ChorusOS man pages section 2K: Kernel System Calls

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

Part No: 806-3326
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: Commands Modifiers Variables Expressions 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

System Calls

NAME

Intro – introduction to ChorusOS error codes and system calls

DESCRIPTION

This manual describes the principal ChorusOS APIs. All system calls and other interfaces provided for use by applications are documented here. In this context “applications” covers a wide range of functions including device drivers, OS personality emulation servers, real time control programs, and so forth.

FEATURES

Each API function is associated with one or more system features. A given interface is available if, and only if, one of its associated features was configured on the target system when that system was built.

Each manual page has a FEATURES section, where the list of features offering this interface is provided.

The features available in ChorusOS are introduced here. For each feature, there is a man page in the 5FEA section which gives an description of the feature and a list of all interfaces associated with the feature.

DATE	time of day service
EVENT	event flag sets
HOT_RESTART	management of restartable actors and persistent memory
IPC	provide OSI stack entry points
IPC_REMOTE	inter-process communication
IPC_REMOTE_COMM	IPC remote communication
LAPBIND	built-in nameserver for local access points
LAPSAFE	safe mode for LAP invocations
LOG	system logging
VIRTUAL_ADDRESS_SPACE	memory management features
ON_DEMAND_PAGING	memory management features
MIPC	message queues
MON	system monitoring
PERF	performance support
RTC	realtime clock
RTMUTEX	real-time mutexes
ROUND_ROBIN	scheduler features
TIMER	system time

SEM	semaphores
USER_MODE	support for user actors
VTIMER	thread execution timing

POSIX-compliant API features

The following features provide POSIX-compatible functions (1003.1, 1003.1b, 1003.1c).

POSIX-THREADS	POSIX 1003.1c `pthread` functions.
POSIX-TIMERS	POSIX 1003.1b real-time clock and timer functions.
POSIX_MQ	POSIX 1003.1b real-time message queues.
POSIX_SHM	POSIX 1003.1b real-time shared memory objects.
POSIX_SOCKETS	POSIX 1003.1j (standard BSD compatible) socket functions.

The POSIX 1003.1c pthread functions and 1003.1b real-time clock and timer functions are delivered into the standard `libcx.a` library.

ERRORS

Warning: For reasons of compatibility with existing standards, different system calls use distinct error return conventions. Always refer to individual man pages when interpreting error codes.

Most ChorusOS system calls are of the integer type and return a direct value which is zero or positive when the call is successful, and negative to indicate an error. The possible error return values are listed below.

The PRIVATE-DATA calls follow a similar convention (error code returned directly by the function) but the error codes are positive, not negative. PRIVATE-DATA error values are indicated below.

The POSIX-THREADS functions whose names begin with `pthread_` follow a similar convention, but with POSIX error return values.

All other POSIX features (POSIX-TIMERS, POSIX_MQ, POSIX_SHM, POSIX_SOCKETS, remaining POSIX-THREADS calls) and in addition the ChorusOS ACTOR_EXTENDED_MNGT, return –1 in case of error and set the *errno* variable. Note that *errno* is not cleared on successive calls, so it should be tested only after an error has been indicated via a function return code. In multi-threaded actors, *errno* is implemented not as a global variable but as a macro which refers to a per-thread *errno* value. A complete list of the POSIX error codes (possible *errno* values) can be found in the file `<errno.h>`.

Standard ChorusOS error codes

The list of error codes is given below, with general meanings. However, this general definition may be overwritten by certain system calls. The meaning should be interpreted according to the type and the circumstances of the call.

When a thread that executes in supervisor address space performs a system call, the arguments of the call are not fully checked by the system. This applies to threads executing in supervisor actors and to threads executing in user actors when the VIRTUAL_ADDRESS_SPACE feature is disabled. In general, when such a thread invokes a system call with wrong arguments the result may be unpredictable.

The implication is that when a man page states that a system call will return a given error code in a given error situation, this will be fully enforced by the system only for threads executing in user address space. For threads executing in supervisor address space, the error situations may not be detected by the system. In particular this is the case for the K_EFAULT error situations.

0 K_OK No error
The call returned successfully.

-1 K_EINVAL Invalid argument
An invalid argument has been provided.

-2 K_ENOMEM Out of resources
The kernel has been unable to perform the call due to a lack of resources.

-3 K_ETOOMUCH Message too big
Attempted to send a message body longer than K_CMSGSIZEMAX bytes.

-4 K_ENOPORT Port invalid
The port local identifier is invalid (the port was never created, or was deleted, or has migrated before or during the call).

-5 K_EFAULT Bad address
The kernel encountered a hardware access fault in attempting to access one of the arguments of the call.

-6 K_ENOTIMP Not implemented
The call, or one of its options, is not implemented in the current version.

-7 K_EUNKNOWN Unreachable destination
Attempt to send an RPC request to an unreachable port.

-8 K_ETIMEOUT Time-out occurred
A system call failed because the time-out occurred.

-9 K_EFAIL Transaction failed
A system call failed because the actor which received the request failed.

-10 K_EABORT Aborted wait

Abortable calls (for example. *ipcReceive, ipcCall, threadSuspend)* have been interrupted because the waiting thread has been aborted.

-11 K_EFULL System queues saturated
An *ipcSend*, *ipcReply* or *ipcCall* failed because the local destination port queue is full, or the local communication system is saturated.

-12 K_EROUND Alignment error
Attempted to allocate a memory region whose starting address is not aligned to a page boundary.

-13 K_EADDR The address of an argument is invalid
Attempted to allocate a memory region overlapping another.

-14 K_EPRIV Privilege violation
Attempted to perform a forbidden operation.

-15 K_EBADMODE Invalid addressing mode
Attempted to send a message with an invalid addressing mode.

-16 K_ESIZE Size error
The size given in the parameter is incompatible with other parameters.

-17 K_EBUSY Memory locking conflict
Attempted to flush a part of a local cache locked in memory.

-18 K_EABORTRPC Aborted RPC
Aborted a remote IPC notification.

-19 K_EMAPPER Mapper access error
Unable to access a mapper.

-20 K_ENOEMPTY Non empty address space
Attempted to duplicate regions within a non-empty address space.

-21 K_EOFFSET Bad segment offset
Attempted to map an impossible part of a segment.

-22 K_EOVERLAP Region overlapping
Attempted to allocate a memory region that would overlap with another one.

-23 K_EPROT Region access rights violation
Attempted to write in a read-only region.

-24 K_ESPACE Address space violation
Attempted to allocate a region outside the valid range of virtual addresses for the address space.

-25 K_EUNDEF Undefined default mapper
The system default mapper is not defined.

-26 K_EARGS Arguments inconsistent
The combination of a number of the call arguments is inconsistent.

-28 K_EBUSYPORT Port in use
Attempted to migrate a port to which a message handler is attached.

-29 K_ELINKFAILURE Link failure
A remote communication protocol failure has been detected during remote IPC.

-30 K_EBADMSG Invalid message identifier
The message identifier parameter is not a valid message address, or the corresponding message has already been consumed.

-31 K_ENOTAVAILABLE System call not available
Attempted to invoke a system call that is not available within this configuration.

PRIVATE-DATA error codes

0 K_OK No error

The call returned successfully.

1 PD_ENOKEY No more keys available
The key table is full, no more keys can be created.

2 PD_ENOMEM Out of resources
The kernel has been unable to perform the call because of a lack of resources.

3 PD_EINVAL Invalid argument
An invalid argument has been given.

4 PD_ESERVER Cannot connect to server
The Private Data Manager is not accessible.

5 PD_EUSER Called from user mode

DEFINITIONS AND ARGUMENT TYPES

All system call arguments are integer or pointer sized. In other words, if an argument is described as "a structure ...", it is actually "a pointer to a structure ...". The following argument types may occur in system calls exported by the Core Executive or by any other module, with the semantics indicated and constant option values.

KnCap actorcap* Usually, *actorcap* is a pointer to a user memory location which either contains an actor capability (input parameter) or will hold an actor capability (output parameter). As an input parameter,

actorcap may contain one of the following special values instead:

K_MYACTOR	Designates the home actor of the calling thread.
K_SVACTOR	Designates the supervisor address space.

KnThreadLid threadli

Usually, *threadli* is a thread local identifier. Instead, *threadli* may contain the special value K_MYSELF, which designates the calling thread.

KnTimeVal waitLimit*

This argument is used in system calls that make the calling thread block until an explicit event or action occurs. It allows the caller to control the maximum waiting time and abortability of the wait. Usually, *waitLimit* is a pointer to a user memory location containing a timeout interval (expressed as a (second,nanosecond) pair, see sysTime(2K)). In this case, if the call blocks, it is ABORTABLE. Instead, *waitLimit* may contain one of the following special values:

K_NOBLOCK

Specifies a polling call. If the relevant condition is not satisfied immediately, the call returns with an error code and does not wait. The call is NONABORTABLE in the sense that it neither consumes the abort state (if the current thread was previously aborted) nor returns K_EABORT. For compatibility with previous versions of the system K_NOBLOCK has the value 0.

K_NOTIMEOUT

Specifies an unlimited wait. The call is ABORTABLE. For compatibility with previous versions of the system K_NOTIMEOUT has the value -1.

K_NOTIMEOUT_NOABORT

Also specifies an unlimited wait. The call is NONABORTABLE.

Site number

The Chorus kernel has the concept of "Site Number" which has the following properties

- There is a unique Site Number per site.
- The site number is a 32 bit unsigned integer
- The site Number is embedded into all structures of the type "KnUniqueId" which are associated to the following kernel entities:
  - IPC ports
  - IPC Group Capabilities (KnCap)
  - Actor Capabilities (KnCap)

The site number of the target is provided to the kernel when you boot the system and can be provided in two different ways:

1. By the boot program within the "siteNumber" field of the "bootConf" structure; in this case, it is the responsability of the boot program to determine the site number of the target. The way the site number is found by the boot is fully "boot dependent", and may be specific to the target board. For instance, it may be stored in the "NVRAM" memory, or computed more dynamically from some (unique) board identifier. When the target is booted with the standard ChorusOS network boot monitor, it is the whole IP address used by the boot monitor which is provided as the site number of the target.
2. Within the boot image through the "chorusSiteId" kernel tunable; in this case, the site number is statically fixed within the boot image. This approach is less flexible that the case above, as the same boot image cannot be booted on multiple similar target boards. For this reason, it should only be used when there is no way for the site number of the target board to be determined when you boot the system.

Note -

- the value fixed through the tunable takes precedence over the value provided by the boot program.
- by default, the site number is set to zero. If the REMOTE IPC feature of the kernel has been configured, a WARNING message is displayed on the console and the REMOTE IPC feature is not activated. In other words, only local ICP communications are enabled when the site number has not been set.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

Name	Description
`_exit`(2K)	terminate a c_actor
`acap`(2K)	get a c_actor capability
`aconf`(2K)	get configurable system variables
`acreate`(2K)	creates a c_actor
`acred`(2K)	get/set c_actor credentials
`actorCreate`(2K)	create an actor
`actorDelete`(2K)	delete an actor
`actorName`(2K)	get and/or set the symbolic name of an actor
`actorPi`(2K)	get and/or set the protection identifier of an actor
`actorPrivilege`(2K)	get and/or set the privilege of an actor
`actorSelf`(2K)	get the current actor capability

actorStart(2K)	See actorStop(2K)
actorStat(2K)	get the status of all actors on a site
actorStop(2K)	Stop an actor; Start an actor
afexec(2K)	create a new c_actor
afexecl(2K)	See afexec(2K)
afexecle(2K)	See afexec(2K)
afexeclp(2K)	See afexec(2K)
afexecv(2K)	See afexec(2K)
afexecve(2K)	See afexec(2K)
afexecvp(2K)	See afexec(2K)
agetId(2K)	get c_actor's ID
agetalparam(2K)	See aload(2K)
akill(2K)	kill or restart an actor
alParamBuild(2K)	See aload(2K)
alParamUnpack(2K)	See aload(2K)
aload(2K)	Manage the loading of an actor
astart(2K)	activates a c_actor
astat(2K)	list all active c_actors
atrace(2K)	c_actor trace
await(2K)	wait for c_actor to terminate or stop
awaits(2K)	See await(2K)
dladdr(2K)	translate address to symbolic information
dlclose(2K)	close a dynamic object
dlerror(2K)	get diagnostic information
dlopen(2K)	gain access to a dynamic object file
dlsym(2K)	get the address of a symbol in a dynamic object
ethIpcStackAttach(2K)	attach an

ethOsiStackAttach(2K)	attach an
eventClear(2K)	See eventInit(2K)
eventInit(2K)	initialize an event flag set; Clear events in an event flag set; Post events in an event flag set; Wait for events in an event flag set
eventPost(2K)	See eventInit(2K)
eventWait(2K)	See eventInit(2K)
grpAllocate(2K)	allocate a port group capability
grpPortInsert(2K)	insert a port into a port group; remove a port from a port group
grpPortRemove(2K)	See grpPortInsert(2K)
ipcCall(2K)	send an RPC request and wait for the reply
ipcGetData(2K)	get the current message body
ipcReceive(2K)	receive a message
ipcReply(2K)	reply to the current message
ipcRestore(2K)	See ipcSave(2K)
ipcSave(2K)	Save the current message; Restore a saved message as the current message
ipcSend(2K)	send a message
ipcSysInfo(2K)	get system information about the current message
ipcTarget(2K)	build a message target
lapDescDup(2K)	See svLapCreate(2K)
lapDescIsZero(2K)	See svLapCreate(2K)
lapDescZero(2K)	See svLapCreate(2K)
lapInvoke(2K)	invoke a lap handler
lapResolve(2K)	See svLapBind(2K)
msgAllocate(2K)	allocates a message from a message space

msgFree(2K)	free a message of a message space
msgGet(2K)	retrieves the first message of a message queue
msgPut(2K)	post a message to a message queue
msgRemove(2K)	remove a message from a message queue
msgSpaceCreate(2K)	create a message space
msgSpaceOpen(2K)	open a message space
mutexGet(2K)	See mutexInit(2K)
mutexInit(2K)	initialize a mutex; acquire a mutex; release a mutex; try to acquire a mutex
mutexRel(2K)	See mutexInit(2K)
mutexTry(2K)	See mutexInit(2K)
padGet(2K)	return actor-specific values associated with keys
padKeyCreate(2K)	create a private key for an actor
padKeyDelete(2K)	delete an actor private key
padSet(2K)	set the actor's key to a specific value
portCreate(2K)	create a port; declare a port
portDeclare(2K)	See portCreate(2K)
portDelete(2K)	delete a port
portEnable(2K)	Enable a port portDisable; Disable a port
portGetSeqNum(2K)	See portMigrate(2K)
portLi(2K)	See portUi(2K)
portMigrate(2K)	Migrate a port; Get a port sequence number
portPi(2K)	get and/or set the protection identifier of a port
portUi(2K)	Get the unique identifier of a port, given its local identifier; Get the local

	identifier of a port, given its unique identifier
`ptdErrnoAddr`(2K)	return a thread-specific errno address
`ptdGet`(2K)	return thread-specific value associated with key
`ptdKeyCreate`(2K)	create a thread-specific data key
`ptdKeyDelete`(2K)	delete a thread-specific data key
`ptdRemoteGet`(2K)	return a thread-specific data value for another thread
`ptdRemoteSet`(2K)	set a thread-specific data value for another thread
`ptdSet`(2K)	set a thread-specific value
`ptdThreadDelete`(2K)	delete all thread-specific values and call destructors
`ptdThreadId`(2K)	return the thread ID
`rgnAllocate`(2K)	allocate a region in an actor address space
`rgnDup`(2K)	duplicate an actor address space
`rgnFree`(2K)	deallocate regions of an actor address space
`rgnInitFromActor`(2K)	allocate a region in an actor address space and initialise it from another region
`rgnMapFromActor`(2K)	create a region in an actor address space and map another region to it
`rgnPhysMap`(2K)	create a region in an actor address space and map (on demand) to it physical memory specified by the caller
`rgnSetInherit`(2K)	Change inheritance options associated with a region; Change paging options associated with a region; Change inheritance options associated with a region; Change

	opaque values associated with a region
rgnSetOpaque(2K)	See rgnSetInherit(2K)
rgnSetPaging(2K)	See rgnSetInherit(2K)
rgnSetProtect(2K)	change protection options associated with a region
rgnStat(2K)	get the statistics of a region of an actor address space
rtMutexGet(2K)	See rtMutexInit(2K)
rtMutexInit(2K)	Initialize a realtime mutex; Acquire a realtime mutex; Release a realtime mutex; Try to acquire a realtime mutex
rtMutexRel(2K)	See rtMutexInit(2K)
rtMutexTry(2K)	See rtMutexInit(2K)
schedAdmin(2K)	scheduling classes administration
semInit(2K)	initialize a semaphore; wait on a semaphore; signal a semaphore
semP(2K)	See semInit(2K)
semV(2K)	See semInit(2K)
svAbortHandler(2K)	See svExcHandler(2K)
svActorAbortHandler(2K)	Connect an actor abort handler; Disconnect an actor abort handler; Get an actor abort handler
svActorAbortHandlerConnect(2K)	See svActorAbortHandler(2K)
svActorAbortHandlerDisconnect(2K)	See svActorAbortHandler(2K)
svActorAbortHandlerGetConnected(2K)	See svActorAbortHandler(2K)
svActorExcHandler(2K)	Connect an actor exception handler; Disconnect an actor exception handler; Get an actor exception handler

svActorExcHandlerConnect(2K)	See svActorExcHandler(2K)
svActorExcHandlerDisconnect(2K)	See svActorExcHandler(2K)
svActorExcHandlerGetConnected(2K)	See svActorExcHandler(2K)
svActorStopHandler(2K)	Actor stop handler management: Connect an actor stop handler; Disconnect an actor stop handler; Get an actor stop handler
svActorStopHandlerConnect(2K)	See svActorStopHandler(2K)
svActorStopHandlerDisconnect(2K)	See svActorStopHandler(2K)
svActorStopHandlerGetConnected(2K)	See svActorStopHandler(2K)
svActorVirtualTimeout(2K)	Set an actor's virtual timeout; Cancel an actor's virtual timeout
svActorVirtualTimeoutCancel(2K)	See svActorVirtualTimeout(2K)
svActorVirtualTimeoutSet(2K)	See svActorVirtualTimeout(2K)
svCopyIn(2K)	Copy from trap caller space; Copy string from trap caller space; Copy to trap caller space
svCopyInString(2K)	See svCopyIn(2K)
svCopyOut(2K)	See svCopyIn(2K)
svExcHandler(2K)	Define an exception handler; Define an abort handler
svGetInvoker(2K)	get handler invoker
svLapBind(2K)	bind a symbolic name to a lap descriptor; unbind the symbolic name bound to a lap descriptor; get a lap descriptor from a lap symbolic name
svLapCreate(2K)	create a lap; reset a lap descriptor; test if a lap descriptor has been initialized; duplicate a lap descriptor

svLapDelete(2K)	delete a lap
svLapUnbind(2K)	See svLapBind(2K)
svMaskAll(2K)	Disable interrupt processing; Enable interrupt processing; Reenable interrupt processing
svMaskedLockGet(2K)	See svMaskedLockInit(2K)
svMaskedLockInit(2K)	Initialize a spin lock; Disable interrupts and acquire a spin lock; Try to disable interrupts and acquire a spin lock; Release a spin lock and enable interrupts; Initialize a spin lock; Acquire a spin lock; Try to acquire a spin lock; Release a spin lock
svMaskedLockRel(2K)	See svMaskedLockInit(2K)
svMaskedLockTry(2K)	See svMaskedLockInit(2K)
svMemRead(2K)	Copy from supervisor caller space; Copy to supervisor caller space
svMemWrite(2K)	See svMemRead(2K)
svMsgHandler(2K)	Connect / disconnect a message handler; Prepare a reply to a handled message
svMsgHdlReply(2K)	See svMsgHandler(2K)
svPagesAllocate(2K)	supervisor address space memory allocator
svPagesFree(2K)	See svPagesAllocate(2K)
svSpinLockGet(2K)	See svMaskedLockInit(2K)
svSpinLockInit(2K)	See svMaskedLockInit(2K)
svSpinLockRel(2K)	See svMaskedLockInit(2K)
svSpinLockTry(2K)	See svMaskedLockInit(2K)
svSysCtx(2K)	get system context table address
svSysPanic(2K)	trigger the invocation of the panic handler

svSysTimeout(2K)	Request a timeout; Cancel a timeout; Get timeout resolution
svSysTimeoutCancel(2K)	See svSysTimeout(2K)
svSysTimeoutSet(2K)	See svSysTimeout(2K)
svSysTrapHandler(2K)	Connect a trap handler; Disconnect a trap handler; Get a trap handler
svSysTrapHandlerConnect(2K)	See svSysTrapHandler(2K)
svSysTrapHandlerDisconnect(2K)	See svSysTrapHandler(2K)
svSysTrapHandlerGetConnected(2K)	See svSysTrapHandler(2K)
svThreadVirtualTimeout(2K)	Set a thread's virtual timeout; Cancel a thread's virtual timeout
svThreadVirtualTimeoutCancel(2K)	See svThreadVirtualTimeout(2K)
svThreadVirtualTimeoutSet(2K)	See svThreadVirtualTimeout(2K)
svTimeoutGetRes(2K)	See svSysTimeout(2K)
svTrapConnect(2K)	Connect a trap handler; Disconnect a trap handler
svTrapDisConnect(2K)	See svTrapConnect(2K)
svUnmask(2K)	See svMaskAll(2K)
svUnmaskAll(2K)	See svMaskAll(2K)
svVirtualTimeoutCancel(2K)	See svVirtualTimeoutSet(2K)
svVirtualTimeoutSet(2K)	Set a virtual timeout; Cancel a virtual timeout
sysBench(2K)	kernel benchmark utility
sysGetConf(2K)	Get Chorus module configuration value
sysGetEnv(2K)	Get a value from the Chorus configuration environment
sysLog(2K)	log a message in the kernel's cyclical buffer

sysPoll(2K)	See sysRead(2K)
sysRead(2K)	Read characters from the system console; Write characters to the system console; Poll characters from the system console
sysReboot(2K)	request a reboot of the local site
sysSetEnv(2K)	Set a value in the ChorusOS configuration environment
sysShutdown(2K)	shut down or restart the system
sysTime(2K)	get system time; get system time resolution
sysTimeGetRes(2K)	See sysTime(2K)
sysTimer(2K)	system timer management
sysTimerGetCounterFrequency(2K)	See sysTimer(2K)
sysTimerGetCounterPeriod(2K)	See sysTimer(2K)
sysTimerReadCounter(2K)	See sysTimer(2K)
sysTimerStartFreerun(2K)	See sysTimer(2K)
sysTimerStartPeriodic(2K)	See sysTimer(2K)
sysTimerStop(2K)	See sysTimer(2K)
sysUnsetEnv(2K)	delete a value from the ChorusOS configuration environment
sysWrite(2K)	See sysRead(2K)
threadAbort(2K)	Abort a thread; Check whether the current thread has been aborted
threadAborted(2K)	See threadAbort(2K)
threadActivate(2K)	make a thread active
threadBind(2K)	bind a thread to a processor
threadContext(2K)	get and/or set the context of a thread
threadCreate(2K)	create a thread
threadDelay(2K)	delay the current thread

threadDelete(2K)	delete a thread
threadLoadR(2K)	Get the current thread's valid soft register value; Reset the current thread's valid soft register value
threadName(2K)	get and/or set the symbolic name of a thread
threadResume(2K)	See threadSuspend(2K)
threadScheduler(2K)	get and/or set thread scheduling parameters
threadSelf(2K)	get the current thread local identifier
threadSemInit(2K)	Initialize a thread semaphore; Signal a thread semaphore; Wait on a thread semaphore
threadSemPost(2K)	See threadSemInit(2K)
threadSemWait(2K)	See threadSemInit(2K)
threadStart(2K)	Stop a thread; Start a thread
threadStat(2K)	obtain the descriptions of the threads running in an actor
threadStop(2K)	See threadStart(2K)
threadStoreR(2K)	See threadLoadR(2K)
threadSuspend(2K)	Suspend a thread; Resume a thread
threadTimes(2K)	get thread execution time
timerCreate(2K)	create a timer
timerDelete(2K)	delete a timer
timerGetRes(2K)	get the timer resolution
timerSet(2K)	start, cancel or query a timer
timerThreadPoolInit(2K)	initialize a timer thread pool
timerThreadPoolWait(2K)	wait for a timer expiration event
uiBuild(2K)	Build a user-defined unique identifier; Clear a unique identifer; Compare two unique identifers; Extract the site number from a

	unique identifier; Check a unique identifier's locality; Get a pre-defined site unique identifier; Check whether an unique identifier has been cleared
uiClear(2K)	See uiBuild(2K)
uiEqual(2K)	See uiBuild(2K)
uiGetSite(2K)	See uiBuild(2K)
uiIsLocal(2K)	See uiBuild(2K)
uiLocalSite(2K)	get the local site number
uiSite(2K)	See uiBuild(2K)
uiValid(2K)	See uiBuild(2K)
univTime(2K)	Get time-of-day; Set time-of-day; Adjust time-of-day univTimeGetRes; Get time-of-day resolution
univTimeAdjust(2K)	See univTime(2K)
univTimeSet(2K)	See univTime(2K)
virtualTimeGetRes(2K)	get virtual time resolution
vmCopy(2K)	copy data between actor address spaces
vmFree(2K)	free memory
vmLock(2K)	Fix data in memory; Data in memory
vmPageSize(2K)	get the minimum allocatable memory block size
vmPhysAddr(2K)	get the physical address corresponding to a virtual address
vmSetPar(2K)	set the memory management parameters
vmStat(2K)	get memory management statistics
vmUnLock(2K)	See vmLock(2K)

NAME

acap – get a c_actor capability

SYNOPSIS

```
#include <am/afexec.h>
int acap(int aid, KnCap *cactorcap);
```

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

`acap()` copies, at the location pointed to by *cactorcap*, the capability of the c_actor designated by the identifier *aid*. See `intro`(2K).

