect (SMCI)

The Shared Memory Cluster Interconnect (SMCI) is a System Communications Services (SCS) port for communications between Galaxy instances. When an OpenVMS instance is booted as both a Galaxy and as an OpenVMS Cluster member, the SMCI driver is loaded. This SCS port driver communicates with other cluster instances in the same Galaxy through shared memory. This capability provides one of the major performance benefits of the OpenVMS Galaxy Software Architecture. The ability to communicate to another clustered instance through shared memory provides dramatic performance benefits over traditional cluster interconnects.

15.1.1 SYS$PBDRIVER Port Devices

When booting as both a Galaxy and a cluster member, SYS$PBDRIVER is loaded by default. The loading of this driver creates a device PBA*x*, where *x* represents the Galaxy partition ID. As other instances are booted, they also create PBA*x* devices. The SMCI quickly identifies the other instances and creates communications channels to them. Unlike traditional cluster interconnects, a new device is created to communicate with the other instances. This device also has the name PBA*x*, where *x* represents the Galaxy partition ID for the instance with which this device is communicating.

For example, consider an OpenVMS Galaxy that consists of two instances: MILKY and WAY. MILKY is instance 0 and WAY is instance 1. When node MILKY boots, it creates device PBA0. When node WAY boots, it creates PBA1. As the two nodes “find” each other, MILKY creates PBA1 to talk to WAY and WAY creates PBA0 to talk to MILKY.

```
MILKY                WAY

PBA0:                PBA1:

PBA1:   <------->    PBA0:
```

15.1.2 Multiple Clusters in a Single Galaxy

SYS$PBDRIVER can support multiple clusters in the same Galaxy. This is done in the same way that SYS$PEDRIVER allows support for multiple clusters on the same LAN. The cluster group number and password used by SYS$PEDRIVER are also used by SYS$PBDRIVER to distinguish different clusters in the same Galaxy community. If your Galaxy instances are also clustered with other OpenVMS instances over the LAN, the cluster group number is set appropriately by CLUSTER_CONFIG. To determine the current cluster group number:

```
$ MCR SYMAN
SYSMAN> CONFIGURATION SHOW CLUSTER_AUTHORIZATION
Node: MILKY   Cluster group number: 0
Multicast address: xx-xx-xx-xx-xx-xx
SYSMAN>
```

If you are not clustering over a LAN and you want to run multiple clusters in the same Galaxy community, then you must set the cluster group number. You must ensure that the group number and password are the same for all Galaxy instances that you want to be in the same cluster as follows:

```
$ MCR SYSMAN
SYSMAN> CONFIGURATION SET CLUSTER_AUTHORIZATION/GROUP_NUMBER=222/PASSWORD=xxxx
SYSMAN>
```

If your Galaxy instances are also clustering over the LAN, CLUSTER_CONFIG asks for a cluster group number, and the Galaxy instances use that group number. If you are not clustering over a LAN, the group number defaults to zero. This means that all instances in the Galaxy will be in the same cluster.

15.1.3 SYSGEN Parameters for SYS$PBDRIVER

In most cases, the default settings for SYS$PBDRIVER should be appropriate; however, several SYSGEN parameters are provided. Two SYSGEN parameters control SYS$PBDRIVER: SMCI_PORTS and SMCI_FLAGS.

15.1.3.1 SMCI_PORTS

The SYSGEN parameter SMCI_PORTS controls initial loading of SYS$PBDRIVER. This parameter is a bitmask in which bits 0 through 25 each represent a controller letter. If bit 0 is set, PBA*x* will be loaded; this is the default setting. If bit 1 is set, PBB*x* will be loaded, and so on all the way up to bit 25, which will cause PBZ*x* to be loaded. For OpenVMS Alpha Version 7.2–1, Compaq recommends leaving this parameter at the default value of 1.

Loading additional ports allows for multiple paths between Galaxy instances. For OpenVMS Alpha Version 7.2–1, having multiple communications channels does not provide any advantages because SYS$PBDRIVER will initially not support Fast Path. A future release of OpenVMS will provide Fast Path support for SYS$PBDRIVER. When Fast Path support is enabled, instances with multiple CPUs can achieve improved throughput by having multiple communications channels between instances.

15.1.4 SMCI_FLAGS

The SYSGEN parameter SMCI_FLAGS controls operational aspects of SYS$PBDRIVER. The only currently defined flag is bit 1. This controls whether or not the port device supports communications with itself. Supporting SCS communications to itself is primarily used for test purposes. By default, this bit will be turned off and thus support for SCS communication locally is disabled, which saves system resources. This parameter is dynamic and by turning this bit on, an SCS virtual circuit should soon form.

Bit	Mask	Description
0	0	0 = Do not create local communications channels (SYSGEN default). Local SCS communications are primarily used in test situations and not needed for normal operations. Leaving this bit off saves resources and overhead. 1 = Create local communications channels.
1	2	0 = Load SYS$PBDRIVER if booting into both a Galaxy and a Cluster (SYSGEN Default). 1 = Load SYS$PBDRIVER if booting into a Galaxy.
2	4	0 = Minimal console output (SYSGEN default) 1 = Full console output, SYS$PBDRIVER will display console messages when creating communication channels and tearing down communication channels.

15.2 LAN Shared Memory Device Driver

Local area network (LAN) communications between OpenVMS Galaxy instances are supported by the Ethernet LAN shared memory driver. This LAN driver communicates to other instances in the same OpenVMS Galaxy system through shared memory. Communicating with other instances through shared memory provides performance benefits over traditional LANs.

To load the LAN shared memory driver SYS$EBDRIVER, enter the following command:

```
$ MCR SYSMAN
SYSMAN> IO CONN EBA/DRIVER=SYS$EBDRIVER/NOADAPTER
```

For OpenVMS Version 7.2–1, in order for LAN protocols to automatically start over this LAN device (EBA*n*, where *n* is the unit number), the procedure for loading this driver should be added to the configuration procedure: `SYS$MANAGER:SYCONFIG.COM`.

The LAN driver emulates an Ethernet LAN with frame formats the same as Ethernet/IEEE 802.3 but with maximum frame size increased from 1518 to 7360 bytes. The LAN driver presents a standard OpenVMS QIO and VCI interface to applications. All existing QIO and VCI LAN applications should work unchanged.

In a future release, the SYS$EBDRIVER device driver will be loaded automatically.

16
Shared Memory Programming Interfaces

A **shared memory global section** maps some amount of memory that can be accessed on all instances in a sharing community. These objects are also called **Galaxywide shared sections**. Each such object has a name, a version, and protection characteristics.

16.1 Using Shared Memory

Application programs access shared memory by mapping Galaxywide shared sections. The programming model is the same as for standard OpenVMS global sections; that is, you create, map, unmap, and delete them on each instance where you want to use them. Some shared memory global section characteristics are:

- Pages start out as demand zero with preallocated shared PFNs.
- Pages are not counted against your working set.
- Once the page is valid in your process' page table, it stays valid until it is deleted; shared memory section pages are never paged to disk.
- You must create the shared section on each instance where you want to access shared memory.
- Sections can be temporary or permanent.
- Sections can be group or system global sections.
- Galaxywide shared sections use a different name space than traditional global sections.
- Section versions specified in the ident_64 field are validated throughout the Galaxy.
- Only one shared section with a given name and UIC group can exist in a sharing community. This is different from traditional global sections, in which multiple versions can coexist.
- The SHMEM privilege is required to create a shared memory section.

