ChorusOS 4.0 Hot Restart Programmer's Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 806-3722-10
December 1999

Contents

Figures

Preface

The *ChorusOS 4.0 Hot Restart Programmer's Guide* provides information for developers of high-availability applications that use the ChorusOS™ 4.0 hot restart feature and associated API. Hot restart provides a means of reducing the time it takes to restart an application or entire system when a serious failure occurs, based on the use of persistent memory. This guide provides a high-level overview of the hot restart architecture, and then looks in detail at how the hot restart API is used.

Who Should Use This Book

Use this book if you need to develop ChorusOS 4.0 actors that can be rapidly restarted in the event of failure, or if you need to use persistent memory in your applications. You will also find the first chapter useful if you are simply interested in learning what hot restart is and what it can do.

Before You Read This Book

If you are simply interested in learning what hot restart is, you will need to be familiar with C programming, and with the high-level architecture of the ChorusOS system before reading this guide. If you will be developing applications with hot restart, you are also expected to be familiar with programming ChorusOS actors, and the Sun Embedded WorkShop™ development tools. All of the ChorusOS prerequisite topics are covered in the *ChorusOS 4.0 Introduction*.

You will need access to a working ChorusOS host machine and target platform if you want to run the hot restart demonstration and example programs.

How This Book Is Organized

This book is divided into four chapters and two appendixes that present different aspects of the hot restart feature.

- Chapter 1 provides a general introduction to what hot restart is and does, and how it is incorporated in the ChorusOS 4.0 system architecture.
- Chapter 2 is a step-by-step guide to getting up and running with hot restart, from configuring the system to running the hot restart demonstration application.
- Chapter 3 takes an in-depth look at how the persistent memory API provided by hot restart can be used. The persistent memory API can be used by any ChorusOS actor.
- Chapter 4 describes the API used for developing restartable actors, and explains how the hot restart mechanism works from a programming point of view.
- Appendix A provides information of use to developers for compiling actors that use the hot restart API.
- Appendix B lists the code of the examples used in the body of this guide.

Related Books

The following documents contain information that is related to the material covered in this guide:

- The *ChorusOS 4.0 Introduction* comprises a basic overview and getting started guide for new users of the ChorusOS 4.0 system. Familiarity with the information in the *ChorusOS 4.0 Introduction* is a prerequisite for users of this Programmer's Guide.
- The *ChorusOS 4.0 Installation Guide*, *ChorusOS Target Family Guides* and related documents cover installation and configuration of the ChorusOS 4.0 system and the Sun Embedded Workshop development tools.
- For a complete reference to the APIs available in the ChorusOS 4.0 system, see the man pages in the *ChorusOS 4.0 Reference Manual Collection*. The *2RESTART* section in this collection covers the API exported by the hot restart feature. Other related man pages are also referenced in the body of this Programmer's Guide.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks selected product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com[SM] Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`

TABLE P–1 Typographic Conventions *(continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **rm** *filename.*
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide.* These are called *class* options. You must be *root* to do this.

Shell Prompts

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	#

Directory Conventions

The following table describes several of the directory conventions used in this book.

TABLE P–3 Directory Conventions

Name	Meaning	Example
install_dir	Directory into which you install the ChorusOS 4.0 product.	`/opt/SUNWconn/SEW`
build_dir	Directory in which you build your ChorusOS system image from the installed product.	`/home/user/ChorusOS`

CHAPTER 1

Introduction

The purpose of this chapter is to provide an introduction to hot restart.

- Section 1.1 “What Is Hot Restart?” on page 15 is a brief introduction to the motivation behind the development of hot restart, and what hot restart actually is.
- Section 1.2 “Basic Concepts” on page 16 presents the four concepts central to the hot restart feature: persistent memory, actor restart, restart groups and site restart.
- Section 1.3 “Architecture Components” on page 21 summarizes the high-level architecture of hot restart, and how it relates to some of the principal ChorusOS™ actors.

By the end of this chapter, you should have sufficient knowledge of hot restart to understand the information provided in the rest of this book.

1.1 What Is Hot Restart?

The ChorusOS™ 4.0 system’s hot restart feature has been designed and implemented to address the high-availability requirements of ChorusOS system builders. Hot restart provides an advanced mechanism for restarting ChorusOS applications or the entire system when a serious error or failure occurs. Traditionally, system recovery from such errors or failures involves terminating applications and reloading them from stable storage, or rebooting the system. This causes system downtime, and can mean that important application data is lost. Such behavior is unacceptable for system builders seeking ’7 by 24’ or ’five nines’ system availability.

The ChorusOS 4.0 hot restart feature solves the problem of downtime and data loss by using *persistent memory*, that is, memory which can persist beyond the lifetime of a particular run-time instance of an actor. When an actor which uses the hot restart feature fails, or terminates abnormally, the system uses the actor data stored in

persistent memory to reconstruct the actor without accessing stable storage. This reconstruction of an actor from persistent memory instead of from stable storage is known as *hot restarting* (or simply *restarting*) the actor.

Hot restarting one or more actors is significantly faster than conventional failure recovery techniques (application reload or cold system reboot) because it protects critical information that allows the failed portions of a system to be reconstructed quickly, with minimal interruption in service.

1.1.1 Feature Services

ChorusOS hot restart comprises an API and run-time architecture which offer the following services:

- *persistent memory allocation*

  The hot restart API allows actors to allocate and free portions of persistent memory while they are executing. This service is available to all ChorusOS actors once hot restart is configured.

- *actor restart*

  With hot restart, the system is capable of detecting the abnormal termination of one or more actors and restarting them automatically from persistent memory. In addition, actors are organized into *restart groups*, enabling the simultaneous restart of all actors in a predefined group when a single actor in the group fails.

- *site restart*

  With hot restart, in addition to restarting one or more actors, the system is capable of restarting all restartable actors, plus the kernel and boot actors, for a given ChorusOS site.

The combination of these services provides a powerful framework for highly-available systems and applications, dramatically reducing the time it takes for a failed system or component to return to service.

1.2 Basic Concepts

This section introduces the basic concepts central to the hot restart feature and services. These concepts are: persistent memory, restartable actor, restart group, and site restart.

1.2.1 Persistent Memory

The foundation of the hot restart mechanism is the use of persistent memory to store data which can persist across an actor or site restart. Persistent memory is used internally by the system, to store the actor image (text and data) from which a restartable actor can be reconstructed. Any actor can also allocate persistent memory to store data. This data could, for example, be used to checkpoint application execution.

At the lowest level, persistent memory is a bank of memory loaded by the ChorusOS kernel at cold boot. The content of this bank of memory is preserved across an actor or site restart. In the current implementation, the only supported medium for the persistent memory bank is RAM: in other words, persistent memory is simply a reserved area of physical memory. For this reason, persistent memory will resist a hot restart, but not a board reset. The size of the area of RAM reserved for persistent memory is governed by a tunable parameter.

The allocation and de-allocation (freeing) of persistent memory are managed by a ChorusOS actor known as the *Persistent Memory Manager* (PMM). The Persistent Memory Manager exports an API for this purpose. This API is distinct from the API used for allocating and de-allocating traditional ChorusOS memory regions (`rgnAllocate(2K)`, `rgnFree(2K)`, `svPagesAllocate(2K)`, and `svPagesFree(2K)`).

The Persistent Memory Manager API is described in more detail in Chapter 3 and in the `pmmAllocate(2RESTART)`, `pmmFree(2RESTART)` and `pmmFreeAll(2RESTART)` man pages.

1.2.2 Restartable Actors

A *restartable actor* is any actor which can be rapidly restarted without accessing stable storage, when it abnormally terminates. A restartable actor is restarted from an *actor image* which comprises the actor's text and initialized data regions. The actor image is stored in persistent memory (unless the actor is executed in place, in which case the actor image is the actor's executable file, stored in non-persistent, physical memory). Restartable actors can use additional blocks of persistent memory to store their own data.

Figure 1–1 shows the state of a typical restartable actor at its initialization, during execution, and after having been hot restarted as a result of an error. The actor uses persistent memory to store some state data. After hot restart, the actor is reconstructed from its actor image, also in persistent memory. It is then re-executed from its initial entry point, and can retrieve the persistent state data which has been stored.

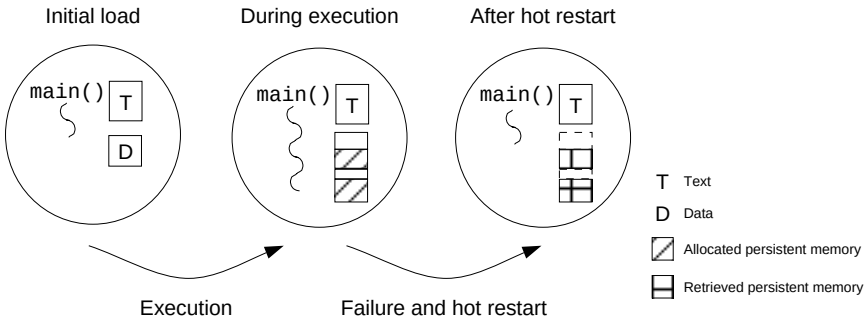

Figure 1–1 Typical restartable actor

In the hot restart architecture, management of restartable actors is assured by a ChorusOS supervisor actor known as the *Hot Restart Controller*. Restartable actors are monitored by the Hot Restart Controller, in that the Hot Restart Controller will detect a restartable actor's abnormal termination and automatically take the appropriate restart action if an abnormal termination occurs. In the context of hot restart, abnormal termination cases include unrecoverable errors such as division by zero, a segmentation fault, unresolved page fault, or invalid op code, and so on.

Restartable actors, like traditional ChorusOS actors, can be run in either user or supervisor mode. In addition, they can be run from the `sysadm.ini` file or C_INIT console, or spawned dynamically during system execution. Indeed, the restartable nature of restartable actors remains transparent to system actors such as the AM actor, responsible for loading and starting restartable actors. This is because restartable actors do not *declare themselves* restartable, but are *run* as restartable actors. More specifically, the *way* a restartable actor is initially run determines how it will be restarted when a restart occurs:

- Restartable actors which are run from the `sysadm.ini` file, or which are run directly from the C_INIT console, are restarted *directly* by the system when a restart occurs. These actors are known as *direct restartable actors.*
- Restartable actors which are spawned dynamically during system execution will be restarted by the actor which initially spawned them. These actors are known as *indirect restartable actors.*

The distinction between direct and indirect restartable actors provides a useful framework for the construction of restartable *groups* of actors, as described in the next section.

