# ChorusOS man pages section
# 2RESTART: Restart System Calls

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

Part No: 806-3329
December 10, 1999


# Contents

# PREFACE

## Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

NAME — This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS — This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ] The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.

. . . Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ‘ "`filename` . . ." .

| Separator. Only one of the arguments separated by this character can be specified at time.

{ } Braces. The options and / or arguments enclosed within braces are

| | |
|---|---|
| | interdependent, such that everything enclosed must be treated as a unit. |
| FEATURES | This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured. |
| DESCRIPTION | This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE. |
| OPTIONS | This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied. |
| OPERANDS | This section lists the command operands and describes how they affect the actions of the command. |
| OUTPUT | This section describes the output - standard output, standard error, or output files - generated by the command. |
| RETURN VALUES | If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES. |
| ERRORS | On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code. |

| | |
|---|---|
| USAGE | This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality:<br>Commands<br>Modifiers<br>Variables<br>Expressions<br>Input Grammar |
| EXAMPLES | This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections. |
| ENVIRONMENT VARIABLES | This section lists any environment variables that the command or function affects, followed by a brief description of the effect. |
| EXIT STATUS | This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions. |
| FILES | This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation. |
| SEE ALSO | This section lists references to other man pages, in-house documentation and outside publications. |
| DIAGNOSTICS | This section lists diagnostic messages with a brief explanation of the condition causing the error. |
| WARNINGS | This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics. |
| NOTES | This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here. |

| | |
|---|---|
| BUGS | This section describes known bugs and wherever possible, suggests workarounds. |

CHAPTER

# Hot Restart and Persistent Memory Services

12

**NAME**

HR_EXIT_HDL – macro to mark a hot restartable actor for clean termination

**SYNOPSIS**

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

**FEATURES**

HOT_RESTART

**DESCRIPTION**

The HR_EXIT_HDL( ) macro is used to add an additional hot restart exit handler to a hot restartable actor's atexit(3STDC) function. The hot restart exit handler informs the Hot Restart Controller that the calling actor has successfully completed its task and can exit normally using exit(3STDC), without triggering a restart. If a restartable actor calls exit(3STDC) without first calling HR_EXIT_HDL( ), the Hot Restart Controller considers this to be an abnormal termination and the actor's group will be restarted.

Call HR_EXIT_HDL( ) shortly before the restartable actor exits with exit(3STDC). If HR_EXIT_HDL( ) is called earlier in application code, and the actor then exits abnormally (before terminating successfully), the Hot Restart Controller will still consider that the actor has terminated successfully and the actor's group will not be restarted.

Once all members registered in a restart group have normally terminated using HR_EXIT_HDL( ) followed by exit(3STDC), the Hot Restart Controller does the following:

- Frees all persistent memory blocks used to store the actor images and executed images for the actors in the group, and unregisters the name of each actor.
- Frees any memory blocks that were allocated by the group's members using the HR_GROUP_KEY deletion key. See the man page for pmmAllocate(2RESTART) for a description of this key.
- Adds the restart group's ID to the list of unused group IDs.

Care must be taken to ensure that all restartable actors registered in a restart group are able to exit cleanly using HR_EXIT_HDL( ). This is important for restartable actors spawned with hrfexec(2RESTART) that are not respawned after a restart. See the man page for hrfexec(2RESTART) for more information concerning clean termination of spawned restartable actors.

**RETURN VALUES**

The HR_EXIT_HDL( ) macro returns 0 if the connection to the exit function succeeds. Otherwise, -1 is returned.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO** `hrfexec`(2RESTART), `hrKillGroup`(2RESTART), `atexit`(3STDC)

**NAME**

hrfexec, hrfexecl, hrfexecv, hrfexecle, hrfexecve, hrfexeclp, hrfexecvp – spawn a hot restartable actor

**SYNOPSIS**

```
#include <hr/hr.h>
int hrfexecl(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecv(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, char *const * argv);

int hrfexecle(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */, char const * envp);

int hrfexecve(PmmName * baseName, const char * path, KnCap * cactorcap, const
AcParam * param, char const * argv, char const * envp);

int hrfexeclp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecvp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, char *const * argv);
```

**FEATURES**

HOT_RESTART

**DESCRIPTION**

The functions in the `hrfexec( )` function family are used to spawn hot restartable actors. `hrfexec( )` functions can only be called by a hot restartable actor. `hrfexec( )` functions provide support for initially loading an actor image (text and data) into persistent memory, and for restarting the actor from this image after a restart. The spawned hot restartable actor is registered and run as a member of the same restart group as the calling actor.

`hrfexec( )` functions accept the same parameters as `afexec`(2K) functions, with the addition of the *baseName* parameter.

The *baseName* parameter is a `PmmName` structure with two fields, `medium` and `name` , as described in the man page for `pmmAllocate`(2RESTART) . The `medium` field must be set to `RAM` . The `name` field is used to register and uniquely identify the spawned hot restartable actor in the Hot Restart Controller. The actor will only be unregistered when the actor's group terminates, or disappears from persistent memory (system reboot). It is the programmer's responsibility to ensure that each hot restartable actor registered with the Hot Restart Controller uses a unique name, as name clashes at run time are not detected by the system, and will cause unpredictable behavior.

When an `hrfexec( )` function is called, the Hot Restart Controller checks whether *baseName->name* identifies an actor which is already registered.

If the actor is not already registered, the Hot Restart Controller does the following:

- Registers the actor with the specified name as a member of the invoking actor's restart group.
- Solicits the Persistent Memory Manager to allocate space in persistent memory for the initialized actor image, and the executed image. The actor image occupies a single persistent memory block containing the actor's text and initialized data. The executed image occupies two persistent memory blocks: one block containing the executed code, and another block containing the initialized data and BSS. Persistent memory blocks used to store an actor's actor image and executing image will only be freed when the actor's restart group terminates cleanly.
- Creates, loads and starts the actor in the space allocated, using `acreate`(2K) , `aload`(2K) and `astart`(2K) . A set of pre-open `stdin` / `stdout` / `stderr` is specified by the invoker of `hrfexec( )` using an `AcParam` structure, which is interpreted in the same way as for `afexec( )` . See `acreate`(2K) for a description of the `AcParam` structure.

If the name is already registered, the Hot Restart Controller considers that the actor needs to be restarted. It ignores the *file* and *param* arguments passed to `hrfexec( )` and re-loads the executed image from the actor image backed-up in persistent memory. The actor is started with a capacity and actor ID which may be different to the capacity and actor ID of previous run-time instances of the actor. The run-time (relocated) code and data of a hot restartable actor are always loaded at the same address. The new start is similar to the initial one, that is, the standard runtime startup routine (usually `_start( )` ) is called. This routine calls the program's `main( )` function.

When a restart occurs, actors launched by `hrfexec( )` in the affected group(s) stop executing but are not restarted automatically. Only actors launched with the C_INIT command `arun`(1M) with the –`g` option are restarted by the system. These actors are then responsible for restarting other hot restartable actors by re-invoking `hrfexec( )` .

**Clean Termination of Spawned Actors**

Actors spawned with an `hrfexec( )` function must cleanly terminate using the macro `HR_EXIT_HDL`(2RESTART) followed by `exit`(3STDC) , as described in the man page for `HR_EXIT_HDL`(2RESTART) . A restartable actor group can only terminate cleanly when every actor that has been run in the group since the group's creation has cleanly terminated. This includes actors spawned by `hrfexec( )` that may not have been restarted after a group restart. This is possible when an actor is spawned subject to a condition that changes after a restart. The spawned actor is registered as a member of the restart group, but will not be able to terminate cleanly as it will not be respawned by its parent after the restart. As a result, the group will not be able to terminate cleanly.