From a programmer's point of view, shared memory global sections are similar to memory resident sections. You use the same system services to create Galaxywide shared sections that you would use to create memory resident sections. Setting the flag SEC$M_SHMGS lets the service operate on a shared memory global section.

In contrast to memory resident sections, the Reserved Memory Registry is not used to allocate space for Galaxywide sections. The SYSMAN RESERVE commands affect only node-private memory. Shared memory is not used for normal OpenVMS paging operations and does not need to be reserved.

There is also no user interface to specify whether shared page tables should be created for Galaxywide sections. Instead, creation of shared page tables for Galaxywide sections is tied to the section size. As of OpenVMS Version 7.2, shared page tables are created for sections of 128 pages (1 MB) or more. Galaxywide shared page tables are shared between all Galaxy instances.

16.2 System Services

The following sections describe new and changed system services that support shared memory global sections.

16.2.1 Enhanced Services

The following system services have been enhanced to recognize the new shared memory global section flag SEC$M_SHMGS:

- SYS$CRMPSC_GDZRO_64
- SYS$CREATE_GDZRO
- SYS$MGBLSC_64
- SYS$DGBLSC

The following system services have been enhanced to work with shared memory, but no interfaces were changed:

- SYS$DELTVA
- SYS$DELTVA_64
- SYS$CREATE_BUFOBJ
- SYS$CREATE_BUFOBJ_64
- SYS$DELETE_BUFOBJ

16.2.2 New Section Flag SEC$M_READ_ONLY_SHPT

The new section flag SEC$M_READ_ONLY_SHPT is recognized by the SYS$CREATE_GDZRO and SYS$CRMPSC_GDZRO_64 services. When this bit is set, it directs the system to create shared page tables for the sections that allow read access only. This feature is particularly useful in an environment where a memory resident or Galaxy shared section is used by many readers but only a single writer.

When you map a Galaxy shared section or a memory resident section that has an associated shared page table section, you have the following options for accessing data:

Shared Page Tables	Read Only	Read and Write
None created	Do not set the SEC$M_WRT flag in the map request. Private page tables will always be used, even if you are specifying a shared page table region into which to map the section.	Set the SEC$M_WRT flag in the map request. Private page tables will always be used, even if you are specifying a shared page table region into which to map the section.
Write access	Do not set the SEC$M_WRT flag in the map request. Ensure that private page tables will be used. Do not specify a shared page table region into which to map the section. If you do, the error status SS$_IVSECFLG is returned.	Set the SEC$M_WRT flag in the map request. The shared page table section will be used for mapping if you specify a shared page table region into which to map the section.
Read access	Do not set the SEC$M_WRT flag in the map request. The shared page table section will be used for mapping if you specify a shared page table region into which to map the section.	Set the SEC$M_WRT flag in the map request. Ensure that private page tables will be used. Do not specify a shared page table region into which to map the section. If you do, the error status SS$_IVSECFLG is returned.

Notes

Shared page tables for Galaxy shared sections are also implemented as Galaxy shared sections. This implies that they allow either read access only on all OpenVMS instances connected to this section or read and write access on all instances. The setting of the SEC$M_READ_ONLY_SHPT flag as requested by the first instance to create the section is used on all instances.

Using the SYS$CRMPSC_GDZRO_64 service always implies that the SEC$M_WRT flag is set and that you want to map the section for writing. If you want to use this service to create a section with shared page tables for read-only access, you must use private page tables and you cannot specify a shared page table region into which to map the section.

16.3 Galaxywide Global Sections

The SHMEM privilege is required to create an object in Galaxy shared memory. The right to map to an existing section is controlled through normal access control mechanisms. SHMEM is not needed to map an existing section. Note that the VMS$MEM_RESIDENT_USER identifier, which is needed to create an ordinary memory resident section, is not required for Galaxywide sections.

Creating and mapping Galaxywide memory sections is accomplished through the same services used to create memory resident sections. The following services now recognize the SEC$M_SHMGS flag:

SYS$CREATE_GDZRO
SYS$CRMPSC_GDZRO_64
SYS$MGBLSC_64
SYS$DGBLSC

SYS$CREATE_GDZRO and SYS$CRMPSC_GDZRO_64 can also return new status codes.

SS$_INV_SHMEM	Shared memory is not valid.
SS$_INSFRPGS	Insufficient free shared pages or private pages.
SS$_NOBREAK	A Galaxy lock is held by another node and was not broken.
SS$_LOCK_ TIMEOUT	A Galaxy lock timed out.

The INSTALL LIST/GLOBAL and SHOW MEMORY commands are also aware of Galaxywide sections.

Galaxywide sections are using their own name space. Just as you could always use the same name to identify system global sections and group global sections for various owner UICs, you can now also have Galaxywide system global sections and Galaxywide group global sections all with the same name.

Galaxywide sections also have their own security classes:

GLXSYS_GLOBAL_SECTION
GLXGRP_GLOBAL_SECTION

These security classes are used with the $GET_SECURITY and $SET_ SECURITY system services, and DCL commands SET/SHOW SECURITY.

These new security classes are only valid in a Galaxy environment. They are not recognized on a non-Galaxy node.

You can only retrieve and affect security attributes of Galaxywide global sections if they exist on your sharing instance.

Audit messages for Galaxywide sections look like this:

```
%%%%%%%%%  OPCOM  20-MAR-1998 10:44:43.71  %%%%%%%% (from node GLX1 at 20-MAR-1998 10:44:43.85)
Message from user AUDIT$SERVER on GLX1
Security alarm (SECURITY) on GLX1, system id: 19955
Auditable event:          Object creation
Event information:        global section map request
Event time:               20-MAR-1998 10:44:43.84
PID:                      2040011A
Process name:             ANDY
Username:                 ANDY
Process owner:            [ANDY]
Terminal name:            RTA1:
Image name:               MILKY$DKA100:[ANDY]SHM_MAP.EXE;1
Object class name:        GLXGRP_GLOBAL_SECTION
Object name:              [47]WAY____D99DDB03_0$MY_SECTION
Secondary object name:    <Galaxywide global section>
Access requested:         READ,WRITE
Deaccess key:             8450C610
Status:                   %SYSTEM-S-CREATED, file or section did not exist; has
been created
```

Note the “Object name” field: the object name displayed here uniquely identifies the section in the OpenVMS Galaxy. The fields are as follows:

[47]	(only for group global sections) identifies the UIC group of the section creator.
WAY____D99DDB03_0$	An identifier for the sharing community.
MY_SECTION	The name of the section as specified by the user.

The user can only specify the section name and class for requests to set or show the security profile. The UIC is always obtained from the current process and the community identifier is obtained from the community in which the process executes.