C_INIT and the Hot Restart Controller provide an interface specifically for running and spawning restartable actors. This interface is described in detail in Chapter 4.

1.2.3 Restart Groups

Many applications are made up of not one but several actors, which cooperate to provide a service. As these actors cooperate closely, any failure in one of them can

have repercussions in the others. For instance, assume that actors A and B cooperate closely (using CHORUS/IPC for instance), and that A fails. Simply terminating, reloading or hot-restarting A will probably not be sufficient, and will most certainly cause B either to fail itself, or to go through some special recovery action. This recovery action may in turn affect other actors which cooperate with actor B. Building cooperating applications which can cope with the large number of potential fault scenarios is a very complex task, as the complexity grows exponentially with the number of actors.

In response to this problem, the hot restart feature uses the concept of *restart group*. A restart group in its most common sense is a group of cooperating restartable actors which can be restarted in the event of the failure or abnormal termination of one or more actors within the group. In other words, when one actor in the group fails, all actors in the group will be stopped and then restarted (either directly, by the system, or indirectly, through spawning). In this way, closely cooperating actors are guaranteed a consistent, combined operating state.

Every restartable actor in a ChorusOS 4.0 system is a member of a restart group. Restart groups of actors are mutually exclusive: a running actor can only be a member of one actor group (declared when the actor is run), and group containment is not permitted. A restart group is created dynamically when a direct actor is declared to be a member of the group: thus, each group contains at least one direct actor. An indirect actor is always a member of the same group as the actor which spawned it. A restart group is therefore populated through spawning from one or more direct restartable actors.

Figure 1–2 illustrates the possible organization of restartable actors in groups within a system.

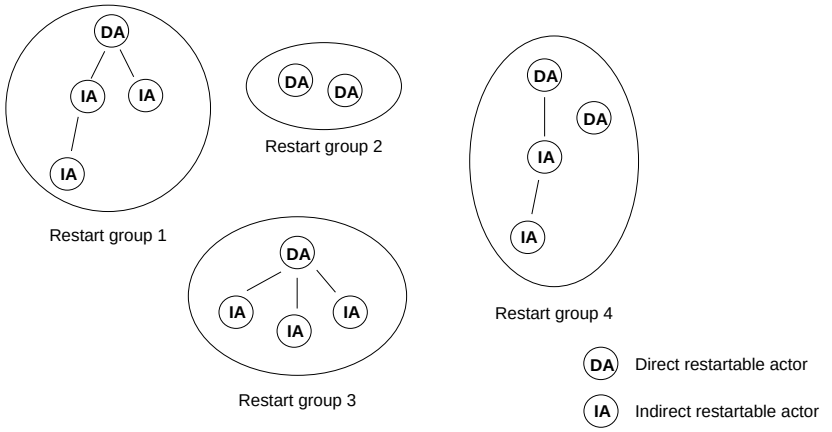

Figure 1–2 Restart Groups in a ChorusOS System

When a group is restarted, it is restarted *from the point at which it initially started.* Figure 1–3 shows the state of a group of restartable actors when it is initially created, during execution, and when it is restarted following the failure of one of its member actors. The group contains two direct actors and one indirect (spawned) actor. The

failure of the indirect actor causes a group restart: the two direct actors automatically re-execute their code from their initial entry point. Time runs vertically down the page.

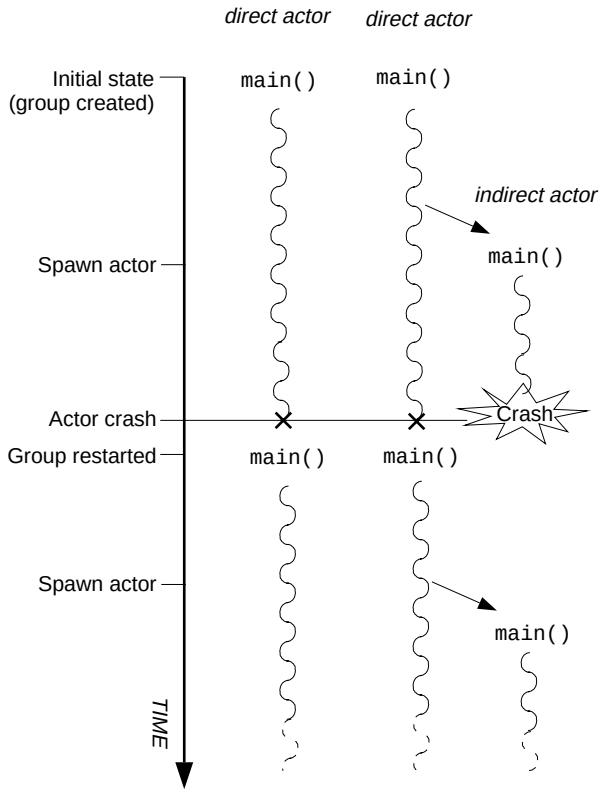

Figure 1–3 Group restart

Of course, simply restarting a group of actors may still not bring the system to the error-free state desired. Such a situation is possible when the failure which provokes an actor group restart is in fact the consequence of an error or failure elsewhere in the system. For this reason, the hot restart feature supports the concept of *site restart*, as described in the next section.

1.2.4 Site Restart

A site restart is the reinitialization of an entire ChorusOS site (system) following the repeated failure of a group of restartable actors. It is the most severe action which

can be automatically provoked by the Hot Restart Controller. A site restart involves the following:

- The kernel and boot actors are re-initialized from the system image. This step is sometimes termed a 'hot reboot' of the system (as opposed to a cold reboot, which involves a board reset and initial system loading steps, as described in the *ChorusOS 4.0 Porting Guide*).
- All restartable actor groups are restarted.

The precise frequency of group restarts which provokes a site restart is determined by the system's *restart policy*. The basic policy implemented by the hot restart feature is based on a set of system tunable parameters described in Chapter 2. You can extend this basic restart policy within your own applications, for example by choosing to provoke a group or site restart when particular application-specific exceptions are raised, or particular events occur.

1.3 Architecture Components

As described in the previous sections, the hot restart feature uses the following two restart-specific actors to implement hot restart services:

- A supervisor actor called the *Persistent Memory Manager* (`PMM`), which offers services for allocating and freeing persistent memory blocks.
- A supervisor actor called the *Hot Restart Controller*, (`HR_CTRL`). It offers the system calls that create and kill restartable actors, monitors restartable actors for abnormal termination, and takes the appropriate restart action when a failure occurs.

The Persistent Memory Manager and Hot Restart Controller principally use the services of the following:

- The C_INIT actor, for the interpretation of hot restart-specific commands entered on the target or host console.
- The system actor AM, solicited by the Hot Restart Controller for loading and running restartable actors.
- The ChorusOS microkernel, for the low-level allocation of persistent memory, and for support for site restart.

The resulting architecture is summarized in the following diagram. Hot restart-specific components appear in gray, together with the API calls they provide. Other components appear in white. Arrows from A to B say that A calls functions which are implemented in B.

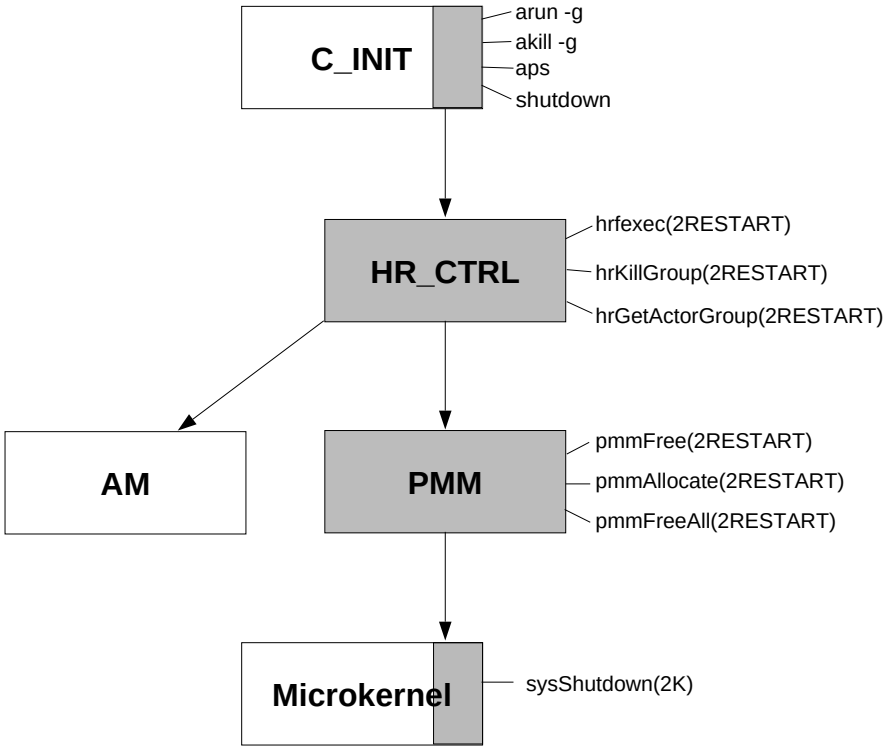

Figure 1–4 Hot Restart Architecture

Further information about the hot restart API is provided in the rest of this guide, and in the corresponding man pages.

CHAPTER 2

Getting Started With Hot Restart

This chapter describes how to set up your ChorusOS 4.0 system to use the hot restart feature. It covers the following:

- Configuring your ChorusOS 4.0 system for hot restart: see Section 2.1 “System Configuration” on page 23.
- Running the graphical hot restart demonstration program provided with Sun Embedded Workshop: see Section 2.2 “Running the Hot Restart Demonstration Program” on page 26.

Note - This chapter assumes that you have already correctly installed Sun Embedded Workshop on a host machine, and that you have a target machine which can be booted from a network boot server. You should also be familiar with configuring your ChorusOS 4.0 system and building a system image. For more information on these topics, see the related documents cited in the Preface of this guide.

This chapter does not cover hints for linking and building your own hot restartable applications. For information on this topic, see Appendix A.

2.1 System Configuration

Before beginning to program and run actors which use the hot restart feature, you will need to update and configure your system for hot restart. System configuration for hot restart involves the following steps:

- including the necessary ChorusOS optional features in your system
- ensuring that the settings for the tunable parameters used by the hot restart feature are suitable for your system

These steps are described in the sections which follow.