Because of this behavior, care must be taken to ensure that all actors spawned with `hrfexec( )` are given a chance to terminate cleanly, for example by

implementing a clean-up actor to restart any blocked spawned actors before group termination. This is described in the *ChorusOS 4.0 Hot Restart Programmer's Guide* .

**LIMITATIONS**

`gzip` -compressed executables cannot be used as restartable actors, even if the GZ_FILE feature is set. Attempting to pass a `gzip` -compressed file to an `hrfexec()` function will result in an `ENOEXEC` error.

Dynamic libraries cannot be used with restartable actors, even if the DYNAMIC_LIB feature is set. Attempting to run a dynamically-linked actor as a restartable actor will result in an `ENOEXEC` error.

The Hot Restart Controller does not detect name sharing between two distinct actors. It is the programmer's responsibility to ensure that two distinct actors to be registered in the Hot Restart Controller do not use the same name.

**RETURN VALUES**

If successful, `hrfexec()` functions return a non-negative integer that is the actor ID of the new hot restartable actor. Otherwise, `hrfexec()` functions return `-1` and set `errno` to indicate one of the following error conditions.

**ERRORS**

| | |
|---|---|
| [E2BIG] | The number of bytes used by the new actor image's argument list and environment list is greater than `ARG_MAX` bytes. |
| [EACCES] | The invoking actor is not hot restartable. |
| [EFAULT] | Some of the arguments provided are outside the invoking actor's address space. |
| [ENOEXEC] | The actor file is not in a valid executable format, or cannot be run as a hot restartable actor (see the restrictions for `hrfexec()` ). |
| [ENOMEM] | No memory is available for performing the task. |
| [ENOSPC] | Attempt to create more than `hrCtrl.maxActors` hot restartable actors. |
| [EPERM] | Attempt to spawn a registered restartable actor that is already executing on the system. |

Attempt to spawn an actor which is registered in a restart group different to the restart group of the invoking actor.

---

**Note -** Error codes generated by `read`(2POSIX) and `open`(2POSIX) may also be returned.

---

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

HR_EXIT_HDL(2RESTART) , hrKillGroup(2RESTART) , hrGetActorGroup(2RESTART) , pmmAllocate(2RESTART) , afexec(2K) , acreate(2K) , aload(2K) , astart(2K) , arun(1M) .

**NAME**

hrfexec, hrfexecl, hrfexecv, hrfexecle, hrfexecve, hrfexeclp, hrfexecvp – spawn a hot restartable actor

**SYNOPSIS**

```
#include <hr/hr.h>
int hrfexecl(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecv(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, char *const * argv);

int hrfexecle(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */, char const * envp);

int hrfexecve(PmmName * baseName, const char * path, KnCap * cactorcap, const
AcParam * param, char const * argv, char const * envp);

int hrfexeclp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecvp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, char *const * argv);
```

**FEATURES**

HOT_RESTART

**DESCRIPTION**

The functions in the `hrfexec( )` function family are used to spawn hot restartable actors. `hrfexec( )` functions can only be called by a hot restartable actor. `hrfexec( )` functions provide support for initially loading an actor image (text and data) into persistent memory, and for restarting the actor from this image after a restart. The spawned hot restartable actor is registered and run as a member of the same restart group as the calling actor.

`hrfexec( )` functions accept the same parameters as `afexec`(2K) functions, with the addition of the *baseName* parameter.

The *baseName* parameter is a `PmmName` structure with two fields, `medium` and `name` , as described in the man page for `pmmAllocate`(2RESTART) . The `medium` field must be set to `RAM` . The `name` field is used to register and uniquely identify the spawned hot restartable actor in the Hot Restart Controller. The actor will only be unregistered when the actor's group terminates, or disappears from persistent memory (system reboot). It is the programmer's responsibility to ensure that each hot restartable actor registered with the Hot Restart Controller uses a unique name, as name clashes at run time are not detected by the system, and will cause unpredictable behavior.

When an `hrfexec( )` function is called, the Hot Restart Controller checks whether *baseName->name* identifies an actor which is already registered.

If the actor is not already registered, the Hot Restart Controller does the following:

- Registers the actor with the specified name as a member of the invoking actor's restart group.
- Solicits the Persistent Memory Manager to allocate space in persistent memory for the initialized actor image, and the executed image. The actor image occupies a single persistent memory block containing the actor's text and initialized data. The executed image occupies two persistent memory blocks: one block containing the executed code, and another block containing the initialized data and BSS. Persistent memory blocks used to store an actor's actor image and executing image will only be freed when the actor's restart group terminates cleanly.
- Creates, loads and starts the actor in the space allocated, using `acreate`(2K) , `aload`(2K) and `astart`(2K) . A set of pre-open `stdin` / `stdout` / `stderr` is specified by the invoker of `hrfexec( )` using an `AcParam` structure, which is interpreted in the same way as for `afexec( )` . See `acreate`(2K) for a description of the `AcParam` structure.

If the name is already registered, the Hot Restart Controller considers that the actor needs to be restarted. It ignores the *file* and *param* arguments passed to `hrfexec( )` and re-loads the executed image from the actor image backed-up in persistent memory. The actor is started with a capacity and actor ID which may be different to the capacity and actor ID of previous run-time instances of the actor. The run-time (relocated) code and data of a hot restartable actor are always loaded at the same address. The new start is similar to the initial one, that is, the standard runtime startup routine (usually `_start( )` ) is called. This routine calls the program's `main( )` function.

When a restart occurs, actors launched by `hrfexec( )` in the affected group(s) stop executing but are not restarted automatically. Only actors launched with the C_INIT command `arun`(1M) with the –`g` option are restarted by the system. These actors are then responsible for restarting other hot restartable actors by re-invoking `hrfexec( )` .

**Clean Termination of Spawned Actors**

Actors spawned with an `hrfexec( )` function must cleanly terminate using the macro `HR_EXIT_HDL`(2RESTART) followed by `exit`(3STDC) , as described in the man page for `HR_EXIT_HDL`(2RESTART) . A restartable actor group can only terminate cleanly when every actor that has been run in the group since the group's creation has cleanly terminated. This includes actors spawned by `hrfexec( )` that may not have been restarted after a group restart. This is possible when an actor is spawned subject to a condition that changes after a restart. The spawned actor is registered as a member of the restart group, but will not be able to terminate cleanly as it will not be respawned by its parent after the restart. As a result, the group will not be able to terminate cleanly.

Because of this behavior, care must be taken to ensure that all actors spawned with `hrfexec( )` are given a chance to terminate cleanly, for example by

implementing a clean-up actor to restart any blocked spawned actors before group termination. This is described in the *ChorusOS 4.0 Hot Restart Programmer's Guide* .

**LIMITATIONS**

`gzip` -compressed executables cannot be used as restartable actors, even if the GZ_FILE feature is set. Attempting to pass a `gzip` -compressed file to an `hrfexec()` function will result in an `ENOEXEC` error.

Dynamic libraries cannot be used with restartable actors, even if the DYNAMIC_LIB feature is set. Attempting to run a dynamically-linked actor as a restartable actor will result in an `ENOEXEC` error.