The output for a Galaxywide system global section differs only in the fields "Object class name" and "Objects name." The object name for this type of section does not include a group identification field:

Object class name:	GLXSYS_GLOBAL_SECTION
Object name:	WAY_ _ _ _D99DDB03_0$SYSTEM_SECTION

Important Security Notes

Security attributes for a Galaxywide memory section must appear identical to a process no matter on what instance it is executing.

This can be achieved by having all instances participating in this sharing community also participate in a "homogeneous" OpenVMS Cluster, where all nodes share the security-related files:

SYSUAF.DAT, SYSUAFALT.DAT (system authorization file)
RIGHTSLIST.DAT (rights database)
VMS$OBJECTS.DAT (objects database)

In particular, automatic propagation of protection changes to a Galaxywide section requires that the same physical file (VMS$OBJECTS.DAT) is used by all sharing instances.

If your installation does not share these files throughout the Galaxy, the creator of a Galaxywide shared section must ensure that the section has the same security attributes on each instances. This may require manual intervention.

17
OpenVMS Galaxy Device Drivers

This chapter describes OpenVMS Alpha Version 7.3 direct-mapped DMA window information for PCI drivers.

17.1 Direct-Mapped DMA Window Changes

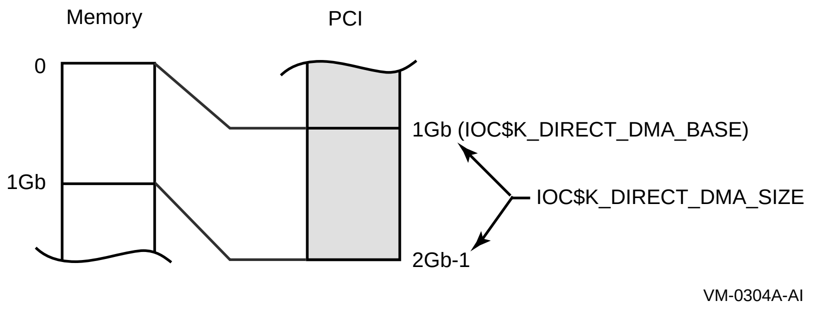
The changes described in this chapter were made in OpenVMS Version 7.2 to support OpenVMS Galaxy and memory holes. The change involves moving the direct-mapped DMA window away from physical memory location 0. This chapter should provide enough background and information for you to update your driver if you have not yet updated it to OpenVMS Version 7.2 or later.

Note that this chapter does not cover bus-addressable pool (BAP).

17.2 How PCI Direct-Mapped DMA Works Prior to OpenVMS Version 7.2

On all PCI-based machines, the direct-mapped DMA window begins at (usually) 1 Gb in PCI space and covers physical memory beginning at 0 for 1 Gb as shown in Figure 17–1.

Figure 17–1 PCI-Based DMA



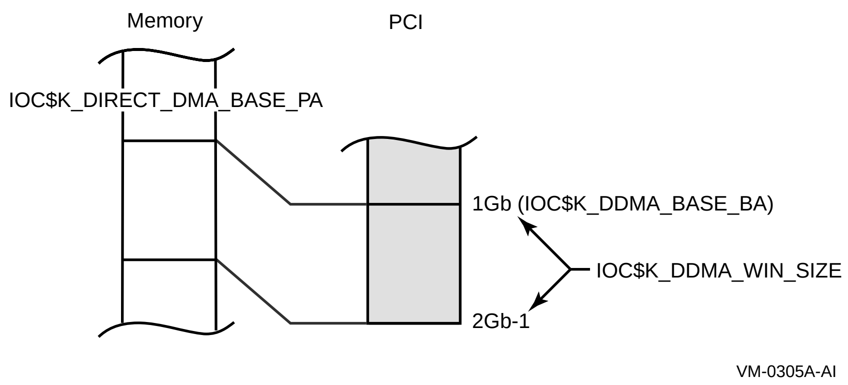
Typically drivers compare their buffer addresses against the length of the window returned by calling IOC$NODE_DATA with the IOC$K_DIRECT_DMA_SIZE function code. This assumes that the window on the memory side starts at zero. Another popular method for determining whether map registers are necessary involves looking at MMG$GL_MAXPFN. This is also not likely to work correctly in OpenVMS Version 7.3.

For a much better picture and explanation, see the *Writing OpenVMS Device Alpha Drivers in C* book.

17.3 How PCI Direct-Mapped DMA Works in Current Versions of OpenVMS

Galaxy and memory-hole considerations force OpenVMS to change the placement of the direct-mapped DMA window, as shown in Figure 17–2.

Figure 17–2 OpenVMS DMA



It is unknown from the drivers perspective where in memory the base of the direct-mapped DMA window will be. Simply comparing a buffer address against the length of the window will no longer be sufficient to determine whether a buffer is within the direct-mapped DMA window. Also, comparing against MMG$GL_MAXPFN will no longer guarantee that all of pool is within the window. The correct cell to check is MMG$GL_MAX_NODE_PFN. additionally, alignment concerns may require that a slightly different offset be incorporated into physical bus address calculations.

17.4 IOC$NODE_DATA Changes to Support Nonzero Direct-Mapped DMA Windows

To alleviate this problem, new function codes have been added to IOC$NODE_DATA. Here is a list of all the codes relating to direct-mapped DMA, and a description of what the data means.

IOC$K_DIRECT_DMA_BASE	This is the base address on the PCI side, or bus address. There is a synonym for this function code called IOC$K_DDMA_BASE_BA. A 32-bit result will be returned.
IOC$DIRECT_DMA_SIZE	On non-Galaxy machines, this returns the size of the direct-mapped DMA window (in megabytes). On a system where the direct-mapped DMA window does not start at zero, the data returned is zero, implying that no direct-mapped DMA windows exist. A 32-bit result will be returned.
IOC$K_DDMA_WIN_SIZE	On all systems, this will always return the size of the direct-mapped DMA window (in megabytes). A 32-bit result will be returned.
IOC$K_DIRECT_DMA_BASE_PA	This is the base physical address in memory of the direct-mapped DMA window. A 32-bit result will be returned.

The address returned with the IOC$K_DIRECT_DMA_BASE_PA code is necessary to compute the offset. (This usually used to be the 1 Gb difference between the memory PA and the bus address.) The offset is defined as the signed difference between the base bus address and the base memory address. This is now not necessarily 1 Gb.

A
OpenVMS Galaxy CPU Load Balancer Program

This appendix contains an example program of a privileged-code application that dynamically reassigns CPU resources among instances in an OpenVMS Galaxy.

A.1 CPU Load Balancer Overview

The OpenVMS Galaxy CPU Load Balancer program is a privileged application that dynamically reassigns CPU resources among instances in an OpenVMS Galaxy.

The program must be run on each participating instance. Each image will create, or map to, a small shared-memory section and periodically post information regarding the depth of that instance's COM queues. Based upon running averages of this data, each instance will determine the most and the least busy instances. If these factors exist for a specified duration, the least busy instance having available secondary processors will reassign one of its processors to the most busy instance, thereby effectively balancing processor usage across the OpenVMS Galaxy. The program provides command-line arguments to allow tuning of the load-balancing algorithm. The program is admittedly shy on error handling.