2.1.1 Features

To incorporate hot restart in your ChorusOS 4.0 system, use the `ews` graphical tool or the `configurator(1CC)` command line utility to include the following optional features in your system profile:

- HOT_RESTART. This feature exports the hot restart API and restart mechanism.
- ACTOR_EXTENDED_MNGT, LAPSAFE, LAPBIND and ADMIN_SHUTDOWN. These features provide necessary support for the HOT_RESTART feature.

2.1.2 Memory Requirements and Design Considerations

As stated in Chapter 1, the hot restart feature implements persistent memory as a portion of physical memory (RAM) on the target device. Although the persistent memory bank does not itself use virtual memory or swapping, hot restart is compatible with all three of the main memory models: flat, protected, and virtual.

The size of the persistent memory bank is defined in bytes by a system tunable parameter, `pmm.rambankSize`. The value of this parameter is static: its value cannot be modified while the system is running. In addition, because the RAM persistent memory bank does not use virtual memory or swapping, objects in persistent memory are locked in memory until they are freed. For these two reasons, it is important to make sure that `pmm.rambankSize` is set to a value realistic for the amount of data likely to be stored in persistent memory at any one time.

A portion of space reserved for an object in the persistent memory bank is termed a *persistent memory block*. A block is a contiguous set of memory pages, which means that the size of a block is always a multiple of the page size. Use `vmPageSize(2K)` to find out the page size for your platform.

For each running restartable actor, the system stores the following data in persistent memory:

- The text and initialized data which were loaded into memory from stable storage. This is known as the *actor image*. The actor image occupies a single block of persistent memory.
- The executed text, initialized data and BSS (data initialized to zero), from which the actor is running. This is known as the actor's *executing image*. The executing image occupies two blocks of persistent memory: one block for the text and one block for the data. The heap and stack for the executing actor are stored in non-persistent memory.

The persistent memory blocks used to store the actor image and executing image will only be freed when the actor's group terminates cleanly (note that this may be some

time after the actor itself has terminated). The actor can also allocate its own blocks of persistent memory to store run-time data while it is executing.

Although it can be difficult to predict the likely required value of `pmm.rambankSize` early in the development cycle, the following rule of thumb, derived from the statements above, may be of use to developers at the system design stage:

- Restartable actors occupy an absolute minimum of twice their size in persistent memory. This minimum will accommodate the actor's actor image and executing image (although it does not allow for rounding up of memory block sizes to the nearest page). The actor may also allocate additional portions of memory. `pmm.rambankSize` should therefore be greater than twice the combined size of the restartable actors expected to run simultaneously on a system.

Note - Sharing persistent memory blocks between user actors, or between user and supervisor actors is not supported. Persistent memory blocks can only be shared between supervisor actors.

The default value of the `pmm.rambankSize` tunable parameter is 1024*1024 bytes, that is, one megabyte.

2.1.3 Tunable Parameters

The HOT_RESTART feature uses a number of system tunable parameters. Each parameter has a default value which can serve as a guideline and is generally suitable for getting started with hot restart programming. All tunable parameters are static: they cannot be modified while the system is running.

Two parameters define limits for persistent memory occupation in the system's persistent memory bank:

- `pmm.rambankSize` is the maximum amount of persistent memory available in the system, in bytes. The default value is one megabyte (`0x100000`). See the previous section for guidelines on setting this parameter to suit your system. If you want to run the hot restart demonstration program, you will need to increase the value of this parameter to four megabytes (`0x400000`).
- `pmm.maxBlocks` is the maximum number of recorded persistent memory blocks which can be allocated in the persistent memory bank. A block is a variable-sized number of contiguous pages of RAM. Each time an actor (supervisor or user) issues a request to store a piece of data in persistent memory, a block of the appropriate size, rounded up to the nearest whole page, is allocated. The default value is 30.

Two parameters control the maximum number of restartable actors and restart groups permitted in the system:

- `hrCtrl.maxActors` is the maximum number of hot restartable actors which can be registered in the system. An actor is registered in the system when it is first

run, and remains registered until all the actors in its group have terminated normally. The default value is 32. If `hrCtrl.maxActors` is greater than 65536, 65536 is used instead.

- `hrCtrl.maxGroups` is the maximum number of restart groups which can be present in the system at the same time. Its default value is 32.

Two parameters define the system's *restart policy* (see Section 1.2.4 "Site Restart" on page 20). These parameters are quite sensitive: different values can produce very different behavior in the system. The system manages a *restart counter* for each restart group. Each time a group is restarted, the system increases its restart counter by one.

- `hrCtrl.interval` is the frequency with which a group's restart counter is *de*creased, in seconds. Every `hrCtrl.interval` seconds, the system will decrease the group's restart counter by one (until the counter reaches zero). The default value for `hrCtrl.interval` is 3 seconds.
- `hrCtrl.maxBadness` is the maximum value a group's restart counter can reach before it triggers a site restart. In other words, when a group's restart counter reaches this value, a site restart is automatically performed. The default value is 25. If set to zero, the system never triggers a site restart.

2.1.4 Building the System Image

Once you have updated your system's features for hot restart and the tunable parameter settings are appropriate for your needs, you are ready to build the system image.

If you want to run the hot restart demonstration and examples, ensure that you include the examples directory and X11 library in your system build paths if they are not already included. For information on building a system image for your particular target platform, see the corresponding document in the *ChorusOS 4.0 Target Family Guide* collection.

After the system image has been correctly built, copy it to your boot server and reboot your target machine. You are now ready to begin programming and running applications which use the hot restart feature.

2.2 Running the Hot Restart Demonstration Program

Sun Embedded Workshop includes a graphical demonstration of the hot restart feature. The demonstration is based on the well-known program Xmaze, which has been slightly modified to make it hot restartable. Some of the program's data is

stored in persistent memory, which means that when the program is restarted, it starts at a point close to the point it had reached prior to the restart. The resulting application is a ChorusOS actor called `xdemo_s`.

To run the hot restart demonstration program, do the following:

1. Ensure that your system features are correctly set for hot restart: see Section 2.1.1 “Features” on page 24.
2. Adjust the following system tunable parameters to suit the memory requirements of the Xmaze demonstration program, using `ews` or the `configurator(1CC)` command line utility:

Tunable parameter	Description	Required value
`pmm.rambankSize`	Size of persistent memory bank, in bytes	0x400000
`kern.exec.dflSysStackSize`	Default system stack size, in bytes	0x8000

3. Configure your system image build to include the X11 library and ChorusOS examples directory, if this is not already the case.
4. If you have made changes to the system image since the previous build, rebuild the system, copy the system image to the appropriate location (for example, the boot directory if you are using tftp-based boot) , and reboot the target machine.
5. Ensure that a copy of the `xdemo_s` actor is present in a directory which is mounted on the target machine. If you use the `make root` command, a copy of the actor is already stored in *build_dir*`/root/bin/examples`. If this directory is not mounted, or you prefer to use a different mounted directory:

```
$ cp build_dir/BUILD_EXAMPLES/restartDemo/xdemo_s example_directory
```

6. Set the target machine’s `DISPLAY` environment variable to the host machine which you are currently working on:

```
$ rsh target setenv DISPLAY host_IP_address:0.0
```

7. Run the restartable actor:

```
$ rsh target arun -g 0 example_directory/xdemo_s
```

   The actor will be run as a member of the restart group with group ID `0`.

The Xmaze demonstration appears on the screen. As the demonstration runs, it periodically stores its state as data in persistent memory. Let the demonstration advance a little, then restart the actor by typing the following on the host console:

```
$ rsh akill aid
```

aid is the actor identifier which is printed on the host console when the actor starts. The actor is restarted, and the Xmaze demonstration continues from a point close to the point it had reached before the restart.

The `akill` command provoked the restart because it was not passed with the restart-specific option –g. To kill the Xmaze demonstration actor without restarting it, type:

```
$ rsh target akill -g 0
```

As the `xdemo_s` actor is run from the command line, it is a direct actor, and will be started automatically by the system when the site is restarted. To check this, rerun the actor, and then provoke a site restart by typing the following:

```
$ rsh target restart
```

When the system has been re-initialized, the demonstration will be restarted.

Of course, this is a very simple illustration of the use of hot restart. The site restart is provoked manually from the command line. As an alternative, try restarting the actor (using `akill` –g) sufficiently frequently to trigger an automatic site restart. To do this, you will first need to set the system's restart policy to be more sensitive to actor failure. The following configuration will cause a site restart if the actor is restarted twice in the space of four seconds:

Tunable parameter	Value
`hrCtrl.interval`	4
`hrCtrl.maxBadness`	2

CHAPTER 3

Programming With Persistent Memory

This chapter provides a detailed description of the API exported by the Persistent Memory Manager. In particular, it covers the following topics:

- How persistent memory is managed in the system: see Section 3.1 “Introduction to Persistent Memory Programming” on page 30.
- Allocating and retrieving blocks of persistent memory with the Persistent Memory Manager API: see Section 3.3 “Allocating and Retrieving a Persistent Memory Block” on page 33.
- Freeing blocks of persistent memory with the Persistent Memory Manager API: see Section 3.4 “Freeing a Persistent Memory Block” on page 34.

In this chapter, an example “hello world” program is used to illustrate different aspects of the Persistent Memory Manager interface. The code for this example is given in Section 3.2 “A Simple Application” on page 30.

To run the example, you will need to compile it and then copy it to a directory which is mounted by the target machine. See Section B.1 “Compiling and Running the Examples” on page 51 for information about compiling and running the hot restart examples.

Note - Before reading this chapter, make sure that you are familiar with the basic persistent memory architecture described in Section 2.1.2 “Memory Requirements and Design Considerations” on page 24.

3.1 Introduction to Persistent Memory Programming

Within a running ChorusOS system, access to persistent memory is provided by a ChorusOS actor known as the Persistent Memory Manager. The Persistent Memory Manager exports a specific API for allocating and freeing blocks of memory in the persistent memory bank. This API is distinct from the API used for allocating and de-allocating traditional ChorusOS memory regions (`rgnAllocate(2K)`, `rgnFree(2K)`, `svPagesAllocate(2K)`, and `svPagesFree(2K)`, for the following reasons:

- Persistent memory blocks, by definition, persist across an actor or site restart. The API provided for manipulating traditional ChorusOS memory regions is not sufficiently rich to allow memory to be recovered after a restart.
- Persistent memory blocks, unlike traditional memory regions, are *named*. This name is used to retrieve a block of memory which is allocated in the persistent memory bank.
- Persistent memory blocks, unlike traditional memory regions, can be *grouped*, for the purposes of simultaneous de-allocation. In other words, a single API call can free multiple blocks of persistent memory, which may have been allocated by different actors in the ChorusOS system.