The Hot Restart Controller does not detect name sharing between two distinct actors. It is the programmer's responsibility to ensure that two distinct actors to be registered in the Hot Restart Controller do not use the same name.

**RETURN VALUES**

If successful, `hrfexec()` functions return a non-negative integer that is the actor ID of the new hot restartable actor. Otherwise, `hrfexec()` functions return `-1` and set `errno` to indicate one of the following error conditions.

**ERRORS**

| | |
|---|---|
| [E2BIG] | The number of bytes used by the new actor image's argument list and environment list is greater than `ARG_MAX` bytes. |
| [EACCES] | The invoking actor is not hot restartable. |
| [EFAULT] | Some of the arguments provided are outside the invoking actor's address space. |
| [ENOEXEC] | The actor file is not in a valid executable format, or cannot be run as a hot restartable actor (see the restrictions for `hrfexec()` ). |
| [ENOMEM] | No memory is available for performing the task. |
| [ENOSPC] | Attempt to create more than `hrCtrl.maxActors` hot restartable actors. |
| [EPERM] | Attempt to spawn a registered restartable actor that is already executing on the system. |

Attempt to spawn an actor which is registered in a restart group different to the restart group of the invoking actor.

---

**Note -** Error codes generated by `read`(2POSIX) and `open`(2POSIX) may also be returned.

---

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

HR_EXIT_HDL(2RESTART) , hrKillGroup(2RESTART) , hrGetActorGroup(2RESTART) , pmmAllocate(2RESTART) , afexec(2K) , acreate(2K) , aload(2K) , astart(2K) , arun(1M) .

**NAME**

hrfexec, hrfexecl, hrfexecv, hrfexecle, hrfexecve, hrfexeclp, hrfexecvp – spawn a hot restartable actor

**SYNOPSIS**

```
#include <hr/hr.h>
int hrfexecl(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecv(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, char *const * argv);

int hrfexecle(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */, char const * envp);

int hrfexecve(PmmName * baseName, const char * path, KnCap * cactorcap, const
AcParam * param, char const * argv, char const * envp);

int hrfexeclp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecvp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, char *const * argv);
```

**FEATURES**

HOT_RESTART

**DESCRIPTION**

The functions in the `hrfexec( )` function family are used to spawn hot restartable actors. `hrfexec( )` functions can only be called by a hot restartable actor. `hrfexec( )` functions provide support for initially loading an actor image (text and data) into persistent memory, and for restarting the actor from this image after a restart. The spawned hot restartable actor is registered and run as a member of the same restart group as the calling actor.

`hrfexec( )` functions accept the same parameters as `afexec`(2K) functions, with the addition of the *baseName* parameter.

The *baseName* parameter is a `PmmName` structure with two fields, `medium` and `name` , as described in the man page for `pmmAllocate`(2RESTART) . The `medium` field must be set to `RAM` . The `name` field is used to register and uniquely identify the spawned hot restartable actor in the Hot Restart Controller. The actor will only be unregistered when the actor's group terminates, or disappears from persistent memory (system reboot). It is the programmer's responsibility to ensure that each hot restartable actor registered with the Hot Restart Controller uses a unique name, as name clashes at run time are not detected by the system, and will cause unpredictable behavior.

When an `hrfexec( )` function is called, the Hot Restart Controller checks whether *baseName->name* identifies an actor which is already registered.

If the actor is not already registered, the Hot Restart Controller does the following:

- Registers the actor with the specified name as a member of the invoking actor's restart group.
- Solicits the Persistent Memory Manager to allocate space in persistent memory for the initialized actor image, and the executed image. The actor image occupies a single persistent memory block containing the actor's text and initialized data. The executed image occupies two persistent memory blocks: one block containing the executed code, and another block containing the initialized data and BSS. Persistent memory blocks used to store an actor's actor image and executing image will only be freed when the actor's restart group terminates cleanly.
- Creates, loads and starts the actor in the space allocated, using `acreate`(2K) , `aload`(2K) and `astart`(2K) . A set of pre-open `stdin` / `stdout` / `stderr` is specified by the invoker of `hrfexec( )` using an `AcParam` structure, which is interpreted in the same way as for `afexec( )` . See `acreate`(2K) for a description of the `AcParam` structure.

If the name is already registered, the Hot Restart Controller considers that the actor needs to be restarted. It ignores the *file* and *param* arguments passed to `hrfexec( )` and re-loads the executed image from the actor image backed-up in persistent memory. The actor is started with a capacity and actor ID which may be different to the capacity and actor ID of previous run-time instances of the actor. The run-time (relocated) code and data of a hot restartable actor are always loaded at the same address. The new start is similar to the initial one, that is, the standard runtime startup routine (usually `_start( )` ) is called. This routine calls the program's `main( )` function.

When a restart occurs, actors launched by `hrfexec( )` in the affected group(s) stop executing but are not restarted automatically. Only actors launched with the C_INIT command `arun`(1M) with the –`g` option are restarted by the system. These actors are then responsible for restarting other hot restartable actors by re-invoking `hrfexec( )` .

**Clean Termination of Spawned Actors**

Actors spawned with an `hrfexec( )` function must cleanly terminate using the macro `HR_EXIT_HDL`(2RESTART) followed by `exit`(3STDC) , as described in the man page for `HR_EXIT_HDL`(2RESTART) . A restartable actor group can only terminate cleanly when every actor that has been run in the group since the group's creation has cleanly terminated. This includes actors spawned by `hrfexec( )` that may not have been restarted after a group restart. This is possible when an actor is spawned subject to a condition that changes after a restart. The spawned actor is registered as a member of the restart group, but will not be able to terminate cleanly as it will not be respawned by its parent after the restart. As a result, the group will not be able to terminate cleanly.

Because of this behavior, care must be taken to ensure that all actors spawned with `hrfexec( )` are given a chance to terminate cleanly, for example by

implementing a clean-up actor to restart any blocked spawned actors before group termination. This is described in the *ChorusOS 4.0 Hot Restart Programmer's Guide* .

**LIMITATIONS**

`gzip` -compressed executables cannot be used as restartable actors, even if the GZ_FILE feature is set. Attempting to pass a `gzip` -compressed file to an `hrfexec( )` function will result in an `ENOEXEC` error.

Dynamic libraries cannot be used with restartable actors, even if the DYNAMIC_LIB feature is set. Attempting to run a dynamically-linked actor as a restartable actor will result in an `ENOEXEC` error.

The Hot Restart Controller does not detect name sharing between two distinct actors. It is the programmer's responsibility to ensure that two distinct actors to be registered in the Hot Restart Controller do not use the same name.

**RETURN VALUES**

If successful, `hrfexec( )` functions return a non-negative integer that is the actor ID of the new hot restartable actor. Otherwise, `hrfexec( )` functions return `-1` and set `errno` to indicate one of the following error conditions.

**ERRORS**

| | |
|---|---|
| [E2BIG] | The number of bytes used by the new actor image's argument list and environment list is greater than `ARG_MAX` bytes. |
| [EACCES] | The invoking actor is not hot restartable. |
| [EFAULT] | Some of the arguments provided are outside the invoking actor's address space. |
| [ENOEXEC] | Attempt to spawn an actor which is registered in a restart group different to the restart group of the invoking actor. The actor file is not in a valid executable format, or cannot be run as a hot restartable actor (see the restrictions for `hrfexec( )` ). |
| [ENOMEM] | No memory is available for performing the task. |
| [ENOSPC] | Attempt to create more than `hrCtrl.maxActors` hot restartable actors. |
| [EPERM] | Attempt to spawn a registered restartable actor that is already executing on the system. |

**Note -** Error codes generated by `read`(2POSIX) and `open`(2POSIX) may also be returned.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

HR_EXIT_HDL(2RESTART) , hrKillGroup(2RESTART) , hrGetActorGroup(2RESTART) , pmmAllocate(2RESTART) , afexec(2K) , acreate(2K) , aload(2K) , astart(2K) , arun(1M) .