This program uses the following OpenVMS Galaxy system services:

SYS$CPU_TRANSITION	CPU reassignment
SYS$CRMPSC_GDZRO_64	Shared memory creation
SYS$SET_SYSTEM_EVENT	OpenVMS Galaxy event notification
SYS$*_GALAXY_LOCK_*	OpenVMS Galaxy locking

Because OpenVMS Galaxy resources are always reassigned via a push model, where only the owner instance can release its resources, one copy of this process must run on each instance in the OpenVMS Galaxy.

This program can be run only in an OpenVMS Version 7.2 or later multiple-instance Galaxy.

A.1.1 Required Privileges

The CMKRNL privilege is required to count CPU queues. The SHMEM privilege is required to map shared memory.

A.1.2 Build and Copy Instructions

Compile and link the example program as described below, or copy the precompiled image found in SYS$EXAMPLES:GCU$BALANCER.EXE to SYS$COMMON:[SYSEXE]GCU$BALANCER.EXE.

If your OpenVMS Galaxy instances use individual system disks, you will need to perform this action for each instance.

If you change the example program, compile and link it as follows:

```
$ CC GCU$BALANCER.C+SYS$LIBRARY:SYS$LIB_C/LIBRARY
$ LINK/SYSEXE GCU$BALANCER
```

A.1.3 Startup Options

You must establish a DCL command for this program. We have provided a sample command table file for this purpose. To install the new command, do the following:

```
$ SET COMMAND/TABLE=SYS$LIBRARY:DCLTABLES -
_$ /OUT=SYS$COMMON:[SYSLIB]DCLTABLES GCU$BALANCER.CLD
```

This command inserts the new command definition into DCLTABLES.EXE in your common system directory. The new command tables will take effect when the system is rebooted. If you would like to avoid a reboot, do the following:

```
$ INSTALL REPLACE SYS$COMMON:[SYSLIB]DCLTABLES.EXE
```

After this command, you will need to log out, then log back in to use the command from any active processes. Alternatively, if you would like to avoid logging out, do the following from each process you would like to run the balancer from:

```
$ SET COMMAND GCU$BALANCER.CLD
```

Once your command has been established, you may use the various command line parameters to control the balancer algorithm.

```
$ CONFIGURE BALANCER[/STATISTICS] x y time
```

In this command, *x* is the number of load samples to take, *y* is the number of queued processes required to trigger resource reassignment, and *time* is the delta time between load sampling.

The /STATISTICS qualifier causes the program to display a continuous status line. This is useful for tuning the parameters. This output is not visible if the balancer is run detached, as is the case if it is invoked via the GCU. The /STATISTICS qualifier is intended to be used only when the balancer is invoked directly from DCL in a DECterm window. For example:

```
$ CONFIG BAL 3 1 00:00:05.00
```

Starts the balancer which samples the system load every 5 seconds. After three samples, if the instance has one or more processes in the COM queue, a resource (CPU) reassignment will occur, giving this instance another CPU.

A.1.4 Starting the Load Balancer from the GCU

The GCU provides a menu item for launching SYS$SYSTEM:GCU$BALANCER.EXE and a dialog for altering the balancer algorithm. These features will only work if the balancer image is properly installed as described the following paragraphs.

To use the GCU-resident balancer startup option, you must:

- Compile, link, or copy the balancer image as described previously.
- Invoke the GCU using the following command:

  ```
  $ CONFIGURE GALAXY
  ```

  You might need to set your DECwindows display to a suitably configured workstation or PC.

- Choose the CPU Balancer item from the Galaxy menu.
- Select appropriate values for your system. This can take some testing. By default, the values are set aggressively so that the balancer action can be readily observed. If your system is very heavily loaded, you will need to increase the values accordingly to avoid excessive resource reassignment. The GCU does not currently save these values, so you may want to write them down once you are satisfied.
- Select the instances you want to have participate, then select the Start function, then click on **OK**. The GCU should launch the process GCU$BALANCER on all selected instances. You might want to verify that these processes have been started.

A.1.5 Shutdown Warning

In an OpenVMS Galaxy, no process may have shared memory mapped on an instance when it leaves the Galaxy—for example, during a shutdown. To stop the process if the GCU$BALANCER program is run from a SYSTEM UIC, you must modify SYS$MANAGER:SYSHUTDWN.COM. Processes in the SYSTEM UIC group are not terminated by SHUTDWN.COM when shutting down or rebooting OpenVMS. If a process still has shared memory mapped when an instance leaves the Galaxy, the instance will crash with a GLXSHUTSHMEM bugcheck.

To make this work, SYS$MANAGER:SYSHUTDWN.COM must stop the process as shown in the following example. Alternatively, the process can be run under a suitably privileged, non-SYSTEM UIC.

```
** SYSHUTDWN.COM EXAMPLE - Paste into SYS$MANAGER:SYSHUTDWN.COM
**
**    $!
**    $! If the GCU$BALANCER image is running, stop it to release shmem.
**    $!
**    $ procctx = f$context("process",ctx,"prcnam","GCU$BALANCER","eql")
**    $ procid  = f$pid(ctx)
**    $ if procid .NES. "" then $ stop/id='procid'
```

Note that you could also use a `$ STOP GCU$BALANCER` statement.