The Persistent Memory Manager API is available to all ChorusOS 4.0 actors (not just restartable actors). The aim of this chapter is to describe in detail the use of this API.

3.2 A Simple Application

Before proceeding with a description of the different functions in the Persistent Memory Manager API, consider the following simple restartable application, an implementation of the familiar “hello world” example. When the actor is run for the first time, it displays the following message on the host console:

```
Hello world!
```

When the actor is restarted, it displays the following message on the target console:

```
Hello again! I have been restarted.
```

The basic flow of execution is as follows:

- The restartable actor begins at the start of its `main()` program, initializing its program data.

- The actor uses the `pmmAllocate(2RESTART)` function to allocate a block of persistent memory. This block is used to store a status counter, which the actor sets to zero.
- The first message is displayed, and the counter is increased by one.
- The actor attempts to access an invalid pointer value, causing a crash so that the actor is restarted. Note that the ChorusOS VIRTUAL_ADDRESS_SPACE optional feature should be set to true for this crash to work.
- The restartable actor recommences execution at the start of its `main()` program, and calls `pmmAllocate()` a second time to retrieve the value of the status counter.
- As the counter is no longer zero, the actor displays the second message.
- The actor calls `pmmFree()` to free the persistent memory block used to store the counter, and then exits cleanly.

CODE EXAMPLE 3–1 The “Hello world” Restartable Actor

```
#include <stdio.h>
#include <pmm/chPmm.h>
#include <hr/hr.h>

#define HR_GROUP "HELLO_GROUP"

    int
main()
{

 int res;
 int any = 1;
 int* counter_p;  /* It will be stored in persistent memory */
 long *p;
 PmmName name;
 KnRgnDesc rgn;

            /*
             * Initialize the name and medium fields
             * to identify the persistent memory block in the system.
             */
 bzero(&name, sizeof(name));
 strcpy(name.medium,"RAM");
 strcpy(name.name,"PM1");

            /*
             * Initialize the block fields
             */
 bzero(&rgn, sizeof(rgn));
 rgn.options = K_ANYWHERE | K_RESERVED;
 rgn.size    = vmPageSize();
 res = rgnAllocate(K_MYACTOR, &rgn);
 if (res != K_OK) {
            printf("rgnAllocate() failed res=%d\n", res);
            HR_EXIT_HDL();
            exit(-1);
 }
```

```
p = (long*) rgn.startAddr;

           /*
            * From now on p is a bad pointer, since
            * VIRTUAL_ADDRESS_SPACE is true.
            */

           /*
            * Allocate the persistent memory block that stores
            * counter_p.
            */
res=pmmAllocate((VmAddr *)&counter_p,
                       &name,sizeof(int),
                       HR_GROUP,
                       sizeof(HR_GROUP));

if (res != K_OK) {
           printf("Cannot allocate or map the persistent memory block called %s."
                  " Error = %d\n", name.name, res);
           HR_EXIT_HDL();
           exit(-1);
}

           /*
            * From the value of *counter_p the actor detects
            * whether it has been hot restarted or not.
            */
if ( *counter_p==0 ) {
               /*
                * This is the first time the actor is run.
                */
           printf("Hello world!\n");

               /*
                * Increment the counter
                */
           (*counter_p)++;

               /*
                * Normally the next instruction causes a core dump and
                * a hot restart of the actor
                */
           *p = 0xDeadBeef;

} else {
               /*
                * The actor has been restarted
                * NOTE: this message will appear on the console!
                */
           printf("The actor has been restarted.\n");

               /*
                * Free the persistent memory block before exiting
                */
           res = pmmFree(&name);
           if (res != K_OK) {
               printf(" pmmFree failed, res=%d. Exit\n", res);
               HR_EXIT_HDL();
               exit(-1);
           }
```

```
            /*
             * Terminate cleanly.
             */
        printf("Example finished. Exit.\n");
        HR_EXIT_HDL();
        exit(0);
 }
        /* Never reached */
}
```

The aspects of this program which are of interest for users of the Persistent Memory Manager API are discussed in the rest of this chapter.

3.3 Allocating and Retrieving a Persistent Memory Block

The “hello world” application uses a block of persistent memory to store a counter indicating whether it has been restarted. The value of the counter controls the program’s flow of execution. This is a very common use of persistent memory. A counter or flag such as this is usually necessary as it is the only way an actor can know whether it has been restarted.

A block of persistent memory is described in the system by a structure of the following type:

```
#include <chPmm.h>
typedef struct { PmmMedium   medium = "RAM";
                 PmmMemName  name; }
PmmName;
```

Within the structure, *medium* is a character string which identifies the memory bank to be used. In the current implementation, it must always be set to `RAM`. *name* is a user-defined, null-terminated character string which uniquely identifies the block of memory in the memory bank. The lifetime of a block name is the same as the lifetime of the block itself in persistent memory. A system tunable parameter, `pmm.maxBlocks`, defines the number of distinct persistent memory blocks (and therefore names) which can be allocated at any one time. The default value is 30.

Caution - Sharing persistent memory blocks between user actors, or between user and supervisor actors is not supported. Persistent memory blocks can only be shared between supervisor actors.

To allocate a block of persistent memory, or retrieve a block of memory which has already been allocated, use the `pmmAllocate()` function call, defined as follows:

```
#include <chPmm.h>
KnError pmmAllocate(  VmAddr     *addr,
                      PmmName    *name,
                      size_t      size,
                      PmmDelKey   delKey,
                      size_t      delKeySize);
```

If no memory block corresponding to the specified `PmmName` structure is present in persistent memory, `pmmAllocate()` allocates a block of size *size* in persistent memory, fills it with nulls, and returns the pointer **addr* to the address of the block. The address is determined by the system and cannot be specified or changed.

If a block identified with the specified `PmmName` already exists in persistent memory, `pmmAllocate()` simply returns a pointer to the existing memory block as an address (**addr*), and the *size* parameter is ignored. Persistent memory blocks are always mapped at the same address. In other words, the address returned by the first and subsequent calls to `pmmAllocate()` is always the same for a given block.

As a result of this dual functionality of the `pmmAllocate()` call, the difference between initially allocating and subsequently retrieving a persistent memory block is transparent at the programming level. The first time the code of the “hello world” example is executed, the call to `pmmAllocate()` allocates an integer-sized block of persistent memory which contains the initialized value of `counter` (0).

```
res=pmmAllocate((VmAddr *)&counter_p,
                      &name,sizeof(int),
                      HR_GROUP,
                      sizeof(HR_GROUP));
```

The second time the code is executed, the same function call returns a pointer to the value of `counter` in persistent memory.

The *delKey* and *delKeySize* parameters passed to `pmmAllocate()` are used to define the *deletion key* associated with the memory block. A deletion key is a user-defined binary array, used to mark a set of persistent memory blocks which can be freed simultaneously, using the `pmmFreeAll()` function, described in Section 3.4.2 “Freeing a Persistent Memory Block Explicitly” on page 35.

3.4 Freeing a Persistent Memory Block

This section describes the API calls used for freeing persistent memory blocks.

3.4.1 Responsibility

A persistent memory block can remain in memory beyond the lifetime of a run-time instance of the actor which allocates the block. This immediately raises the question

of responsibility for freeing blocks of persistent memory. When a traditional ChorusOS 4.0 user actor terminates, any memory regions it allocated (using `rgnAllocate(2K)`) are automatically freed. Clearly, this simple rule makes little sense in the case of persistent memory blocks, which can survive beyond such a termination.

The hot restart feature provides two solutions to this problem:

- Actors can explicitly free blocks of persistent memory using the API function `pmmFree()` or `pmmFreeAll()`. This is the only solution available for non-restartable actors which use persistent memory: for these actors, freeing persistent memory is entirely the programmer's responsibility.

  If persistent memory needs to survive beyond the persistent lifetime of the allocating actor (that is, even after the actor has cleanly terminated), implementing this solution will require either careful application design or the presence of a garbage collection actor.

  Explicit freeing of persistent memory blocks is described in the next section.

- Hot restartable actors can benefit from an automatic clean-up mechanism provided by the Hot Restart Controller. This is described in more detail in Section 4.2.3 "Freeing Persistent Memory" on page 42.

In both cases, freeing a persistent memory block has the same effect: the block is immediately and permanently freed (cannot be retrieved), and the name which identified it can be re-used to identify a different memory block.

3.4.2 Freeing a Persistent Memory Block Explicitly

Use the `pmmFree()` or `pmmFreeAll()` function to explicitly free a persistent memory block. The explicit freeing of a given memory block can be performed by any actor, not necessarily the actor which originally allocated the block. It is the programmer's responsibility to ensure that the persistent memory block which will be freed is no longer in use.

Use `pmmFree()` to free a single memory block identified by a `PmmName`:

```
#include <chPmm.h>
int pmmFree( PmmName *name )
```

Use `pmmFreeAll()` to free a *group* of persistent memory blocks which were allocated with the same deletion key. The deletion key for a persistent memory block is specified when the block is allocated with `pmmAllocate()`.

```
#include <chPmm.h>
int pmmFreeAll( PmmDelKey    delkey,
                size_t       delKeySize );
```

A typical use of a deletion key is to mark all persistent memory blocks used by an actor or a group of actors with the same key, and then have a separate, independent

actor that frees all the blocks when a particular job is completed or a particular event occurs. For example, the “hello world” example uses `pmmFree()` to free the single memory block it allocates before it terminates. If the “hello world” actor did not free its own persistent memory block, the following call to `pmmFreeAll()` from another actor would free the block, along with any other blocks marked with the deletion key `HR_GROUP`.

```
pmmFreeAll( HR_GROUP, sizeof(HR_GROUP) );
```

CHAPTER 4

Programming With Restartable Actors

This chapter covers programming and running restartable actors on a ChorusOS 4.0 system. In particular, it provides the following:

- An overview of how restartable actors are represented and managed in the system: see Section 4.1 “Introduction” on page 37.
- A description of the API and C_INIT commands used for loading, restarting, and terminating restartable actors: see Section 4.2 “The Restartable Actor Lifecycle” on page 39 and Section 4.3 “Killing Restartable Actors” on page 46.
- A description of the API and C_INIT commands used to restart the site: see Section 4.4 “Site Restart” on page 46.
- An introduction to the `restartSpawn` example program, used to illustrate the use of the Hot Restart Controller and Persistent Memory Manager APIs: see Section 4.5 “Putting It All Together: the restartSpawn Example Program” on page 47.