RETURN VALUES

Upon successful completion, `acap()` returns `0`. Otherwise it returns `–1` and sets `errno` to indicate one of the following error conditions:

[EINVAL]	*aid* is zero.
[ESRCH]	*aid* does not exist.
[EPERM]	the calling c_actor is not a trusted c_actor.
[EFAULT]	*cactorcap* points outside the allocated address space of the c_actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`afexec`(2K), `agetId`(2K), `intro`(2K)

NAME

aconf – get configurable system variables

SYNOPSIS

```
#include <am/aconf.h>
int aconf(int name);
```

FEATURES

ACTOR_EXTENDED_MGMT

DESCRIPTION

The `aconf()` function provides a method for the application to determine the current value of configurable system limits, modules or options (variables) defined by the ChorusOS 4.0 AM.

The *name* argument represents the system variable/module to be queried. The symbolic constants used for *name* are defined in `<am/aconf.h>` and appear in the table that follows.

Variable name	Value
AMDS	_AC_M_AMDS Dump module.
ARG_MAX	_AC_C_ARG_MAX Maximum length of arguments for the exec functions, in bytes, including environment.

Other values may be obtained by using `sysctl`(3POSIX) for values defined by the IOM, or by using `sysconf`(3POSIX).

RETURN VALUE

If *name* is an invalid value, `aconf()` returns –1.

Otherwise, `aconf()` returns the current variable value on the system. The value does not change during the lifetime of the calling actor. If *name* corresponds to an AM module, `aconf()` returns `1` if the module is present, or `0` if the module is not present.

ERRORS

[EINVAL] The *name* argument is invalid.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysconf`(3POSIX)

NAME

acreate – creates a c_actor

SYNOPSIS

#include <am/afexec.h>
int **acreate**(KnCap **cactorcap*, const AcParam **param*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The *acreate* call creates a c_actor according to the arguments specified by *param*.

The *param* argument points to an *AcParam* structure which should contain the specification of the new c_actor. An *AcParam* structure has the following members:

```
int         acSite ;        /* new c_actor execution site ID */
int         acFlags ;       /* c_actor options */
char*       acStdin ;       /* standard input */
char*       acStout ;       /* standard output */
char*       acStderr ;      /* standard error output */
char*       acCurdir ;      /* current directory */
char*       acRootdir ;     /* root directory */
cx_cred_t*  acCred ;        /* c_actor credentials (This member is ignored.) */
```

The new c_actor runs on the local site. *acSite* is ignored.

The *acFlags* values are constructed by using a combination of the following flags. Note that either AFX_USER_SPACE or AFX_SUPERVISER_SPACE, but not both, can be used.

AFX_USER_SPACE — If AFX_TRUSTED is set, the new c_actor is implemented as a `system` actor (see `intro`(2K)) and is a `Trusted` c_actor (see `intro`(2K)), otherwise the new c_actor is implemented as a `user` actor and is not `Trusted`.

AFX_SUPERVISOR_SPACE — The new c_actor is implemented as a `supervisor` actor (see `intro`(2K)) and is a `Trusted` c_actor (see `intro`(2K)).

AFX_ANY_SPACE — AFX_ANY_SPACE is meaningful only with *afexec*; it is not applicable with *acreate*.

The new c_actor privilege (`user` or `supervisor`) will be deduced from binary at load time. This flag takes precedence over the two previous ones. If the AFX_NON_TRUSTED flag is set, the privilege deduced from binary will be the `user` privilege.

AFX_DEBUGGEE — The new c_actor will be started in debug mode. If AFX_WAIT_ATTACH is also set, the new c_actor will be stopped to wait for a debugger to perform an ATRACE_ATTACH command through the (see

	atrace(2K)) system call; otherwise, the calling c_actor is considered as its own debugger.
AFX_EXECUTE_IN_PLACE	The new c_actor will be executed in place, meaning the code will not be copied, but instead executed where found. Data will be copied. This requires the actor to be fully linked.
AFX_KDB_SYMBOLS	A description of the new c_actor symbols will be loaded in memory. This symbol description will enable symbolic debugging of the new c_actor using the kernel debugger. This flag is only meaningful for a supervisor c_actor.
AFX_DEBUGMODE	The new c_actor will be put into the DEBUGMODE mode (see actorCreate(2K)) That means that more information than usual about thread execution will be gathered by the system. Not to be confused with AFX_DEBUGEE.
AFX_STOPPED	The new c_actor will be put into the STOPPED state. Not to be confused with AFX_WAIT_ATTACH, as no ATRACE_ATTACH command is expected.

The *acStdin*, *acStdout*, and *acStderr* fields specify the path name for the standard input, the standard output and the standard error output files for the new c_actor.

The *acCurdir* field allows you to specify the pathname for the current directory of the new c_actor.

The *acRootdir* field allows you to specify the pathname for the root directory of the new c_actor.

The *acCred* field allows you to set the credentials of the new c_actor. This field may only be set by a trusted c_actor.

Unlike *afexec*, *acreate* does not load the new c_actor or activate it.

After *acreate* has been created, the new c_actor exists, a corresponding actor has been created according to the *acFlags* specified, the standard I/O files and directories specified are open, and the new c_actor has the specified credentials. However, the new c_actor does not have any regions or threads. In addition, the new c_actor is not mature, which means that if the calling c_actor is deleted, the new c_actor will be deleted.

If one of the pointer-type members of **param* is NULL, the corresponding parameter is considered un-specified. If a parameter is un-specified,

the new c_actor will inherit the corresponding running attributes of the calling c_actor.

File descriptors with the close-on-afexec flag set (see `afexec`(2K)) will not be inherited by the new c_actor.

LIMITATIONS

Correct functioning of a supervisor actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) is not guaranteed. Even if the supervisor actor functions normally, only one supervisor actor can be executed in place at any one time from a given binary file. Attempting to execute more than one supervisor actor in place from the same binary file will be rejected by the system.

Correct functioning of any actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) when in flat memory mode is not guaranteed. Even if the actor functions normally when in flat memory mode, only one actor can be executed in place at any one time from a given binary file. Attempting to execute more than one actor in place in this situation will be rejected by the system.

RETURN VALUES

Upon successful completion, *acreate* returns a non-negative integer that is the *aid* of the new c_actor. Otherwise *acreate* returns –`1` and sets `errno` to indicate one of the following error conditions:

[EPERM]	The calling c_actor is not `trusted` and is either trying to create a `trusted` c_actor or to set the new c_actor's credentials.
[ENOENT]	One or more components of the new c_actor's I/O files' path names do not exist.
[ENOTDIR]	A component of the new c_actor's I/O files' path prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in one of the new c_actor's I/O files' path prefix.
[EINVAL]	Inconsistent attribute flags were provided.
[EINVAL]	*parame* was `NULL`.
[ENAMETOOLONG]	The length of a component of one of the I/O file's path exceeds `NAME_MAX` characters, or the length of the complete pathname exceeds `PATH_MAX` characters.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `afexec`(2K), `akill`(2K), `aload`(2K), `astart`(2K), `hrfexec`(2RESTART)

NAME

acred – get / set c_actor credentials

SYNOPSIS

#include <cx/cred.h>
int **acred**(const KnCap **cactorcap*, cx_cred_t **oldCred*, const cx_cred_t **newCred*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

If *oldCred* is not the NULL pointer, a copy of the credential structure of the c_actor designated by its capability *cactorcap* will be copied at the location pointed to by *oldCred*.

If *newCred* is not the NULL pointer, the credentials of the c_actor designated by its capability *cactorcap* will be set to the value of the credentials pointed to by *newCred*.

A structure *cx_cred_t* includes the following members:

```
uid_t           cr_uid;       /* c_actor's user ID */
gid_t           cr_gid;       /* c_actor's group ID */
unsigned short  cr_ngroups;   /* number of groups in cr_groups */
gid_t           cr_groups[];  /* supplementary group list */
```

RETURN VALUE

Upon successful completion, *acred* returns 0; otherwise it returns the following error conditions:

[ESRCH]	*cactorcap* doesn't designate a valid c_actor.
[EPERM]	*newCred* is not the NULL pointer and the caller is not a trusted c_actor.
[EFAULT]	*cactorcap*, *oldCred* or *newCred* points outside the allocated address space of the calling c_actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

afexec(2K)

NAME

actorCreate – create an actor

SYNOPSIS

#include <exec/chExec.h>
int **actorCreate**(KnCap **actorinit*, KnCap **actorcap*, KnActorPrivilege *privilege*, KnActorStatus *status*);

FEATURES

CORE

DESCRIPTION

The *actorCreate* call creates a new actor. A capability for this actor is returned in *actorcap*, which is a pointer to a *KnCap* structure whose members are the following:

```
KnUniqueId    ui ;         /* entity name */
KnKey         key ;        /* modification key */
```

Where *ui* is the unique identifier of the actor, and *key* is the key needed to modify the actor (for instance, creating or deleting threads or other resources, or stopping the actor).

The new actor inherits a number of attributes from its init actor, whose capability is given by *actorinit* . If *actorinit* is K_MYACTOR, the current actor is taken as the init actor. The CORE module attributes inherited from the init actor are: the exception handler (see *svExcHandler(2K))* and the abort handler (see *svAbortHandler(2K)).* Furthermore, other kernel modules, such as communication modules, may define attributes which are inherited by the new actor.

The *privilege* field gives the actor's privilege. Two main privileges may be given to an actor: *USER* or *SYSTEM*. A system actor's thread has access to some privileged kernel calls even if the thread is not running with the *SUPERVISOR* privilege level (see *threadCreate*(2K)). A *SYSTEM* actor may only be created by a thread running with the *SUPERVISOR* privilege or if its actor is a *SYSTEM* actor.

In addition, *SYSTEM* actors may be created as *SUPERVISOR* actors: a *SUPERVISOR* actor shares the kernel address space, and no user address space is allocated to it. *SUPERVISOR* actors may only contain *SUPERVISOR* threads.

A *USER* actor is created with *privilege* equal to K_USERACTOR; a *SYSTEM* actor is created with *privilege* equal to K_SYSTEMACTOR; a *SUPERVISOR* actor is created with *privilege* equal to K_SUPACTOR.

When created, the new actor contains neither thread nor memory region.

The *status* parameter is the initial actor status: if the K_STOPPED flag is set in the *status* parameter, the new actor is created in the *STOPPED* state. That means that even if threads are created in the actor (using *threadCreate*(2K)), none of these threads are able to run until the actor is put into the *ACTIVE* state using *actorStart*(2K).

If the K_STOPPED flag is not set, threads that are created in the *ACTIVE* state are able to run immediately.

if the K_DEBUGMODE flag is set in the *status* parameter, the new actor will be put into the *DEBUGMODE* mode. That means that more information than usual about thread execution will be gathered by the system. This helps to support some source level debuggers.

The maximum number of actors that may be created on a site is tunable parameter.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorinit* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorinit* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_ENOMEM]	The system is out of resources.
[K_EPRIV]	The current thread is not allowed to create a *SYSTEM* actor.

RESTRICTIONS

The init actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorDelete`(2K), `actorStart`(2K), `actorStop`(2K), `actorCapability`(2K), `threadCreate`(2K)

NAME

actorDelete – delete an actor

SYNOPSIS

```
#include <exec/chExec.h>
int actorDelete(KnCap *actorcap);
```

FEATURES

CORE

DESCRIPTION

The *actorDelete* call deletes the actor whose capability is given by *actorcap* .

If *actorcap* is K_MYACTOR, the current actor is deleted.

When the actor is deleted, all the actor's resources are deleted. For example, threads are deleted just as if *threadDelete*(2K) had been called.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_EBUSY]	The actor is already in the process of being deleted.

RESTRICTIONS

The deleted actor and the current actor must be located on the same site.

Interrupt, trap, exception or time-out handlers connected by the actor are not disconnected on actor deletion in this version. When deleting *SUPERVISOR* actors, these handlers must be disconnected explicitly before invoking *actorDelete*. Otherwise, unpredictable results may occur.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorCreate`(2K), `threadDelete`(2K)

NAME

actorName – get and / or set the symbolic name of an actor

SYNOPSIS

#include <exec/chExec.h>
int **actorName**(KnCap **actorcap*, char **oldname*, char **newname*);

FEATURES

CORE

DESCRIPTION

The *actorName* call gets and / or sets the symbolic name of the actor whose capability is given by *actorcap* (see *actorCreate*(2K)). If *actorcap* is K_MYACTOR, the current actor is affected.

If *oldname* is not NULL, the actor's symbolic name is copied to the caller address space at the location specified by *oldname.* An actor's symbolic name has a maximum size of K_ACTORNAMEMAX (including the NULL character). If *newname* is not NULL, the actor's symbolic name is set to the new name pointed to by *newname* in the caller address space. The new name is truncated to a maximum size of K_ACTORNAMEMAX (including the NULL character).

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorCreate`(2K)

NAME

actorPi – get and / or set the protection identifier of an actor

SYNOPSIS

```
#include <exec/chExec.h>
#include <ipc/chId.h>
#include <ipc/chIpc.h>
int actorPi(KnCap *actorcap, KnProtId *oldpi, KnProtId *newpi);
```

FEATURES

IPC

DESCRIPTION

The *actorPi* call gets and / or sets the protection identifier of the actor whose capability is given by *actorcap*. If *actorcap* is K_MYACTOR, the operation is applied to the current actor.

The *oldpi* and *newpi* pointers indicate *KnProtId* structures. The structure of *KnProtId* is described in *portPi(2K)*. The corresponding actor's protection identifier is copied to the structure pointed to by *oldpi* in the client address space. The new corresponding context is copied from the structure pointed to by *newpi*.

If *oldpi* is a NULL pointer, the actor's protection identifier is not copied in the caller's address space. If *newpi* is a NULL pointer, the actor's protection identifier is not modified. The calling thread must be a *SUPERVISOR* thread (see *threadCreate*(2K)) or must belong to a *SYSTEM* actor (see *actorCreate*(2K)).

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorCreate`(2K), `threadCreate`(2K)

NAME

actorPrivilege – get and / or set the privilege of an actor

SYNOPSIS

#include <exec/chExec.h>
int **actorPrivilege**(KnCap **actorcap*, KnActorPrivilege **oldpriv*, KnActorPrivilege **newpriv*);

FEATURES

CORE

DESCRIPTION

actorPrivilege gets and / or sets the privilege of the actor the capability of which is given by *actorcap*. If *actorcap* is K_MYACTOR, the operation is applied to the current actor.

oldpriv and *newpriv* are pointers to *KnActorPrivilege* data. *KnActorPrivilege* represents the actor privilege, as described in *actorCreate* (2K).

If *oldpriv* is a NULL pointer, the actor's privilege is not copied in the caller's address space. If *newpriv* is a NULL pointer, the actor's privilege is not modified. The calling thread must be a *SUPERVISOR* thread (see *threadCreate*(2K)) or must belong to a *SYSTEM* actor (see *actorCreate*(2K)).

The only allowed privilege modifications are from USER to SYSTEM or from SYSTEM to USER.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorCreate`(2K), `threadCreate`(2K)

NAME

actorSelf – get the current actor capability

SYNOPSIS

```
#include <exec/chExec.h>
int actorSelf(KnCap *actorcap);
```

FEATURES

CORE

DESCRIPTION

actorSelf returns the capability of the current actor into the structure pointed to by *actorcap.*

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the provided data are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

actorCreate(2K)

NAME

actorStop, actorStart – Stop an actor; Start an actor

SYNOPSIS

```
#include <exec/chExec.h>
int actorStart(KnCap * actorcap);

int actorStop(KnCap * actorcap);
```

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *actorStop* call stops the actor whose capability is given by *actorcap* (see *actorCreate* (2K)).

If *actorcap* is K_MYACTOR, the current actor is stopped.

The effect of *actorStop* is to prevent all the threads from running until the actor is restarted using *actorStart (2K)* .

Performing *actorStop* is equivalent to performing *threadStop* on all the actor's threads.

All other threads executing in the actor and having entered it through safe LAPs, and any thread attempting to enter a stopped actor through a safe LAP will also be stopped.

The effect of *actorStop* is not instantaneous on threads executing a system call that is implemented via a trap, but it guarantees that a thread performing this type of system call will not return from that call.

When an actor has been stopped, the internal status of its threads may, however, change (if messages are delivered or resumed, for example).

It is possible to resume individual threads using *threadStart* (2K).

The *actorStart* call starts the actor whose capability is given by *actorcap*. It may be applied to a newly created actor, or to an actor previously stopped using *actorStop (2K).*

If *actorcap* is K_MYACTOR, the current actor is started.

After *actorStart* , any of the runnable actor's threads may run user code.

Performing *actorStart* is equivalent to performing *threadStart* on all the actor's threads and to resuming all other threads executing in the actor or attempting to enter it through safe LAPs.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.

[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided is out of the current actor's address space.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorCreate`(2K) , `threadContext`(2K) , `threadStart`(2K) , `threadStop`(2K)

NAME

actorStat – get the status of all actors on a site

SYNOPSIS

#include <exec/chExec.h>
int **actorStat**(unsigned int *options*, KnActorStat **stat*, unsigned int *buffsize*);

FEATURES

CORE

DESCRIPTION

The *actorStat* call obtains the status of all actors created on the current site.

The *stat* argument points to a buffer in which the descriptions of existing actors will be returned. The *buffsize* argument gives the size of the buffer. The buffer is considered part of the caller (current user or supervisor) address space.

The *options* argument is reserved for future use and must be set to 0.

On successful return the buffer will contain an array of descriptors, each one describing a single actor. Each descriptor is a *KnActorStat* structure with the following fields:

```
KnCap asCap ;
KnActorStatus asStatus ;
```

The *asCap* field is the actor capability. The *asStatus* field is the actor status (K_STOPPED or K_ACTIVE, see *actorCreate*(2K) for details).

The *actorStat* call is restricted to threads running with *SUPERVISOR* privilege or whose actor is a *SYSTEM* actor.

RETURN VALUE

If successful, *actorStat* returns the number of actors created on the current site. Otherwise a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EPRIV]	The current thread is not allowed to perform this call.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorCreate`(2K)

NAME

actorStop, actorStart – Stop an actor; Start an actor

SYNOPSIS

```
#include <exec/chExec.h>
int actorStart(KnCap * actorcap);

int actorStop(KnCap * actorcap);
```

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *actorStop* call stops the actor whose capability is given by *actorcap* (see *actorCreate* (2K)).

If *actorcap* is K_MYACTOR, the current actor is stopped.

The effect of *actorStop* is to prevent all the threads from running until the actor is restarted using *actorStart (2K)* .

Performing *actorStop* is equivalent to performing *threadStop* on all the actor's threads.

All other threads executing in the actor and having entered it through safe LAPs, and any thread attempting to enter a stopped actor through a safe LAP will also be stopped.

The effect of *actorStop* is not instantaneous on threads executing a system call that is implemented via a trap, but it guarantees that a thread performing this type of system call will not return from that call.

When an actor has been stopped, the internal status of its threads may, however, change (if messages are delivered or resumed, for example).

It is possible to resume individual threads using *threadStart* (2K).

The *actorStart* call starts the actor whose capability is given by *actorcap*. It may be applied to a newly created actor, or to an actor previously stopped using *actorStop (2K)*.

If *actorcap* is K_MYACTOR, the current actor is started.

After *actorStart* , any of the runnable actor's threads may run user code.

Performing *actorStart* is equivalent to performing *threadStart* on all the actor's threads and to resuming all other threads executing in the actor or attempting to enter it through safe LAPs.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.

[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided is out of the current actor's address space.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorCreate`(2K) , `threadContext`(2K) , `threadStart`(2K) , `threadStop`(2K)

NAME

afexec, afexecl, afexecv, afexecle, afexecve, afexeclp, afexecvp – create a new c_actor

SYNOPSIS

```
#include <am/afexec.h>
int afexecve(const char * path, KnCap * cactorcap, const AcParam * param, char const * argv, char const * envp);

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

int afexecv(const char * path, KnCap * cactorcap, const AcParam * param, char *const * argv);

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

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

int afexecvp(const char * file, KnCap * cactorcap, const AcParam * param, char *const * argv);
```

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

afexec family routines create a new c_actor with a single thread (the main thread). This main thread will execute with the priority defined by the *am.afexecschedprio* configuration parameter. It will have an automatically allocated stack, which size is defined by the *am.afexec.userstacksize* configuration parameter for user actors, and by *kern.exec.dflSysStackSize* for supervisor actors. (See intro(2K)).

The main thread starts executing the runtime startup routine (usually called *_start*) specified as the program entry point in the binary header of the *new c_actor file* . After initialization of the threadsafe C library, the runtime startup routine calls the C program as follows:

```
int main (int argc, char* argv[], char* envp[])
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As will be described later, *argc* is conventionally at least 1 , and the first member of the array points to a string containing the name (or pathname) of the executable file.

cactorcap is the returned capability of the new c_actor, and is a pointer to a *KnCap* structure (see actorCreate(2K)).

If *param* is NULL , default parameters are used, *acFlags* is set to AFX_ANY_SPACE and all others to un-specified . See acreate(2K) for a description of the AcParam structure.

By convention, *afexec* routines search for a binary, either fully linked or relocatable, in *path* . If no such binary exists, *afexec* routines search for a file with a `.r` suffix and then with a `.o` suffix. If the GZ_FEATURE is set to `true` , for each step, the *afexec* routines also search for a file with an additional `.gz` (for example: `path` , `path.gz` , `path.r` , `path.r.gz`).

path may also refer to a relocatable binary stored in a memory buffer in supervisor space, using the syntax "ram:0x<start>:0x<length>". <start> is the start address at which the binary is stored. <length> is the length of the binary. The buffer is not used any more after loading, except if the program uses the dynamic library API. Freeing the buffer after *afexec* may have unpredictable results if the program uses the dynamic library API.

Files to be loaded may be fully linked or relocatable files. They may also include dependencies on dynamic, but not shared, libraries. These libraries are looked for and loaded within the actor address space as part of the *afexec* call.

arg0 , ... , *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new c_actor. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or `file` or the last component of *path*).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new c_actor. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or `file` or the last component of *path*). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new c_actor. *envp* is terminated by a null pointer. For *afexecl* , *afexecv* , *afexeclp* and *afexecvp* , the C runtime start-off routine places a pointer to the environment of the calling c_actor in the global object:

```
extern char** environ;
```

and it is used to pass the environment of the calling c_actor to the new c_actor.

afexeclp and *afexecvp* are called with the same arguments as *afexecl* and *afexecv* , but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the PATH environment variable, and `file` must point to an executable file name for the new c_actor.

LIMITATIONS

Incompatible requests (such as *afexec* of a supervisor binary with the AFX_USER_SPACE flag set) are not normally detected. Applications can use the AFX_ANY_SPACE (see `acreate`(2K)) flag to force the system to deduce the type (`user` or `supervisor`) of the binary. In this case incompatible requests will be detected and refused.

Correct functioning of a supervisor actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) is not guaranteed. Even if the supervisor actor functions normally, only one supervisor actor can be executed in place at any one time from a given binary file. Attempting to execute more than one supervisor actor in place from the same binary file will be rejected by the system.

Correct functioning of any actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) when in flat memory mode is not guaranteed. Even if the actor functions normally, when in flat memory mode only one actor can be executed in place at any one time from a given binary file. Attempting to execute more than one actor in place in this situation will be rejected by the system.

RETURN VALUES

Upon successful completion, *afexec* returns a non-negative integer that is the *aid* of the new c_actor. Otherwise *afexec* returns –1 and sets `errno` to indicate one of the following error conditions.

ERRORS

[E2BIG]	The number of bytes used by the new c_actor's image argument list and environment list is greater than `ARG_MAX` bytes.
[EACCES]	Search permission is denied for a directory listed in the new c_actor's file path prefix.
[EBADSYM]	The runtime link-editor cannot find a symbol needed by the new c_actor. Possible reasons are that ChorusOS 4.0 `imake` rules have not been used to compile the program, or the link editor has found unexpected dynamic libraries.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	Inconsistent attribute flags were specified.
[EINVAL]	The address of the text or data region of the new c_actor is not consistent with the AFX_USER_SPACE or AFX_SUPERVISOR_SPACE attribute flag.
[EIO]	An I/O error occurred while reading from the file system.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ENOENT]	The new c_actor is a dynamic actor and one of its dependencies cannot be found either in the directories listed in the `LD_LIBRARY_PATH`

	environment variable, or in its runpath, or in `/usr/lib` .
[ENOEXEC]	The new c_actor file has the appropriate access permission but is not in the correct format.
[ENOTDIR]	A component of the new c_actor path of the file prefix is not a directory.
[EPERM]	The calling c_actor is not Trusted and is trying either to create a Trusted c_actor or set the new c_actor's credentials.
[ELOOP]	Too many symbolic links have been encountered during analysis of *path* .

Note - Error codes generated by *read* and *open* may also be returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`aconf`(2K) , `acreate`(2K) , `actorCreate`(2K) , `akill`(2K) , `aload`(2K) , `astart`(2K) , `astat`(2K) , `await`(2K) , `dlopen`(2K) , `fcntl`(2POSIX) , `intro`(2K)

NAME

afexec, afexecl, afexecv, afexecle, afexecve, afexeclp, afexecvp – create a new c_actor

SYNOPSIS

#include <am/afexec.h>
int **afexecve**(const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char const * *argv*, char const * *envp*);

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

int **afexecv**(const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

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

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

int **afexecvp**(const char * *file*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

afexec family routines create a new c_actor with a single thread (the main thread). This main thread will execute with the priority defined by the *am.afexecschedprio* configuration parameter. It will have an automatically allocated stack, which size is defined by the *am.afexec.userstacksize* configuration parameter for user actors, and by *kern.exec.dflSysStackSize* for supervisor actors. (See intro(2K)).

The main thread starts executing the runtime startup routine (usually called *_start*) specified as the program entry point in the binary header of the *new c_actor file* . After initialization of the threadsafe C library, the runtime startup routine calls the C program as follows:

```
int main (int argc, char* argv[], char* envp[])
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As will be described later, *argc* is conventionally at least 1 , and the first member of the array points to a string containing the name (or pathname) of the executable file.

cactorcap is the returned capability of the new c_actor, and is a pointer to a *KnCap* structure (see actorCreate(2K)).