**NAME**

hrfexec, hrfexecl, hrfexecv, hrfexecle, hrfexecve, hrfexeclp, hrfexecvp – spawn a hot restartable actor

**SYNOPSIS**

```
#include <hr/hr.h>
int hrfexecl(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecv(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, char *const * argv);

int hrfexecle(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */, char const * envp);

int hrfexecve(PmmName * baseName, const char * path, KnCap * cactorcap, const
AcParam * param, char const * argv, char const * envp);

int hrfexeclp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecvp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, char *const * argv);
```

**FEATURES**

HOT_RESTART

**DESCRIPTION**

The functions in the `hrfexec( )` function family are used to spawn hot restartable actors. `hrfexec( )` functions can only be called by a hot restartable actor. `hrfexec( )` functions provide support for initially loading an actor image (text and data) into persistent memory, and for restarting the actor from this image after a restart. The spawned hot restartable actor is registered and run as a member of the same restart group as the calling actor.

`hrfexec( )` functions accept the same parameters as `afexec`(2K) functions, with the addition of the *baseName* parameter.

The *baseName* parameter is a `PmmName` structure with two fields, `medium` and `name` , as described in the man page for `pmmAllocate`(2RESTART) . The `medium` field must be set to `RAM` . The `name` field is used to register and uniquely identify the spawned hot restartable actor in the Hot Restart Controller. The actor will only be unregistered when the actor's group terminates, or disappears from persistent memory (system reboot). It is the programmer's responsibility to ensure that each hot restartable actor registered with the Hot Restart Controller uses a unique name, as name clashes at run time are not detected by the system, and will cause unpredictable behavior.

When an `hrfexec( )` function is called, the Hot Restart Controller checks whether *baseName->name* identifies an actor which is already registered.

If the actor is not already registered, the Hot Restart Controller does the following:

- Registers the actor with the specified name as a member of the invoking actor's restart group.
- Solicits the Persistent Memory Manager to allocate space in persistent memory for the initialized actor image, and the executed image. The actor image occupies a single persistent memory block containing the actor's text and initialized data. The executed image occupies two persistent memory blocks: one block containing the executed code, and another block containing the initialized data and BSS. Persistent memory blocks used to store an actor's actor image and executing image will only be freed when the actor's restart group terminates cleanly.
- Creates, loads and starts the actor in the space allocated, using `acreate`(2K) , `aload`(2K) and `astart`(2K) . A set of pre-open `stdin` / `stdout` / `stderr` is specified by the invoker of `hrfexec( )` using an `AcParam` structure, which is interpreted in the same way as for `afexec( )` . See `acreate`(2K) for a description of the `AcParam` structure.

If the name is already registered, the Hot Restart Controller considers that the actor needs to be restarted. It ignores the *file* and *param* arguments passed to `hrfexec( )` and re-loads the executed image from the actor image backed-up in persistent memory. The actor is started with a capacity and actor ID which may be different to the capacity and actor ID of previous run-time instances of the actor. The run-time (relocated) code and data of a hot restartable actor are always loaded at the same address. The new start is similar to the initial one, that is, the standard runtime startup routine (usually `_start( )` ) is called. This routine calls the program's `main( )` function.

When a restart occurs, actors launched by `hrfexec( )` in the affected group(s) stop executing but are not restarted automatically. Only actors launched with the C_INIT command `arun`(1M) with the –`g` option are restarted by the system. These actors are then responsible for restarting other hot restartable actors by re-invoking `hrfexec( )` .

**Clean Termination of Spawned Actors**

Actors spawned with an `hrfexec( )` function must cleanly terminate using the macro `HR_EXIT_HDL`(2RESTART) followed by `exit`(3STDC) , as described in the man page for `HR_EXIT_HDL`(2RESTART) . A restartable actor group can only terminate cleanly when every actor that has been run in the group since the group's creation has cleanly terminated. This includes actors spawned by `hrfexec( )` that may not have been restarted after a group restart. This is possible when an actor is spawned subject to a condition that changes after a restart. The spawned actor is registered as a member of the restart group, but will not be able to terminate cleanly as it will not be respawned by its parent after the restart. As a result, the group will not be able to terminate cleanly.

Because of this behavior, care must be taken to ensure that all actors spawned with `hrfexec( )` are given a chance to terminate cleanly, for example by

implementing a clean-up actor to restart any blocked spawned actors before group termination. This is described in the *ChorusOS 4.0 Hot Restart Programmer's Guide* .

**LIMITATIONS**

`gzip` -compressed executables cannot be used as restartable actors, even if the GZ_FILE feature is set. Attempting to pass a `gzip` -compressed file to an `hrfexec()` function will result in an `ENOEXEC` error.

Dynamic libraries cannot be used with restartable actors, even if the DYNAMIC_LIB feature is set. Attempting to run a dynamically-linked actor as a restartable actor will result in an `ENOEXEC` error.

The Hot Restart Controller does not detect name sharing between two distinct actors. It is the programmer's responsibility to ensure that two distinct actors to be registered in the Hot Restart Controller do not use the same name.

**RETURN VALUES**

If successful, `hrfexec()` functions return a non-negative integer that is the actor ID of the new hot restartable actor. Otherwise, `hrfexec()` functions return `-1` and set `errno` to indicate one of the following error conditions.

**ERRORS**

| | |
|---|---|
| [E2BIG] | The number of bytes used by the new actor image's argument list and environment list is greater than `ARG_MAX` bytes. |
| [EACCES] | The invoking actor is not hot restartable. |
| [EFAULT] | Some of the arguments provided are outside the invoking actor's address space. |
| [ENOEXEC] | The actor file is not in a valid executable format, or cannot be run as a hot restartable actor (see the restrictions for `hrfexec()` ). |
| [ENOMEM] | No memory is available for performing the task. |
| [ENOSPC] | Attempt to create more than `hrCtrl.maxActors` hot restartable actors. |
| [EPERM] | Attempt to spawn a registered restartable actor that is already executing on the system. |

Attempt to spawn an actor which is registered in a restart group different to the restart group of the invoking actor.

---

**Note -** Error codes generated by `read`(2POSIX) and `open`(2POSIX) may also be returned.

---

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

HR_EXIT_HDL(2RESTART) , hrKillGroup(2RESTART) , hrGetActorGroup(2RESTART) , pmmAllocate(2RESTART) , afexec(2K) , acreate(2K) , aload(2K) , astart(2K) , arun(1M) .