4.1 Introduction

As described in Chapter 1, a restartable actor is an actor which can be rapidly reconstructed from an actor image (text and data) , without accessing stable storage. The management of restartable actors is handled by a ChorusOS supervisor actor known as the Hot Restart Controller. The Hot Restart Controller is responsible for:

- Loading and running restartable actors, and controlling their storage in persistent memory.
- Monitoring restartable actors for abnormal termination, and restarting their restart group if such an abnormal termination occurs.
- Triggering a site restart if a group is restarted too frequently (based on the system’s restart policy, as described in Chapter 2).

This chapter looks at the API provided by the Hot Restart Controller, and the corresponding restart-related commands provided by the C_INIT actor. Before proceeding to a description of the API, however, it is important to understand how restartable actors are managed within the system.

4.1.1 Types of Restartable Actor

As explained in Chapter 1, it is important to understand that actors do not explicitly *declare themselves* restartable, that is, there is no function call to declare an actor restartable at the start of its `main`() program. Instead, an actor can be *run* as a restartable actor. More precisely, an actor can be run as either a *direct* or *indirect* restartable actor:

- Direct restartable actors are loaded and run using the `C_INIT` command `arun(1M)` with the –`g` option.
- Indirect restartable actors are spawned from restartable actors using the `hrfexec(2RESTART)` family of API calls. `hrfexec()` calls function similarly to `afexec(2K)` calls, but provide an additional `PmmName` parameter used to identify them for the purposes of actor restart:

```
#include <hr/hr.h>
int hrfexecve( PmmName *       baseName,
               const char *    path,
               KnCap *         cactorcap,
               const AcParam * param,
               char const *    argv,
               char const *    envp);
(...)
```

This distinction between direct and indirect actors is important for understanding the automatic restart mechanism provided by the Hot Restart Controller. When an error occurs, the Hot Restart Controller will first stop all actors in the group, and then only restart the concerned *direct* restartable actors. These actors, re-executed from their initial entry point, are responsible for restarting any indirect actors they may have spawned. An illustration of this is provided in Figure 1–3.

4.1.2 Restartable Actor Credentials

Restartable actors, just like traditional ChorusOS actors, are identified in the system by a unique capability and identifier (actor ID). Restartable actors also run in a user group (with a user ID), like traditional ChorusOS actors. The lifetime of each of these credentials is the same as the lifetime of a particular run-time instance of the actor: when a restartable actor is restarted, it is given a new capability, actor ID and user ID.

Hot restartable actors also have two additional credentials, which persist across an actor restart, and serve to characterize them in the Hot Restart Controller:

- Each restartable actor has a unique *name*. The maximum number of restartable actors (unique names) which can be registered in the Hot Restart Controller is fixed by the system tunable parameter `hrCtrl.maxActors`.

Note - It is the programmer's responsibility to ensure that each actor running in the system uses a unique name, as this is not checked by the system. Attempting to run two actors which use the same name will give unpredictable results.

- Each restartable actor is a member of a restart group. A restart group is uniquely identified in the system by an integer, known as the group's ID. The maximum number of group IDs allowed in the system is fixed by a system tunable parameter, `hrCtrl.maxGroups`.

4.1.3 Restartable Actors and Persistent Memory

As explained in Section 2.1.2 "Memory Requirements and Design Considerations" on page 24, the system uses persistent memory to store the following data for each executing restartable actor:

- The actor's *actor image*: a copy of the actor's text and initialized data segments from which the actor will be loaded after a restart.
- The actor's *executing image*: a copy of the actor's text and data from which the actor is executed.

This data is stored in three persistent memory blocks: one memory block for the actor image, one memory block for the executed text and one memory block for the actor data. These blocks are allocated and freed by requests from the Hot Restart Controller to the Persistent Memory Manager. Other actors cannot access or free these persistent memory blocks, although restartable actors can place additional blocks which they allocate under the control of the Hot Restart Controller. This is described in Section 4.2.3 "Freeing Persistent Memory" on page 42.

4.2 The Restartable Actor Lifecycle

One approach to understanding how the API provided by the Hot Restart Controller is used, is to consider it in the context of the run-time life-cycle of a restartable actor. Indeed, a restartable actor's code is not simply executed once, from the start of the `main()` program to its final return, but could be re-executed many times if there are many restarts. Data which is initialized and actors which are initially loaded during the first execution will only need to be retrieved or restarted on subsequent executions. This is why it is important to view the restart API in the context of this first execution, and then subsequent executions.

For this reason, this section looks at the way the Hot Restart Controller API is used in the context of the life-cycle of a typical restartable actor.

4.2.1 Initial Load

Use the C_INIT command `arun` with the –`g` option, or the function call `hrfexec()` to load a restartable actor from stable storage into persistent memory. Both `arun` and `hrfexec()` provide support for specifying the persistent credentials of a restartable actor when the actor is initially loaded.

- For a direct actor (run with `arun`), the actor name is system generated, and the group ID is passed using the –`g` option. If the group ID is not already in use, a new group is created which contains the direct actor. If the group ID already exists, the direct actor is simply added to the corresponding restart group. If no ID is passed after –`g`, the actor is started in the restart group with ID 0.

  A restart group can contain any number of direct actors.

- For an indirect actor (run with `hrfexec()`), the actor name is specified using a `PmmName` structure (see the description of this structure in Section 3.3 “Allocating and Retrieving a Persistent Memory Block” on page 33) . An indirect actor is automatically a member of the same actor group as the actor which spawned it.

Actors created directly using `actorCreate(2K)` or `acreate(2K)` are not hot restartable and cannot use the Hot Restart Controller API.

When an actor is run as a restartable actor, the Hot Restart Controller checks whether an actor identified with the specified name is already registered. If this is not the case (as is the case for an initial load), the Hot Restart Controller first solicits the Persistent Memory Manager to allocate the persistent memory blocks which will store the actor’s actor image and executing image. If successful, it registers the name of the new actor as a restartable actor, running in the specified group.

The subsequent load and start of the persistent actor is the same as for an actor run using a member of the `afexec(2K)` function family (see the man page for a description of this process). The difference is that the actor is loaded from its actor image (in persistent memory) and not from stable storage.

Note - A restartable actor’s name remains registered in the Hot Restart Controller for the lifetime of its actor group. The lifetime of the group may extend beyond the lifetime of the actor. It is the programmer’s responsibility to ensure that no two restartable actors will attempt to register with the same name in the Hot Restart Controller.

Once a restartable actor has been registered and loaded, it runs under the control of the Hot Restart Controller. If the actor fails, the failure will provoke the restart of all the direct members of its restart group. These direct actors are then responsible for

restarting any indirect actors registered in the group. To query an actor's restart group, use `hrGetActorGroup(2RESTART)`:

```
#include < hr/hr.h >
hrGetActorGroup(int aid)
```

4.2.2 Group Restart

In the context of hot restart, an actor is considered to have abnormally terminated (and will therefore provoke the restart of its group) if any of the following occur:

- unrecoverable error (division by zero, unresolved page fault, invalid op code, and so on)
- premature exit signal, that is, an exit signal prior to the expected completion of the actor's task
- the actor is killed without using the restart-specific command (`akill(1M)` with the `–g` option) or function call (`hrKillGroup(2RESTART)`) provided for this purpose

There is no single API call which can explicitly force a group of actors to restart. For cases in which it may be desirable to provoke a restart (for example, for testing purposes), the easiest way to do so is to deliberately provoke one of the above cases. In the "hello world" example introduced in the previous chapter, this was done by causing a segmentation fault.

When an actor fails, all actors in the failed actor's restart group stop executing and the Hot Restart Controller restarts all direct actors in the group from their initial entry point. The direct actors are responsible for restarting any indirect actors, using `hrfexec()`. When `hrfexec()` is called with a name which is already registered in the Hot Restart Controller, the Controller recognizes the actor name and simply restarts the actor from the actor's actor image, instead of loading it from stable storage.

A restartable actor is always restarted at the same address. Its capacity, actor ID and user ID are not guaranteed to be the same after restart. All system resources obtained before the restart are lost: in particular, open files, including those that had been inherited at the time of initial creation are lost. This may include the standard I/O connected to an `rsh` connection.

A restarted actor uses the same arguments and environment parameters that were specified when the actor was initially started. For direct restartable actors, a new set of pre-open `stdin`/`stdout`/`stderr` is provided, which are connected to `/dev/console`. For indirect members, a new set of pre-open `stdin`/`stdout`/`stderr` is provided by the invoker of `hrfexec()`, just as for `afexec(2K)`.

4.2.3 Freeing Persistent Memory

Just like any actor, a restartable actor can free persistent memory blocks using `pmmFree()` or `pmmFreeAll()`. This is described in Section 3.4.2 “Freeing a Persistent Memory Block Explicitly” on page 35.

Restartable actors which allocate memory with `pmmAllocate()` can also use a simple, *automatic* de-allocation mechanism provided by the Hot Restart Controller. This saves the actor from having to free its persistent memory explicitly. Instead, the persistent memory will remain allocated for the lifetime of the actor’s group, and then be freed automatically by the Hot Restart Controller when the last member of the actor’s restart group terminates cleanly. The disadvantage of this system is that the lifetime of the restart group may extend well beyond the point at which the memory block is no longer needed. In this case the memory block will take up space in persistent memory unnecessarily.

To mark a persistent memory block for automatic de-allocation by the Hot Restart Controller, pass the macros `HR_GROUP_KEY` and `HR_GROUP_KEYSIZE` as the *delKey* and *delKeySize* arguments respectively in the call to `pmmAllocate()`. These macros tie the lifetime of the persistent memory block to the lifetime of the calling actor’s restart group.

A block marked for automatic de-allocation by the Hot Restart Controller can still be freed explicitly by calling `pmmFree()` with the block’s `PmmName`. However, attempting to call `pmmFreeAll()` by passing the `HR_GROUP_KEY` and `HR_GROUP_KEYSIZE` macros will result in an error, as this is not permitted.

4.2.4 Clean Termination

As described in Section 4.2.2 “Group Restart” on page 41, any actor that exits before the expected completion of its task is considered to have aborted abnormally and will cause the restart of its actor group. This is useful for cases in which the actor does indeed exit prematurely as a result of an error. This mechanism is also useful for provoking an actor restart where this is explicitly desired, for example, when an execution problem is detected.