If *param* is NULL , default parameters are used, *acFlags* is set to AFX_ANY_SPACE and all others to un-specified . See acreate(2K) for a description of the AcParam structure.

By convention, *afexec* routines search for a binary, either fully linked or relocatable, in *path* . If no such binary exists, *afexec* routines search for a file with a `.r` suffix and then with a `.o` suffix. If the GZ_FEATURE is set to `true` , for each step, the *afexec* routines also search for a file with an additional `.gz` (for example: `path` , `path.gz` , `path.r` , `path.r.gz`).

path may also refer to a relocatable binary stored in a memory buffer in supervisor space, using the syntax "ram:0x<start>:0x<length>". <start> is the start address at which the binary is stored. <length> is the length of the binary. The buffer is not used any more after loading, except if the program uses the dynamic library API. Freeing the buffer after *afexec* may have unpredictable results if the program uses the dynamic library API.

Files to be loaded may be fully linked or relocatable files. They may also include dependencies on dynamic, but not shared, libraries. These libraries are looked for and loaded within the actor address space as part of the *afexec* call.

arg0 , ... , *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new c_actor. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or `file` or the last component of *path*).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new c_actor. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or `file` or the last component of *path*). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new c_actor. *envp* is terminated by a null pointer. For *afexecl* , *afexecv* , *afexeclp* and *afexecvp* , the C runtime start-off routine places a pointer to the environment of the calling c_actor in the global object:

```
extern char** environ;
```

and it is used to pass the environment of the calling c_actor to the new c_actor.

afexeclp and *afexecvp* are called with the same arguments as *afexecl* and *afexecv* , but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the PATH environment variable, and `file` must point to an executable file name for the new c_actor.

LIMITATIONS

Incompatible requests (such as *afexec* of a supervisor binary with the AFX_USER_SPACE flag set) are not normally detected. Applications can use the AFX_ANY_SPACE (see `acreate`(2K)) flag to force the system to deduce the type (`user` or `supervisor`) of the binary. In this case incompatible requests will be detected and refused.

Correct functioning of a supervisor actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) is not guaranteed. Even if the supervisor actor functions normally, only one supervisor actor can be executed in place at any one time from a given binary file. Attempting to execute more than one supervisor actor in place from the same binary file will be rejected by the system.

Correct functioning of any actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) when in flat memory mode is not guaranteed. Even if the actor functions normally, when in flat memory mode only one actor can be executed in place at any one time from a given binary file. Attempting to execute more than one actor in place in this situation will be rejected by the system.

RETURN VALUES

Upon successful completion, *afexec* returns a non-negative integer that is the *aid* of the new c_actor. Otherwise *afexec* returns –1 and sets `errno` to indicate one of the following error conditions.

ERRORS

[E2BIG]	The number of bytes used by the new c_actor's image argument list and environment list is greater than `ARG_MAX` bytes.
[EACCES]	Search permission is denied for a directory listed in the new c_actor's file path prefix.
[EBADSYM]	The runtime link-editor cannot find a symbol needed by the new c_actor. Possible reasons are that ChorusOS 4.0 `imake` rules have not been used to compile the program, or the link editor has found unexpected dynamic libraries.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	Inconsistent attribute flags were specified.
[EINVAL]	The address of the text or data region of the new c_actor is not consistent with the AFX_USER_SPACE or AFX_SUPERVISOR_SPACE attribute flag.
[EIO]	An I/O error occurred while reading from the file system.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ENOENT]	The new c_actor is a dynamic actor and one of its dependencies cannot be found either in the directories listed in the `LD_LIBRARY_PATH`

	environment variable, or in its runpath, or in `/usr/lib` .
[ENOEXEC]	The new c_actor file has the appropriate access permission but is not in the correct format.
[ENOTDIR]	A component of the new c_actor path of the file prefix is not a directory.
[EPERM]	The calling c_actor is not Trusted and is trying either to create a Trusted c_actor or set the new c_actor's credentials.
[ELOOP]	Too many symbolic links have been encountered during analysis of *path* .

Note - Error codes generated by *read* and *open* may also be returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`aconf`(2K) , `acreate`(2K) , `actorCreate`(2K) , `akill`(2K) , `aload`(2K) , `astart`(2K) , `astat`(2K) , `await`(2K) , `dlopen`(2K) , `fcntl`(2POSIX) , `intro`(2K)

NAME

afexec, afexecl, afexecv, afexecle, afexecve, afexeclp, afexecvp – create a new c_actor

SYNOPSIS

#include <am/afexec.h>
int **afexecve**(const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char const * *argv*, char const * *envp*);

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

int **afexecv**(const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

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

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

int **afexecvp**(const char * *file*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

afexec family routines create a new c_actor with a single thread (the main thread). This main thread will execute with the priority defined by the *am.afexecschedprio* configuration parameter. It will have an automatically allocated stack, which size is defined by the *am.afexec.userstacksize* configuration parameter for user actors, and by *kern.exec.dflSysStackSize* for supervisor actors. (See intro(2K)).

The main thread starts executing the runtime startup routine (usually called *_start*) specified as the program entry point in the binary header of the *new c_actor file* . After initialization of the threadsafe C library, the runtime startup routine calls the C program as follows:

```
int main (int argc, char* argv[], char* envp[])
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As will be described later, *argc* is conventionally at least 1 , and the first member of the array points to a string containing the name (or pathname) of the executable file.

cactorcap is the returned capability of the new c_actor, and is a pointer to a *KnCap* structure (see actorCreate(2K)).

If *param* is NULL , default parameters are used, *acFlags* is set to AFX_ANY_SPACE and all others to un-specified . See acreate(2K) for a description of the AcParam structure.

By convention, *afexec* routines search for a binary, either fully linked or relocatable, in *path* . If no such binary exists, *afexec* routines search for a file with a `.r` suffix and then with a `.o` suffix. If the GZ_FEATURE is set to `true` , for each step, the *afexec* routines also search for a file with an additional `.gz` (for example: `path` , `path.gz` , `path.r` , `path.r.gz`).

path may also refer to a relocatable binary stored in a memory buffer in supervisor space, using the syntax "ram:0x<start>:0x<length>". <start> is the start address at which the binary is stored. <length> is the length of the binary. The buffer is not used any more after loading, except if the program uses the dynamic library API. Freeing the buffer after *afexec* may have unpredictable results if the program uses the dynamic library API.

Files to be loaded may be fully linked or relocatable files. They may also include dependencies on dynamic, but not shared, libraries. These libraries are looked for and loaded within the actor address space as part of the *afexec* call.

arg0 , ... , *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new c_actor. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or `file` or the last component of *path*).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new c_actor. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or `file` or the last component of *path*). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new c_actor. *envp* is terminated by a null pointer. For *afexecl* , *afexecv* , *afexeclp* and *afexecvp* , the C runtime start-off routine places a pointer to the environment of the calling c_actor in the global object:

```
extern char** environ;
```

and it is used to pass the environment of the calling c_actor to the new c_actor.

afexeclp and *afexecvp* are called with the same arguments as *afexecl* and *afexecv* , but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the PATH environment variable, and `file` must point to an executable file name for the new c_actor.

LIMITATIONS

Incompatible requests (such as *afexec* of a supervisor binary with the AFX_USER_SPACE flag set) are not normally detected. Applications can use the AFX_ANY_SPACE (see `acreate`(2K)) flag to force the system to deduce the type (`user` or `supervisor`) of the binary. In this case incompatible requests will be detected and refused.

Correct functioning of a supervisor actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) is not guaranteed. Even if the supervisor actor functions normally, only one supervisor actor can be executed in place at any one time from a given binary file. Attempting to execute more than one supervisor actor in place from the same binary file will be rejected by the system.

Correct functioning of any actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) when in flat memory mode is not guaranteed. Even if the actor functions normally, when in flat memory mode only one actor can be executed in place at any one time from a given binary file. Attempting to execute more than one actor in place in this situation will be rejected by the system.

RETURN VALUES

Upon successful completion, *afexec* returns a non-negative integer that is the *aid* of the new c_actor. Otherwise *afexec* returns –1 and sets `errno` to indicate one of the following error conditions.

ERRORS

[E2BIG]	The number of bytes used by the new c_actor's image argument list and environment list is greater than `ARG_MAX` bytes.
[EACCES]	Search permission is denied for a directory listed in the new c_actor's file path prefix.
[EBADSYM]	The runtime link-editor cannot find a symbol needed by the new c_actor. Possible reasons are that ChorusOS 4.0 `imake` rules have not been used to compile the program, or the link editor has found unexpected dynamic libraries.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	Inconsistent attribute flags were specified.
[EINVAL]	The address of the text or data region of the new c_actor is not consistent with the AFX_USER_SPACE or AFX_SUPERVISOR_SPACE attribute flag.
[EIO]	An I/O error occurred while reading from the file system.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ENOENT]	The new c_actor is a dynamic actor and one of its dependencies cannot be found either in the directories listed in the `LD_LIBRARY_PATH`

	environment variable, or in its runpath, or in `/usr/lib` .
[ENOEXEC]	The new c_actor file has the appropriate access permission but is not in the correct format.
[ENOTDIR]	A component of the new c_actor path of the file prefix is not a directory.
[EPERM]	The calling c_actor is not Trusted and is trying either to create a Trusted c_actor or set the new c_actor's credentials.
[ELOOP]	Too many symbolic links have been encountered during analysis of *path* .

Note - Error codes generated by *read* and *open* may also be returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`aconf`(2K) , `acreate`(2K) , `actorCreate`(2K) , `akill`(2K) , `aload`(2K) , `astart`(2K) , `astat`(2K) , `await`(2K) , `dlopen`(2K) , `fcntl`(2POSIX) , `intro`(2K)

NAME

afexec, afexecl, afexecv, afexecle, afexecve, afexeclp, afexecvp – create a new c_actor

SYNOPSIS

#include <am/afexec.h>
int **afexecve**(const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char const * *argv*, char const * *envp*);

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

int **afexecv**(const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

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

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

int **afexecvp**(const char * *file*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

afexec family routines create a new c_actor with a single thread (the main thread). This main thread will execute with the priority defined by the *am.afexecschedprio* configuration parameter. It will have an automatically allocated stack, which size is defined by the *am.afexec.userstacksize* configuration parameter for user actors, and by *kern.exec.dflSysStackSize* for supervisor actors. (See intro(2K)).

The main thread starts executing the runtime startup routine (usually called *_start*) specified as the program entry point in the binary header of the *new c_actor file* . After initialization of the threadsafe C library, the runtime startup routine calls the C program as follows:

```
int main (int argc, char* argv[], char* envp[])
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As will be described later, *argc* is conventionally at least 1 , and the first member of the array points to a string containing the name (or pathname) of the executable file.

cactorcap is the returned capability of the new c_actor, and is a pointer to a *KnCap* structure (see actorCreate(2K)).

If *param* is NULL , default parameters are used, *acFlags* is set to AFX_ANY_SPACE and all others to un-specified . See acreate(2K) for a description of the AcParam structure.

By convention, *afexec* routines search for a binary, either fully linked or relocatable, in *path* . If no such binary exists, *afexec* routines search for a file with a `.r` suffix and then with a `.o` suffix. If the GZ_FEATURE is set to `true` , for each step, the *afexec* routines also search for a file with an additional `.gz` (for example: `path` , `path.gz` , `path.r` , `path.r.gz`).

path may also refer to a relocatable binary stored in a memory buffer in supervisor space, using the syntax "ram:0x<start>:0x<length>". <start> is the start address at which the binary is stored. <length> is the length of the binary. The buffer is not used any more after loading, except if the program uses the dynamic library API. Freeing the buffer after *afexec* may have unpredictable results if the program uses the dynamic library API.

Files to be loaded may be fully linked or relocatable files. They may also include dependencies on dynamic, but not shared, libraries. These libraries are looked for and loaded within the actor address space as part of the *afexec* call.

arg0 , ... , *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new c_actor. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or `file` or the last component of *path*).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new c_actor. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or `file` or the last component of *path*). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new c_actor. *envp* is terminated by a null pointer. For *afexecl* , *afexecv* , *afexeclp* and *afexecvp* , the C runtime start-off routine places a pointer to the environment of the calling c_actor in the global object:

```
extern char** environ;
```

and it is used to pass the environment of the calling c_actor to the new c_actor.

afexeclp and *afexecvp* are called with the same arguments as *afexecl* and *afexecv* , but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the PATH environment variable, and `file` must point to an executable file name for the new c_actor.

LIMITATIONS

Incompatible requests (such as *afexec* of a supervisor binary with the AFX_USER_SPACE flag set) are not normally detected. Applications can use the AFX_ANY_SPACE (see `acreate`(2K)) flag to force the system to deduce the type (`user` or `supervisor`) of the binary. In this case incompatible requests will be detected and refused.

Correct functioning of a supervisor actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) is not guaranteed. Even if the supervisor actor functions normally, only one supervisor actor can be executed in place at any one time from a given binary file. Attempting to execute more than one supervisor actor in place from the same binary file will be rejected by the system.

Correct functioning of any actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) when in flat memory mode is not guaranteed. Even if the actor functions normally, when in flat memory mode only one actor can be executed in place at any one time from a given binary file. Attempting to execute more than one actor in place in this situation will be rejected by the system.

RETURN VALUES

Upon successful completion, *afexec* returns a non-negative integer that is the *aid* of the new c_actor. Otherwise *afexec* returns –1 and sets `errno` to indicate one of the following error conditions.

ERRORS

[E2BIG]	The number of bytes used by the new c_actor's image argument list and environment list is greater than `ARG_MAX` bytes.
[EACCES]	Search permission is denied for a directory listed in the new c_actor's file path prefix.
[EBADSYM]	The runtime link-editor cannot find a symbol needed by the new c_actor. Possible reasons are that ChorusOS 4.0 `imake` rules have not been used to compile the program, or the link editor has found unexpected dynamic libraries.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	Inconsistent attribute flags were specified.
[EINVAL]	The address of the text or data region of the new c_actor is not consistent with the AFX_USER_SPACE or AFX_SUPERVISOR_SPACE attribute flag.
[EIO]	An I/O error occurred while reading from the file system.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ENOENT]	The new c_actor is a dynamic actor and one of its dependencies cannot be found either in the directories listed in the `LD_LIBRARY_PATH`

	environment variable, or in its runpath, or in `/usr/lib` .
[ENOEXEC]	The new c_actor file has the appropriate access permission but is not in the correct format.
[ENOTDIR]	A component of the new c_actor path of the file prefix is not a directory.
[EPERM]	The calling c_actor is not Trusted and is trying either to create a Trusted c_actor or set the new c_actor's credentials.
[ELOOP]	Too many symbolic links have been encountered during analysis of *path* .

Note - Error codes generated by *read* and *open* may also be returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`aconf`(2K) , `acreate`(2K) , `actorCreate`(2K) , `akill`(2K) , `aload`(2K) , `astart`(2K) , `astat`(2K) , `await`(2K) , `dlopen`(2K) , `fcntl`(2POSIX) , `intro`(2K)

NAME

afexec, afexecl, afexecv, afexecle, afexecve, afexeclp, afexecvp – create a new c_actor

SYNOPSIS

#include <am/afexec.h>
int **afexecve**(const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char const * *argv*, char const * *envp*);

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

int **afexecv**(const char * *path*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

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

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

int **afexecvp**(const char * *file*, KnCap * *cactorcap*, const AcParam * *param*, char *const * *argv*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

afexec family routines create a new c_actor with a single thread (the main thread). This main thread will execute with the priority defined by the *am.afexecschedprio* configuration parameter. It will have an automatically allocated stack, which size is defined by the *am.afexec.userstacksize* configuration parameter for user actors, and by *kern.exec.dflSysStackSize* for supervisor actors. (See intro(2K)).

The main thread starts executing the runtime startup routine (usually called *_start*) specified as the program entry point in the binary header of the *new c_actor file* . After initialization of the threadsafe C library, the runtime startup routine calls the C program as follows:

```
int main (int argc, char* argv[], char* envp[])
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As will be described later, *argc* is conventionally at least 1 , and the first member of the array points to a string containing the name (or pathname) of the executable file.

cactorcap is the returned capability of the new c_actor, and is a pointer to a *KnCap* structure (see actorCreate(2K)).

If *param* is NULL , default parameters are used, *acFlags* is set to AFX_ANY_SPACE and all others to un-specified . See acreate(2K) for a description of the AcParam structure.

By convention, *afexec* routines search for a binary, either fully linked or relocatable, in *path* . If no such binary exists, *afexec* routines search for a file with a `.r` suffix and then with a `.o` suffix. If the GZ_FEATURE is set to `true` , for each step, the *afexec* routines also search for a file with an additional `.gz` (for example: `path` , `path.gz` , `path.r` , `path.r.gz`).

path may also refer to a relocatable binary stored in a memory buffer in supervisor space, using the syntax "ram:0x<start>:0x<length>". <start> is the start address at which the binary is stored. <length> is the length of the binary. The buffer is not used any more after loading, except if the program uses the dynamic library API. Freeing the buffer after *afexec* may have unpredictable results if the program uses the dynamic library API.

Files to be loaded may be fully linked or relocatable files. They may also include dependencies on dynamic, but not shared, libraries. These libraries are looked for and loaded within the actor address space as part of the *afexec* call.

arg0 , ... , *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new c_actor. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or `file` or the last component of *path*).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new c_actor. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or `file` or the last component of *path*). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new c_actor. *envp* is terminated by a null pointer. For *afexecl* , *afexecv* , *afexeclp* and *afexecvp* , the C runtime start-off routine places a pointer to the environment of the calling c_actor in the global object:

```
extern char** environ;
```

and it is used to pass the environment of the calling c_actor to the new c_actor.

afexeclp and *afexecvp* are called with the same arguments as *afexecl* and *afexecv* , but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the PATH environment variable, and `file` must point to an executable file name for the new c_actor.

LIMITATIONS

Incompatible requests (such as *afexec* of a supervisor binary with the AFX_USER_SPACE flag set) are not normally detected. Applications can use the AFX_ANY_SPACE (see `acreate`(2K)) flag to force the system to deduce the type (`user` or `supervisor`) of the binary. In this case incompatible requests will be detected and refused.

Correct functioning of a supervisor actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) is not guaranteed. Even if the supervisor actor functions normally, only one supervisor actor can be executed in place at any one time from a given binary file. Attempting to execute more than one supervisor actor in place from the same binary file will be rejected by the system.

Correct functioning of any actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) when in flat memory mode is not guaranteed. Even if the actor functions normally, when in flat memory mode only one actor can be executed in place at any one time from a given binary file. Attempting to execute more than one actor in place in this situation will be rejected by the system.

RETURN VALUES

Upon successful completion, *afexec* returns a non-negative integer that is the *aid* of the new c_actor. Otherwise *afexec* returns –1 and sets `errno` to indicate one of the following error conditions.

ERRORS

[E2BIG]	The number of bytes used by the new c_actor's image argument list and environment list is greater than `ARG_MAX` bytes.
[EACCES]	Search permission is denied for a directory listed in the new c_actor's file path prefix.
[EBADSYM]	The runtime link-editor cannot find a symbol needed by the new c_actor. Possible reasons are that ChorusOS 4.0 `imake` rules have not been used to compile the program, or the link editor has found unexpected dynamic libraries.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	Inconsistent attribute flags were specified.
[EINVAL]	The address of the text or data region of the new c_actor is not consistent with the AFX_USER_SPACE or AFX_SUPERVISOR_SPACE attribute flag.
[EIO]	An I/O error occurred while reading from the file system.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ENOENT]	The new c_actor is a dynamic actor and one of its dependencies cannot be found either in the directories listed in the `LD_LIBRARY_PATH`

	environment variable, or in its runpath, or in `/usr/lib` .
[ENOEXEC]	The new c_actor file has the appropriate access permission but is not in the correct format.
[ENOTDIR]	A component of the new c_actor path of the file prefix is not a directory.
[EPERM]	The calling c_actor is not Trusted and is trying either to create a Trusted c_actor or set the new c_actor's credentials.
[ELOOP]	Too many symbolic links have been encountered during analysis of *path* .

Note - Error codes generated by *read* and *open* may also be returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`aconf`(2K) , `acreate`(2K) , `actorCreate`(2K) , `akill`(2K) , `aload`(2K) , `astart`(2K) , `astat`(2K) , `await`(2K) , `dlopen`(2K) , `fcntl`(2POSIX) , `intro`(2K)

NAME

afexec, afexecl, afexecv, afexecle, afexecve, afexeclp, afexecvp – create a new c_actor

SYNOPSIS

```
#include <am/afexec.h>
int afexecve(const char * path, KnCap * cactorcap, const AcParam * param, char const * argv, char const * envp);

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

int afexecv(const char * path, KnCap * cactorcap, const AcParam * param, char *const * argv);

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

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

int afexecvp(const char * file, KnCap * cactorcap, const AcParam * param, char *const * argv);
```

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

afexec family routines create a new c_actor with a single thread (the main thread). This main thread will execute with the priority defined by the *am.afexecschedprio* configuration parameter. It will have an automatically allocated stack, which size is defined by the *am.afexec.userstacksize* configuration parameter for user actors, and by *kern.exec.dflSysStackSize* for supervisor actors. (See intro(2K)).

The main thread starts executing the runtime startup routine (usually called *_start*) specified as the program entry point in the binary header of the *new c_actor file* . After initialization of the threadsafe C library, the runtime startup routine calls the C program as follows:

```
int main (int argc, char* argv[], char* envp[])
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As will be described later, *argc* is conventionally at least 1 , and the first member of the array points to a string containing the name (or pathname) of the executable file.

cactorcap is the returned capability of the new c_actor, and is a pointer to a *KnCap* structure (see actorCreate(2K)).

If *param* is NULL , default parameters are used, *acFlags* is set to AFX_ANY_SPACE and all others to un-specified . See acreate(2K) for a description of the AcParam structure.