A.2 Example Program

```
/*
** COPYRIGHT (c) 1998 BY COMPAQ COMPUTER CORPORATION ALL RIGHTS RESERVED.
**
** THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
** ONLY  IN  ACCORDANCE  OF  THE  TERMS  OF  SUCH  LICENSE  AND WITH THE
** INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR  ANY  OTHER
** COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
** OTHER PERSON.  NO TITLE TO AND  OWNERSHIP OF THE  SOFTWARE IS  HEREBY
** TRANSFERRED.
**
** THE INFORMATION IN THIS SOFTWARE IS  SUBJECT TO CHANGE WITHOUT NOTICE
** AND  SHOULD  NOT  BE  CONSTRUED  AS A COMMITMENT BY COMPAQ COMPUTER
** CORPORATION.
**
** COMPAQ ASSUMES NO RESPONSIBILITY FOR THE USE  OR  RELIABILITY OF ITS
** SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY COMPAQ OR DIGITAL.
**
**=====================================================================
** WARNING - This example is provided for instructional and demo
**           purposes only.  The resulting program should not be
**           run on systems which make use of soft-affinity
**           features of OpenVMS, or while running applications
**           which are tuned for precise processor configurations.
**           We are continuing to explore enhancements such as this
**           program which will be refined and integrated into
**           future releases of OpenVMS.
**=====================================================================
**
** GCU$BALANCER.C - OpenVMS Galaxy CPU Load Balancer.
**
** This is an example of a privileged application which dynamically
** reassigns CPU resources among instances in an OpenVMS Galaxy.  The
** program must be run on each participating instance.  Each image
** will create, or map to, a small shared memory section and periodically
** post information regarding the depth of that instances' COM queues.
** Based upon running averages of this data, each instance will
** determine the most, and least busy instance.  If these factors
** exist for a specified duration, the least busy instance having
** available secondary processors, will reassign one of its processors
** to the most busy instance, thereby effectively balancing processor
** usage across the OpenVMS Galaxy.  The program provides command line
** arguments to allow tuning of the load balancing algorithm.
** The program is admittedly shy on error handling.
**
** This program uses the following OpenVMS Galaxy system services:
**
**      SYS$CPU_TRANSITION   - CPU reassignment
**      SYS$CRMPSC_GDZRO_64  - Shared memory creation
**      SYS$SET_SYSTEM_EVENT - OpenVMS Galaxy event notification
**      SYS$*_GALAXY_LOCK_*  - OpenVMS Galaxy locking
**
** Since OpenVMS Galaxy resources are always reassigned via a "push"
** model, where only the owner instance can release its resources,
** one copy of this process must run on each instance in the OpenVMS
** Galaxy.
**
** ENVIRONMENT: OpenVMS V7.2 Multiple-instance Galaxy.
**
** REQUIRED PRIVILEGES:  CMKRNL required to count CPU queues
**                       SHMEM  required to map shared memory
**
** BUILD/COPY INSTRUCTIONS:
**
```

```
** Compile and link the example program as described below, or copy the
** precompiled image found in SYS$EXAMPLES:GCU$BALANCER.EXE to
** SYS$COMMON:[SYSEXE]GCU$BALANCER.EXE
**
** If your OpenVMS Galaxy instances utilize individual system disks, you
** will need to do the above for each instance.
**
** If you change the example program, compile and link it as follows:
**
**   $ CC GCU$BALANCER.C+SYS$LIBRARY:SYS$LIB_C/LIBRARY
**   $ LINK/SYSEXE GCU$BALANCER
**
** STARTUP OPTIONS:
**
** You must establish a DCL command for this program.  We have provided a
** sample command table file for this purpose.  To install the new command,
** do the following:
**
**    $ SET COMMAND/TABLE=SYS$LIBRARY:DCLTABLES -
**      /OUT=SYS$COMMON:[SYSLIB]DCLTABLES GCU$BALANCER.CLD
**
** This command inserts the new command definition into DCLTABLES.EXE
** in your common system directory.  The new command tables will take
** effect when the system is rebooted.  If you would like to avoid a
** reboot, do the following:
**
**    $ INSTALL REPLACE SYS$COMMON:[SYSLIB]DCLTABLES.EXE
**
** After this command, you will need to log out, then log back in to
** use the command from any active processes.  Alternatively, if you
** would like to avoid logging out, do the following from each process
** you would like to run the balancer from:
**
**    $ SET COMMAND GCU$BALANCER.CLD
**
** Once your command has been established, you may use the various
** command line parameters to control the balancer algorithm.
**
**    $ CONFIGURE BALANCER{/STATISTICS} x y time
**
** Where: "x" is the number of load samples to take.
**        "y" is the number of queued processes required to trigger
**            resource reassignment.
**        "time" is the delta time between load sampling.
**
** The /STATISTICS qualifier causes the program to display a
** continuous status line.  This is useful for tuning the parameters.
** This output is not visible if the balancer is run detached, as is
** the case if it is invoked via the GCU.  It is intended to be used
** only when the balancer is invoked directly from DCL in a DECterm
** window.
**
** For example: $ CONFIG BAL 3 1 00:00:05.00
**
**        Starts the balancer which samples the system load every
**        5 seconds.  After 3 samples, if the instance has one or
**        more processes in the COM queue, a resource (CPU)
**        reassignment will occur, giving this instance another CPU.
**
** GCU STARTUP:
**
** The GCU provides a menu item for launching SYS$SYSTEM:GCU$BALANCER.EXE
** and a dialog for altering the balancer algorithm.  These features will
** only work if the balancer image is properly installed as described
** the the following paragraphs.
```

```
**
** To use the GCU-resident balancer startup option, you must:
**
** 1) Compile, link, or copy the balancer image as described previously.
** 2) Invoke the GCU via: $ CONFIGURE GALAXY   You may need to set your
**    DECwindows display to a suitably configured workstation or PC.
** 3) Select the "CPU Balancer" entry from the "Galaxy" menu.
** 4) Select appropriate values for your system.  This may take some
**    testing.  By default, the values are set aggressively so that
**    the balancer action can be readily observed.  If your system is
**    very heavily loaded, you will need to increase the values
**    accordingly to avoid excessive resource reassignment.  The GCU
**    does not currently save these values, so you may want to write
**    them down once you are satisfied.
** 5) Select the instance/s you wish to have participate, then select
**    the "Start" function, then press OK.  The GCU should launch the
**    process GCU$BALANCER on all selected instances.  You may want to
**    verify these processes have been started.
**
** SHUTDOWN WARNING:
**
** In an OpenVMS Galaxy, no process may have shared memory mapped on an
** instance when it leaves the Galaxy, as during a shutdown. Because of
** this, SYS$MANAGER:SYSHUTDWN.COM must be modified to stop the process
** if the GCU$BALANCER program is run from a SYSTEM UIC.  