To allow a restartable actor to terminate cleanly without causing a restart, use the `HR_EXIT_HDL()` macro prior to the call to `exit(3STDC)`:

```
#include <hr/hr.h>
HR_EXIT_HDL();
```

The purpose of this macro is to add an additional hot restart exit handler to the actor’s `atexit(3STDC)` function. The hot restart exit handler effectively removes the concerned actor from the Hot Restart Controller’s responsibility: once an actor has called `HR_EXIT_HDL()`, the Hot Restart Controller will no longer monitor it for abnormal termination. As a result, when the actor exits, it will terminate cleanly and no longer trigger a restart.

Call the HR_EXIT_HDL() macro shortly before the actor exits. Calling the macro earlier in the actor code will mean that any unexpected exit between the macro call and the final exit will not be detected by the Hot Restart Controller. As a result, the actor will not be restarted if it exits abnormally.

Cleanly terminating an actor does not deregister the actor in the Hot Restart Controller, or remove the actor's actor image and executing image from persistent memory. This is because a cleanly terminated actor will still be restarted if its group is restarted, since a group is always restarted in its initial state. In other words, when a group is restarted, *all* direct restartable actors will recommence execution at their initial entry point, regardless of whether or not they had already exited before the restart occurred. This is shown in the following diagram. Both direct actor 1 (DA1) and indirect actor 2 (A2) terminate cleanly, but are restarted when direct actor 2 (DA2) crashes.

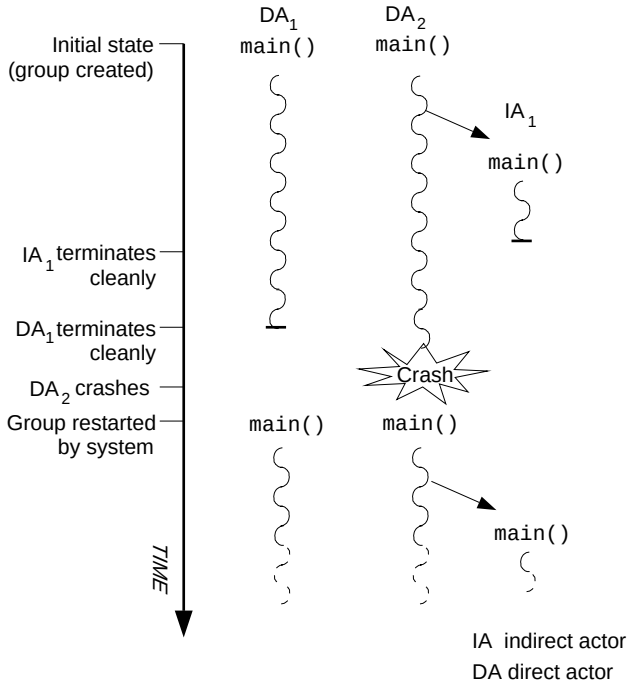

Figure 4–1 Restart of Cleanly Terminated Actors

Because of this behavior, it can be useful to record the clean termination of restartable actors which will never need to be re-executed completely during a group's lifetime by setting a flag in persistent memory. A restarted actor can check the state of this flag at the start of its execution, and thus detect whether it should re-execute or not.

4.2.4.1 Group Termination

For each group of restartable actors present in a ChorusOS system, the Hot Restart Controller stores a list of the actors in the group in a persistent memory block. An actor is added to the list when it is first started. When an actor cleanly terminates, the Hot Restart Controller notes this in the list. When all actors in the list have terminated cleanly, the Hot Restart Controller does the following:

- De-allocates the persistent memory blocks used to store the images of the terminated actors, as well as blocks which were allocated using the `HR_GROUP_KEY` and `HR_GROUP_KEYSIZE` deletion key macros (see Section 4.2.3 “Freeing Persistent Memory” on page 42). The actor names used by the actors can then be reused by other restartable actors, which will be loaded into memory as new actors.
- Adds the group’s ID to the list of available IDs for new actor groups.

A group of actors can only terminate if all of its member actors terminate cleanly. This is important to remember in situations where not all indirect actors are restarted after a group restart. This is a matter of execution flow: if certain conditions in a direct actor change the actor’s flow from one execution to the next, the direct actor may not restart an indirect actor which was running prior to the restart. As a result, the indirect actor will never terminate cleanly and so the group will not be able to terminate.

For example, consider the situation in the following figure. The direct actor spawns the indirect actor only if a certain condition is fulfilled. This condition is fulfilled the first time the direct actor runs. After the direct actor restarts, the condition is no longer fulfilled, so the indirect actor is no longer spawned.

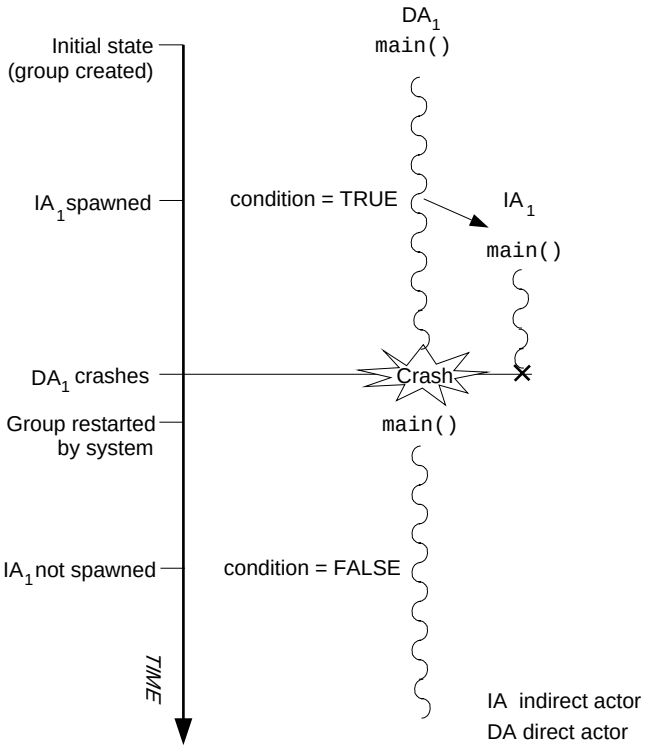

Figure 4–2 Conditional Spawning of a Restartable Actor

In the situation illustrated above, the actor group will not be able to terminate until the indirect actor has been rerun using `hrfexec()`, and has terminated cleanly.

When a restart group cannot terminate because of one or more direct actors in this situation, the Hot Restart Controller detects the fact and prints the following message on the target console:

```
HR_CTRL: group gid blocked, some members have not terminated: list_of_actors
```

gid is the ID of the group concerned, and *list_of_actors* provides the name of each actor which prevents the group from terminating. When this message appears, a basic solution is to kill the actor group using the `akill` command with the `–g` option, as described in Section 4.3 “Killing Restartable Actors” on page 46. This solution is only useful if none of the indirect actors need to be run to complete the group’s task.

A better solution is to use careful application design. If the situation is likely to occur, flags can be stored in persistent memory to indicate indirect actors which have not terminated cleanly. An actor can then be made responsible for cleaning up the group, that is, restarting each indirect actor which is flagged. This clean-up actor can be run using the `arun –g` command when the Hot Restart Controller notification appears on the target console. Alternatively, the group could be designed so that the

clean-up actor is always run just before the group is expected to terminate, in which case the problem is solved without the need for access to the C_INIT console.

4.3 Killing Restartable Actors

At times it may be desirable to circumvent the automatic restart mechanism provided by the Hot Restart Controller and explicitly terminate (kill) a restartable actor. Actors which are killed will not be restarted. Killing an actor automatically kills all actors within the actor's restart group. This is because a restart group must remain consistent, and may not be able to function properly if an actor is no longer present.

Restartable actors can be explicitly killed using either of the following:

- the C_INIT command `akill(1M)` with the `–g` option,
- the API call `hrKillGroup(2RESTART)` with the actor's group ID:

```
#include <hr/hr.h>
   int hrKillGroup (int groupId);
```

The group ID can be queried using the `hrGetActorGroup(2RESTART)` call:

```
#include <hr/hr.h>
   int hrGetActorGroup(int aid);
```

Either method has the same result: all actors in the associated restart group are killed, and the Hot Restart Controller terminates the group as though all actors had exited cleanly (see Section 4.2.4.1 "Group Termination" on page 44).

4.4 Site Restart

A site restart is a hot restart of the whole system. All data of boot actors are reset to their original values from the previously loaded archive, and the system enters its start-up phase again. As C_INIT restarts, `sysadm.ini` is executed again. Any calls to start restartable actors in the `sysadm.ini` file are ignored for a site restart, as all direct restartable actors are restarted automatically by the system once `sysadm.ini` has been read.

Note - When the system is restarted, previously mounted disks are not automatically remounted. To solve this problem, ensure that they are mounted in the `sysadm.ini` file, or create a hot restartable actor that will mount the disks.

A site restart can be provoked automatically, by the Hot Restart Controller, according to the tunable parameters defining the system's restart policy. This is described in Section 2.1.3 "Tunable Parameters" on page 25.

To provoke a site restart programmatically, use the `sysShutdown(2K)` function call with the `–i 1` arguments:

```
int sysShutdown (int argv, char** argc)
```

To provoke a site restart from the C_INIT command-line console, use the command `shutdown –i 1` or `restart(1M)`.

4.5 Putting It All Together: the restartSpawn Example Program

A programming example, `restartSpawn`, illustrates many of the function calls covered in this chapter and previous chapters. The example is provided as a framework illustration of the restart mechanism and the use of persistent memory. Parts of the example could be used as the basis of a more complex user application that incorporates hot restart.

The `restartSpawn` example uses two restartable actors, a parent actor, `HR_parent.r` and a child, `HR_child.r` which is spawned by the parent. Both actors should be compiled as supervisor actors. The source code for the two actors is provided in Appendix B. The example can be summarized as follows:

- The parent actor uses a set of control structures stored in persistent memory. It spawns the child actor using `hrfexec()` , then explicitly crashes, causing itself to be restarted by the system. It indirectly restarts the child actor each time it runs, through the call to `hrfexec()`.
- The child actor also uses a set of control structures stored in persistent memory. It executes a four-step loop which causes the following to be printed:

```
=========== Message ===========
   STEP 1  STEP 2  STEP 3  STEP 4
  ======== End of message========
```

The message is printed independently of the number of times the parent actor crashes or the site is restarted.