By convention, *afexec* routines search for a binary, either fully linked or relocatable, in *path* . If no such binary exists, *afexec* routines search for a file with a `.r` suffix and then with a `.o` suffix. If the GZ_FEATURE is set to `true` , for each step, the *afexec* routines also search for a file with an additional `.gz` (for example: `path` , `path.gz` , `path.r` , `path.r.gz`).

path may also refer to a relocatable binary stored in a memory buffer in supervisor space, using the syntax "ram:0x<start>:0x<length>". <start> is the start address at which the binary is stored. <length> is the length of the binary. The buffer is not used any more after loading, except if the program uses the dynamic library API. Freeing the buffer after *afexec* may have unpredictable results if the program uses the dynamic library API.

Files to be loaded may be fully linked or relocatable files. They may also include dependencies on dynamic, but not shared, libraries. These libraries are looked for and loaded within the actor address space as part of the *afexec* call.

arg0 , ... , *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new c_actor. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or `file` or the last component of *path*).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new c_actor. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or `file` or the last component of *path*). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new c_actor. *envp* is terminated by a null pointer. For *afexecl* , *afexecv* , *afexeclp* and *afexecvp* , the C runtime start-off routine places a pointer to the environment of the calling c_actor in the global object:

```
extern char** environ;
```

and it is used to pass the environment of the calling c_actor to the new c_actor.

afexeclp and *afexecvp* are called with the same arguments as *afexecl* and *afexecv* , but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the PATH environment variable, and `file` must point to an executable file name for the new c_actor.

LIMITATIONS

Incompatible requests (such as *afexec* of a supervisor binary with the AFX_USER_SPACE flag set) are not normally detected. Applications can use the AFX_ANY_SPACE (see `acreate`(2K)) flag to force the system to deduce the type (`user` or `supervisor`) of the binary. In this case incompatible requests will be detected and refused.

Correct functioning of a supervisor actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) is not guaranteed. Even if the supervisor actor functions normally, only one supervisor actor can be executed in place at any one time from a given binary file. Attempting to execute more than one supervisor actor in place from the same binary file will be rejected by the system.

Correct functioning of any actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) when in flat memory mode is not guaranteed. Even if the actor functions normally, when in flat memory mode only one actor can be executed in place at any one time from a given binary file. Attempting to execute more than one actor in place in this situation will be rejected by the system.

RETURN VALUES

Upon successful completion, *afexec* returns a non-negative integer that is the *aid* of the new c_actor. Otherwise *afexec* returns –1 and sets `errno` to indicate one of the following error conditions.

ERRORS

[E2BIG]	The number of bytes used by the new c_actor's image argument list and environment list is greater than `ARG_MAX` bytes.
[EACCES]	Search permission is denied for a directory listed in the new c_actor's file path prefix.
[EBADSYM]	The runtime link-editor cannot find a symbol needed by the new c_actor. Possible reasons are that ChorusOS 4.0 `imake` rules have not been used to compile the program, or the link editor has found unexpected dynamic libraries.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	Inconsistent attribute flags were specified.
[EINVAL]	The address of the text or data region of the new c_actor is not consistent with the AFX_USER_SPACE or AFX_SUPERVISOR_SPACE attribute flag.
[EIO]	An I/O error occurred while reading from the file system.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ENOENT]	The new c_actor is a dynamic actor and one of its dependencies cannot be found either in the directories listed in the `LD_LIBRARY_PATH`

	environment variable, or in its runpath, or in `/usr/lib` .
[ENOEXEC]	The new c_actor file has the appropriate access permission but is not in the correct format.
[ENOTDIR]	A component of the new c_actor path of the file prefix is not a directory.
[EPERM]	The calling c_actor is not Trusted and is trying either to create a Trusted c_actor or set the new c_actor's credentials.
[ELOOP]	Too many symbolic links have been encountered during analysis of *path* .

Note - Error codes generated by *read* and *open* may also be returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`aconf`(2K) , `acreate`(2K) , `actorCreate`(2K) , `akill`(2K) , `aload`(2K) , `astart`(2K) , `astat`(2K) , `await`(2K) , `dlopen`(2K) , `fcntl`(2POSIX) , `intro`(2K)

NAME

afexec, afexecl, afexecv, afexecle, afexecve, afexeclp, afexecvp – create a new c_actor

SYNOPSIS

```
#include <am/afexec.h>
int afexecve(const char * path, KnCap * cactorcap, const AcParam * param, char const * argv, char const * envp);

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

int afexecv(const char * path, KnCap * cactorcap, const AcParam * param, char *const * argv);

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

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

int afexecvp(const char * file, KnCap * cactorcap, const AcParam * param, char *const * argv);
```

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

afexec family routines create a new c_actor with a single thread (the main thread). This main thread will execute with the priority defined by the *am.afexecschedprio* configuration parameter. It will have an automatically allocated stack, which size is defined by the *am.afexec.userstacksize* configuration parameter for user actors, and by *kern.exec.dflSysStackSize* for supervisor actors. (See intro(2K)).

The main thread starts executing the runtime startup routine (usually called *_start*) specified as the program entry point in the binary header of the *new c_actor file* . After initialization of the threadsafe C library, the runtime startup routine calls the C program as follows:

```
int main (int argc, char* argv[], char* envp[])
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As will be described later, *argc* is conventionally at least 1 , and the first member of the array points to a string containing the name (or pathname) of the executable file.

cactorcap is the returned capability of the new c_actor, and is a pointer to a *KnCap* structure (see actorCreate(2K)).

If *param* is NULL , default parameters are used, *acFlags* is set to AFX_ANY_SPACE and all others to un-specified . See acreate(2K) for a description of the AcParam structure.

By convention, *afexec* routines search for a binary, either fully linked or relocatable, in *path* . If no such binary exists, *afexec* routines search for a file with a `.r` suffix and then with a `.o` suffix. If the GZ_FEATURE is set to `true` , for each step, the *afexec* routines also search for a file with an additional `.gz` (for example: `path` , `path.gz` , `path.r` , `path.r.gz`).

path may also refer to a relocatable binary stored in a memory buffer in supervisor space, using the syntax "ram:0x<start>:0x<length>". <start> is the start address at which the binary is stored. <length> is the length of the binary. The buffer is not used any more after loading, except if the program uses the dynamic library API. Freeing the buffer after *afexec* may have unpredictable results if the program uses the dynamic library API.

Files to be loaded may be fully linked or relocatable files. They may also include dependencies on dynamic, but not shared, libraries. These libraries are looked for and loaded within the actor address space as part of the *afexec* call.

arg0 , ... , *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new c_actor. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or `file` or the last component of *path*).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new c_actor. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or `file` or the last component of *path*). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new c_actor. *envp* is terminated by a null pointer. For *afexecl* , *afexecv* , *afexeclp* and *afexecvp* , the C runtime start-off routine places a pointer to the environment of the calling c_actor in the global object:

```
extern char** environ;
```

and it is used to pass the environment of the calling c_actor to the new c_actor.

afexeclp and *afexecvp* are called with the same arguments as *afexecl* and *afexecv* , but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the PATH environment variable, and `file` must point to an executable file name for the new c_actor.

LIMITATIONS

Incompatible requests (such as *afexec* of a supervisor binary with the AFX_USER_SPACE flag set) are not normally detected. Applications can use the AFX_ANY_SPACE (see `acreate`(2K)) flag to force the system to deduce the type (`user` or `supervisor`) of the binary. In this case incompatible requests will be detected and refused.

Correct functioning of a supervisor actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) is not guaranteed. Even if the supervisor actor functions normally, only one supervisor actor can be executed in place at any one time from a given binary file. Attempting to execute more than one supervisor actor in place from the same binary file will be rejected by the system.

Correct functioning of any actor executed in place (using the AFX_EXECUTE_IN_PLACE flag) when in flat memory mode is not guaranteed. Even if the actor functions normally, when in flat memory mode only one actor can be executed in place at any one time from a given binary file. Attempting to execute more than one actor in place in this situation will be rejected by the system.

RETURN VALUES

Upon successful completion, *afexec* returns a non-negative integer that is the *aid* of the new c_actor. Otherwise *afexec* returns –1 and sets `errno` to indicate one of the following error conditions.

ERRORS

[E2BIG]	The number of bytes used by the new c_actor's image argument list and environment list is greater than `ARG_MAX` bytes.
[EACCES]	Search permission is denied for a directory listed in the new c_actor's file path prefix.
[EBADSYM]	The runtime link-editor cannot find a symbol needed by the new c_actor. Possible reasons are that ChorusOS 4.0 `imake` rules have not been used to compile the program, or the link editor has found unexpected dynamic libraries.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	Inconsistent attribute flags were specified.
[EINVAL]	The address of the text or data region of the new c_actor is not consistent with the AFX_USER_SPACE or AFX_SUPERVISOR_SPACE attribute flag.
[EIO]	An I/O error occurred while reading from the file system.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ENOENT]	The new c_actor is a dynamic actor and one of its dependencies cannot be found either in the directories listed in the `LD_LIBRARY_PATH`

	environment variable, or in its runpath, or in `/usr/lib` .
[ENOEXEC]	The new c_actor file has the appropriate access permission but is not in the correct format.
[ENOTDIR]	A component of the new c_actor path of the file prefix is not a directory.
[EPERM]	The calling c_actor is not Trusted and is trying either to create a Trusted c_actor or set the new c_actor's credentials.
[ELOOP]	Too many symbolic links have been encountered during analysis of *path* .

Note - Error codes generated by *read* and *open* may also be returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`aconf`(2K) , `acreate`(2K) , `actorCreate`(2K) , `akill`(2K) , `aload`(2K) , `astart`(2K) , `astat`(2K) , `await`(2K) , `dlopen`(2K) , `fcntl`(2POSIX) , `intro`(2K)

NAME

aload, alParamBuild, alParamUnpack, agetalparam – Manage the loading of an actor

SYNOPSIS

#include <am/afexec.h>
int **aload**(const KnCap * *cactorcap*, const char * *path*, const AlParam * *ldParam*, KnPc *entry*);

void **alParamBuild**(void * *buffer*, unsigned int *bufferSize*, const char *const * *argv*, const char *const * *envp*);

int **alParamUnpack**(const Alparam * *alParamp*, int * *argcp*, char *** *argvp*);

unsigned int **alParamSize**(const Alparam * *alParamp*);

int **agetalparam**(const AlParam ** *argp*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The `aload()` call initializes the text and data regions of the target actor specified by *cactorcap* . This call also initializes the environment and arguments of the target actor.

The target actor is loaded from an executable object file whose pathname is specified by *path* . This file consists of a binary header, a text segment, and the initialized part of the data segment. The file may be in any of the executable file formats supported by the system.

The *param* argument points to an *AlParam* data structure, which is a packed representation of the environment, and arguments which will be copied to a region of the target actor address space. The address of this region can subsequently be obtained by the target actor using the `agetalparam()` call. This region is freed when the target actor starts executing (see `start`(2K)), after the *args* and *env* have been installed in the actor address space (heap or stack).

An *AlParam* structure can be built using `alParamBuild()` .

If *path* is a `NULL` pointer, no executable file is loaded.

If *ldParam* is a `NULL` pointer, the packed arguments and environment currently installed in the actor are not affected.

If *entry* is not a `NULL` pointer, it must point to a valid location large enough to store a `KnPc` object. In that case, the entry-point as determined by the loader is returned into the location pointed to by *entry* .

The `aload()` call can be used on any actor as many times as necessary. It is up to the caller to ensure that the address ranges to be used in the target actor are actually free, and that the target actor is in a state where it is safe to modify its address space.

The `alParamBuild()` command builds an `AlParam` structure from the *argv* and *envp* arrays provided, and stores the result in the buffer *buffer* . The *buffer*

field must point to an area of memory large enough to hold the arguments and environment described by *argv* and *envp* .

The `alParamUnpack( )` command retrieves the arguments and environment from the data structure pointed to by *alParamp* , copies them to the newly allocated memory, builds the environment string pointer arrays and builds the argument's string pointers array whose addresses are stored in the areas pointed to by *argvp* . The number of arguments is returned into the location pointed to by *argcp* .

Do not build an `AlParam` data structure containing more than `ARG_MAX` bytes of data. The only limit is the amount of memory available when unpacking it, and the amount of memory available in the system to store a copy of it in the *restart parameters* of a new actor.

The *alparamsize* field shows the total size used of the `AlParam` object referred to by *alParamp* . The size returned accounts for the `AlParam` object and the strings that follow.

The `agetalparam( )` call returns, at the location pointed to by *argp* , the address where the packed arguments and environment are stored in the calling actor's address space.

Note that `agetalparam( )` and `alParamUnpack( )` are not normally used by the application; they are used by the C library startup code to build the arguments and environment in the traditional UNIX representation. The packed arguments and environment themselves are not affected, nor used by, the `getenv( )` and `setenv( )` library routines. If permitted by the nucleus configuration, the region containing the packed arguments and environment will be read-only.

LIMITATIONS

It should be noted that `aload( )` is usually the first event that allocates regions to an actor created using `acreate( )` .

BUGS

When installing the new packed arguments and environment, `aload( )` does not free the packed arguments and environment region resulting from a previous call to `aload( )` .

RETURN VALUE

Upon successful completion, `aload( )` returns `0` . Otherwise `aload( )` returns `–1` and sets `errno` to indicate one of the following error conditions:

[ENOENT]	One or more components of the new actor's pathname do not exist.
[ENOTDIR]	A component of the new actor's pathname prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new actor file's path prefix.

[ENOEXEC]	The new actor file has the appropriate access permission but is not in the proper format.
[EFAULT]	The new actor file is not as long as indicated by the size values in its header.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	The address of the text or data region of the new actor is not consistent with the new actor privilege (SUPERVISOR, USER, SYSTEM).
[ENAMETOOLONG]	The length of a component of *path* exceeds NAME_MAX characters or the length of *path* exceeds PATH_MAX characters.
[ELOOP]	Too many symbolic links have been encountered during analysis of the file.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ESRCH]	The actor referred to by the given capability does not exist.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

acreate(2K) , afexec(2K) , akill(2K) , astart(2K) , hrfexec(2RESTART)

NAME

agetId – get c_actor's ID

SYNOPSIS

```
#include <am/afexec.h>
int agetId(void);
```

FEATURES

ACTOR_EXTENDED_MNGT

RETURN VALUE

The `agetId()` call returns the current c_actor's ID (see `intro`(2K)).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`afexec`(2K), `acap`(2K)

NAME

akill – kill or restart an actor

SYNOPSIS

```
#include <am/afexec.h>
int akill(const KnCap *cactorcap);
```

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The `akill( )` call stops and deletes the actor identified by *cactorcap* (see `afexec`(2K)), unless the actor is a restartable actor. If the actor is a restartable actor, calling `akill( )` simply restarts the actor, as if it had terminated abnormally. To permanently kill (and not simply restart) a restartable actor, call `hrKillGroup`(2RESTART).

RETURN VALUES

The `akill( )` call returns `0` if the actor was killed successfully (even if it restarts). Otherwise, `akill( )` returns `–1` and sets `errno` to indicate one of the following error conditions:

[EFAULT]	*cactorcap* points to an illegal address.
[ESRCH]	*cactorcap* points to an illegal capability or the actor with the capability given doesn't exist.
[EPERM]	the calling actor is not a trusted actor or a supervisor actor, and the user-id of the calling actor does not match the user-id of the killed actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`afexec`(2K), `await`(2K), `hrKillGroup`(2RESTART)

NAME

aload, alParamBuild, alParamUnpack, agetalparam – Manage the loading of an actor

SYNOPSIS

#include <am/afexec.h>
int **aload**(const KnCap * *cactorcap*, const char * *path*, const AlParam * *ldParam*, KnPc *entry*);

void **alParamBuild**(void * *buffer*, unsigned int *bufferSize*, const char *const * *argv*, const char *const * *envp*);

int **alParamUnpack**(const Alparam * *alParamp*, int * *argcp*, char *** *argvp*);

unsigned int **alParamSize**(const Alparam * *alParamp*);

int **agetalparam**(const AlParam ** *argp*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The `aload()` call initializes the text and data regions of the target actor specified by *cactorcap* . This call also initializes the environment and arguments of the target actor.

The target actor is loaded from an executable object file whose pathname is specified by *path* . This file consists of a binary header, a text segment, and the initialized part of the data segment. The file may be in any of the executable file formats supported by the system.

The *param* argument points to an *AlParam* data structure, which is a packed representation of the environment, and arguments which will be copied to a region of the target actor address space. The address of this region can subsequently be obtained by the target actor using the `agetalparam()` call. This region is freed when the target actor starts executing (see `start`(2K)), after the *args* and *env* have been installed in the actor address space (heap or stack).

An *AlParam* structure can be built using `alParamBuild()` .

If *path* is a `NULL` pointer, no executable file is loaded.

If *ldParam* is a `NULL` pointer, the packed arguments and environment currently installed in the actor are not affected.

If *entry* is not a `NULL` pointer, it must point to a valid location large enough to store a `KnPc` object. In that case, the entry-point as determined by the loader is returned into the location pointed to by *entry* .

The `aload()` call can be used on any actor as many times as necessary. It is up to the caller to ensure that the address ranges to be used in the target actor are actually free, and that the target actor is in a state where it is safe to modify its address space.

The `alParamBuild()` command builds an `AlParam` structure from the *argv* and *envp* arrays provided, and stores the result in the buffer *buffer* . The *buffer*

field must point to an area of memory large enough to hold the arguments and environment described by *argv* and *envp* .

The `alParamUnpack()` command retrieves the arguments and environment from the data structure pointed to by *alParamp* , copies them to the newly allocated memory, builds the environment string pointer arrays and builds the argument's string pointers array whose addresses are stored in the areas pointed to by *argvp* . The number of arguments is returned into the location pointed to by *argcp* .

Do not build an `AlParam` data structure containing more than `ARG_MAX` bytes of data. The only limit is the amount of memory available when unpacking it, and the amount of memory available in the system to store a copy of it in the *restart parameters* of a new actor.

The *alparamsize* field shows the total size used of the `AlParam` object referred to by *alParamp* . The size returned accounts for the `AlParam` object and the strings that follow.

The `agetalparam()` call returns, at the location pointed to by *argp* , the address where the packed arguments and environment are stored in the calling actor's address space.

Note that `agetalparam()` and `alParamUnpack()` are not normally used by the application; they are used by the C library startup code to build the arguments and environment in the traditional UNIX representation. The packed arguments and environment themselves are not affected, nor used by, the `getenv()` and `setenv()` library routines. If permitted by the nucleus configuration, the region containing the packed arguments and environment will be read-only.

LIMITATIONS

It should be noted that `aload()` is usually the first event that allocates regions to an actor created using `acreate()` .

BUGS

When installing the new packed arguments and environment, `aload()` does not free the packed arguments and environment region resulting from a previous call to `aload()` .

RETURN VALUE

Upon successful completion, `aload()` returns `0` . Otherwise `aload()` returns `–1` and sets `errno` to indicate one of the following error conditions:

[ENOENT]	One or more components of the new actor's pathname do not exist.
[ENOTDIR]	A component of the new actor's pathname prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new actor file's path prefix.

[ENOEXEC]	The new actor file has the appropriate access permission but is not in the proper format.
[EFAULT]	The new actor file is not as long as indicated by the size values in its header.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	The address of the text or data region of the new actor is not consistent with the new actor privilege (SUPERVISOR, USER, SYSTEM).
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ELOOP]	Too many symbolic links have been encountered during analysis of the file.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ESRCH]	The actor referred to by the given capability does not exist.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`acreate`(2K) , `afexec`(2K) , `akill`(2K) , `astart`(2K) , `hrfexec`(2RESTART)

NAME

aload, alParamBuild, alParamUnpack, agetalparam – Manage the loading of an actor

SYNOPSIS

```
#include <am/afexec.h>
int aload(const KnCap * cactorcap, const char * path, const AlParam * ldParam, KnPc entry);

void alParamBuild(void * buffer, unsigned int bufferSize, const char *const * argv, const
char *const * envp);

int alParamUnpack(const Alparam * alParamp, int * argcp, char *** argvp);

unsigned int alParamSize(const Alparam * alParamp);

int agetalparam(const AlParam ** argp);
```

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The `aload()` call initializes the text and data regions of the target actor specified by *cactorcap* . This call also initializes the environment and arguments of the target actor.

The target actor is loaded from an executable object file whose pathname is specified by *path* . This file consists of a binary header, a text segment, and the initialized part of the data segment. The file may be in any of the executable file formats supported by the system.

The *param* argument points to an *AlParam* data structure, which is a packed representation of the environment, and arguments which will be copied to a region of the target actor address space. The address of this region can subsequently be obtained by the target actor using the `agetalparam()` call. This region is freed when the target actor starts executing (see `start`(2K)), after the *args* and *env* have been installed in the actor address space (heap or stack).

An *AlParam* structure can be built using `alParamBuild()` .

If *path* is a `NULL` pointer, no executable file is loaded.

If *ldParam* is a `NULL` pointer, the packed arguments and environment currently installed in the actor are not affected.

If *entry* is not a `NULL` pointer, it must point to a valid location large enough to store a `KnPc` object. In that case, the entry-point as determined by the loader is returned into the location pointed to by *entry* .

The `aload()` call can be used on any actor as many times as necessary. It is up to the caller to ensure that the address ranges to be used in the target actor are actually free, and that the target actor is in a state where it is safe to modify its address space.

The `alParamBuild()` command builds an `AlParam` structure from the *argv* and *envp* arrays provided, and stores the result in the buffer *buffer* . The *buffer*

field must point to an area of memory large enough to hold the arguments and environment described by *argv* and *envp* .

The `alParamUnpack( )` command retrieves the arguments and environment from the data structure pointed to by *alParamp* , copies them to the newly allocated memory, builds the environment string pointer arrays and builds the argument's string pointers array whose addresses are stored in the areas pointed to by *argvp* . The number of arguments is returned into the location pointed to by *argcp* .

Do not build an `AlParam` data structure containing more than `ARG_MAX` bytes of data. The only limit is the amount of memory available when unpacking it, and the amount of memory available in the system to store a copy of it in the *restart parameters* of a new actor.

The *alparamsize* field shows the total size used of the `AlParam` object referred to by *alParamp* . The size returned accounts for the `AlParam` object and the strings that follow.

The `agetalparam( )` call returns, at the location pointed to by *argp* , the address where the packed arguments and environment are stored in the calling actor's address space.

Note that `agetalparam( )` and `alParamUnpack( )` are not normally used by the application; they are used by the C library startup code to build the arguments and environment in the traditional UNIX representation. The packed arguments and environment themselves are not affected, nor used by, the `getenv( )` and `setenv( )` library routines. If permitted by the nucleus configuration, the region containing the packed arguments and environment will be read-only.

LIMITATIONS

It should be noted that `aload( )` is usually the first event that allocates regions to an actor created using `acreate( )` .

BUGS

When installing the new packed arguments and environment, `aload( )` does not free the packed arguments and environment region resulting from a previous call to `aload( )` .

RETURN VALUE

Upon successful completion, `aload( )` returns `0` . Otherwise `aload( )` returns `–1` and sets `errno` to indicate one of the following error conditions:

[ENOENT]	One or more components of the new actor's pathname do not exist.
[ENOTDIR]	A component of the new actor's pathname prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new actor file's path prefix.

[ENOEXEC]	The new actor file has the appropriate access permission but is not in the proper format.
[EFAULT]	The new actor file is not as long as indicated by the size values in its header.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	The address of the text or data region of the new actor is not consistent with the new actor privilege (SUPERVISOR, USER, SYSTEM).
[ENAMETOOLONG]	The length of a component of *path* exceeds NAME_MAX characters or the length of *path* exceeds PATH_MAX characters.
[ELOOP]	Too many symbolic links have been encountered during analysis of the file.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ESRCH]	The actor referred to by the given capability does not exist.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

acreate(2K) , afexec(2K) , akill(2K) , astart(2K) , hrfexec(2RESTART)

NAME

aload, alParamBuild, alParamUnpack, agetalparam – Manage the loading of an actor

SYNOPSIS

#include <am/afexec.h>
int **aload**(const KnCap * *cactorcap*, const char * *path*, const AlParam * *ldParam*, KnPc *entry*);

void **alParamBuild**(void * *buffer*, unsigned int *bufferSize*, const char *const * *argv*, const char *const * *envp*);

int **alParamUnpack**(const Alparam * *alParamp*, int * *argcp*, char *** *argvp*);

unsigned int **alParamSize**(const Alparam * *alParamp*);

int **agetalparam**(const AlParam ** *argp*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The `aload()` call initializes the text and data regions of the target actor specified by *cactorcap* . This call also initializes the environment and arguments of the target actor.

The target actor is loaded from an executable object file whose pathname is specified by *path* . This file consists of a binary header, a text segment, and the initialized part of the data segment. The file may be in any of the executable file formats supported by the system.

The *param* argument points to an *AlParam* data structure, which is a packed representation of the environment, and arguments which will be copied to a region of the target actor address space. The address of this region can subsequently be obtained by the target actor using the `agetalparam()` call. This region is freed when the target actor starts executing (see `start`(2K)), after the *args* and *env* have been installed in the actor address space (heap or stack).

An *AlParam* structure can be built using `alParamBuild()` .

If *path* is a `NULL` pointer, no executable file is loaded.

If *ldParam* is a `NULL` pointer, the packed arguments and environment currently installed in the actor are not affected.

If *entry* is not a `NULL` pointer, it must point to a valid location large enough to store a `KnPc` object. In that case, the entry-point as determined by the loader is returned into the location pointed to by *entry* .

The `aload()` call can be used on any actor as many times as necessary. It is up to the caller to ensure that the address ranges to be used in the target actor are actually free, and that the target actor is in a state where it is safe to modify its address space.

The `alParamBuild()` command builds an `AlParam` structure from the *argv* and *envp* arrays provided, and stores the result in the buffer *buffer* . The *buffer*

field must point to an area of memory large enough to hold the arguments and environment described by *argv* and *envp* .

The `alParamUnpack( )` command retrieves the arguments and environment from the data structure pointed to by *alParamp* , copies them to the newly allocated memory, builds the environment string pointer arrays and builds the argument's string pointers array whose addresses are stored in the areas pointed to by *argvp* . The number of arguments is returned into the location pointed to by *argcp* .

Do not build an `AlParam` data structure containing more than `ARG_MAX` bytes of data. The only limit is the amount of memory available when unpacking it, and the amount of memory available in the system to store a copy of it in the *restart parameters* of a new actor.

The *alparamsize* field shows the total size used of the `AlParam` object referred to by *alParamp* . The size returned accounts for the `AlParam` object and the strings that follow.

The `agetalparam( )` call returns, at the location pointed to by *argp* , the address where the packed arguments and environment are stored in the calling actor's address space.

Note that `agetalparam( )` and `alParamUnpack( )` are not normally used by the application; they are used by the C library startup code to build the arguments and environment in the traditional UNIX representation. The packed arguments and environment themselves are not affected, nor used by, the `getenv( )` and `setenv( )` library routines. If permitted by the nucleus configuration, the region containing the packed arguments and environment will be read-only.

LIMITATIONS

It should be noted that `aload( )` is usually the first event that allocates regions to an actor created using `acreate( )` .

BUGS

When installing the new packed arguments and environment, `aload( )` does not free the packed arguments and environment region resulting from a previous call to `aload( )` .

RETURN VALUE

Upon successful completion, `aload( )` returns `0` . Otherwise `aload( )` returns `–1` and sets `errno` to indicate one of the following error conditions:

[ENOENT]	One or more components of the new actor's pathname do not exist.
[ENOTDIR]	A component of the new actor's pathname prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new actor file's path prefix.

[ENOEXEC]	The new actor file has the appropriate access permission but is not in the proper format.
[EFAULT]	The new actor file is not as long as indicated by the size values in its header.
[EFAULT]	*path* , *argv* , or *envp* point to an illegal address.
[EINVAL]	The address of the text or data region of the new actor is not consistent with the new actor privilege (SUPERVISOR, USER, SYSTEM).
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ELOOP]	Too many symbolic links have been encountered during analysis of the file.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ESRCH]	The actor referred to by the given capability does not exist.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`acreate`(2K) , `afexec`(2K) , `akill`(2K) , `astart`(2K) , `hrfexec`(2RESTART)

NAME

astart – activates a c_actor

SYNOPSIS