Processes in the
** SYSTEM UIC group are not terminated by SHUTDOWN.COM when shutting down
** or rebooting OpenVMS. If a process still has shared memory mapped when
** an instance leaves the Galaxy, the instance will crash with a
** GLXSHUTSHMEM bugcheck.
**
** To make this work, SYS$MANAGER:SYSHUTDWN.COM must stop the process as
** shown in the example below.  Alternatively, the process can be run
** under a suitably privileged, non-SYSTEM UIC.
**
** SYSHUTDWN.COM EXAMPLE - Paste into SYS$MANAGER:SYSHUTDWN.COM
**
**    $!
**    $! If the GCU$BALANCER image is running, stop it to release shmem.
**    $!
**    $ procctx = f$context("process",ctx,"prcnam","GCU$BALANCER","eql")
**    $ procid  = f$pid(ctx)
**    $ if procid .NES. "" then $ stop/id='procid'
**
** Note, you could also just do a "$ STOP GCU$BALANCER" statement.
**
** OUTPUTS:
**
**    If the logical name GCU$BALANCER_VERIFY is defined, notify the
**    SYSTEM account when CPUs are reassigned.  If the /STATISTICS
**    qualifier is specified, a status line is continually displayed,
**    but only when run directly from the command line.
**
** REVISION HISTORY:
**
** 02-Dec-1998 Greatly improved instructions.
** 03-Nov-1998 Improved instructions.
** 24-Sep-1998 Initial code example and integration with GCU.
*/
#include <BRKDEF>
#include <BUILTINS>
#include <CSTDEF>
#include <DESCRIP>
#include <GLOCKDEF>
#include <INTS>
#include <PDSCDEF>
```

```
#include <PSLDEF>
#include <SECDEF>
#include <SSDEF>
#include <STARLET>
#include <STDIO>
#include <STDLIB>
#include <STRING>
#include <SYIDEF>
#include <SYSEVTDEF>
#include <VADEF>
#include <VMS_MACROS>
#include <CPUDEF>
#include <IOSBDEF.H>
#include <EFNDEF.H>
/* For CLI */
#include <cli$routines.h>
#include <chfdef.h>
#include <climsgdef.h>

#define HEARTBEAT_RESTART     0 /* Flags for synchronization            */
#define HEARTBEAT_ALIVE       1
#define HEARTBEAT_TRANSPLANT  2

#define GLOCK_TIMEOUT    100000 /* Sanity check, max time holding gLock */
#define _failed(x) (!((x) & 1))

$DESCRIPTOR(system_dsc, "SYSTEM");              /* Brkthru account name     */
$DESCRIPTOR(gblsec_dsc, "GCU$BALANCER");        /* Global section name      */

struct  SYI_ITEM_LIST {                         /* $GETSYI item list format */
  short buflen,item;
  void *buffer,*length;
};

/* System information and an item list to use with $GETSYI */

static unsigned long total_cpus;
static uint64   partition_id;
static long     max_instances = 32;
iosb            g_iosb;

struct SYI_ITEM_LIST syi_itemlist[3] = {
     {sizeof (long), SYI$_ACTIVECPU_CNT,&total_cpus,  0},
     {sizeof (long), SYI$_PARTITION_ID, &partition_id,0},
     {0,0,0,0}};

extern uint32 *SCH$AQ_COMH;             /* Scheduler COM queue address  */
unsigned long PAGESIZE;                 /* Alpha page size              */
uint64       glock_table_handle;        /* Galaxy lock table handle     */

/*
** Shared Memory layout (64-bit words):
** ===================================
** 0  to  n-1:  Busy count, where 100 = 1 process in a CPU queue
** n  to  2n-1: Heartbeat (status) for each instance
** 2n to  3n-1: Current CPU count on each instance
** 3n to  4n-1: Galaxy lock handles for modifying heartbeats
**
** where n = max_instances * sizeof(long).
**
** We assume the entire table (easily) fits in two Alpha pages.
*/

/* Shared memory pointers must be declared volatile */

volatile uint64  gs_va = 0;             /* Shmem section address        */
volatile uint64  gs_length = 0;         /* Shmem section length         */
volatile uint64 *gLocks;                /* Pointers to gLock handles    */
volatile uint64 *busycnt,*heartbeat,*cpucount;
```

```
/*************************************************************************/
/* FUNCTION init_lock_tables - Map to the Galaxy locking table and      */
/* create locks if needed. Place the lock handles in a shared memory    */
/* region, so all processes can access the locks.                       */
/*                                                                      */
/* ENVIRONMENT: Requires SHMEM and CMKRNL to create tables.             */
/* INPUTS:      None.                                                   */
/* OUTPUTS:     Any errors from lock table creation.                    */
/*************************************************************************/
int init_lock_tables (void)
{
    int status,i;
    unsigned long sanity;
    uint64 handle;
    unsigned int min_size, max_size;

    /* Lock table names are 15-byte padded values, unique across a Galaxy. */
    char table_name[] = "GCU_BAL_GLOCK  ";

    /* Lock names are 15-byte padded values, but need not be unique. */
    char lock_name[] = "GCU_BAL_LOCK   ";

    /* Get the size of a Galaxy lock */
    status = sys$get_galaxy_lock_size(&min_size,&max_size);
    if (_failed(status)) return (status);

    /*
    ** Create or map to a process space Galaxy lock table. We assume
    ** one page is enough to hold the locks. This will work for up
    ** to 128 instances.
    */
    status = sys$create_galaxy_lock_table(table_name,PSL$C_USER,
                PAGESIZE,GLCKTBL$C_PROCESS,0,min_size,&glock_table_handle);
    if (_failed(status)) return (status);

    /*
    ** Success case 1: SS$_CREATED
    ** We created the table, so  populate it with locks and
    ** write the handles to shared memory so the other partitions
    ** can access them. Only one instance can receive SS$_CREATED
    ** for a given lock table; all other mappers will get SS$_NORMAL.
    */
    if (status == SS$_CREATED)
    {
      printf ("%%GCU$BALANCER-I-CRELOCK, Creating G-locks\n");
      for (i=0; i<max_instances; i++)
      {
        status = sys$create_galaxy_lock(glock_table_handle,lock_name,
                   min_size,0,0,0,&handle);
        gLocks[i] = handle;
        if (_failed(status)) return (status);
      }
    }
    else
    {
    /*
    ** Success case 2: SS$_NORMAL
    ** We mapped the table, but did not create it. Spin until
    ** the creator fills the lock handles. NOTE: If the creator
    ** fails in the loop above and does not finish creating the
    ** locks, then we will be stuck waiting forever - so we
    ** use a sanity check here. Process space lock tables and
    ** memory regions are automatically deleted when all
    ** processes mapping them are deleted, so the worst case
    ** is this:
    **
    ** - Process 1 starts, creates gLock table
    ** - Processes 2-n start and are waiting on gLock creation
    ** - Process 1 dies before completing gLock creation
    ** - Processes 2-n time out and exit; the half-initialized
    **   section and lock tables are deleted by VMS.
    ** - The user (or script) receives SS$_TIMEOUT and can
    **   now restart all processes with a "clean slate".
    */
      sanity = 0;
      printf ("%%GCU$BALANCER-I-WAITLOCK, Waiting for G-lock creation...\n");
      while (gLocks[max_instances-1] == 0)
      {
        if (sanity++ > 1000000) return (SS$_TIMEOUT);