APPENDIX **A**

Hot Restart Programming Environment

This appendix describes the environment used for programming and compiling applications which use the API exported by the hot restart feature. For more general information about compiling and linking ChorusOS actors, see the *ChorusOS 4.0 Introduction*.

A.1 Hot Restart Header Files and Directories

The hot restart programming interface is declared in the following files:

For the Persistent Memory Manager API:

install_dir/chorus-*family*/kernel/include/chorus/pmm/chPmm.h

For the Hot Restart Controller API:

install_dir/chorus-*family*/os/include/chorus/hr/hr.h
install_dir/chorus-*family*/os/include/chorus/hr/hrCtrl.h

Detailed descriptions of each function call are available in the ChorusOS man page collection.

A.2 Make Environment

A restartable actor can be compiled using any of the following standard Imakefile macros

- UserActorTarget

- SupActorTarget
- CXXUserActorTarget
- CXXSupActorTarget

Actors using dynamic libraries (compiled with Imake macros of the type Dynamic...Target) cannot be hot restartable.

Use hints in the following table to link actors that use the API exported by the hot restart feature. Note that all ChorusOS actors are automatically linked with the libcx.a library.

API Function	Library
hrfexec() HR_EXIT_HDL() hrKillGroup() hrGetActorGroup()	libcx.a
pmmAllocate() pmmFree() pmmFreeAll()	pmmlib.a

The following is an example Imakefile for a restartable actor which uses the Persistent Memory Manager API:

```
SRCS = HR_actor.c

UserActorTarget(HR_actor_s, HR_actor.o, $(NUCLEUS_DIR)/lib/pmm/pmmlib.a)
Depend($(SRCS))
```

APPENDIX **B**

Example Application Code

This appendix provides the following:

- instructions for compiling and running the “hello world” and actor spawn examples described in this guide.
- source code and makefiles for these examples.

The source code for the example applications is also provided in *install_dir*/`chorus-`*family*`/src/opt/examples` once the examples package has been installed on your system.

B.1 Compiling and Running the Examples

Two examples which are designed to illustrate the use of the hot restart API are provided with the Sun Embedded WorkShop™ product. These examples are as follows:

- `helloRestart`: a very simple illustration of persistent memory programming using a ’hello world’ actor. This example is discussed in Chapter 3. `helloRestart` can be compiled as either a supervisor or a user actor.
- `restartSpawn`: an example which illustrates how a hot-restartable actor can be spawned from an actor. This example is introduced in Chapter 4. Both of the actors in the `restartSpawn` example are supervisor actors.

To compile the examples, make sure that the examples directory is included in your system image build configuration. Binaries for all of the examples are provided in *build_dir*/`build-EXAMPLES` once the examples directory has been built.

To run the examples, first copy them to a directory which is mounted on the target, or use the `make root` command to build a root directory to mount.

Use the C_INIT command `arun` with the –g option to run a restartable actor from the command line. For example, to run the 'hello world' restart example:

```
$ rsh target arun -g 0 example_directory/HR_hello_u
```

where *target* is the target name, and *example_directory* is the directory mounted on the target machine where the restartable hello world actor binary is stored. The –g 0 option runs the hello world restartable actor as a member of a restart group with ID 0.

B.2 The "hello world" Restartable Actor

The restartable "hello world" actor is a simple illustration of the use of persistent memory.

See Chapter 3 for a discussion of this actor.

B.2.1 helloRestart.c

```
#include <stdio.h>
#include <pmm/chPmm.h>
#include <hr/hr.h>

#define HR_GROUP "HELLO_GROUP"

    int
main()
{

 int res;
 int any = 1;
 int* counter_p;  /* It will be stored in persistent memory */
 long *p;
 PmmName name;
 KnRgnDesc rgn;

            /*
             * Initialize the name and medium fields
             * to identify the persistent memory block in the system.
             */
 bzero(&name, sizeof(name));
 strcpy(name.medium,"RAM");
 strcpy(name.name,"PM1");

            /*
             * Initialize the block fields
             */
 bzero(&rgn, sizeof(rgn));
 rgn.options = K_ANYWHERE | K_RESERVED;
 rgn.size    = vmPageSize();
```

```
res = rgnAllocate(K_MYACTOR, &rgn);
if (res != K_OK) {
           printf("rgnAllocate() failed res=%d\n", res);
           HR_EXIT_HDL();
           exit(-1);
}

p = (long*) rgn.startAddr;

           /*
            * From now on p is a bad pointer, since
            * VIRTUAL_ADDRESS_SPACE is true.
            */

           /*
            * Allocate the persistent memory block that stores
            * counter_p.
            */
res=pmmAllocate((VmAddr *)&counter_p,
                       &name,sizeof(int),
                       HR_GROUP,
                       sizeof(HR_GROUP));

if (res != K_OK) {
           printf("Cannot allocate or map the persistent memory block called %s."
                  " Error = %d\n", name.name, res);
           HR_EXIT_HDL();
           exit(-1);
}

           /*
            * From the value of *counter_p the actor detects
            * whether it has been hot restarted or not.
            */
if ( *counter_p==0 ) {
               /*
                * This is the first time the actor is run.
                */
           printf("Hello world!\n");

               /*
                * Increment the counter
                */
           (*counter_p)++;

               /*
                * Normally the next instruction causes a core dump and
                * a hot restart of the actor
                */
           *p = 0xDeadBeef;

} else {
               /*
                * The actor has been restarted
                * NOTE: this message will appear on the console!
                */
           printf("The actor has been restarted.\n");

               /*
                * Free the persistent memory block before exiting
```

```
             */
            res = pmmFree(&name);
            if (res != K_OK) {
                printf(" pmmFree failed, res=%d. Exit\n", res);
                HR_EXIT_HDL();
                exit(-1);
            }
             /*
              * Terminate cleanly.
              */
            printf("Example finished. Exit.\n");
            HR_EXIT_HDL();
            exit(0);
 }
            /* Never reached */
}
```

B.2.2 Imakefile for helloRestart.c

```
SRCS = helloRestart.c

SupActorTarget(helloRestart.r, helloRestart.o, $(NUCLEUS_DIR)/lib/pmm/pmmlib.a)
UserActorTarget(helloRestart_u, helloRestart.o, $(NUCLEUS_DIR)/lib/pmm/pmmlib.a)

Depend($(SRCS))
```

B.3 The restartSpawn Example

The restartSpawn example comprises two actors: `HR_parent.r` and `HR_child.r`. Both actors must be compiled as supervisor actors. See Section 4.5 “Putting It All Together: the restartSpawn Example Program” on page 47 for an overview of the restartSpawn example.

B.3.1 HR_parent.c

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <am/afexec.h>
#include <pmm/chPmm.h>
#include <exec/chModules.h>
#include <hr/hr.h>
#include <err.h>
#include <errno.h>
```

```
#define PM_MEDIUM "RAM"
#define PM_NAME "PARENT_PM"
#define MAX_LOOPS 8

/*
 * Some static variables
 */
char baseName[PATH_MAX];
char last_global_data;

/*
 * Declaration of objects that will be stored in persistent memory.
 * restarted: number of times the actor has been restarted.
 * counter: number of times the actor's main loop is run.
 */
typedef struct _HR_Status {
    int restarted;
    int counter;
} HR_Status;

    /*
     * Wait "sec" seconds.
     */
    void
waitSec(int sec)
{
    KnTimeVal delay;
    delay.tmSec = sec;
    delay.tmNSec = 0;

    threadDelay(&delay);
}

    /*
     * Create a child hot restartable actor.
     * Start the child actor only if the parent has
     * not been hot restarted.
     */
    void
childCreate()
{
    KnCap childCap;
    KnActorPrivilege curActPriv;
    PmmName childName;
    int res;
    int childAid = -1;
    char path[PATH_MAX];
    char* argv[3];

    res = actorPrivilege(K_MYACTOR, &curActPriv, NULL);
    if (res != K_OK) {
        printf("actorPrivilege failed, res=%d\n", res);
 HR_EXIT_HDL();
 exit(-1);
    }

    if (curActPriv != K_SUPACTOR) {
     argv[0] = "HR_child";
    } else {
```

```
            argv[0] = "HR_child_u";
        }

        argv[1] = NULL;
        argv[2] = NULL;

        strcpy(childName.medium, "RAM");
        strcpy(childName.name, "CHILD");

        strcpy(path, baseName);
        if (curActPriv == K_SUPACTOR) {
            strcat(path, "HR_child");
        } else {
            strcat(path, "HR_child_u");
        }

        childAid = hrfexecv(&childName, path, &childCap, NULL, argv);

        if (childAid == -1) {
            printf("Cannot hrfexecv(%s), error=%d\n", path, errno);
            HR_EXIT_HDL();
            exit(-1);
        }
    }

        /*
         * Cause a hot restart by exiting without
         * first calling HR_EXIT_HDL().
         */
        void
    crash_exit()
    {
        printf("\nPARENT hot-restarts (exits with no HR_EXIT_HDL)!\n");
        exit(1);
    }

        /*
         * Cause a segmentation fault.
         */
        void
    crash_seg()
    {
        KnRgnDesc      rgn;
        unsigned long* badSupPtr;
        int            res;

        rgn.options = K_ANYWHERE | K_RESERVED;
        rgn.size    = vmPageSize();
        rgn.opaque1 = NULL;
        rgn.opaque2 = 0;
        res = rgnAllocate(K_MYACTOR, &rgn);
        if (res != K_OK) {
            printf("unable to allocate a page res=%d\n", res);
            return;
        }

        badSupPtr = (unsigned long*) rgn.startAddr;

        printf("\nPARENT crashes (segmentation fault)!\n");
```

```
        /*
         * Generate an unrecoverable page fault, since
         * VIRTUAL_ADDRESS_SPACE is true
         */
    *badSupPtr = (unsigned long) 0xffffffff;

        /*
         * it should never return with
         */
    printf("Can't generate a crash\n");
    return;
}

    /*
     * Cause a failure due to division by 0.
     * Note: This does not crash on some platforms.
     */
    int
crash_div()
{
    int i;
    int z;
    int x = 1;

    printf("\nPARENT tries to crash with division by 0!\n");
    for (i = 10; i > -1; i--) {
        z = x/i;
    }
    return z;
}

    /*
     * Perform a site restart.
     */
    void
site_restart()
{
    char*       argv[3];
    int         res;

    argv[0] = "shutdown";
    argv[1] = "-i";
    argv[2] = "1";

    res = sysShutdown (3, argv);

    if (res) {
        printf("parent error=%d\n", res);