```
#include <am/afexec.h>
int astart(const KnCap *cactorcap);
```

FEATURES

RESTART_EXTENDED

DESCRIPTION

The `astart()` call creates one thread in the c_actor specified by *cactorcap*. That thread executes the c_actor's text at the entry-point defined by the executable file header determined during the last use of the `aload`(2K) operation applied to that c_actor.

The *cactorcap* pointer indicates the capability of the c_actor to be activated.

Although unlikely, it is possible that you may want to start a number of threads at the actor's entry point by calling `astart()` repeatedly.

LIMITATIONS

The `astart()` cannot act on a remote c_actor. If the capability specified refers to a remote actor, this call fails and returns the `ENOTIMPLEMENTED` error code.

RETURN VALUE

Upon successful completion, these routines return `0`. Otherwise they return `–1` and set `errno` to indicate the following error condition:

[ESRCH] The c_actor refered to by the given capability does not exist.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`acreate`(2K), `afexec`(2K), `akill`(2K), `aload`(2K)

NAME

astat – list all active c_actors

SYNOPSIS

#include <am/astat.h>
int **astat**(astatEntry **entries*, int *maxEntries*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The `astat()` call fills an entry defined by the *entries* array for each c_actor present. The maximum number of entries is defined by *maxEntries*.

The *entries* array is a structure containing the following members:

```
int     astatFlags;     /* c_actor's status  */
int     astatAid;       /* c_actor's ID */
int     astatUid;       /* c_actor's user ID  */
char    astatName[];    /* c_actor's name */
```

The status information *astatFlags* is bit-encoded using the following bits:

```
ASTAT_DEBUGGED_CACTOR  the c_actor is being debugged
```

RETURN VALUE

Upon successful completion, *astat* returns the number of active c_actors; otherwise it returns –1 and sets `errno` to indicate the following error condition:

[EFAULT] *entries* points outside the allocated address space of the c_actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`afexec`(2K)

NAME

atrace – c_actor trace

SYNOPSIS

```
#include <chorus.h>
#include <am/reg.h>
#include <am/atrace.h>
int atrace(int request, const KnCap *cactorcap, char *addr, int data, char *addr2);
```

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The `atrace()` call allows a c_actor to control the execution of another c_actor. This system call is similar to the `ptrace()` system call in UNIX. Its primary use is for the implementation of breakpoint debugging. The traced c_actor behaves normally until it encounters an exception, at which point it enters a stopped state and the debugger is notified using `awaits`(2K).

The system maintains the concept of a `target thread`, which is the identifier of the thread that encountered the exception. Some `atrace()` requests apply to the thread identified by the value of the target thread. (ATRACE_SINGLESTEP, ATRACE_SETREGS and ATRACE_GETREGS.) The target thread is (re)set by the system as soon as a thread causes the c_actor to stop. It can also be reset by the debugger using a specific `atrace()` request.

When the c_actor is stopped, the debugger can examine and modify its *core image* using `atrace()`. The debugger can also make the c_actor terminate or continue. By using the ATRACE_THREADSUSPEND and ATRACE_THREADRESUME requests before making the c_actor continue execution using ATRACE_CONT, the debugger can restart a subset of the threads within the c_actor.

The *request* argument determines the precise action to be taken by `atrace()`. For each request, *cactorcap* must designate a debugged c_actor. If the semantics of the *addr*, *data* and/or *addr2* arguments are not specified for a request, those arguments are ignored. The debugged c_actor must be stopped before the requests can be made.

ATRACE_PEEKTEXT, ATRACE_PEEKDATA
: These requests return the word at the location *addr* in the address space of the c_actor to the debugger. These two requests will fail if *addr* is not the start address of a word or is outside the c_actor's address space; in this case, a value of `–1` is returned to the debugger and `errno` is set to EIO.

ATRACE_POKETEXT, ATRACE_POKEDATA
: These requests put the value given by the *data* argument into the address space of the c_actor at the location *addr*. Upon successful completion, the value written into the address space of the c_actor is returned to the debugger. These two requests will fail if *addr* is not the start address of a word or is outside the c_actor's address space; in this case, a value of `–1` is returned to the debugger and `errno` is set to EFAULT.

ATRACE_CONT
This request causes the c_actor to resume execution. The target thread resumes execution at the address *addr*. All other threads will also resume execution, unless suspended by ATRACE_THREADSUSPEND. For this request, the *data* argument must be equal to 0. Upon successful completion, 0 is returned to the debugger; otherwise –1 is returned and errno is set to EFAULT.

ATRACE_KILL
This request causes the c_actor to terminate with the same consequences as _exit(2K).

ATRACE_SINGLESTEP
This request sets the trace bit in the Processor Status Word of the target thread, it then executes the same steps listed above for the ATRACE_CONT request, with the exception that only the target thread will be restarted. All other threads remain in a stopped state. The trace bit causes an exception upon completion of one machine instruction, which allows single-stepping of the target thread.

ATRACE_ATTACH
Start tracing a running c_actor, designated by *cactorcap*.

ATRACE_DETACH
Stop tracing the c_actor designated by *cactorcap*. The c_actor continues its execution from the address *addr*. If *addr* is defined as (char *) 1 the execution continues from where it stopped. The *data* argument must be 0.

ATRACE_GETREGS, ATRACE_SETREGS
ATRACE_GETREGS returns the registers of the target thread in the c_actor designated by *cactorcap* to a table pointed to by *addr*. The registers table type is regs as described in <am/reg.h>. ATRACE_SETREGS writes the registers of the target thread in the c_actor designated by *cactorcap* from a table pointed to by *addr*. The registers table type is regs as described in <am/reg.h>.

ATRACE_GETTHREADREGS, ATRACE_SETTHREADREGS
ATRACE_GETTHREADREGS returns the registers of the thread identified by *data* in the c_actor designated by *cactorcap* to a structure pointed to by *addr*. The registers table type is regs as described in <am/reg.h>. ATRACE_SETTHREADREGS writes the registers of the thread identified by *data* in the c_actor designated by *cactorcap* from a structure pointed to by *addr*.The registers table type is regs as described in <am/reg.h>.

ATRACE_GETTARGETTHREAD
The identifier of the current target thread is returned.

ATRACE_SETTARGETTHREAD
Set the target thread to the thread identified by the *data* argument.

ATRACE_THREADSUSPEND
Suspend the thread identified by *data* (see threadSuspend(2K)). If that thread is the thread which caused the c_actor to stop, the thread will only be suspended at the time the debugger restarts the c_actor using ATRACE_CONT or ATRACE_DETACH.

ATRACE_THREADRESUME
Resume the thread identified by *data* (see threadResume(2K)).

ATRACE_THREADLIST
Return the number of threads in the c_actor designated by *cactorcap*. For each thread within the c_actor, return the KnThreadStat structure in an array pointed to by *addr*. The number of bytes copied to *addr* is limited to the amount specified in *data* (see threadStat(2K)).

ATRACE_GETTHREADNAME, ATRACE_SETTHREADNAME
Get or set the symbolic name of the thread identified by *data*.

Note: –1 can be a legitimate return value of atrace( ) in the case of a PEEK request. Therefore, the only way to check whether atrace( ) has failed is to set errno to 0 before calling atrace( ) and then check the value of the errno variable.

RETURN VALUES

If atrace( ) fails, it will set errno to one of the following values:

[EIO]	*request* is an illegal number.
[EFAULT]	One or more of the arguments provided is outside the caller's memory space.
[ESRCH]	The thread designated by *data* does not correspond to a valid thread identifier within the c_actor designated by *cactorcap*.
[ESRCH]	*cactorcap* designated a c_actor that does not exist or is not debugged by the calling c_actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

awaits(2K), threadSuspend(2K), threadResume(2K), threadStat(2K), _exit(2K)

NAME

await, awaits – wait for c_actor to terminate or stop

SYNOPSIS

```
#include <am/await.h>
int awaits(const KnCap * cactorcap, int * statusp);

IS_EXIT_EVT(int status);

GET_EXIT_EVT(int status);

IS_KILL_EVT(int status);

GET_KILL_EVT(int status);

IS_STP_EVT(int status);

GET_STP_EVT(int status);

int await(const KnCap * cactorcap);
```

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The awaits( ) call blocks the caller until the c_actor identified by *cactorcap* (see afexec(2K)) terminates or stops due to tracing.

If *statusp* is not NULL , the status of the c_actor designated by *cactorcap* is stored in the location pointed to by *statusp* upon successful return from awaits( ) . It indicates why the c_actor terminated or stopped.

The IS_EXIT_EVT , GET_EXIT_EVT , IS_KILL_EVT , GET_KILL_EVT , IS_STP_EVT and GET_STP_EVT macros take an argument of the int type as returned by awaits( ) in the location pointed to by *statusp* . If the c_actor terminated due to a call to _exit(2K) , IS_EXIT_EVT will be true (1) and GET_EXIT_EVT will be the low-order byte of the argument the c_actor passed to _exit( ) . If the c_actor terminated due to an asynchronous event (exception, actorDelete(2K) , and so forth) IS_KILL_EVT will be true (1). If the c_actor was being debugged and caught an exception, IS_STP_EVT will be true (1). In both of these cases GET_KILL_EVT or GET_STP_EVT will be the number of the event which caused the c_actor to terminate or stop.

The following events are valid:

EVT_ILL	(illegal instruction exception)
EVT_BKPT	(breakpoint exception)
EVT_FPE	(floating point exception)
EVT_SEGV	(memory protection exception)
EVT_AKILL	(c_actor killed by akill(2K))
EVT_DELETE	(c_actor killed by actorDelete(2K))

The `await( )` call performs the same function as `awaits( )`, and should be used when *statusp* is a `NULL` pointer.

RETURN VALUES

Both `awaits( )` and `await( )` return the *aid* of the c_actor which terminated or stopped due to tracing; otherwise they return `–1` and set `errno` to indicate one of the following error conditions:

[EINVAL]	*cactorcap* refers to the calling c_actor.
[ESRCH]	*cactorcap* doesn't designate a valid c_actor.
[EFAULT]	*cactorcap* or *statusp* points to an illegal address.
[EINTR]	The calling thread was aborted before the execution of the c_actor was terminated.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorDelete`(2K), `afexec`(2K), `akill`(2K), `_exit`(2K)

RESTRICTIONS

It is not currently possible to distinguish between terminated c_actors and invalid actor capabilities.

NAME

await, awaits – wait for c_actor to terminate or stop

SYNOPSIS

#include <am/await.h>
int **awaits**(const KnCap * *cactorcap*, int * *statusp*);

IS_EXIT_EVT(int *status*);

GET_EXIT_EVT(int *status*);

IS_KILL_EVT(int *status*);

GET_KILL_EVT(int *status*);

IS_STP_EVT(int *status*);

GET_STP_EVT(int *status*);

int **await**(const KnCap * *cactorcap*);

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The awaits( ) call blocks the caller until the c_actor identified by *cactorcap* (see afexec(2K)) terminates or stops due to tracing.

If *statusp* is not NULL , the status of the c_actor designated by *cactorcap* is stored in the location pointed to by *statusp* upon successful return from awaits( ) . It indicates why the c_actor terminated or stopped.

The IS_EXIT_EVT , GET_EXIT_EVT , IS_KILL_EVT , GET_KILL_EVT , IS_STP_EVT and GET_STP_EVT macros take an argument of the int type as returned by awaits( ) in the location pointed to by *statusp* . If the c_actor terminated due to a call to _exit(2K) , IS_EXIT_EVT will be true (1) and GET_EXIT_EVT will be the low-order byte of the argument the c_actor passed to _exit( ) . If the c_actor terminated due to an asynchronous event (exception, actorDelete(2K) , and so forth) IS_KILL_EVT will be true (1). If the c_actor was being debugged and caught an exception, IS_STP_EVT will be true (1). In both of these cases GET_KILL_EVT or GET_STP_EVT will be the number of the event which caused the c_actor to terminate or stop.

The following events are valid:

EVT_ILL	(illegal instruction exception)
EVT_BKPT	(breakpoint exception)
EVT_FPE	(floating point exception)
EVT_SEGV	(memory protection exception)
EVT_AKILL	(c_actor killed by akill(2K))
EVT_DELETE	(c_actor killed by actorDelete(2K))

The `await( )` call performs the same function as `awaits( )`, and should be used when *statusp* is a `NULL` pointer.

RETURN VALUES

Both `awaits( )` and `await( )` return the *aid* of the c_actor which terminated or stopped due to tracing; otherwise they return `–1` and set `errno` to indicate one of the following error conditions:

[EINVAL]	*cactorcap* refers to the calling c_actor.
[ESRCH]	*cactorcap* doesn't designate a valid c_actor.
[EFAULT]	*cactorcap* or *statusp* points to an illegal address.
[EINTR]	The calling thread was aborted before the execution of the c_actor was terminated.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorDelete`(2K), `afexec`(2K), `akill`(2K), `_exit`(2K)

RESTRICTIONS

It is not currently possible to distinguish between terminated c_actors and invalid actor capabilities.

NAME

dladdr – translate address to symbolic information

SYNOPSIS

#include <cx/dlfcn.h>
int **dladdr**(void **address*, Dl_info **dlip*);

FEATURES

ACTOR_EXTENDED_MNGT, DYNAMIC_LIB

DESCRIPTION

`dladdr()` is a member of the dynamic linking API, a family of routines that give the user direct access to dynamic linking functionality.

`dladdr()` is *only* available to programs and libraries compiled with `imake` rules of the form `Dynamic`*Type*`Target()`, where *Type* is one of `User`, `Sup`, `CXXUser`, `CXXSup` or `Library`.

See *ChorusOS 4.0 Introduction* for more information on building executables with dynamic libraries.

`dladdr()` determines whether the specified address is located within one of the mapped objects that make up the address space of the current application. An address is considered to fall within a mapped object when it is between the base address and the `_end` address of that object. If a mapped object fits this criterion, the symbol table made available to the run-time linker is searched to locate the symbol nearest the address specified. The nearest symbol is the one that has a value less than or equal to the required address.

PARAMETERS

The `Dl_info` structure must be preallocated by the user. The structure members are filled by `dladdr()` based on the specified *address*. The `Dl_info` structure includes the following members:

const char ***dli_fname*	Contains a pointer to the filename of the containing object.
void ***dli_fbase*	Contains the base address of the containing object.
const char ***dli_sname*	Contains a pointer to the symbol name nearest to the address specified. This symbol either has the same address or is the nearest symbol with a lower address.
void ***dli_saddr*	Contains the actual address of the above symbol.

RETURN VALUES

If the specified *address* cannot be matched to a mapped object, `0` is returned. Otherwise, a non-zero value is returned and the associated *Dl_info* elements are filled.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `dlclose`(2K), `dlerror`(2K), `dlopen`(2K), `dlsym`(2K)

ChorusOS 4.0 Introduction

NOTES The *Dl_info* pointer elements point to addresses within the mapped objects. These addresses may become invalid if objects are removed before the pointer elements are used (see `dlclose`(2K)). If no symbol is found to describe the specified address, both `dli_sname` and `dli_saddr` members are set to `0`.

NAME

dlclose – close a dynamic object

SYNOPSIS

#include <cx/dlfcn.h>
int **dlclose**(void **handle*);

FEATURES

ACTOR_EXTENDED_MNGT, DYNAMIC_LIB

DESCRIPTION

`dlclose()` is a member of the dynamic linking API, a family of routines that give the user direct access to dynamic linking functionality.

`dlclose()` is *only* available to programs and libraries compiled with `imake` rules of the form `Dynamic`*Type*`Target()`, where *Type* is one of `User`, `Sup`, `CXXUser`, `CXXSup` or `Library`.

See the *ChorusOS 4.0 Introduction* for more information on building executables with dynamic libraries.

`dlclose()` dissociates a dynamic object previously opened by `dlopen()` from the current actor. Once an object has been closed using `dlclose()`, its symbols are no longer available to `dlsym()`. All objects loaded automatically as a result of invoking `dlopen()` on the referenced object are also closed. The handle argument is the value returned by `dlopen()`.

RETURN VALUES

Upon successful completion, `0` is returned. If the object could not be closed, or if *handle* does not refer to an open object, `dlclose()` returns a non-zero value. Detailed diagnostic information is available through `dlerror()`.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dladdr`(2K), `dlerror`(2K), `dlopen`(2K), `dlsym`(2K)

ChorusOS 4.0 Introduction

NOTES

Successful invocation of `dlclose()` does not guarantee that the objects associated with *handle* will actually be removed from the address space of the actor. The same object may be opened multiple times. Objects loaded automatically when the program was started, or as a result of invoking `dlopen()` on another object, may also be loaded by another invocation of `dlopen()`. An object is not removed from the address space until all references to that object through explicit `dlopen()` calls have been closed and all other objects implicitly referencing that object have also been closed.

Once an object has been closed by `dlclose()`, referencing symbols contained in that object can cause undefined behavior.

NAME

dlerror – get diagnostic information

SYNOPSIS

```
#include <cx/dlfcn.h>
char *dlerror(void);
```

FEATURES

ACTOR_EXTENDED_MNGT, DYNAMIC_LIB

DESCRIPTION

`dlerror( )` is a member of the dynamic linking API, a family of routines that give the user direct access to dynamic linking functionality.

`dlerror( )` is *only* available to programs and libraries compiled with `imake` rules of the form `Dynamic`*Type*`Target( )`, where *Type* is one of `User`, `Sup`, `CXXUser`, `CXXSup` or `Library`.

See *ChorusOS 4.0 Introduction* for more information on building executables with dynamic libraries.

`dlerror( )` returns a null-terminated character string with no trailing newline that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of `dlerror( )`, `dlerror( )` returns `NULL`. Therefore, invoking `dlerror( )` twice in a row returns `NULL`.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dladdr`(2K), `dlclose`(2K), `dlopen`(2K), `dlsym`(2K)

Introducint ChorusOS 4.0

NOTES

Messages returned by `dlerror( )` may reside in a static buffer that is overwritten on each call to `dlerror( )`. Application code should not write to this buffer. Programs that need to preserve an error message should make their own copies of the message.

NAME

dlopen – gain access to a dynamic object file

SYNOPSIS

```
#include <cx/dlfcn.h>
void *dlopen(const char *pathname, int mode);
```

FEATURES

ACTOR_EXTENDED_MNGT, DYNAMIC_LIB

DESCRIPTION

`dlopen()` is a member of the dynamic linking API, a family of routines that give the user direct access to dynamic linking functionality.

`dlopen()` is *only* available to programs and libraries compiled with `imake` rules of the form `Dynamic`*Type*`Target()`, where *Type* is one of `User`, `Sup`, `CXXUser`, `CXXSup` or `Library`.

See the *ChorusOS 4.0 Introduction* for more information on building executables with dynamic libraries.

`dlopen()` makes a dynamic object file available to a running actor. `dlopen()` returns a *handle* that an actor may use when calling `dlsym()` and `dlclose()`.

PARAMETERS

The value of the *handle* should not be interpreted in any way by the actor. The *pathname* is the path name of the object to be opened. Any *pathname* containing `/` is interpreted as an absolute path or as a path relative to the current directory. Otherwise, the set of search paths currently in effect for the runtime linker is used to locate the specific file. See NOTES.

Any dependencies recorded within the *pathname* object are also loaded as part of the `dlopen()` call. The dependencies are searched in the order they are loaded to locate any additional dependencies. The search continues until all dependencies of *pathname* are loaded.

If the value of *pathname* is `0`, then `dlopen()` provides a *handle* for the global symbol object. The global symbol object provides access to the symbols from an ordered set of objects consisting of the original program image file and any dependencies loaded at program startup, and any objects loaded using `dlopen()`. Because the set of objects loaded using `dlopen()` can change during actor execution, the set identified by *handle* can also change dynamically.

An object may be opened multiple times by `dlopen()`. As long as the object has not been removed from memory using `dlclose()`, the same *handle* is returned each time `dlopen()` is called. A counter is maintained internallly to record the number of consecutive openings.

The *mode* parameter describes how `dlopen()` operates on *pathname* with regard to relocation processing and scope of symbols from *pathname* and its dependencies. When an object is brought into the address space of an actor, it may contain references to symbols for which addresses are not known until the object is loaded. These types of references must be relocated before the

symbols can be accessed. The *mode* parameter governs how these relocations are processed. The following *mode*s are supported:

RTLD_NOW	Performs all necessary relocations when the object is first loaded.
RTLD_GLOBAL	Makes the object's global symbols available for relocation processing of any other object.
RTLD_LOCAL	Makes the object's global symbols available only for relocation processing of other objects in the same group.

The program image file and any objects loaded at program startup use the RTLD_GLOBAL *mode*. By default, all objects accessed using `dlopen()` use RTLD_LOCAL *mode*. Any local object may depend on more than one group. Any object using RTLD_LOCAL *mode* referenced as a dependency of an object using RTLD_GLOBAL *mode* is promoted to RTLD_GLOBAL, and RTLD_LOCAL is ignored. Any object loaded by `dlopen()` that requires relocation of global symbols can reference the symbols in any RTLD_GLOBAL object, including, at least, the program image file, any objects loaded at program startup, the object itself and any dependencies that the object references. However, the *mode* parameter may also be bitwise-ORed with the following values that affect the scope of symbol availability:

RTLD_GROUP	Makes only symbols from the associated group available for relocation. The group is the object and all dependencies of the object. A group must be completely self-contained. All dependencies between members of the group must satisfy the relocation requirements of each object in the group.
RTLD_WORLD	Makes only symbols from RTLD_GLOBAL objects available for relocation.

The default *mode*s for `dlopen()` are RTLD_WORLD and RTLD_GROUP. Both *mode*s are bitwise-ORed together if the object is required by different dependencies specifying different *mode*s.

RTLD_LAZY (lazy-binding) and RTLD_PARENT are not supported.

The following *mode*s provide additional functionality beyond relocation processing:

RTLD_NODELETE	Does not delete the specified object from the address space as part of `dlclose()`. The opening counter is still decremented, however.

RTLD_NOLOAD	Does not load the specified object as part of the `dlopen()` call, but a valid *handle* is returned if the object already exists in the actor address space. Additional *mode*s can be specified and are bitwise-ORed with the current *mode* of the object and its dependencies. `RTLD_NOLOAD` provides a means of querying the presence, or promoting the *mode*s, of an existing dependency. Note that `RTLD_NOLOAD` does not prevent the opening counter of the object from being incremented. Therefore, using it on an existing object requires an additional `dlclose()` to effectively remove the object from memory.

RETURN VALUES

If the specified *pathname* cannot be found, cannot be opened for reading, is not a relocatable object, or if an error occurs while loading *pathname* or while relocating its symbolic references, `dlopen()` returns `NULL`.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dladdr`(2K), `dlclose`(2K), `dlerror`(2K), `dlsym`(2K)

ChorusOS 4.0 Introduction

NOTES

If other objects are were link-editied with *pathname* when *pathname* was built — that is, if *pathname* has dependencies on other objects — those objects are automatically loaded by `dlopen()`. Unless *pathname* contains `/`, the directory search path used to find both *pathname* and the objects it depends upon may be set using either the environment variable `LD_LIBRARY_PATH`, which is analyzed at actor startup, or the runpath setting within the object calling `dlopen()`. Objects whose names resolve to the same absolute or relative *pathname* may be opened any number of times using `dlopen()`. However, the object referenced is loaded only once into the address space of the current actor.

NAME

dlsym – get the address of a symbol in a dynamic object

SYNOPSIS