      }
    }
    return (SS$_NORMAL);
}

/************************************************************************/
/* FUNCTION update_cpucount - Update the number of CPUs in this instance*/
/*                                                                      */
/* ENVIRONMENT: Called directly or via a system event AST.              */
/* INPUTS:      None.                                                   */
/* OUTPUTS:     Updates this instance’s CPU count in shared memory.     */
/************************************************************************/
void update_cpucount(int unused)
{
   sys$getsyiw(EFN$C_ENF,0,0,&syi_itemlist,&g_iosb,0,0);
   cpucount[partition_id] = total_cpus;
}

/************************************************************************/
/* FUNCTION cpu_q - Count the number of processes in CPU COM queues     */
/*                                                                      */
/* ENVIRONMENT: OpenVMS Kernel Mode.                                    */
/* INPUTS:      None.                                                   */
/* OUTPUTS:     Returns the number of processes on the COM queues.      */
/************************************************************************/
long cpu_q(void)
{
   uint32 *head, *tmp;
   long procs = 0;
   int p;

   head = SCH$AQ_COMH;          /* Head of 1st COM queue                */
   sys_lock(SCHED,1,0);         /* Obtain SCHED spinlock                */

   for (p=64; p>0; p--)         /* Queues to scan (32 COM + 32 COMO)    */
   {
     tmp = (uint32 *) *head;    /* Look at first flink                  */
     while (tmp != head)        /* Compare vs. head of queue            */
     {
       procs++;                 /* Different, count a job waiting       */
       tmp = (uint32 *) *tmp;   /* Go to next queue entry               */
     }
     head = head + 2;           /* Go to next queue (increment by 2*32) */
   }                            /* And scan it (loop to "for p...")     */

   sys_unlock(SCHED,0,0);       /* Release SCHED spinlock               */
   return procs;
}
```

```
/*********************************************************************/
/* FUNCTION lockdown - Lock the cpu_q routine into the working set      */
/*                    so that it can't pagefault while at elevated IPL  */
/*                                                                      */
/* ENVIRONMENT: Requires CMKRNL privilege.                              */
/* INPUTS:      None.                                                   */
/* OUTPUTS:     None.                                                   */
/*********************************************************************/
void lockdown(void)
{
   struct pdscdef *proc_desc = (void *)cpu_q;
   unsigned long sub_addr[2], locked_head[2], locked_code[2];
   unsigned long status;

   sub_addr[0] = (unsigned long) cpu_q;
   sub_addr[1] = sub_addr[0] + PAGESIZE;
   if (__PAL_PROBER((void *)sub_addr[0],sizeof(int),PSL$C_USER) != 0)
     sub_addr[1] = sub_addr[0];

   status = sys$lkwset(sub_addr,locked_head,PSL$C_USER);
   if (_failed(status)) exit(status);

   sub_addr[0] = proc_desc->pdsc$q_entry[0];
   sub_addr[1] = sub_addr[0] + PAGESIZE;
   if (__PAL_PROBER((void *)sub_addr[0],sizeof(int),PSL$C_USER) != 0)
        sub_addr[1] = sub_addr[0];

   status = sys$lkwset(sub_addr,locked_code,PSL$C_USER);
   if (_failed(status)) exit(status);
}

/*********************************************************************/
/* FUNCTION reassign_a_cpu - Reassign a single CPU to another instance. */
/*                                                                      */
/* ENVIRONMENT: Requires CMKRNL privilege.                              */
/* INPUTS:      most_busy_id: partition ID of destination.              */
/* OUTPUTS:     None.                                                   */
/*                                                                      */
/* Donate one CPU at a time - then wait for the remote instance to      */
/* reset its heartbeat and recalculate its load.                        */
/*********************************************************************/
void reassign_a_cpu(int most_busy_id)
{
  int status,i;
  static char op_msg[255];
  static char iname_msg[1];
  $DESCRIPTOR(op_dsc,op_msg);
  $DESCRIPTOR(iname_dsc,"");
  iname_dsc.dsc$w_length = 0;

  /* Update CPU info */

  status = sys$getsyiw(EFN$C_ENF,0,0,&syi_itemlist,&g_iosb,0,0);
  if (_failed(status)) exit(status);

  /* Don't attempt reassignment if we are down to one CPU */

  if (total_cpus > 1)
  {
    status = sys$acquire_galaxy_lock(gLocks[most_busy_id],GLOCK_TIMEOUT,0);
    if (_failed(status)) exit(status);
    heartbeat[most_busy_id] = HEARTBEAT_TRANSPLANT;
    status = sys$release_galaxy_lock(gLocks[most_busy_id]);
    if (_failed(status)) exit(status);
    status = sys$cpu_transitionw(CST$K_CPU_MIGRATE,CST$K_ANY_CPU,0,
                                 most_busy_id,0,0,0,0,0,0);
    if (status & 1)
    {
      if (getenv ("GCU$BALANCER_VERIFY"))
      {
        sprintf(op_msg,
                "\n\n*****GCU$BALANCER: Reassigned a CPU to instance %li\n",
                most_busy_id);
        op_dsc.dsc$w_length = strlen(op_msg);
        sys$brkthru(0,&op_dsc,&system_dsc,BRK$C_USERNAME,0,0,0,0,0,0,0);
      }
      update_cpucount(0);  /* Update the CPU count after donating one */
    }
  }
}

/**********************************************************************/
/* IMAGE ENTRY - MAIN                                                 */
/*                                                                    */
/* ENVIRONMENT: OpenVMS Galaxy                                        */
/* INPUTS:      None.                                                 */
/* OUTPUTS:     None.                                                 */
/**********************************************************************/
int main(int argc, char **argv)
{
   int           show_stats = 0;
   long          busy,most_busy,nprocs;
   int64         delta;
   unsigned long status,i,j,k,system_cpus,instances;
   unsigned long arglst        = 0;
   uint64        version_id[2]  = {0,1};
   uint64        region_id      = VA$C_P0;
   uint64        most_busy_id,cpu_hndl = 0;

/* Static descriptors for storing parameters.  Must match CLD defs */

   $DESCRIPTOR(p1_desc,"P1");
   $DESCRIPTOR(p2_desc,"P2");
   $DESCRIPTOR(p3_desc,"P3");
   $DESCRIPTOR(p4_desc,"P4");
   $DESCRIPTOR(stat_desc,"STATISTICS");

/* Dynamic descriptors for retrieving parameter values */

   struct dsc$descriptor_d samp_desc = {0,DSC$K_DTYPE_T,DSC$K_CLASS_D,0};
   struct dsc$descriptor_d proc_desc = {0,DSC$K_DTYPE_T,DSC$K_CLASS_D,0};
   struct dsc$descriptor_d time_desc = {0,DSC$K_DTYPE_T,DSC$K_CLASS_D,0};

   struct SYI_ITEM_LIST syi_pagesize_list[3] = {
     {sizeof (long), SYI$_PAGE_SIZE      ,&PAGESIZE     ,0},
     {sizeof (long), SYI$_GLX_MAX_MEMBERS,&max_instances,0},
     {0,0,0,0}};
/*