    } else {

        waitSec(5);
        printf("Timeout ! \n");
    }
}

    /*
     * Kill the group actors and free persistent memory
     * blocks allocated by the parent actor.
     */
```

```
    void
clean_up(PmmName *np)
{
    int res;
    int group = 1;
    int actId;

    actId = agetId();

    res=pmmFree(np);
    if (res != K_OK) {
        printf("\nCannot free the persistent memory block called %s."
               " Error = %d\n", np->name, res);
        HR_EXIT_HDL();
        exit(-1);
    }
    printf("\nPersistent memory has been freed.\n");

    group=hrGetActorGroup(actId);
    if (group < 0) {
        printf("Cannot get actor group. Error = %s\n", errno);
        HR_EXIT_HDL();
        exit(-1);
    }

    printf("Example finished. Exit.\n");

    res=hrKillGroup(group);
    if (res != K_OK) {
        printf("Cannot kill actor group %d. Error = %d\n", group, res);
        HR_EXIT_HDL();
        exit(-1);
    }
}

    /*
     * main
     */
    int
main(int argc, char** argv, char**envp)
{

    int res;
    int counter;
    int ref;
    int* mem_version;
    static PmmName name;
    HR_Status* st;
    char* endPath;
    KnActorPrivilege curActPriv;

        /*
         * Check that argc != 0. Otherwise exit.
         */
    if(argc==0) {
        printf("Cannot start this test. argc == %d. Exit.\n", argc);
        HR_EXIT_HDL();
        exit(-1);
    }
```

```
res = actorPrivilege(K_MYACTOR, &curActPriv, NULL);
if (res != K_OK) {
    printf("actorPrivilege failed, res=%d\n", res);
    HR_EXIT_HDL();
    exit(-1);
}

if (curActPriv != K_SUPACTOR) {
    printf("This example can only be run in supervisor mode. Exit.\n");
    HR_EXIT_HDL();
    exit(-1);
}

    /*
     * If the example runs in flat memory mode, it will not work.
     * Some of the failures will not always cause a hot-restart.
     * Print an error message and exit.
     */
res = sysGetConf(K_MODULE_MEM_NAME, K_GETCONF_VERSION, mem_version);
if (res != K_OK) {
    printf("Cannot get memory configuration."
           " res=%d\n", res);
    HR_EXIT_HDL();
    exit(-1);
}

if (*mem_version==K_MEM_VERSION_FLM) {
    printf("Sorry. The example cannot be run in flat memory"
           " configuration. Exit.\n");
    HR_EXIT_HDL();
    exit(-1);
}

    /*
     * Get the directory of the current actor.
     */
strcpy(baseName, argv[0]);
endPath = strrchr(baseName, '/');
*(endPath+1) = '\0';

    /*
     * Initialize the name and medium fields to identify
     * the HR_Status structure.
     */
bzero(&name, sizeof(name));
strcpy(name.medium,PM_MEDIUM);
strcpy(name.name,PM_NAME);

    /*
     * Allocate or map the data in st in persistent memory.
     */
res=pmmAllocate((VmAddr *)&st,
                &name,
                sizeof(HR_Status),
                HR_GROUP_KEY,
                HR_GROUP_KEYSIZE);

if (res != K_OK) {
    printf("Cannot allocate or map the persistent memory block called %s."
           " Error = %d, errno=%d\n", name.name, res, errno);
```

```
    HR_EXIT_HDL();
    exit(-1);
}

    /*
     * If the actor has been restarted, print out a message.
     */
if (st->restarted>0) {
    printf("PARENT RESTARTS (%d-th time)\n", st->restarted);
}

    /*
     * Increase the "restarted" counter.
     */
st->restarted++;


    /*
     * Create a child hot-restartable actor.
     */
childCreate();

    /*
     * main loop
     * provokes different faults in the parent actor.
     * This causes the parent AND the child to hot restart.
     */
while ( st->counter<MAX_LOOPS ) {
    waitSec(2 + rand() % 2);
    st->counter++;
    ref = (st->counter%5);
    switch ( ref ) {
        case 1:
            crash_seg();
            break;
        case 2:
            res = crash_div();
                /*
                 * If you get here, it means that division by 0 does not
                 * crash your system!
                 */
            printf("The parent actor does not crash"
                   " with division by 0. Continue.\n");
            break;
        case 3:
            crash_exit();
            break;
        case 4:
            site_restart();
            break;
        default:
            break;
    }
}

    /*
     * Example complete. Free persistent memory blocks and exit.
     */
clean_up(&name);
```

```
}
```

B.3.2 HR_child.c

```
#include <stdio.h>
#include <strings.h>
#include <pmm/chPmm.h>
#include <hr/hr.h>
#include <exec/chExec.h>
#include <pd/chPd.h>
#include <errno.h>

#define PM_MEDIUM "RAM"
#define PM_NAME "CHILD_PM"
#define MESSAGE_NAME "CHILD_MESSAGE"
#define MESSAGE_SIZE 100

typedef struct _HR_Status {
    int restarted;
    int checkpoint;
} HR_Status;

/*
 * Static variables
 */
static HR_Status *st;

    /*
     * Wait "sec" seconds.
     */
    void
waitSec(int sec)
{
    KnTimeVal delay;
    delay.tmSec = sec;
    delay.tmNSec = 0;

    (void) threadDelay(&delay);
}

    /*
     * General operations in all steps.
     */
    void
gen_step (char** message, char* m_out)
{
    strcat(*message, m_out);
    printf("%s", m_out);
    fflush(NULL);

        /*
         * st is stored in persistent memory.
         * If the actor does not reach the end of the next instruction
         * before a hot restart, the current step will be repeated.
         */
    st->checkpoint=++(st->checkpoint) % 4;
}

    /*
```

```
     * step1
     */
    void
step1 (char** message)
{
    gen_step(message, " STEP 1 ");
}

    /*
     * step2
     */
    void
step2 (char** message)
{
    gen_step(message, " STEP 2 ");
}

    /*
     * step3
     */
    void
step3 (char** message)
{
    gen_step(message, " STEP 3 ");
}

    /*
     * step4
     */
    void
step4 (char** message)
{
    gen_step(message, " STEP 4 ");

        /*
         * Print out the entire message at the end of the cycle.
         * The entire message is printed even if the child actor is
  * restarted during a cycle.
  *
         * =========== Message ===========
         *  STEP 1  STEP 2  STEP 3  STEP 4
         * ======== End of message========
         *
         * Note that output from the parent actor may garble
  * this output.
         */
    printf("\n\n=========== Message ===========\n");
    printf("%s", *message);
    printf("\n======== End of message========\n\n");

        /*
         * Reset the message.
         */
    bzero(*message, MESSAGE_SIZE);
}

    /*
     * Function to be executed before the actor exits for any reason.
     */
    void
```

```
before_exit()
{
    printf("CHILD EXITS!\n");
}

    /*
     * main
     */
    int
main(int argc, char** argv, char**envp)
{

    int res;
    int counter;
    static PmmName name;
    static PmmName m_name;
    size_t size;
    PdKey key;
    char message[MESSAGE_SIZE];
    KnActorPrivilege curActPriv;

    res = actorPrivilege(K_MYACTOR, &curActPriv, NULL);
    if (res != K_OK) {
        printf("actorPrivilege failed, res=%d\n", res);
        HR_EXIT_HDL();
        exit(-1);
    }
    if (curActPriv == K_SUPACTOR) {
            /*
             * Create a private actor data key with a destructor associated
      * with it.
             */
        res = padKeyCreate(&key, (KnPdHdl)before_exit);
        if(res != 0) {
            printf("Couldn't create PD key. Exit with errno %d\n", errno);
            HR_EXIT_HDL();

            exit(-1);
        }

        res = padSet(K_MYACTOR, key, "M");
        if (res != K_OK) {
            printf("Cannot set the PD key, error %d\n", res);
            HR_EXIT_HDL();
            exit(-1);
        }
    } else {
        res=atexit(&before_exit);

            /*
             * atexit() accepts up to 32 functions so this cannot fail.
             */
    }

        /*
         * Initialize the name and medium fields for the HR_Status structure.
         */
    bzero(&name, sizeof(name));
    strcpy(name.medium,PM_MEDIUM);
    strcpy(name.name,PM_NAME);
```

```
    /*
     * Allocate or map the data in st in persistent memory.
     */
res=pmmAllocate((VmAddr *)&st,
                &name,
                sizeof(HR_Status),
                HR_GROUP_KEY,
                HR_GROUP_KEYSIZE);


if (res != K_OK) {
    printf("Cannot allocate or map the persistent memory block called %s."
           " Error = %d\n", name.name, res);
    HR_EXIT_HDL();
    exit(-1);
}

    /*
     * Initialize the name and medium fields for the message char buffer.
     */
bzero(&m_name, sizeof(m_name));
strcpy(m_name.medium,PM_MEDIUM);
strcpy(m_name.name,MESSAGE_NAME);

    /*
     * Allocate or map the message data in persistent memory.
     */
res=pmmAllocate((VmAddr *)&message,
                &m_name,
                MESSAGE_SIZE,
                HR_GROUP_KEY,
                HR_GROUP_KEYSIZE);

if (res != K_OK) {
    printf("Cannot allocate or map the persistent memory block called %s."
           " Error = %d\n", name.name, res);
    HR_EXIT_HDL();
    exit(-1);
}

    /*
     * If the actor has been restarted, print out a message.
     */
if (st->restarted>0) {
    printf("CHILD RESTARTS (%d-th time)\n", st->restarted);
}

    /*
     * Increase the "restarted" counter.
     */
st->restarted++;

    /*
     * Loop forever.
     * Each time the parent actor crashes, the child actor will be
     * stopped with it since they belong
     * to the same group.
     */
while ( 1 ) {
```

```
        waitSec(1);
        switch ( st->checkpoint ) {
            case 0:
                step1(&message);
                break;
            case 1:
                step2(&message);
                break;
            case 2:
                step3(&message);
                break;
            case 3:
                step4(&message);
                break;
            default:
                break;
        }
    }
    return 0;

}
```

B.3.3 Imakefile for HR_parent.c and HR_child.c

```
SRCS = HR_child.c HR_parent.c

SupActorTarget(HR_child.r,HR_child.o,$(NUCLEUS_DIR)/lib/pmm/pmmlib.a)
SupActorTarget(HR_parent.r,HR_parent.o,$(NUCLEUS_DIR)/lib/pmm/pmmlib.a)

Depend($(SRCS))
```

Index

R

S

T