```
#include <cx/dlfcn.h>
void *dlsym(void *handle, const char *name);
```

FEATURES

ACTOR_EXTENDED_MNGT, DYNAMIC_LIB

DESCRIPTION

`dlsym( )` is a member of the dynamic linking API, a family of routines that give the user direct access to dynamic linking functionality.

`dlsym( )` is *only* available to programs and libraries compiled with `imake` rules of the form `Dynamic`*Type*`Target( )`, where *Type* is one of `User`, `Sup`, `CXXUser`, `CXXSup` or `Library`.

See the *ChorusOS 4.0 Introduction* for more information on building executables with dynamic libraries.

`dlsym( )` allows an actor to obtain the address of a symbol defined within a dynamic object. The *handle* is either the value returned from a call to `dlopen( )` or one of the special flags `RTLD_NEXT` or `RTLD_DEFAULT`. The *name* is the symbol's name as a character string.

If the *handle* used has been returned by `dlopen( )`, the corresponding dynamic object must not have been closed using `dlclose( )`. `dlsym( )` searches for the named symbol in all dynamic objects loaded automatically as a result of loading the object referenced by *handle*. For details, see `dlopen`(2K).

If the *handle* used is `RTLD_NEXT`, `dlsym( )` searches for the named symbol in the objects that were loaded following the object from which the `dlsym( )` call is being made.

If the *handle* used is `RTLD_DEFAULT`, `dlsym( )` searches for the named symbol, starting with the first object loaded and proceeding through the list of loaded objects until a match is found. This search follows the default model employed to relocate all objects within the actor.

If the *handle* used is either `RTLD_NEXT` or `RTLD_DEFAULT` and if the objects being searched have been loaded using `dlopen( )`, `dlsym( )` searches the object only if the caller is part of the same `dlopen( )` dependency hierarchy, or if the object was given global search access. See `dlopen`(2K) for a discussion of the `RTLD_GLOBAL` mode.

RETURN VALUES

If the specified *handle* does not refer to a valid object opened using `dlopen( )`, is not the special flag `RTLD_NEXT`, or if the named symbol cannot be found within any of the objects associated with *handle*, then `dlsym( )` returns `NULL`. More detailed diagnostic information is available through `dlerror( )`.

EXAMPLES

The following example shows how to use `dlopen( )` and `dlsym( )` to access either function or data objects. For simplicity, error checking has been omitted.

```
void *handle;
int *iptr, (*fptr)(int);
/* open the object needed */
handle = dlopen("/usr/home/me/libfoo.so.1", RTLD_NOW);

/* find the address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);
```

The following example shows how to use `dlsym( )` to check that a particular function is defined and to call it only if it is defined.

```
int (*fptr)();
if ((fptr = (int (*)())dlsym(RTLD_DEFAULT,
    "my_function")) != NULL) {
 (*ftpr)();
}
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dladdr`(2K), `dlclose`(2K), `dlerror`(2K), `dlopen`(2K)

ChorusOS 4.0 Introduction

NAME

ethIpcStackAttach – attach an IPC stack to an Ethernet device

SYNOPSIS

```
#include <ddi/net/netFrame.h>
#include <iom/ipcStackAttach.h>
int ethIpcStackAttach(const *device);
```

FEATURES

IOM_IPC, IPC_REMOTE

DESCRIPTION

If the `IOM_IPC` feature is set to `true`, the `IOM` actor includes an `IPC` stack. `ethIpcStackAttach()` attaches the IPC stack to an Ethernet *device* handled by the `IOM`.

`ethIpcStackAttach()` should be executed only after the corresponding Ethernet *device* is up and running and has been assigned an IP address.

`ethIpcStackAttach()` works by causing the IOM to scan its network its `ifnet` interfaces until it finds the interface attached to the corresponding Ethernet hardware *device*. Once that interface is found, it processes IPC traffic. If *device* is `NULL`, the first `IOM` Ethernet network interface is used.

Once attached, the IPC stack cannot be detached. Furthermore, the IPC stack may only be attached to a single Ethernet hardware *device*.

RETURN VALUES

Upon successful completion, `ethIpcStackAttach()` returns `0`; otherwise it returns `-1` and sets `errno` to indicate the error.

ERRORS

[EACCES] `ethIpcStackAttach()` was called by a user actor.

Only supervisor actors can invoke `ethIpcStackAttach()`.

[ENODEV] No network interface attached to *device* was found.

[EINVAL] The IPC stack could not be correctly initialized.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`C_INIT`(1M)

NAME

ethOsiStackAttach – attach an OSI stack to an Ethernet device

SYNOPSIS

```
#include <ddi/net/netFrame.h>
#include <iom/osiStackAttach.h>
int ethOsiStackAttach(const *device, OsiToEthConf* in, EthToOsiConf* out);
```

FEATURES

IOM_OSI

DESCRIPTION

`ethOsiStackAttach()` attaches an OSI stack that you provide to an Ethernet *device* handled by the `IOM`.

`ethOsiStackAttach()` works by causing the IOM to scan its network its `ifnet` interfaces until it finds the interface attached to the corresponding Ethernet hardware *device*. Once that interface is found, it processes OSI traffic. If *device* is `NULL`, the first `IOM` Ethernet network interface is used.

Once attached, the OSI stack cannot be detached. OSI stacks may be attached to multiple Ethernet hardware *device*s and each *device* may be attached to multiple OSI stacks.

EXTENDED DESCRIPTION

The *in* and *out* parameters are defined as follows:

```
typedef void    (*EthFrameSend)    (void* ethCookie, struct NetFrame* outFrame);
typedef void    (*OsiFrameReceive) (void* osiCookie, struct NetFrame* inFrame);

typedef struct {
        void*           osiCookie;
        OsiFrameReceive osiFrameReceive;
} OsiToEthConf;

typedef struct {
        void*        ethCookie;
        char         etherAddr[6];
        EthFrameSend ethFrameSend;
} EthToOsiConf;
```

The OSI stack provider must provide both (i) a cookie and (ii) an upcall for processing OSI network frames received in the *in* parameter.

It must also return (i) an Ethernet cookie, (ii) the Ethernet address of the Ethernet *device*, and (iii) a function pointer for sending OSI frames in the *out* parameter.

After `ethOsiStackAttach()` succeeds, the `IOM` will invoke the `osiFrameReceive()` function you provide each time it receives an OSI network frame. The first argument is the cookie given before in the *in* parameter passed to `ethOsiStackAttach()`, the second parameter is the frame whose type is defined as follows (<ddi/net/netFrame.h>):

```
typedef struct NetBuf {
        struct NetBuf* next;    /* next buffer in the list or NULL */
```

```
        uint32_f       bufSize; /* size of memory buffer */
        char*          bufAddr; /* starting address of memory buffer */
} NetBuf;

typedef struct NetFrame* NetFramePtr;
typedef void (*NetFrameFree)(NetFramePtr);

typedef struct NetFrame {
        struct NetFrame* next;      /* to build list of NetFrames */
        uint32_f         frameSize; /* total length of frame */
        NetBuf*          bufList;   /* list of memory buffers holding the data */
        NetFrameFree     freeFrame; /* the free function of the frame */
} NetFrame;
```

The OSI stack is given ownership of the `NetFrame`. Once done with it, it must invoke the `freeFrame()` function to free it.

The `osiFrameReceive()` function is called at interrupt level. It it highly recommended that you optimize it in order to process the frame as fast as possible.

RETURN VALUES

Upon successful completion, `ethOsiStackAttach()` returns `0`; otherwise it returns `-1` and sets `errno` to indicate the error.

ERRORS

[EACCES] `ethOsiStackAttach()` was called by a user actor.

Only supervisor actors can invoke `ethOsiStackAttach()`.

[ENODEV] No network interface attached to *device* was found.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

eventInit, eventClear, eventPost, eventWait – initialize an event flag set; Clear events in an event flag set; Post events in an event flag set; Wait for events in an event flag set

SYNOPSIS

```
#include <sync/chEvent.h>
int eventInit(KnEventSet * eventSet);

int eventClear(KnEventSet * eventSet, unsigned int mask);

int eventPost(KnEventSet * eventSet, unsigned int mask);

int eventWait(KnEventSet * eventSet, unsigned int inMask, unsigned int option, unsigned
int * outMask, KnTimeVal * waitLimit);
```

FEATURES

EVENT

DESCRIPTION

Event flag sets are *KnEventSet* structures allocated in user memory. An event flag set is a set of bits in memory associated with a thread wait queue. Each bit is associated with one event. The set is implemented as an unsigned integer; the maximum number of flags in a set is *8*sizeof(int)* . Within a set, each event flag is designated by an integer number *([0 .. 8*sizeof(int)-1])* .

When a flag is set, it is said to be POSTED, and the associated event is considered to have occurred. Otherwise, the flag is said to be UNPOSTED, and the associated event has not yet occurred. Both threads and interrupt handlers can use event flag sets for signaling purposes.

The *eventInit* call initializes the event flag set whose address is *eventSet*. All events are initialized in the UNPOSTED state.

The *eventPost* call posts (sets to POSTED) one or more event flags within *eventSet*. Any thread waiting for posted events will be awakened, and subsequently *eventWait* calls referring to the events posted will return immediately. The event flags remain in the posted state until explicitly cleared using *eventClear*. The *mask* field specifies the mask of posted events (event number *i* is posted if bit *i* of *mask* is set). You can call *eventPost* within an interrupt handler, or with preemption disabled.

The *eventWait* call makes the current thread wait conditionally for one or more events on the event flag set *eventSet*.

The *inMask* field specifies the mask of awaited events (event number { *i* } is awaited if the bit *i* of *inMask* is set). If *inMask* is zero (no awaited flag), *eventWait* returns immediately, regardless the value of *waitLimit* . This can be used to obtain the current state of the event flag set.

The *option* field can be:

K_EVENT_OR	The thread waits for an OR condition to be satisfied by the events indicated in *inMask;* it is sufficient that one of the events be posted for the thread to be awakened.
K_EVENT_AND	The thread waits for an AND condition to be satified by the events indicated in *inMask* ; all the events in the mask must be posted for the thread to be awakened.

Posted events remain posted after *eventWait* returns, until cleared explicitly using *eventClear.* If a subsequent *eventWait* with the same *inMask* is launched before clearing, it will return immediately with the same status as the previous call.

The *outMask* field shows the state of the event flag set when the thread was awakened.

If the thread event-set state does not satisfy the conditions described in *inMask* and *option,* the thread is blocked according to the options described in *waitLimit* in *intro(K).* The *waitLimit* pointer indicates a *KnTimeVal* structure which contains a timeout interval as described in *sysTime(K).*

The *eventClear* call clears one or more event flags within the event flag set *eventSet.* The *mask* field shows the mask of cleared events (event number *i* is cleared if the bit *i* of *mask* is set).

RESTRICTIONS

A user application and a supervisor application may not share an event.

However, two user applications may share an event by mapping it in both user address spaces.

RETURN VALUE

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

ERRORS

[K_EABORT]	*eventWait* has been aborted.
[K_EFAULT]	*eventSet* or *outMask* points outside the current actor's address space.
[K_EINVAL]	The *eventSet* structure has not been correctly initialized, or the *waitLimit* is an invalid *KnTimeVal.*
[K_ETIMEOUT]	The time- out occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

eventInit, eventClear, eventPost, eventWait – initialize an event flag set; Clear events in an event flag set; Post events in an event flag set; Wait for events in an event flag set

SYNOPSIS

```
#include <sync/chEvent.h>
int eventInit(KnEventSet * eventSet);

int eventClear(KnEventSet * eventSet, unsigned int mask);

int eventPost(KnEventSet * eventSet, unsigned int mask);

int eventWait(KnEventSet * eventSet, unsigned int inMask, unsigned int option, unsigned
int * outMask, KnTimeVal * waitLimit);
```

FEATURES

EVENT

DESCRIPTION

Event flag sets are *KnEventSet* structures allocated in user memory. An event flag set is a set of bits in memory associated with a thread wait queue. Each bit is associated with one event. The set is implemented as an unsigned integer; the maximum number of flags in a set is *8*sizeof(int)* . Within a set, each event flag is designated by an integer number *([0 .. 8*sizeof(int)-1])* .

When a flag is set, it is said to be POSTED, and the associated event is considered to have occurred. Otherwise, the flag is said to be UNPOSTED, and the associated event has not yet occurred. Both threads and interrupt handlers can use event flag sets for signaling purposes.

The *eventInit* call initializes the event flag set whose address is *eventSet*. All events are initialized in the UNPOSTED state.

The *eventPost* call posts (sets to POSTED) one or more event flags within *eventSet*. Any thread waiting for posted events will be awakened, and subsequently *eventWait* calls referring to the events posted will return immediately. The event flags remain in the posted state until explicitly cleared using *eventClear*. The *mask* field specifies the mask of posted events (event number *i* is posted if bit *i* of *mask* is set). You can call *eventPost* within an interrupt handler, or with preemption disabled.

The *eventWait* call makes the current thread wait conditionally for one or more events on the event flag set *eventSet*.

The *inMask* field specifies the mask of awaited events (event number { *i* } is awaited if the bit *i* of *inMask* is set). If *inMask* is zero (no awaited flag), *eventWait* returns immediately, regardless the value of *waitLimit* . This can be used to obtain the current state of the event flag set.

The *option* field can be:

K_EVENT_OR	The thread waits for an OR condition to be satisfied by the events indicated in *inMask;* it is sufficient that one of the events be posted for the thread to be awakened.
K_EVENT_AND	The thread waits for an AND condition to be satified by the events indicated in *inMask* ; all the events in the mask must be posted for the thread to be awakened.

Posted events remain posted after *eventWait* returns, until cleared explicitly using *eventClear.* If a subsequent *eventWait* with the same *inMask* is launched before clearing, it will return immediately with the same status as the previous call.

The *outMask* field shows the state of the event flag set when the thread was awakened.

If the thread event-set state does not satisfy the conditions described in *inMask* and *option,* the thread is blocked according to the options described in *waitLimit* in *intro(K).* The *waitLimit* pointer indicates a *KnTimeVal* structure which contains a timeout interval as described in *sysTime(K).*

The *eventClear* call clears one or more event flags within the event flag set *eventSet.* The *mask* field shows the mask of cleared events (event number *i* is cleared if the bit *i* of *mask* is set).

RESTRICTIONS

A user application and a supervisor application may not share an event.

However, two user applications may share an event by mapping it in both user address spaces.

RETURN VALUE

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

ERRORS

[K_EABORT]	*eventWait* has been aborted.
[K_EFAULT]	*eventSet* or *outMask* points outside the current actor's address space.
[K_EINVAL]	The *eventSet* structure has not been correctly initialized, or the *waitLimit* is an invalid *KnTimeVal.*
[K_ETIMEOUT]	The time- out occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

eventInit, eventClear, eventPost, eventWait – initialize an event flag set; Clear events in an event flag set; Post events in an event flag set; Wait for events in an event flag set

SYNOPSIS

```
#include <sync/chEvent.h>
int eventInit(KnEventSet * eventSet);

int eventClear(KnEventSet * eventSet, unsigned int mask);

int eventPost(KnEventSet * eventSet, unsigned int mask);

int eventWait(KnEventSet * eventSet, unsigned int inMask, unsigned int option, unsigned
int * outMask, KnTimeVal * waitLimit);
```

FEATURES

EVENT

DESCRIPTION

Event flag sets are *KnEventSet* structures allocated in user memory. An event flag set is a set of bits in memory associated with a thread wait queue. Each bit is associated with one event. The set is implemented as an unsigned integer; the maximum number of flags in a set is *8*sizeof(int)* . Within a set, each event flag is designated by an integer number *([0 .. 8*sizeof(int)-1])* .

When a flag is set, it is said to be POSTED, and the associated event is considered to have occurred. Otherwise, the flag is said to be UNPOSTED, and the associated event has not yet occurred. Both threads and interrupt handlers can use event flag sets for signaling purposes.

The *eventInit* call initializes the event flag set whose address is *eventSet*. All events are initialized in the UNPOSTED state.

The *eventPost* call posts (sets to POSTED) one or more event flags within *eventSet*. Any thread waiting for posted events will be awakened, and subsequently *eventWait* calls referring to the events posted will return immediately. The event flags remain in the posted state until explicitly cleared using *eventClear*. The *mask* field specifies the mask of posted events (event number *i* is posted if bit *i* of *mask* is set). You can call *eventPost* within an interrupt handler, or with preemption disabled.

The *eventWait* call makes the current thread wait conditionally for one or more events on the event flag set *eventSet*.

The *inMask* field specifies the mask of awaited events (event number { *i* } is awaited if the bit *i* of *inMask* is set). If *inMask* is zero (no awaited flag), *eventWait* returns immediately, regardless the value of *waitLimit* . This can be used to obtain the current state of the event flag set.

The *option* field can be:

K_EVENT_OR	The thread waits for an OR condition to be satisfied by the events indicated in *inMask;* it is sufficient that one of the events be posted for the thread to be awakened.
K_EVENT_AND	The thread waits for an AND condition to be satified by the events indicated in *inMask* ; all the events in the mask must be posted for the thread to be awakened.

Posted events remain posted after *eventWait* returns, until cleared explicitly using *eventClear.* If a subsequent *eventWait* with the same *inMask* is launched before clearing, it will return immediately with the same status as the previous call.

The *outMask* field shows the state of the event flag set when the thread was awakened.

If the thread event-set state does not satisfy the conditions described in *inMask* and *option,* the thread is blocked according to the options described in *waitLimit* in *intro(K).* The *waitLimit* pointer indicates a *KnTimeVal* structure which contains a timeout interval as described in *sysTime(K).*

The *eventClear* call clears one or more event flags within the event flag set *eventSet.* The *mask* field shows the mask of cleared events (event number *i* is cleared if the bit *i* of *mask* is set).

RESTRICTIONS

A user application and a supervisor application may not share an event.

However, two user applications may share an event by mapping it in both user address spaces.

RETURN VALUE

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

ERRORS

[K_EABORT]	*eventWait* has been aborted.
[K_EFAULT]	*eventSet* or *outMask* points outside the current actor's address space.
[K_EINVAL]	The *eventSet* structure has not been correctly initialized, or the *waitLimit* is an invalid *KnTimeVal.*
[K_ETIMEOUT]	The time- out occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

eventInit, eventClear, eventPost, eventWait – initialize an event flag set; Clear events in an event flag set; Post events in an event flag set; Wait for events in an event flag set

SYNOPSIS

```
#include <sync/chEvent.h>
int eventInit(KnEventSet * eventSet);

int eventClear(KnEventSet * eventSet, unsigned int mask);

int eventPost(KnEventSet * eventSet, unsigned int mask);

int eventWait(KnEventSet * eventSet, unsigned int inMask, unsigned int option, unsigned
int * outMask, KnTimeVal * waitLimit);
```

FEATURES

EVENT

DESCRIPTION

Event flag sets are *KnEventSet* structures allocated in user memory. An event flag set is a set of bits in memory associated with a thread wait queue. Each bit is associated with one event. The set is implemented as an unsigned integer; the maximum number of flags in a set is *8*sizeof(int)* . Within a set, each event flag is designated by an integer number *([0 .. 8*sizeof(int)-1])* .

When a flag is set, it is said to be POSTED, and the associated event is considered to have occurred. Otherwise, the flag is said to be UNPOSTED, and the associated event has not yet occurred. Both threads and interrupt handlers can use event flag sets for signaling purposes.

The *eventInit* call initializes the event flag set whose address is *eventSet*. All events are initialized in the UNPOSTED state.

The *eventPost* call posts (sets to POSTED) one or more event flags within *eventSet*. Any thread waiting for posted events will be awakened, and subsequently *eventWait* calls referring to the events posted will return immediately. The event flags remain in the posted state until explicitly cleared using *eventClear*. The *mask* field specifies the mask of posted events (event number *i* is posted if bit *i* of *mask* is set). You can call *eventPost* within an interrupt handler, or with preemption disabled.

The *eventWait* call makes the current thread wait conditionally for one or more events on the event flag set *eventSet*.

The *inMask* field specifies the mask of awaited events (event number { *i* } is awaited if the bit *i* of *inMask* is set). If *inMask* is zero (no awaited flag), *eventWait* returns immediately, regardless the value of *waitLimit* . This can be used to obtain the current state of the event flag set.

The *option* field can be:

K_EVENT_OR	The thread waits for an OR condition to be satisfied by the events indicated in *inMask;* it is sufficient that one of the events be posted for the thread to be awakened.
K_EVENT_AND	The thread waits for an AND condition to be satified by the events indicated in *inMask* ; all the events in the mask must be posted for the thread to be awakened.

Posted events remain posted after *eventWait* returns, until cleared explicitly using *eventClear.* If a subsequent *eventWait* with the same *inMask* is launched before clearing, it will return immediately with the same status as the previous call.

The *outMask* field shows the state of the event flag set when the thread was awakened.

If the thread event-set state does not satisfy the conditions described in *inMask* and *option,* the thread is blocked according to the options described in *waitLimit* in *intro(K).* The *waitLimit* pointer indicates a *KnTimeVal* structure which contains a timeout interval as described in *sysTime(K).*

The *eventClear* call clears one or more event flags within the event flag set *eventSet.* The *mask* field shows the mask of cleared events (event number *i* is cleared if the bit *i* of *mask* is set).

RESTRICTIONS

A user application and a supervisor application may not share an event.

However, two user applications may share an event by mapping it in both user address spaces.

RETURN VALUE

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

ERRORS

[K_EABORT]	*eventWait* has been aborted.
[K_EFAULT]	*eventSet* or *outMask* points outside the current actor's address space.
[K_EINVAL]	The *eventSet* structure has not been correctly initialized, or the *waitLimit* is an invalid *KnTimeVal.*
[K_ETIMEOUT]	The time- out occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

_exit – terminate a c_actor

SYNOPSIS

```
#include <unistd.h>
void _exit(int status);
```

FEATURES

ACTOR_EXTENDED_MNGT

DESCRIPTION

The *_exit* call terminates the calling c_actor with the following consequences:

- All open file descriptors in the calling c_actor are closed.
- If other c_actors are executing an *await*(2K) or an *awaits*(2K) system call, they are notified of the calling c_actor's termination.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`exit`(3STDC), `awaits`(2K)

NAME

grpAllocate – allocate a port group capability

SYNOPSIS

```
#include <ipc/chIpc.h>
int grpAllocate(int options, KnCap *groupcap, int stamp);
```

FEATURES

IPC

DESCRIPTION

grpAllocate allocates a port group capability.

groupcap is the returned capability for the port group, and is a pointer to a *KnCap* structure the members of which are the following:

```
KnUniqueId ui ;    /* entity name */
KnKey      key ;   /* modification key */
```

ui is the unique identifier of the port group, used to send messages to the group (see *ipcSend*(2K) and *ipcTarget*(2K)). *key* is the key needed for modification of the group (i.e. inserting or removing ports by *grpPortInsert*(2K) and *grpPortRemove*(2K)).

options may have one of the following values, which describe the way in which the port group capability is to be allocated:

`K_DYNAMIC` The call will return a new unique capability for a port group. *stamp* is ignored.