**NAME**

hrfexec, hrfexecl, hrfexecv, hrfexecle, hrfexecve, hrfexeclp, hrfexecvp – spawn a hot restartable actor

**SYNOPSIS**

```
#include <hr/hr.h>
int hrfexecl(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecv(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, char *const * argv);

int hrfexecle(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */, char const * envp);

int hrfexecve(PmmName * baseName, const char * path, KnCap * cactorcap, const
AcParam * param, char const * argv, char const * envp);

int hrfexeclp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecvp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, char *const * argv);
```

**FEATURES**

HOT_RESTART

**DESCRIPTION**

The functions in the `hrfexec( )` function family are used to spawn hot restartable actors. `hrfexec( )` functions can only be called by a hot restartable actor. `hrfexec( )` functions provide support for initially loading an actor image (text and data) into persistent memory, and for restarting the actor from this image after a restart. The spawned hot restartable actor is registered and run as a member of the same restart group as the calling actor.

`hrfexec( )` functions accept the same parameters as `afexec`(2K) functions, with the addition of the *baseName* parameter.

The *baseName* parameter is a `PmmName` structure with two fields, `medium` and `name` , as described in the man page for `pmmAllocate`(2RESTART) . The `medium` field must be set to `RAM` . The `name` field is used to register and uniquely identify the spawned hot restartable actor in the Hot Restart Controller. The actor will only be unregistered when the actor's group terminates, or disappears from persistent memory (system reboot). It is the programmer's responsibility to ensure that each hot restartable actor registered with the Hot Restart Controller uses a unique name, as name clashes at run time are not detected by the system, and will cause unpredictable behavior.

When an `hrfexec( )` function is called, the Hot Restart Controller checks whether *baseName->name* identifies an actor which is already registered.

If the actor is not already registered, the Hot Restart Controller does the following:

- Registers the actor with the specified name as a member of the invoking actor's restart group.
- Solicits the Persistent Memory Manager to allocate space in persistent memory for the initialized actor image, and the executed image. The actor image occupies a single persistent memory block containing the actor's text and initialized data. The executed image occupies two persistent memory blocks: one block containing the executed code, and another block containing the initialized data and BSS. Persistent memory blocks used to store an actor's actor image and executing image will only be freed when the actor's restart group terminates cleanly.
- Creates, loads and starts the actor in the space allocated, using `acreate`(2K) , `aload`(2K) and `astart`(2K) . A set of pre-open `stdin` / `stdout` / `stderr` is specified by the invoker of `hrfexec( )` using an `AcParam` structure, which is interpreted in the same way as for `afexec( )` . See `acreate`(2K) for a description of the `AcParam` structure.

If the name is already registered, the Hot Restart Controller considers that the actor needs to be restarted. It ignores the *file* and *param* arguments passed to `hrfexec( )` and re-loads the executed image from the actor image backed-up in persistent memory. The actor is started with a capacity and actor ID which may be different to the capacity and actor ID of previous run-time instances of the actor. The run-time (relocated) code and data of a hot restartable actor are always loaded at the same address. The new start is similar to the initial one, that is, the standard runtime startup routine (usually `_start( )` ) is called. This routine calls the program's `main( )` function.

When a restart occurs, actors launched by `hrfexec( )` in the affected group(s) stop executing but are not restarted automatically. Only actors launched with the C_INIT command `arun`(1M) with the –`g` option are restarted by the system. These actors are then responsible for restarting other hot restartable actors by re-invoking `hrfexec( )` .

**Clean Termination of Spawned Actors**

Actors spawned with an `hrfexec( )` function must cleanly terminate using the macro `HR_EXIT_HDL`(2RESTART) followed by `exit`(3STDC) , as described in the man page for `HR_EXIT_HDL`(2RESTART) . A restartable actor group can only terminate cleanly when every actor that has been run in the group since the group's creation has cleanly terminated. This includes actors spawned by `hrfexec( )` that may not have been restarted after a group restart. This is possible when an actor is spawned subject to a condition that changes after a restart. The spawned actor is registered as a member of the restart group, but will not be able to terminate cleanly as it will not be respawned by its parent after the restart. As a result, the group will not be able to terminate cleanly.

Because of this behavior, care must be taken to ensure that all actors spawned with `hrfexec( )` are given a chance to terminate cleanly, for example by

implementing a clean-up actor to restart any blocked spawned actors before group termination. This is described in the *ChorusOS 4.0 Hot Restart Programmer's Guide* .

**LIMITATIONS**

`gzip` -compressed executables cannot be used as restartable actors, even if the GZ_FILE feature is set. Attempting to pass a `gzip` -compressed file to an `hrfexec()` function will result in an `ENOEXEC` error.

Dynamic libraries cannot be used with restartable actors, even if the DYNAMIC_LIB feature is set. Attempting to run a dynamically-linked actor as a restartable actor will result in an `ENOEXEC` error.

The Hot Restart Controller does not detect name sharing between two distinct actors. It is the programmer's responsibility to ensure that two distinct actors to be registered in the Hot Restart Controller do not use the same name.

**RETURN VALUES**

If successful, `hrfexec()` functions return a non-negative integer that is the actor ID of the new hot restartable actor. Otherwise, `hrfexec()` functions return `-1` and set `errno` to indicate one of the following error conditions.

**ERRORS**

| | |
|---|---|
| [E2BIG] | The number of bytes used by the new actor image's argument list and environment list is greater than `ARG_MAX` bytes. |
| [EACCES] | The invoking actor is not hot restartable. |
| [EFAULT] | Some of the arguments provided are outside the invoking actor's address space. |
| [ENOEXEC] | The actor file is not in a valid executable format, or cannot be run as a hot restartable actor (see the restrictions for `hrfexec()` ). |
| [ENOMEM] | No memory is available for performing the task. |
| [ENOSPC] | Attempt to create more than `hrCtrl.maxActors` hot restartable actors. |
| [EPERM] | Attempt to spawn a registered restartable actor that is already executing on the system. |

Attempt to spawn an actor which is registered in a restart group different to the restart group of the invoking actor.

---

**Note -** Error codes generated by `read`(2POSIX) and `open`(2POSIX) may also be returned.

---

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`HR_EXIT_HDL`(2RESTART) , `hrKillGroup`(2RESTART) , `hrGetActorGroup`(2RESTART) , `pmmAllocate`(2RESTART) , `afexec`(2K) , `acreate`(2K) , `aload`(2K) , `astart`(2K) , `arun`(1M) .

**NAME**

hrfexec, hrfexecl, hrfexecv, hrfexecle, hrfexecve, hrfexeclp, hrfexecvp – spawn a hot restartable actor

**SYNOPSIS**

```
#include <hr/hr.h>
int hrfexecl(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecv(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, char *const * argv);

int hrfexecle(PmmName * baseName, const char * path, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */, char const * envp);

int hrfexecve(PmmName * baseName, const char * path, KnCap * cactorcap, const
AcParam * param, char const * argv, char const * envp);

int hrfexeclp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, const char * arg0, (...), const char * argn, char * /* NULL */);

int hrfexecvp(PmmName * baseName, const char * file, KnCap * cactorcap, const AcParam
* param, char *const * argv);
```

**FEATURES**

HOT_RESTART

**DESCRIPTION**

The functions in the `hrfexec( )` function family are used to spawn hot restartable actors. `hrfexec( )` functions can only be called by a hot restartable actor. `hrfexec( )` functions provide support for initially loading an actor image (text and data) into persistent memory, and for restarting the actor from this image after a restart. The spawned hot restartable actor is registered and run as a member of the same restart group as the calling actor.

`hrfexec( )` functions accept the same parameters as `afexec`(2K) functions, with the addition of the *baseName* parameter.