** num_samples and time_desc determine how often the balancer should check
** to see if any other instance needs more CPUs. num_samples determines the
** number of samples used to calculate the running average, and sleep_dsc
** determines the amount of time between samples.
**
** For example, a sleep_dsc of 30 seconds and a num_samples of 20 means that
** a running average over the last 10 minutes (20 samples * 30 secs) is used
** to balance CPUs.
**
** load_tolerance is the minimum load difference which triggers a CPU
** migration. 100 is equal to 1 process in the computable CPU queue.
*/
   int num_samples;     /* Number of samples in running average      */
   int load_tolerance;  /* Minimum load diff to trigger reassignment */
```

```
/* Parse the CLI */
                                                /* CONFIGURE VERB */
   status      = CLI$PRESENT(&p1_desc);         /* BALANCER       */
   if (status != CLI$_PRESENT) exit(status);
   status      = CLI$PRESENT(&p2_desc);         /* SAMPLES        */
   if (status != CLI$_PRESENT) exit(status);
   status      = CLI$PRESENT(&p3_desc);         /* PROCESSES      */
   if (status != CLI$_PRESENT) exit(status);
   status      = CLI$PRESENT(&p4_desc);         /* TIME           */
   if (status != CLI$_PRESENT) exit(status);

   status     = CLI$GET_VALUE(&p2_desc,&samp_desc);
   if (_failed(status)) exit(status);
   status     = CLI$GET_VALUE(&p3_desc,&proc_desc);
   if (_failed(status)) exit(status);
   status     = CLI$GET_VALUE(&p4_desc,&time_desc);
   if (_failed(status)) exit(status);
   status     = CLI$PRESENT(&stat_desc);
   show_stats = (status == CLI$_PRESENT) ? 1 : 0;

   num_samples = atoi(samp_desc.dsc$a_pointer);
   if (num_samples <= 0) num_samples = 3;

   load_tolerance = (100 * (atoi(proc_desc.dsc$a_pointer)));
   if (load_tolerance <= 0) load_tolerance = 100;

   if (show_stats)
     printf("Args: Samples: %d, Processes: %d, Time: %s\n",
        num_samples,load_tolerance/100,time_desc.dsc$a_pointer);

   lockdown();                  /* Lock down the cpu_q subroutine */

   /* Get the page size and max members for this system */

   status = sys$getsyiw(EFN$C_ENF,0,0,&syi_pagesize_list,&g_iosb,0,0);
   if (_failed(status)) return (status);

   if (max_instances == 0) max_instances = 1;

   /* Get our partition ID and initial CPU info */

   status = sys$getsyiw(EFN$C_ENF,0,0,&syi_itemlist,&g_iosb,0,0);
   if (_failed(status)) return (status);

   /* Map two pages of shared memory */

   status = sys$crmpsc_gdzro_64(&gblsec_dsc,version_id,0,PAGESIZE+PAGESIZE,
              &region_id,0,PSL$C_USER,(SEC$M_EXPREG|SEC$M_SYSGBL|SEC$M_SHMGS),
              &gs_va,&gs_length);
   if (_failed(status)) exit(status);

   /* Initialize the pointers into shared memory */

   busycnt   = (uint64 *) gs_va;
   heartbeat = (uint64 *) gs_va     + max_instances;
   cpucount  = (uint64 *) heartbeat + max_instances;
   gLocks    = (uint64 *) cpucount  + max_instances;

   cpucount[partition_id] = total_cpus;

   /* Create or map the Galaxy lock table */

   status = init_lock_tables();
   if (_failed(status)) exit(status);

   /* Initialize delta time for sleeping */

   status = sys$bintim(&time_desc,&delta);
   if (_failed(status)) exit(status);
```

```
/*
** Register for CPU migration events. Whenever a CPU is added to
** our active set, the routine "update_cpucount" will fire.
*/
status = sys$set_system_event(SYSEVT$C_ADD_ACTIVE_CPU,
           update_cpucount,0,0,SYSEVT$M_REPEAT_NOTIFY,&cpu_hndl);
if (_failed(status)) exit(status);

/* Force everyone to resync before we do anything */

for (j=0; j<max_instances; j++)
{
  status = sys$acquire_galaxy_lock(gLocks[j],GLOCK_TIMEOUT,0);
  if (_failed(status)) exit(status);
  heartbeat[j] = HEARTBEAT_RESTART;
  status = sys$release_galaxy_lock (gLocks[j]);
  if (_failed(status)) exit(status);
}

printf("%%GCU$BALANCER-S-INIT, CPU balancer initialized.\n\n");

/*** Main loop ***/
do
{
  /* Calculate a running average and update it */

  nprocs = sys$cmkrnl(cpu_q,&arglst) * 100;

  /* Check out our state... */

  switch (heartbeat[partition_id])
  {
    case HEARTBEAT_RESTART: /* Mark ourself for reinitializition. */
    {
      update_cpucount(0);
      status = sys$acquire_galaxy_lock(gLocks[partition_id],GLOCK_TIMEOUT,0);
      if (_failed(status)) exit(status);
      heartbeat[partition_id] = HEARTBEAT_ALIVE;
      status = sys$release_galaxy_lock(gLocks[partition_id]);
      if (_failed(status)) exit(status);
      break;
    }
    case HEARTBEAT_ALIVE: /* Update running average and continue. */
    {
      busy = (busycnt[partition_id]*(num_samples-1)+nprocs)/num_samples;
      busycnt[partition_id] = busy;
      break;
    }
    case HEARTBEAT_TRANSPLANT:  /* Waiting for a new CPU to arrive. */
    {
      /*
      ** Someone just either reset us, or gave us a CPU and put a wait on
      ** further donations.  Reassure the Galaxy that we're alive, and
      ** calculate a new busy count.
      */
      busycnt[partition_id] = nprocs;
      status = sys$acquire_galaxy_lock(gLocks[partition_id],GLOCK_TIMEOUT,0);
      if (_failed(status)) exit(status);
      heartbeat[partition_id] = HEARTBEAT_ALIVE;
      status = sys$release_galaxy_lock(gLocks[partition_id]);
      if (_failed(status)) exit(status);
      break;
    }
    default:         /* This should never happen. */
    {
      exit(0);
      break;
    }
```

```
     }

     /* Determine the most_busy instance. */

     for (most_busy_id=most_busy=i=0; i<max_instances; i++)
     {
       if (busycnt[i] > most_busy)
       {
         most_busy_id = (uint64) i;
         most_busy    = busycnt[i];
       }
     }

     if (show_stats)
       printf("Current Load: %3Ld, Busiest Instance: %Ld, Queue Depth: %4d\r",
             busycnt[partition_id],most_busy_id,(nprocs/100));

     /* If someone needs a CPU and we have an extra, dontate it. */

     if ((most_busy > busy + load_tolerance) &&
         (cpucount[partition_id] > 1) &&
         (heartbeat[most_busy_id] != HEARTBEAT_TRANSPLANT) &&
         (most_busy_id != partition_id))
     {
        reassign_a_cpu(most_busy_id);
     }

     /* Hibernate for a while and do it all again. */

     status = sys$schdwk(0,0,&delta,0);
     if (_failed(status)) exit(status);
     status = sys$hiber();
     if (_failed(status)) exit(status);

   } while (1);
   return (1);
}
```

B
Common Values for Environment Variables

Table B–1 lists common values for Galaxy environment variables. All values are expressed in hexadecimal bytes.

Table B–1 Common Values for Environment Variables

1 Megabyte	0x 10 0000
2 Megabytes	0x 20 0000
4 Megabytes	0x 40 0000
8 Megabytes	0x 80 0000
16 Megabytes	0x 100 0000
32 Megabytes	0x 200 0000
64 Megabytes	0x 400 0000
128 Megabytes	0x 800 0000
256 Megabytes	0x 1000 0000
448 Megabytes	0x1C00 0000
512 Megabytes	0x 2000 0000
1 Gigabyte	0x 4000 0000
2 Gigabytes	0x 8000 0000
4 Gigabytes	0x 1 0000 0000
8 Gigabytes	0x 2 0000 0000
16 Gigabytes	0x 4 0000 0000
32 Gigabytes	0x 8 0000 0000
64 Gigabytes	0x 10 0000 0000
128 Gigabytes	0x 20 0000 0000
256 Gigabytes	0x 40 0000 0000
512 Gigabytes	0x 80 0000 0000
1 Terabyte	0x 100 0000 0000

Index