`K_STATUSER` The group is a static user port group. The call will not return a new group capability, but a capability of one of the pre-allocated user port group capabilities. The right group is identified by *stamp* which may vary between 0 and 0xffffffff.

`K_STATSYS` The group is a static system port group. The call will not return a new group name but rather the name of one of the pre-allocated system port group capabilities. The right group is identified by *stamp* which may vary between 0 and 0xffffffff. This option is only allowed for threads of *SYSTEM* actors or *SUPERVISOR* threads.

Static and dynamic groups only differ by the way their name is allocated. They are used (by *ipcSend*(2K) or *ipcCall*(2K)) and managed (by *grpPortInsert*(2K) or *grpPortRemove*(2K)) in exactly the same way. Static groups are used in order to ease applications binding. Their main property resides in the fact that *grpAllocate* will always return the same capability for a given value of *stamp*. The different actors of an application independently invoke *grpAllocate*. After this operation, some of them insert ports in the group. The others simply send messages to the group. Dynamic groups are used when applications need new groups with

unique names. Only one actor of the application invokes *grpAllocate*. It needs to transmit the group UI to other application actors to allow them to use the group.

RETURN VALUE

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

ERRORS

[K_EPRIV]	The K_STATSYS option is not allowed to the current thread.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`grpPortInsert`(2K), `ipcSend`(2K), `ipcTarget`(2K), `grpPortRemove`(2K)

NAME

grpPortInsert, grpPortRemove – insert a port into a port group; remove a port from a port group

SYNOPSIS

```
#include <ipc/chIpc.h>
int grpPortInsert(KnCap * groupcap, KnUniqueId * portui);

int grpPortRemove(KnCap * groupcap, KnUniqueId * portui);
```

FEATURES

IPC

DESCRIPTION

grpPortInsert inserts the port, the name of which is given by *portui* into the port group, the capability of which is pointed to by *groupcap*.

grpPortRemove removes the port, the name of which is given by *portui* from the port group, the capability of which is pointed to by *groupcap*.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a negative error code is returned. *grpPortInsert* returns 0 if the port was already in the port group.

ERRORS

[K_EINVAL]	*groupcap* is an invalid port group capability, or there was an attempt to remove a port from a port group to which it did not belong.
[K_EUNKNOWN]	*portui* is an invalid port name.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_ENOMEM]	The system is out of resources.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor, and attempts to insert a port or remove a port in a system static group.

RESTRICTIONS

The port and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`grpAllocate`(2K)

NAME

grpPortInsert, grpPortRemove – insert a port into a port group; remove a port from a port group

SYNOPSIS

```
#include <ipc/chIpc.h>
int grpPortInsert(KnCap * groupcap, KnUniqueId * portui);

int grpPortRemove(KnCap * groupcap, KnUniqueId * portui);
```

FEATURES

IPC

DESCRIPTION

grpPortInsert inserts the port, the name of which is given by *portui* into the port group, the capability of which is pointed to by *groupcap*.

grpPortRemove removes the port, the name of which is given by *portui* from the port group, the capability of which is pointed to by *groupcap*.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a negative error code is returned. *grpPortInsert* returns 0 if the port was already in the port group.

ERRORS

[K_EINVAL]	*groupcap* is an invalid port group capability, or there was an attempt to remove a port from a port group to which it did not belong.
[K_EUNKNOWN]	*portui* is an invalid port name.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_ENOMEM]	The system is out of resources.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor, and attempts to insert a port or remove a port in a system static group.

RESTRICTIONS

The port and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`grpAllocate`(2K)

NAME

ipcCall – send an RPC request and wait for the reply

SYNOPSIS

#include <ipc/chIpc.h>
int **ipcCall**(KnMsgDesc *_reqmsg_, int _reqsrc_, KnIpcDest *_reqdest_, KnMsgDesc *_repmsg_, int _delay_);

FEATURES

IPC

DESCRIPTION

The *ipcCall* function sends a message in transactional (*RPC*) mode, and blocks the caller until a reply to the request is received or the optional *delay* expires.

The *delay* field is the maximum waiting time, expressed in milliseconds. If *delay* has a negative value, the blocking time is unlimited.

The *reqdest* field defines the destination of the message, and is a pointer to a *KnIpcDest* structure. Its meaning is described in *ipcSend*(2K). Use of the K_BROADMODE addressing mode on a port group is forbidden.

The *reqsrc* field is a local identifier for the source port of the message. If *reqsrc* is K_DEFAULTPORT, the default port of the current actor is used.

The *reqmsg* field points to a descriptor for the request message to be sent, on condition that *reqmsg* points to a *KnMsgDesc* structure, as described in *ipcSend*(2K).

The *repmsg* field is a descriptor for the reply message to be received, as described in *ipcReceive*(2K). As in *ipcReceive*(2K), the K_ABORTABLE flag may be present in the *flags* field of this structure. In this case, the *ipcCall* is *ABORTABLE*. Otherwise, it is *NONABORTABLE*. These two different states determine the behavior of the thread when a *threadAbort*(2K) is applied to it. If *threadAbort(2K)* is applied to a *NONABORTABLE ipcCall*, the thread replies and stays in the *ABORTED* state (see *threadAbort*(2K)). If *threadAbort(2K)* is applied to an *ABORTABLE ipcCall*, the kernel attempts to pass the abortion on to the thread which is processing the request messages:

- If the request message is queued behind the destination port, the message is deleted and the *ipcCall* returns K_EABORT.
- If a thread (called the *server*) currently is processing the request message, the *server* is aborted.

This propagation mechanism is recursive: in the case of nested *ipcCall* (2K), (the *server* is itself blocked in an *ABORTABLE ipcCall*), the abortion of the *server* is also passed on to its *server*.

Unless explicitly shown by the return of the K_EABORT error code, a *client* thread aborted during its *ipcCall* always enters the *ABORTED* state upon return (the abortion still has to be processed; it appears as if the abortion occurred after the *ipcCall* returned).

Upon return, the reply received does not become the current message for the current thread. As a consequence, *ipcReply*(2K), *ipcSave*(2K), *ipcGetData*(2K) and *ipcSysInfo*(2K) may never be applied to the reply of an *ipcCall*.

RETURN VALUE

Upon successful completion, the size of the message body received is returned. Otherwise, a negative error code is returned. The error codes are those listed below, except when the destination port is connected to a message handler. In this case the error code is the return value of the handler function.

ERRORS

[K_EABORT]	The thread was aborted while waiting; the request message was never delivered to its destination.
[K_EBADMODE]	Attempt to broadcast an *RPC* request.
[K_EFAIL]	The *ipcCall* transaction has failed.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_EFULL]	The destination port is local and its queue is full, or the local remote-communication subsystem is saturated.
[K_EINVAL]	*reqsrc* is not a valid local port identifier.
[K_ETIMEOUT]	The timeout has occurred.
[K_ETOOMUCH]	The request message body is too big.
[K_EUNKNOWN]	Unreachable destination. When the destination port is connected to a message handler, the error code is the return value of the handler function.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ipcSend`(2K), `ipcReply`(2K), `ipcReceive`(2K), `ipcTarget`(2K), `threadAbort`(2K), `svMsgHandler`(2K)

NAME

ipcGetData – get the current message body

SYNOPSIS

```
#include <ipc/chIpc.h>
int ipcGetData(KnMsgDesc *msg);
```

FEATURES

IPC

DESCRIPTION

The *ipcGetData* call delivers the body size of the current thread's current message to the thread's address space.

The message body may only be obtained if the message was received with a NULL *bodyAddr* argument (see *ipcReceive*(2K)). The message body may only be obtained once; system buffers are freed after the operation.

The *msg* field points to a *KnMsgDesc* structure, as described in *ipcReceive*(2K).

This function returns the real body size delivered.

RETURN VALUE

Upon successful completion, the body size delivered is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	There is no current message (no message has been received by the current thread since the last *ipcReply*(2K) or *ipcSave*(2K)).
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

ipcReceive(2K)

NAME

ipcReceive – receive a message

SYNOPSIS

#include <ipc/chIpc.h>
int **ipcReceive**(KnMsgDesc **msg*, int **portli*, int *delay*);

FEATURES

IPC

DESCRIPTION

ipcReceive blocks the caller until a message (sent by *ipcSend*(2K), or *ipcReply*(2K), or *ipcCall*(2K)) is received on the port(s) specified by *portli* or the optional *delay* expires.

delay is the blocking waiting time, expressed in milliseconds. A delay of 0 means a non blocking attempt. If *delay* has a negative value, the blocking time is unbound.

At call time, *portli* is a pointer to the local identifier of the port on which a message is expected. If *portli* points to the value K_DEFAULTPORT, the default port of the current actor is used.

If *portli* points to the value K_ANYENABLED, a message is expected on any of the enabled ports owned by the current actor (see *portEnable*(2K)). In that case, the value pointed by *portli* is set, at return-time, to the local identifier of the port on which a message has been received. If more than one enabled port are holding messages at the time of the call, a message is received on the enabled port with the highest priority.

msg points to a message descriptor for the expected message. *msg* is a pointer to a *KnMsgDesc* structure whose members are the following:

```
unsigned int   flags ;      /* message structure definition */
unsigned int   bodySize ;   /* body size */
VmAddr         bodyAddr ;   /* body address */
VmAddr         annexAddr ;  /* fixed size annex address */
KnEvtNum       seqNum ;     /* sequence number */
```

The *annexAddr* member of the message descriptor gives the starting address at which the system must copy the message annex in the receiver's address space. If this member is set to NULL, the annex is not delivered. Otherwise, K_CMSGANNEXSIZE bytes are copied by the system in the area starting at *annexAddr* in the receiver's address space. If the message was not sent with an annex (see *ipcSend*(2K)), this area is not affected.

The *bodySize* and *bodyAddr* members of the message descriptor respectively give the size of the expected message body and the starting address at which the system must deliver it in the receiver's address space. If the expected size is smaller than the real message size, only the expected amount is received and the remainder is discarded. If *bodyAddr* is aligned on a page boundary, and if *bodySize* is a multiple of the page size (see *vmPageSize*(2K)), the body will be delivered without copy.

If *bodyAddr* is NULL, and if the message contains a body, the message body is not delivered to the receiver address space but rather is kept in system buffers. If the body was not delivered at receive time (NULL *bodyAddr)*, it can be delivered by an invocation of *ipcGetData*(2K). This allows the receiver to fix the message body location in its address space only when the message body size is known.

The *seqNum* member of the message descriptor is an output value indicating the count of messages that has been received on the port, before and including the delivered message. This member is only updated when the *ipcReceive* return value is positive or null.

This call invalidates the current thread's previous current message. If the *ipcReceive* is successful, the received message becomes the current thread's current message, on which *ipcReply*(2K), *ipcSysInfo*(2K), *ipcGetData*(2K) or *ipcSave*(2K) may be applied.

The *flags* field of the message descriptor is a logical combination of the following options:

K_ABORTABLE If this flag is set, *ipcReceive* is *ABORTABLE*. Otherwise, it is *NONABORTABLE*. These two different states determine the behavior of the thread when a *threadAbort*(2K) is applied to it. See *threadAbort*(2K) for a description of this behavior in the two different cases.

K_USERBODY If the caller thread is a *SUPERVISOR* thread, this flag indicates that the message body destination address is part of the current user address space. If this flag is not set while the caller is a *SUPERVISOR* thread, the message body destination is assumed to be part of the kernel address space. This flag is only to be used by trap handling routines, when a received message body has to be directly received in the user address space, without copying it into the kernel address space. If the caller thread is not a *SUPERVISOR* thread, this flag is ignored (the message body destination is always assumed to be part of the user address space).

K_USERANNEX This flag has exactly the same meaning as the K_USERBODY, but concerns the message annex instead of the message body.

If *flag* is equal to 0, none of the previous options is selected: the blocking is *NONABORTABLE*, and message and annex are assumed to be part of the user (kernel) address space when the caller is a *USER* (*SUPERVISOR*) thread.

RETURN VALUE

Upon successful completion:

- If *bodyAddr* was not NULL, the size in bytes of the delivered message body. If the system encountered a memory access fault while copying the message body to the receiver address space, the returned value is set to the successfully delivered body size.
- If *bodyAddr* was NULL, the size in bytes of the message body.

Otherwise, a negative error code is returned.

ERRORS

[K_ENOPORT]	No port corresponds to *portli*.
[K_EINVAL]	A handler is attached to the port.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_ETIMEOUT]	The time out occurred.
[K_EABORT]	The thread has been aborted while waiting.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ipcSend`(2K), `ipcReply`(2K), `ipcCall`(2K), `ipcSave`(2K), `ipcSysInfo`(2K), `ipcGetData`(2K), `threadAbort`(2K), `svMsgHandler`(2K)

NAME

ipcReply – reply to the current message

SYNOPSIS

```
#include <ipc/chIpc.h>
int ipcReply(KnMsgDesc *msg);
```

FEATURES

IPC

DESCRIPTION

The *ipcReply* call replies to the sender of the current thread's current message.

The *msg* field points to a descriptor for the destination of the message (a *KnMsgDesc* structure, as described in *ipcSend*(2K)).

The message is sent asynchronously; the sender is only blocked during the time needed by the system to process the request.

After the reply, if the call is successful, the current thread will not have a current message.

RETURN VALUE

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

ERRORS

[K_EFULL]	The destination port is local and its queue is full, or the local remote-communication subsystem is saturated.
[K_EINVAL]	There is no current message (no message has been received by the current thread since the last *ipcReply*(2K), or *ipcSave*(2K)).
[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_ETOOMUCH]	The message body is too big.
[K_EUNKNOWN]	Unreachable destination.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ipcSend`(2K), `ipcReceive`(2K), `ipcCall`(2K), `ipcGetData`(2K)

NAME

ipcSave, ipcRestore – Save the current message; Restore a saved message as the current message

SYNOPSIS

#include <ipc/chIpc.h>
int **ipcSave**(int *portli*);

int **ipcRestore**(int *msgid*);

FEATURES

IPC

DESCRIPTION

The *ipcSave* call saves the current thread's current message to a system buffer, and returns an identifier for the saved message. The *portli* argument reserved for future use. The caller should use the K_NONEPORT value for this parameter.

Messages used to be saved when a thread receives several messages and has to reply in a different order (*ipcReply* (2K) is always applied to the current message; receiving a message supplants the current message). After *ipcSave*, there is no current message, and *ipcReply* (2K), *ipcGetData* (2K) or *ipcSysInfo* (2K) cannot be used unless a message is received *(ipcReceive* (2K)) or restored *(ipcRestore* (2K)).

The *ipcRestore* call restores a saved message as the current message. If the call is successful, the previous current message is replaced by the current message .

The number of messages that an actor may save is limited to K_CMSGSAVEDMAX.

RETURN VALUE

Upon successful completion, *ipcRestore* returns 0 and *ipcSave* returns a positive identifier for the saved message. Otherwise, a negative error code is returned. In case of failure, the current message is not erased.

ERRORS

[K_ENOMEM]	*ipcSave* : K_CMSGSAVEDMAX messages have already been saved by the calling actor.
[K_EINVAL]	*ipcSave* : no current message; *ipcRestore* : invalid identifier.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ipcReceive`(2K)

NAME

ipcSave, ipcRestore – Save the current message; Restore a saved message as the current message

SYNOPSIS

```
#include <ipc/chIpc.h>
int ipcSave(int portli);

int ipcRestore(int msgid);
```

FEATURES

IPC

DESCRIPTION

The *ipcSave* call saves the current thread's current message to a system buffer, and returns an identifier for the saved message. The *portli* argument reserved for future use. The caller should use the K_NONEPORT value for this parameter.

Messages used to be saved when a thread receives several messages and has to reply in a different order (*ipcReply* (2K) is always applied to the current message; receiving a message supplants the current message). After *ipcSave*, there is no current message, and *ipcReply* (2K), *ipcGetData* (2K) or *ipcSysInfo* (2K) cannot be used unless a message is received *(ipcReceive* (2K)) or restored *(ipcRestore* (2K)).

The *ipcRestore* call restores a saved message as the current message. If the call is successful, the previous current message is replaced by the current message .

The number of messages that an actor may save is limited to K_CMSGSAVEDMAX.

RETURN VALUE

Upon successful completion, *ipcRestore* returns 0 and *ipcSave* returns a positive identifier for the saved message. Otherwise, a negative error code is returned. In case of failure, the current message is not erased.

ERRORS

[K_ENOMEM]	*ipcSave* : K_CMSGSAVEDMAX messages have already been saved by the calling actor.
[K_EINVAL]	*ipcSave* : no current message; *ipcRestore* : invalid identifier.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ipcReceive`(2K)

NAME

ipcSend – send a message

SYNOPSIS

#include <ipc/chIpc.h>
int **ipcSend**(KnMsgDesc *_msg_, int _msgsrc_, KnIpcDest *_msgdest_);

FEATURES

IPC

DESCRIPTION

ipcSend sends an asynchronous message. *msgdest* defines the destination of the message. It is a pointer to a *KnIpcDest* structure the members of which are the following:

```
KnUniqueId       target ;          /* port or group */
KnUniqueId       coTarget ;        /* site qualifier */
```

If *target* is a port name, the message will be sent to that port. *target* may also be a port group name in which an addressing mode has been set using *ipcTarget*(2K). If the addressing mode is K_BROADMODE, the message will be sent to each reachable member of the group (i.e. ports which have been inserted in the group - see *grpPortInsert*(2K)). If the addressing mode is K_FUNCMODE, the message will be sent to one of the reachable members of the group. If the addressing mode is K_FUNCUMODE, the message will be sent to one of the reachable members of the group which reside on the site denoted by *coTarget*. *coTarget* is a unique identifier. This addressing mode is called *associative functional mode*. A site is denoted by such a unique identifier, if this unique identifier has been declared on the site. For example, the names of the ports owned by the actors located on the site and the pre-defined site unique identifiers (see *uiSite*(2K)) are identifiers which are automatically registered on a site. If the addressing mode is K_FUNCXMODE, the message will be sent to one of the reachable members of the group, assuming that the UI of the selected member is different from the UI given by *coTarget*. This addressing mode is called *exclusive functional mode*.

msgsrc is the local identifier of a port owned by the sender, and specifies the source port of the message. If one of the receivers of the message performs a *ipcReply*(2K) on it, the reply will be sent to that port. The unique identifier of this port is given to a receiver which performs a *ipcSysInfo*(2K), as the name of the source port of the message. If *msgsrc* is K_DEFAULTPORT, the current actor default port is used.

msg points to a descriptor for the message to be sent, and is a pointer to a *KnMsgDesc* structure, the members of which are the following:

```
unsigned int     flags ;      /* message structure definition */
unsigned int     bodySize ;   /* body size */
VmAddr           bodyAddr ;   /* body address */
VmAddr           annexAddr ;  /* fixed size annex address */
KnEvtNum         seqNum ;     /* not used */
```

The message data is composed of a `message body`, a byte string of variable size (limited to K_CMSGSIZEMAX), to which a `message annex`, a small fixed sized (K_CMSGANNEXSIZE) byte string, might be associated. The body transmission may benefit from the (virtual) memory management to avoid physical copies (see below). When present, the annex is always physically copied from the sender address space to the receiver address space.

The *bodySize* and *bodyAddr* members of the message descriptor respectively give the size of the message body and its starting address in the sender address space. *bodySize* is limited to K_CMSGSIZEMAX. If *bodySize* is set to 0, the message contains no body data.

The *annexAddr* member of the message descriptor gives the starting address of the message annex in the sender address space. If this field is NULL, no annex is sent with the message body.

The *flags* field of the message descriptor may be a combination of the following options, which describe the way in which the message body is to be transmitted:

K_MOVE (for virtual memory management only) This option indicates that the sender will not retain the contents of the message body in its address space after the send operation. In that case, the system will try to transfer the message body without local copy, by transferring only the page descriptors. After the send, the virtual addresses corresponding to the message body are still valid, but their contents are undefined.

If the K_MOVE option is not set, the system will copy the message body.

K_MOVEAL (for virtual memory management only) This option has the same effect as the K_MOVE option, except that even if the message body end is not aligned on a page boundary (see *vmPageSize*(2K)), the last page partially covered by the message may be moved by the system. That means that the user must be ready to lose the contents of its address space from the beginning of the message body to the next page boundary following the end of the message body.

K_USERBODY If the caller thread is a *SUPERVISOR* thread, this flag indicates that the message body is part of the current user address space. If this flag is not set while the caller is a *SUPERVISOR* thread, the message body is assumed to be part of the kernel address space. This flag is only to be used by trap handling routines, when data located in the user address space have to be directly used as a message

body, without copying them into the kernel address space. If the caller thread is not a *SUPERVISOR* thread, this flag is ignored (the message body is always assumed to be part of the user address space).

K_USERANNEX This flag has exactly the same meaning as the K_USERBODY, but concerns the message annex instead of the message body.

If *flag* is equal to 0, none of the previous options is selected: the message body will be copied (never "moved"), and message and annex are assumed to be part of the user (kernel) address space when the caller is a *USER* (*SUPERVISOR*) thread.

The message is sent asynchronously: the sender is only blocked during the time needed by the system to queue the send request.

RETURN VALUE

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

ERRORS

[K_EFULL]	The destination port is local and its queue is full, or the local communication subsystem is saturated.
[K_EINVAL]	*msgsrc* is not a valid local port identifier.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_ETOOMUCH]	*bodysize* is too big.
[K_EUNKNOWN]	Unreachable destination.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ipcReply`(2K), `ipcReceive`(2K), `ipcTarget`(2K), `ipcSysInfo`(2K)

NAME

ipcSysInfo – get system information about the current message

SYNOPSIS

```
#include <ipc/chIpc.h>
int ipcSysInfo(KnMsgHead *msghead);
```

FEATURES

IPC

DESCRIPTION

The *ipcSysInfo* call returns system information about the current message received. The *msghead* field points to a *KnMsgHead* structure whose members are the following:

```
int         bodySize ;       /* Body size */
char        flags [4] ; /* Flags */
KnUniqueId  srcPort ;        /* source port */
KnUniqueId  target ;         /* destination port or group */
KnUniqueId  coTarget ;       /* site qualifier */
KnTransId   transId ;        /* message transaction id. */
KnProtId    actorId ;        /* sender actor protection id. */
KnProtId    portId ;         /* source port protection id. */
```

The *bodySize* field is the size of the message body.

The *srcPort* field is the unique identifier of the source port of the message.

The *target* and *coTarget* parameters specified the destination when the message was sent (see *ipcSend*(2K) and *ipcCall*(2K)).

The *transId* field is the transaction identifier of the message.

The protection identifier of the actor originating the message is specified using *actorId*.

The protection identifier of the source port of the message is specified using *portId*.

Note that *ipcReply*(2K) does not affect the source port, the protection identifiers and the transaction identifier of a message.

RETURN VALUE

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

ERRORS

[K_EINVAL]	There is no current message (no message has been received by the current thread since the last *ipcReply*(2K) or *ipcSave*(2K)).
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

The fields *flags* and *transId* are obsolete and are never set by the kernel.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `ipcReply`(2K), `ipcReceive`(2K), `ipcSend`(2K), `ipcCall`(2K)

NAME

ipcTarget – build a message target

SYNOPSIS

#include <ipc/chIpc.h>
int **ipcTarget**(KnUniqueId *_groupui_, int _mode_);

FEATURES

IPC

DESCRIPTION

ipcTarget builds a message destination using the port group unique identifier given by *groupui* and the addressing mode *mode*. It affects the port group identifier given by *groupui*. This identifier may then be used as a target for sending messages by *ipcSend*(2K) or *ipcCall*(2K).

The following addressing modes are possible:

K_BROADMODE	The message will be sent to each reachable member of the group (*ie.* ports which have been inserted in the group - see *grpPortInsert*(2K)).
K_FUNCMODE	The message will be sent to one of the reachable members of the group.
K_FUNCUMODE	Tthe message will be sent to one of the reachable members of the group which resides on the site qualified by the *coTarget* member specified at send time (see *ipcSend*(2K)).
K_FUNCXMODE	The message will be sent to one of the reachable members of the group, assuming that the UI of this member must be different from the UI given by the *coTarget* member specified at send time (see *ipcSend*(2K)).