The *baseName* parameter is a `PmmName` structure with two fields, `medium` and `name` , as described in the man page for `pmmAllocate`(2RESTART) . The `medium` field must be set to `RAM` . The `name` field is used to register and uniquely identify the spawned hot restartable actor in the Hot Restart Controller. The actor will only be unregistered when the actor's group terminates, or disappears from persistent memory (system reboot). It is the programmer's responsibility to ensure that each hot restartable actor registered with the Hot Restart Controller uses a unique name, as name clashes at run time are not detected by the system, and will cause unpredictable behavior.

When an `hrfexec( )` function is called, the Hot Restart Controller checks whether *baseName->name* identifies an actor which is already registered.

If the actor is not already registered, the Hot Restart Controller does the following:

- Registers the actor with the specified name as a member of the invoking actor's restart group.
- Solicits the Persistent Memory Manager to allocate space in persistent memory for the initialized actor image, and the executed image. The actor image occupies a single persistent memory block containing the actor's text and initialized data. The executed image occupies two persistent memory blocks: one block containing the executed code, and another block containing the initialized data and BSS. Persistent memory blocks used to store an actor's actor image and executing image will only be freed when the actor's restart group terminates cleanly.
- Creates, loads and starts the actor in the space allocated, using `acreate`(2K) , `aload`(2K) and `astart`(2K) . A set of pre-open `stdin` / `stdout` / `stderr` is specified by the invoker of `hrfexec( )` using an `AcParam` structure, which is interpreted in the same way as for `afexec( )` . See `acreate`(2K) for a description of the `AcParam` structure.

If the name is already registered, the Hot Restart Controller considers that the actor needs to be restarted. It ignores the *file* and *param* arguments passed to `hrfexec( )` and re-loads the executed image from the actor image backed-up in persistent memory. The actor is started with a capacity and actor ID which may be different to the capacity and actor ID of previous run-time instances of the actor. The run-time (relocated) code and data of a hot restartable actor are always loaded at the same address. The new start is similar to the initial one, that is, the standard runtime startup routine (usually `_start( )` ) is called. This routine calls the program's `main( )` function.

When a restart occurs, actors launched by `hrfexec( )` in the affected group(s) stop executing but are not restarted automatically. Only actors launched with the C_INIT command `arun`(1M) with the –`g` option are restarted by the system. These actors are then responsible for restarting other hot restartable actors by re-invoking `hrfexec( )` .

**Clean Termination of Spawned Actors**

Actors spawned with an `hrfexec( )` function must cleanly terminate using the macro `HR_EXIT_HDL`(2RESTART) followed by `exit`(3STDC) , as described in the man page for `HR_EXIT_HDL`(2RESTART) . A restartable actor group can only terminate cleanly when every actor that has been run in the group since the group's creation has cleanly terminated. This includes actors spawned by `hrfexec( )` that may not have been restarted after a group restart. This is possible when an actor is spawned subject to a condition that changes after a restart. The spawned actor is registered as a member of the restart group, but will not be able to terminate cleanly as it will not be respawned by its parent after the restart. As a result, the group will not be able to terminate cleanly.

Because of this behavior, care must be taken to ensure that all actors spawned with `hrfexec( )` are given a chance to terminate cleanly, for example by

implementing a clean-up actor to restart any blocked spawned actors before group termination. This is described in the *ChorusOS 4.0 Hot Restart Programmer's Guide* .

**LIMITATIONS**

`gzip` -compressed executables cannot be used as restartable actors, even if the GZ_FILE feature is set. Attempting to pass a `gzip` -compressed file to an `hrfexec()` function will result in an `ENOEXEC` error.

Dynamic libraries cannot be used with restartable actors, even if the DYNAMIC_LIB feature is set. Attempting to run a dynamically-linked actor as a restartable actor will result in an `ENOEXEC` error.

The Hot Restart Controller does not detect name sharing between two distinct actors. It is the programmer's responsibility to ensure that two distinct actors to be registered in the Hot Restart Controller do not use the same name.

**RETURN VALUES**

If successful, `hrfexec()` functions return a non-negative integer that is the actor ID of the new hot restartable actor. Otherwise, `hrfexec()` functions return `-1` and set `errno` to indicate one of the following error conditions.

**ERRORS**

| | |
|---|---|
| [E2BIG] | The number of bytes used by the new actor image's argument list and environment list is greater than `ARG_MAX` bytes. |
| [EACCES] | The invoking actor is not hot restartable. |
| [EFAULT] | Some of the arguments provided are outside the invoking actor's address space. |
| [ENOEXEC] | The actor file is not in a valid executable format, or cannot be run as a hot restartable actor (see the restrictions for `hrfexec()` ). |
| [ENOMEM] | No memory is available for performing the task. |
| [ENOSPC] | Attempt to create more than `hrCtrl.maxActors` hot restartable actors. |
| [EPERM] | Attempt to spawn a registered restartable actor that is already executing on the system. |

Attempt to spawn an actor which is registered in a restart group different to the restart group of the invoking actor.

---

**Note -** Error codes generated by `read`(2POSIX) and `open`(2POSIX) may also be returned.

---

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`HR_EXIT_HDL`(2RESTART) , `hrKillGroup`(2RESTART) , `hrGetActorGroup`(2RESTART) , `pmmAllocate`(2RESTART) , `afexec`(2K) , `acreate`(2K) , `aload`(2K) , `astart`(2K) , `arun`(1M) .

**NAME**

hrfexec, hrfexecl, hrfexecv, hrfexecle, hrfexecve, hrfexeclp, hrfexecvp – spawn a hot restartable actor

**SYNOPSIS**

#include <hr/hr.h>

int **hrfexecl**(PmmName * *baseName*, const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, const char * *arg0*, (...), const char * *argn*, char * /* *NULL* */);

int **hrfexecv**(PmmName * *baseName*, const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

int **hrfexecle**(PmmName * *baseName*, const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, const char * *arg0*, (...), const char * *argn*, char * /* *NULL* */, char const * *envp*);

int **hrfexecve**(PmmName * *baseName*, const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char const * *argv*, char const * *envp*);

int **hrfexeclp**(PmmName * *baseName*, const char * *file*, KnCap * *cactorcap*, const AcParam * *param*, const char * *arg0*, (...), const char * *argn*, char * /* *NULL* */);

int **hrfexecvp**(PmmName * *baseName*, const char * *file*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

**FEATURES**

HOT_RESTART

**DESCRIPTION**

The functions in the `hrfexec( )` function family are used to spawn hot restartable actors. `hrfexec( )` functions can only be called by a hot restartable actor. `hrfexec( )` functions provide support for initially loading an actor image (text and data) into persistent memory, and for restarting the actor from this image after a restart. The spawned hot restartable actor is registered and run as a member of the same restart group as the calling actor.

`hrfexec( )` functions accept the same parameters as `afexec`(2K) functions, with the addition of the *baseName* parameter.

The *baseName* parameter is a `PmmName` structure with two fields, `medium` and `name` , as described in the man page for `pmmAllocate`(2RESTART) . The `medium` field must be set to `RAM` . The `name` field is used to register and uniquely identify the spawned hot restartable actor in the Hot Restart Controller. The actor will only be unregistered when the actor's group terminates, or disappears from persistent memory (system reboot). It is the programmer's responsibility to ensure that each hot restartable actor registered with the Hot Restart Controller uses a unique name, as name clashes at run time are not detected by the system, and will cause unpredictable behavior.

When an `hrfexec( )` function is called, the Hot Restart Controller checks whether *baseName->name* identifies an actor which is already registered.

If the actor is not already registered, the Hot Restart Controller does the following:

- Registers the actor with the specified name as a member of the invoking actor's restart group.
- Solicits the Persistent Memory Manager to allocate space in persistent memory for the initialized actor image, and the executed image. The actor image occupies a single persistent memory block containing the actor's text and initialized data. The executed image occupies two persistent memory blocks: one block containing the executed code, and another block containing the initialized data and BSS. Persistent memory blocks used to store an actor's actor image and executing image will only be freed when the actor's restart group terminates cleanly.
- Creates, loads and starts the actor in the space allocated, using `acreate`(2K) , `aload`(2K) and `astart`(2K) . A set of pre-open `stdin` / `stdout` / `stderr` is specified by the invoker of `hrfexec( )` using an `AcParam` structure, which is interpreted in the same way as for `afexec( )` . See `acreate`(2K) for a description of the `AcParam` structure.

If the name is already registered, the Hot Restart Controller considers that the actor needs to be restarted. It ignores the *file* and *param* arguments passed to `hrfexec( )` and re-loads the executed image from the actor image backed-up in persistent memory. The actor is started with a capacity and actor ID which may be different to the capacity and actor ID of previous run-time instances of the actor. The run-time (relocated) code and data of a hot restartable actor are always loaded at the same address. The new start is similar to the initial one, that is, the standard runtime startup routine (usually `_start( )` ) is called. This routine calls the program's `main( )` function.

When a restart occurs, actors launched by `hrfexec( )` in the affected group(s) stop executing but are not restarted automatically. Only actors launched with the C_INIT command `arun`(1M) with the –`g` option are restarted by the system. These actors are then responsible for restarting other hot restartable actors by re-invoking `hrfexec( )` .

**Clean Termination of Spawned Actors**

Actors spawned with an `hrfexec( )` function must cleanly terminate using the macro `HR_EXIT_HDL`(2RESTART) followed by `exit`(3STDC) , as described in the man page for `HR_EXIT_HDL`(2RESTART) . A restartable actor group can only terminate cleanly when every actor that has been run in the group since the group's creation has cleanly terminated. This includes actors spawned by `hrfexec( )` that may not have been restarted after a group restart. This is possible when an actor is spawned subject to a condition that changes after a restart. The spawned actor is registered as a member of the restart group, but will not be able to terminate cleanly as it will not be respawned by its parent after the restart. As a result, the group will not be able to terminate cleanly.

Because of this behavior, care must be taken to ensure that all actors spawned with `hrfexec( )` are given a chance to terminate cleanly, for example by

implementing a clean-up actor to restart any blocked spawned actors before group termination. This is described in the *ChorusOS 4.0 Hot Restart Programmer's Guide* .

**LIMITATIONS**

`gzip` -compressed executables cannot be used as restartable actors, even if the GZ_FILE feature is set. Attempting to pass a `gzip` -compressed file to an `hrfexec()` function will result in an `ENOEXEC` error.

Dynamic libraries cannot be used with restartable actors, even if the DYNAMIC_LIB feature is set. Attempting to run a dynamically-linked actor as a restartable actor will result in an `ENOEXEC` error.

The Hot Restart Controller does not detect name sharing between two distinct actors. It is the programmer's responsibility to ensure that two distinct actors to be registered in the Hot Restart Controller do not use the same name.

**RETURN VALUES**

If successful, `hrfexec()` functions return a non-negative integer that is the actor ID of the new hot restartable actor. Otherwise, `hrfexec()` functions return `-1` and set `errno` to indicate one of the following error conditions.

**ERRORS**

| | |
|---|---|
| [E2BIG] | The number of bytes used by the new actor image's argument list and environment list is greater than `ARG_MAX` bytes. |
| [EACCES] | The invoking actor is not hot restartable. |
| [EFAULT] | Some of the arguments provided are outside the invoking actor's address space. |
| [ENOEXEC] | The actor file is not in a valid executable format, or cannot be run as a hot restartable actor (see the restrictions for `hrfexec()` ). |
| [ENOMEM] | No memory is available for performing the task. |
| [ENOSPC] | Attempt to create more than `hrCtrl.maxActors` hot restartable actors. |
| [EPERM] | Attempt to spawn a registered restartable actor that is already executing on the system. |

Attempt to spawn an actor which is registered in a restart group different to the restart group of the invoking actor.

---

**Note -** Error codes generated by `read`(2POSIX) and `open`(2POSIX) may also be returned.

---

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`HR_EXIT_HDL`(2RESTART) , `hrKillGroup`(2RESTART) , `hrGetActorGroup`(2RESTART) , `pmmAllocate`(2RESTART) , `afexec`(2K) , `acreate`(2K) , `aload`(2K) , `astart`(2K) , `arun`(1M) .

**NAME**

hrGetActorGroup – query the restart group ID for a restartable actor

**SYNOPSIS**

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

**FEATURES**

HOT_RESTART

**DESCRIPTION**

This function returns the group ID of the hot restartable actor with actor ID *aid*.

**RETURN VALUES**

If successful, `hrGetActorGroup( )` returns a non-negative integer that is the ID of the restart group containing the actor with ID *aid*. Otherwise `hrGetActorGroup( )` returns -1 and sets `errno` to indicate the error.

**ERRORS**

[ESRCH] *aid* is not an actor ID for a hot restartable actor.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`hrKillGroup`(2RESTART), `hrfexec`(2RESTART), `agetId`(2K), `arun`(1M)

**NAME**

hrKillGroup – kill a group of restartable actors

**SYNOPSIS**

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

**FEATURES**

HOT_RESTART

**DESCRIPTION**

`hrKillGroup( )` can only be invoked by supervisor actors or trusted user actors.

`hrKillGroup( )` permanently kills all hot restartable actors in the restart group with ID *groupId*, and unregisters them in the Hot Restart Controller. All persistent memory blocks used to store the actor images and executed images for these actors are freed. All persistent memory blocks created using the group's deletion key (`HR_GROUP_KEY`, as described in the man page for `pmmAllocate`(2RESTART)) are also freed. It is the programmer's responsibility to free any persistent memory blocks allocated by the killed actors that use a different deletion key.

When a group has been successfully killed, the group ID that identified it can be used to identify a different restartable actor group.

Calling `hrKillGroup( )` is the only way to kill a restartable actor programmatically. Individual restartable actors cannot be killed independently of their group, as this would leave the group in an inconsistent state after a restart. Calling `akill`(2K) on a restartable actor will simply restart the actor's group, and not kill the actor.

No error is returned if *groupId* is not currently used by a restart group.

`hrKillGroup( )` is functionally equivalent to the C_INIT command `akill`(1M) with the `–g` option.

**RETURN VALUES**

If successful, `hrKillGroup( )` returns 0. Otherwise `hrKillGroup( )` returns -1 and sets `errno` to indicate the error.

**ERRORS**

| | |
|---|---|
| [EACCESS] | The calling actor is neither a trusted actor nor a supervisor actor. |
| [EINVAL] | *groupId* is greater than or equal to the value of the tunable system parameter `hrCtrl.maxGroups`. |

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`hrfexec`(2RESTART), `hrGetActorGroup`(2RESTART), `akill`(1M)

**NAME**

pmmAllocate – allocate a block of persistent memory, retrieve a block of persistent memory

**SYNOPSIS**

#include <pmm/chPmm.h>
KnError **pmmAllocate** (VmAddr **addr*, PmmName **blockName*, size_t *memSize*, PmmDelKey *delKey*, size_t *delKeySize*);

**FEATURES**

HOT_RESTART

**DESCRIPTION**

`pmmAllocate()` allocates or retrieves a persistent memory block and maps it into the caller's address space. A persistent memory block is identified in the system by a `PmmName` structure, defined as follows:

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

In a `PmmName` structure, *medium* is a null-terminated character string that identifies the persistent memory medium in which the block is to be allocated. The only medium supported is `"RAM"`. It is a reserved, fixed-size bank of system memory. The size of the bank is fixed by the `pmm.rambankSize` tunable system parameter. The contents of the bank persist across a hot restart but not across a cold restart.

In a `PmmName` structure, *name* is a user-defined, null-terminated character string that uniquely identifies the memory block in the selected medium.

The *blockName* parameter passed to `pmmAllocate()` must be a valid `PmmName` structure, and the *size* parameter specifies its size in bytes.

`pmmAllocate()` returns the mapping address of the block as the value of *addr*. If *blockName* does not identify an existing persistent memory block, the returned *addr* will be a pointer to a newly-allocated, zero-filled block of persistent memory. If *blockName* identifies an existing memory block, the returned *addr* will be a pointer to the existing block. As long as it is not destroyed, a persistent block is guaranteed to always be mapped at the same address; in other words, *addr* will always point to the same address during the lifetime of the block. If this address is not available in the caller's space, the block cannot be mapped and an error is raised.

The deletion key *delKey* is a binary array that is used to group several blocks and destroy all of them at once, using `pmmFreeAll`(2RESTART). *delKeySize* must be the size of the array, in bytes. If *delKey* is `NULL` or *delKeySize* is zero, `pmmFreeAll()` will not be able to destroy the block. The *delKey* and *delKeySize* parameters are only taken into account when a block is first allocated. If

*blockName* identifies an existing memory block, the *delKey* and *delKeySize* parameters are ignored.

Passing the macros `HR_GROUP_KEY( )` and `HR_GROUP_KEY_SIZE( )` as the *delKey* and *delKeySize* parameters respectively, marks the persistent memory block for automatic freeing by the Hot Restart Controller. Persistent memory blocks that use this special deletion key will be automatically freed when the calling actor's group terminates or is killed, and will not need to be explicitly freed.

Block names and deletion keys are stored in the same medium as the block they are associated with. As the only medium is "`RAM`", all block names and deletion keys are deleted when a cold reboot occurs.

The actor that uses a persistent memory block at any given time does not have to be the same as the actor that created the block, and does not even have to be in any way related to the actor that created the block. However, the following rules apply:

- Persistent memory blocks can only be shared between supervisor actors. No support is provided for sharing blocks of persistent memory with user actors or trusted user actors.

  It is the programmer's responsibility to ensure that this rule is respected. The system does not detect block sharing with user or trusted user actors.

  To share areas of memory with user actors and/or trusted user actors, use the `rgnMapFromActor`(2K) system call.
- `pmmAllocate( )` guarantees that if the requested block of persistent memory already exists, it will be mapped at the same address as at creation, but it is up to the invoker to make sure that the corresponding range of addresses is not already occupied. If the address range is occupied, `pmmAllocate( )` will fail and the block will not be mapped at all.
- Using a block of persistent memory created in a different address space is not supported.

**RETURN VALUE**

Upon successful completion a value of 0 is returned. Otherwise, a negative error code is returned.

**ERRORS**

| | |
|---|---|
| [K_EINVAL] | *delKeySize* is larger than the size of *delKey*. |
| | The medium name is invalid. |
| [K_EFAULT] | Some of the data provided is outside the current actor's address space. |

| | |
|---|---|
| [K_EPRIV] | *blockName->name* begins with "__SYS", that is, identifies a persistent memory block used internally by the system. |
| [K_ENOMEM] | The system limit of persistent memory blocks has been reached. |
| [K_EOVERLAP] | *blockName* identifies an existing block that was created by an actor with higher privileges than the invoking actor. |
| [K_ESPACE] | *blockName* identifies an existing block whose address range would overlap with an existing region in the invoking actor's address space. |

**RESTRICTIONS**

The system will not always detect that the same block of persistent memory is requested by more than one running actor, or by actors running in incompatible address spaces. As no support is provided for sharing persistent memory blocks with user actors or trusted user actors, it is the programmer's responsibility to ensure that this situation does not occur. This restriction does not apply to supervisor actors, as sharing persistent memory blocks between two or more supervisor actors is supported.

The system uses persistent memory blocks internally. Their *blockName->name* parameter and deletion key begin with "__SYS". Using these types of names and deletion keys in application code will cause unpredictable behavior.

Calling `rgnFree`(2K) with an argument that overlaps a memory block returned by `pmmAllocate()` will have unpredictable results.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`pmmFree`(2RESTART), `pmmFreeAll`(2RESTART), `rgnAllocate`(2K)

**NAME**

pmmFree – free a persistent memory block

**SYNOPSIS**

KnError **pmmFree**(PmmName **blockName*);

**FEATURES**

HOT_RESTART

**DESCRIPTION**

`pmmFree()` frees the persistent memory block identified by *blockName*, and removes its name from the system.

`pmmFree()` accepts the `PmmName` of any block that has been successfully allocated using `pmmAllocate`(2RESTART). This includes blocks that use the special deletion key `HR_GROUP_KEY`.

See the man page for `pmmAllocate`(2RESTART) for a description of the `PmmName` structure.

**RETURN VALUES**

If successful, K_OK is returned. Otherwise, a negative error code is returned.

**ERRORS**

| | |
|---|---|
| [K_EFAULT] | *blockName* is outside the caller's address space. |
| [K_EINVAL] | *blockName* identifies a block on an invalid medium, that is, *blockName->medium* is not "`RAM`". |
| [K_ENOMEM] | The system does not have enough resources to identify an existing persistent memory block. |
| [K_EPRIV] | *blockName* identifies a persistent memory block on which is defined the system and cannot be freed by the user. |

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`pmmAllocate`(2RESTART), `pmmFreeAll`(2RESTART)

**NAME**

pmmFreeAll – free a group of persistent memory blocks

**SYNOPSIS**

KnError **pmmFreeAll**(PmmDelKey **delKey*, size_t *delKeySize*);

**FEATURES**

HOT_RESTART

**DESCRIPTION**

`pmmFreeAll( )` permanently frees all persistent memory blocks that have been created with the deletion key *delKey*. *delKey* must be a valid deletion key (see `pmmAllocate`(2RESTART)), and *delKeySize* is its size.

The *delKey* parameter must not be `NULL`. The `HR_GROUP_KEY` and `HR_GROUP_KEY_SIZE` macros cannot be used with `pmmFreeAll( )`. Attempting to do so will cause a `K_EPRIV` error.

**RETURN VALUES**

If successful, K_OK is returned. Otherwise a negative error code is returned.

**ERRORS**

| | |
|---|---|
| [K_EPRIV] | *delKey* is an internal system key (beginning with __SYS) and cannot be used. |
| | *delKey* is associated with one or more persistent memory blocks which are internal to the system. These blocks can only be freed by the system. |
| [K_EINVAL] | *delKeySize* is larger than the size of `PmmDelKey`. |
| [K_EFAULT] | *delKey* does not identify an existing or valid persistent memory block. |

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`pmmAllocate`(2RESTART), `pmmFree`(2RESTART)

# Index