A port group name may be used directly as a message target, without applying *ipcTarget*(2K) on it. In that case, the default addressing mode K_BROADMODE is used.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*groupui* is not a port group name, or *mode* is not a valid addressing mode.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `grpAllocate`(2K), `ipcSend`(2K), `ipcCall`(2K), `grpPortInsert`(2K)

NAME

svLapCreate, lapDescZero, lapDescIsZero, lapDescDup – create a lap; reset a lap descriptor; test if a lap descriptor has been initialized; duplicate a lap descriptor

SYNOPSIS

#include <lap/chLap.h>
int **svLapCreate**(KnCap * *actorcap*, KnLapHdl *laphdl*, void * *cookie*, unsigned int *options*, KnLapDesc * *lapdesc*);

void **lapDescZero**(KnLapDesc * *lapdesc*);

int **lapDescIsZero**(KnLapDesc * *lapdesc*);

void **lapDescDup**(KnLapDesc * *olddesc*, KnLapDesc * *newdesc*);

FEATURES

CORE

DESCRIPTION

The *svLapCreate* (2K) system call creates a new local access point (lap) for the actor designated by the *actorcap* capability. If *actorcap* is equal to K_MYACTOR, the home actor of the calling thread is considered. If the call succeeds, the lap descriptor pointed to by *lapdesc* represents this new lap and can be passed to client threads that will use it to invoke the lap handler.

On lap invocation (see *lapInvoke* (2K)), the *laphdl* handler is called. This handler is a function which takes two arguments:

```
void handler (arg, cookie)
void         *arg ;
void         *cookie ;
```

Where *arg* is the argument specified by *lapInvoke* (2K), and *cookie* is the value of the cookie specified by *svLapCreate* (2K).

By default as a consequence of a lap invocation, the execution actor of the calling thread are set to the lap owning actor for the duration of the lap handler execution.

Additional actions may take place during the lap invocation, depending on the combination of flags specified in the *options* parameter:

K_LAP_SAFE — A " *lap frame* " descriptor is allocated to register the calling thread as a temporary resource of the invoked actor. Each " *lap frame* " has an associated level (lap frame level) which represents the number of lap frames for the considered thread.

This option is only valid if the LAPSAFE feature has been specified into the system. It will enforce a stronger checking during lap invocation. This guarantees in particular that the kernel will synchronize the *svLapDelete* (2K) operation with concurrent lap invocations.

Futhermore, full context of the calling thread is saved prior to the lap invocation. This guarantees that the calling thread can return from its invocation even if a failure (exception, deletion of the lap owning actor, etc.) occurs during the execution of the lap handler.

This option is mandatory for laps to be called from user mode.

The two utility routines *lapDescZero* (2K) and *lapDescIsZero* (2K) are available to manipulate the state of a lap descriptor with the following semantics: if a lap descriptor has been initialized with *lapDescZero* (2K) or is implemented in a zero-filled memory region, *lapDescIsZero* (2K) returns a non-zero value until the lap descriptor has been successfully initialized by *svLapCreate* (2K) or *lapResolve* (2K).

lapDescDup (2K) must be used to duplicate the contents of a lap descriptor.

RETURN VALUE

On success, *svLapCreate* (2K) returns K_OK. Otherwise, a negative error code is returned.

lapDescIsZero (2K) returns a non-zero value if the lap descriptor is not initialized.

lapDescZero (2K) and *lapDescDup* (2K) have no return values.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_ENOMEM]	The system is out of resources.

RESTRICTIONS

The current implementation does not support the K_LAP_SETJMP option, which is silently ignored.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `lapResolve`(2K) , `svLapBind`(2K) , `svLapDelete`(2K) , `svLapUnbind`(2K) , `threadStat`(2K)

NAME

svLapCreate, lapDescZero, lapDescIsZero, lapDescDup – create a lap; reset a lap descriptor; test if a lap descriptor has been initialized; duplicate a lap descriptor

SYNOPSIS

#include <lap/chLap.h>
int **svLapCreate**(KnCap * *actorcap*, KnLapHdl *laphdl*, void * *cookie*, unsigned int *options*, KnLapDesc * *lapdesc*);

void **lapDescZero**(KnLapDesc * *lapdesc*);

int **lapDescIsZero**(KnLapDesc * *lapdesc*);

void **lapDescDup**(KnLapDesc * *olddesc*, KnLapDesc * *newdesc*);

FEATURES

CORE

DESCRIPTION

The *svLapCreate* (2K) system call creates a new local access point (lap) for the actor designated by the *actorcap* capability. If *actorcap* is equal to K_MYACTOR, the home actor of the calling thread is considered. If the call succeeds, the lap descriptor pointed to by *lapdesc* represents this new lap and can be passed to client threads that will use it to invoke the lap handler.

On lap invocation (see *lapInvoke* (2K)), the *laphdl* handler is called. This handler is a function which takes two arguments:

```
void handler (arg, cookie)
void         *arg ;
void         *cookie ;
```

Where *arg* is the argument specified by *lapInvoke* (2K), and *cookie* is the value of the cookie specified by *svLapCreate* (2K).

By default as a consequence of a lap invocation, the execution actor of the calling thread are set to the lap owning actor for the duration of the lap handler execution.

Additional actions may take place during the lap invocation, depending on the combination of flags specified in the *options* parameter:

K_LAP_SAFE — A " *lap frame* " descriptor is allocated to register the calling thread as a temporary resource of the invoked actor. Each " *lap frame* " has an associated level (lap frame level) which represents the number of lap frames for the considered thread.

This option is only valid if the LAPSAFE feature has been specified into the system. It will enforce a stronger checking during lap invocation. This guarantees in particular that the kernel will synchronize the *svLapDelete* (2K) operation with concurrent lap invocations.

Futhermore, full context of the calling thread is saved prior to the lap invocation. This guarantees that the calling thread can return from its invocation even if a failure (exception, deletion of the lap owning actor, etc.) occurs during the execution of the lap handler.

This option is mandatory for laps to be called from user mode.

The two utility routines *lapDescZero* (2K) and *lapDescIsZero* (2K) are available to manipulate the state of a lap descriptor with the following semantics: if a lap descriptor has been initialized with *lapDescZero* (2K) or is implemented in a zero-filled memory region, *lapDescIsZero* (2K) returns a non-zero value until the lap descriptor has been successfully initialized by *svLapCreate* (2K) or *lapResolve* (2K).

lapDescDup (2K) must be used to duplicate the contents of a lap descriptor.

RETURN VALUE

On success, *svLapCreate* (2K) returns K_OK. Otherwise, a negative error code is returned.

lapDescIsZero (2K) returns a non-zero value if the lap descriptor is not initialized.

lapDescZero (2K) and *lapDescDup* (2K) have no return values.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_ENOMEM]	The system is out of resources.

RESTRICTIONS

The current implementation does not support the K_LAP_SETJMP option, which is silently ignored.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `lapResolve`(2K) , `svLapBind`(2K) , `svLapDelete`(2K) , `svLapUnbind`(2K) , `threadStat`(2K)

NAME

svLapCreate, lapDescZero, lapDescIsZero, lapDescDup – create a lap; reset a lap descriptor; test if a lap descriptor has been initialized; duplicate a lap descriptor

SYNOPSIS

#include <lap/chLap.h>
int **svLapCreate**(KnCap * *actorcap*, KnLapHdl *laphdl*, void * *cookie*, unsigned int *options*, KnLapDesc * *lapdesc*);

void **lapDescZero**(KnLapDesc * *lapdesc*);

int **lapDescIsZero**(KnLapDesc * *lapdesc*);

void **lapDescDup**(KnLapDesc * *olddesc*, KnLapDesc * *newdesc*);

FEATURES

CORE

DESCRIPTION

The *svLapCreate* (2K) system call creates a new local access point (lap) for the actor designated by the *actorcap* capability. If *actorcap* is equal to K_MYACTOR, the home actor of the calling thread is considered. If the call succeeds, the lap descriptor pointed to by *lapdesc* represents this new lap and can be passed to client threads that will use it to invoke the lap handler.

On lap invocation (see *lapInvoke* (2K)), the *laphdl* handler is called. This handler is a function which takes two arguments:

```
void handler (arg, cookie)
void         *arg ;
void         *cookie ;
```

Where *arg* is the argument specified by *lapInvoke* (2K), and *cookie* is the value of the cookie specified by *svLapCreate* (2K).

By default as a consequence of a lap invocation, the execution actor of the calling thread are set to the lap owning actor for the duration of the lap handler execution.

Additional actions may take place during the lap invocation, depending on the combination of flags specified in the *options* parameter:

K_LAP_SAFE — A " *lap frame* " descriptor is allocated to register the calling thread as a temporary resource of the invoked actor. Each " *lap frame* " has an associated level (lap frame level) which represents the number of lap frames for the considered thread.

This option is only valid if the LAPSAFE feature has been specified into the system. It will enforce a stronger checking during lap invocation. This guarantees in particular that the kernel will synchronize the *svLapDelete* (2K) operation with concurrent lap invocations.

Futhermore, full context of the calling thread is saved prior to the lap invocation. This guarantees that the calling thread can return from its invocation even if a failure (exception, deletion of the lap owning actor, etc.) occurs during the execution of the lap handler.

This option is mandatory for laps to be called from user mode.

The two utility routines *lapDescZero* (2K) and *lapDescIsZero* (2K) are available to manipulate the state of a lap descriptor with the following semantics: if a lap descriptor has been initialized with *lapDescZero* (2K) or is implemented in a zero-filled memory region, *lapDescIsZero* (2K) returns a non-zero value until the lap descriptor has been successfully initialized by *svLapCreate* (2K) or *lapResolve* (2K).

lapDescDup (2K) must be used to duplicate the contents of a lap descriptor.

RETURN VALUE

On success, *svLapCreate* (2K) returns K_OK. Otherwise, a negative error code is returned.

lapDescIsZero (2K) returns a non-zero value if the lap descriptor is not initialized.

lapDescZero (2K) and *lapDescDup* (2K) have no return values.

ERRORS

[K_EINVAL] *actorcap* is an inconsistent actor capability.

[K_ENOMEM] The system is out of resources.

RESTRICTIONS

The current implementation does not support the K_LAP_SETJMP option, which is silently ignored.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `lapResolve`(2K) , `svLapBind`(2K) , `svLapDelete`(2K) , `svLapUnbind`(2K) , `threadStat`(2K)

NAME

lapInvoke – invoke a lap handler

SYNOPSIS

```
#include <lap/chLap.h>
int lapInvoke(KnLapDesc *lapdesc, void *arg);
```

DESCRIPTION

The *lapInvoke*(2K) system call invokes a lap handler in the context of the calling thread (see *svLapCreate*(2K) for a description of the exact invocation semantics). The lap considered here is represented by its lap descriptor pointed to by *lapdesc*.

The lap handler is called with two parameters: the *arg* argument and the cookie specified at lap creation time (see *svLapCreate*(2K)).

RETURN VALUES

On success, *lapInvoke*(2K) returns K_OK, otherwise a negative error code is returned.

ERRORS

[K_EFAIL]	The lap was created with the K_LAP_SAFE option and the lap has been deleted when the invocation is in progress.
[K_EFAIL]	The lap was created with the K_LAP_SETJMP option and a failure occured during the lap invocation.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_EINVAL]	The lap was created with the K_LAP_SAFE option and it has been deleted. *lapInvoke*(2K) is called from user mode and the invoked lap was not created with the K_LAP_SAFE option.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svLapCreate(2K), svLapDelete(2K)

NAME

svLapBind, svLapUnbind, lapResolve – bind a symbolic name to a lap descriptor; unbind the symbolic name bound to a lap descriptor; get a lap descriptor from a lap symbolic name

SYNOPSIS

#include <lap/chLap.h>
int **svLapBind**(KnLapDesc * *lapdesc*, char * *name*, unsigned int *options*);

int **svLapUnbind**(char * *name*);

int **lapResolve**(KnLapDesc * *lapdesc*, char * *name*, unsigned int *options*);

FEATURES

LAPBIND

DESCRIPTION

The *svLapBind* (2K) system call binds the lap descriptor pointed to by *lapdesc* with the symbolic name pointed to by *name* .

name points to a null-terminated string of K_LAPNAMEMAX characters at most (not including the null-terminating character).

If the K_LAP_PROTECTED option is set in the *options* parameter, the binding will only be visible to trusted threads (e.g. threads executing in user actors will get an error from *lapResolve* (2K) on this name).

The *svLapUnbind* (2K) system call removes the lap binding associated to *name* .

The *lapResolve* (2K) system call initializes the lap descriptor pointed to by *lapdesc* with the lap descriptor bound to *name* . *lapResolve* (2K) will block until a lap descriptor is bound to *name* , except if the K_LAP_NOBLOCK option is set in *options* , in which case the call returns immediatly with an error.

svLapBind and *svLapUnbind* are restricted to *SUPERVISOR* threads.

RETURN VALUE

On success, these calls return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EABORT]	The calling thread has been aborted in *lapResolve* (2K).
[K_EBUSY]	The name given to *svLapBind* (2K) is already in use.
[K_EFAULT]	The *lapdesc* or *name* arguments of *lapResolve* points to the outside of the caller's address space.
[K_EINVAL]	The *name* given to *svLapUnbind* (2K) is not bound to any lap, or the *name* given to *lapResolve* (2K) is not bound to any lap, and the K_LAP_NOBLOCK option is set.
[K_ENOMEM]	The *name* given to *svLapBind* (2K), *svLapUnbind* (2K) or *lapResolve* (2K) is too long. [K_ENOMEM] The system is out of resources.

[K_EPRIV] *lapResolve* (2K) is called by a non trusted thread to resolve a name which has been bound with the K_LAP_PROTECTED option.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K)

NAME

msgAllocate – allocates a message from a message space

SYNOPSIS

#include <mipc/chMipc.h>
int **msgAllocate**(int *spaceLid*, unsigned int *poolIndex*, unsigned int *msgSize*, KnTimeVal **waitLimit*, char ***msgAddr*);

FEATURES

MIPC

DESCRIPTION

msgAllocate attempts to allocate a message from the most appropriate message pool of the message space designated by *spaceLid*.

With MIPC_S, the only valid value for *spaceLid* is 0.

poolIndex is either the index of the message pool from which the message must be allocated or the specific K_ANY_MSGPOOL constant.

msgSize is the size of the application message.

waitLimit optionally specifies a timeout, and may also specify a non abortable wait.

msgAddr is the address of a pointer where to store the address of the allocated message.

msgAllocate considers the message pool which is designated by *poolIndex*, if it is not equal to K_ANY_MSGPOOL, or parses the list of message pools of the Message Space to find the pool which fits the requested message size. If the message pool is not empty, *msgAllocate* takes the first free message, computes its address within the addressing space of the caller, and stores it into the *msgAddr* pointer.

If the pool is empty, *msgAllocate* does not try to allocate a message from a pool of bigger messages, if any. If *waitLimit* is equal to K_NOBLOCK(0), *msgAllocate* immediately returns a K_ETIMEOUT error code. Otherwise, it blocks the invoking thread until a message of the pool is freed by *msgFree*(2K) or until the timeout, if any, expires.

RESTRICTIONS

Interrupt handlers must always invoke *msgAllocate* with a delay of 0.

RETURN VALUE

Upon successful completion, *msgAllocate* returns zero. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*spaceLid* is not valid, or *poolIndex* is not valid, or *waitLimit* is not a valid *KnTimeVal*.
[K_ESIZE]	*msgSize* is greater than the maximum message size.
[K_ETIMEOUT]	The maximum waiting time specified in *waitLimit* expired.

[K_EFAULT]	*msgAddr* is not a valid address within the addressing space of the caller.
[K_EABORT]	The invoking thread was aborted while waiting for a message.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`msgSpaceCreate`(2K), `msgSpaceOpen`(2K), `msgFree`(2K), `msgPut`(2K), `msgGet`(2K), `msgRemove`(2K)

NAME

msgFree – free a message of a message space

SYNOPSIS

#include <mipc/chMipc.h>
int **msgFree**(int *spaceLid*, char **msg*);

FEATURES

MIPC

DESCRIPTION

msgFree returns to the message space identified by *spaceLid* a message, which was previously allocated by *msgAllocate*(2K) from the same message space, or which was previously received by *msgGet*(2K) from a message queue of the same message space.

With MIPC_S, the only valid value for *spaceLid* is 0.

msg is the address of the message within the addressing space of the caller.

msgFree converts *msg* into the corresponding internal message resource, from which it finds its associated message pool. If a thread is blocked waiting for a message of the pool to become available, it is awakened, and the freed message is given to it. Otherwise, the freed message is appended to the pool.

RETURN VALUE

Upon successful completion, *msgFree* returns zero. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*spaceLid* is not valid.
[K_EBADMSG]	*msg* is not a valid message address, or the corresponding internal message resource is not in the allocated state.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`msgSpaceCreate`(2K), `msgSpaceOpen`(2K), `msgAllocate`(2K), `msgPut`(2K), `msgGet`(2K), `msgRemove`(2K)

NAME

msgGet – retrieves the first message of a message queue

SYNOPSIS

#include <mipc/chMipc.h>
int **msgGet**(int *spaceLid*, unsigned int *msgQueueId*, KnTimeVal **waitLimit*, char ***msgAddr*, KnUniqueId **srcActor*);

FEATURES

MIPC

DESCRIPTION

msgGet returns to the caller the first message with the highest priority pending behind the message queue of index *msgQueueId* within the set of message queues of the message space designated by *spaceLid*.

waitLimit optionally specifies a timeout, and may also specify a non abortable wait.

msgAddr is the address of a pointer where to store the address of the retrieved message.

srcActor is either equal to the specific NULL constant or points to a structure of type *KnUniqueId* where to store the unique identifier of the source actor of the retrieved message, if any.

The first message with the highest priority pending behind the message queue is dequeued.

If no message is pending behind the message queue, the invoking thread is blocked until a message is sent to the message queue or until the timeout, if any, expires. Multiple threads can be blocked behind the same message queue. In such a case, the policy for ordering incoming messages is a simple FIFO. No data copy is performed. The receiving thread is returned a pointer to the message allocated by the sender. This thread can immediately access the content of the message, and update it before sending the message (without any extra data copy) to another message queue of the same message space.

RETURN VALUE

Upon successful completion, *msgGet* returns 0. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*spaceLid* is not valid, or *waitLimit* is not a valid *KnTimeVal*.
[K_EARGS]	*msgQueueId* is not a valid message queue.
[K_ETIMEOUT]	The maximum waiting time specified in *waitLimit* expired.
[K_EFAULT]	One of the following pointers is invalid: *waitLimit*, *msgAddr* or *srcActor*. If *waitLimit* is an invalid pointer, *msgGet* has no effect on the message queue designated by *msgQueueId* and returns immediately K_EFAULT. If *srcActor* or

	msgAddr is or becomes an invalid pointer (while *msgGet*, the message that has been dequeued is freed. Therefore the message is lost from the faulty application point of view.
[K_EABORT]	The invoking thread was aborted while waiting for a message.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`msgSpaceCreate`(2K), `msgSpaceOpen`(2K), `msgAllocate`(2K), `msgFree`(2K), `msgPut`(2K), `msgRemove`(2K)

NAME

msgPut – post a message to a message queue

SYNOPSIS

#include <mipc/chMipc.h>
int **msgPut**(int *spaceLid*, unsigned int *msgQueueId*, char **msg*, unsigned int *prio*);

FEATURES

MIPC

DESCRIPTION

msgPut sends a message to the message queue of index *msgQueueId* in the message space identified by *spaceLid*.

msg is the address within the addressing space of the caller of a message which was previously allocated (by *msgAllocate*(2K)) from the same message space, or which has been retrieved (by *msgGet*(2K)) from a message queue of the same message space.

prio is the priority of the message. Its value must fall between 0 (the lowest priority) and the specific K_MSG_PRIOMAX constant which is the highest priority. If *prio* is greater than K_MSG_PRIOMAX, it is silently dropped to K_MSG_PRIOMAX.

If a thread is blocked waiting for a message to arrive, it is awakened, and the message is immediately given to it. Otherwise, the message is inserted in the message queue according to its priority.

RETURN VALUE

Upon successful completion, *msgPut* returns 0. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*spaceLid* is not valid.
[K_EARGS]	*msgQueueId* is not a valid message queue.
[K_EBADMSG]	*msg* is not a valid message address, or the corresponding message resource is not in the allocated state.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`msgSpaceCreate`(2K), `msgSpaceOpen`(2K), `msgAllocate`(2K), `msgFree`(2K), `msgGet`(2K), `msgRemove`(2K)

NAME

msgRemove – remove a message from a message queue

SYNOPSIS

#include <mipc/chMipc.h>
int **msgRemove**(int *spaceLid*, unsigned int *msgQueueId*, char **msg*, unsigned int *prio*);

FEATURES

MIPC

DESCRIPTION

The *msgRemove* call removes the message whose address is *msg* from the message queue of the index *msgQueueId* in the message space *spaceLid*.

The *prio* argument must be identical to the priority provided to the *msgPut* system call previously invoked to post the message.

RETURN VALUE

Upon successful completion, *msgRemove* returns 0. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*spaceLid* is not valid.
[K_EARGS]	*msgQueueId* is not a valid message queue.
[K_EBADMSG]	*msg* is not a valid message address, or the corresponding message has already been consumed, or the priority is not the one affected when the message was put in the queue

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`msgSpaceCreate`(2K), `msgSpaceOpen`(2K), `msgAllocate`(2K), `msgFree`(2K), `msgGet`(2K), `msgPut`(2K)

NAME

msgSpaceCreate – create a message space

SYNOPSIS

#include <mipc/chMipc.h>
int **msgSpaceCreate**(KnMsgSpaceId *spaceGid*, unsigned int *msgQueueNb*, unsigned int *msgPoolNb*, const KnMsgPool **msgPools*);

FEATURES

MIPC

DESCRIPTION

`msgSpaceCreate()` creates a message space.

spaceGid is the global identifier of the new message space. It is under the responsibility of applications to assign unique global identifiers to their message spaces. If *spaceGid* is equal to the special `K_PRIVATEID` constant, the message space is private to the invoking actor: its message queues and message pools will only be accessible to threads executing within this actor.

msgQueueNb is the number of message queues of the message space. Each message queue is designated by its index within the set of message queues (by an unsigned integer within the range [0 .. *msgQueueNb* - 1]).

msgPoolNb is the number of message pools of the communication space. Its value must be within the range [1 .. `K_MSG_POOLMAX`] where `K_MSG_POOLMAX` (16) is a constant defined at kernel compile time.

msgPool is a pointer to an array of `poolNb` structures of type `KnMsgPool`. The `KnMsgPool` structure includes the following information:

```
unsigned int msgSize ;
unsigned int msgNumber ;
```

RETURN VALUES

Upon successful completion, the memory allocated for the message pools is mapped within the addressing space of the actor, and `msgSpaceCreate()` returns a positive local identifier of the message space. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL] *spaceGid* is not equal to `K_PRIVATEID`, and is already assigned to another existing message space.

[K_EOVERLAP] There is not sufficient room in the addressing space of the actor to map the message pools of the created message space.

[K_EARGS] The value of *msgQueueNb* is equal to zero, or the value of *msgPoolNb* is out of the range [1 … `K_MSG_POOLMAX`].

[K_EFAULT] Some of the data provided are outside the current actor's address space.

[K_ENOMEM] The system was not able to allocate the memory needed to create the message space.

This value is returned, for example, when the system attemps to exceed the limit defined by the tunable parameter `ipc.maxOpenSpaceNumber`.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`msgAllocate`(2K), `msgFree`(2K), `msgGet`(2K), `msgPut`(2K), `msgRemove`(2K), `msgSpaceOpen`(2K)

NAME

msgSpaceOpen – open a message space

SYNOPSIS

```
#include <mipc/chMipc.h>
int msgSpaceOpen(KnMsgSpaceId spaceGid);
```

FEATURES

MIPC

DESCRIPTION

The `msgSpaceOpen( )` call assigns a local private identifier associated with the message space opened to the calling actor. It maps the message pools of the message to the calling actor's addressing space.

The *spaceGid* field is the global identifier of a message space created previously using `msgSpaceCreate`(2K).

The `msgSpaceOpen( )` call transparently selects the address within the addressing space of the actor at which the set of message pools of the message space are to be mapped.

RETURN VALUES

Upon successful completion, `msgSpaceOpen( )` returns the local identifier of the message space assigned to the actor. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL] The message space does not exist.

[K_EOVERLAP] There is insufficient room in the addressing space of the actor to map the message pools of the opened message space.

[K_ENOMEM] The system is out of resources.

This value is returned, for example, when the system attemps to exceed the limit defined by the tunable `kernel.mipc.maxOpenSpaceNumber`.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`msgAllocate`(2K), `msgFree`(2K), `msgGet`(2K), `msgPut`(2K), `msgRemove`(2K), `msgSpaceCreate`(2K)

NAME

mutexInit, mutexGet, mutexRel, mutexTry – initialize a mutex; acquire a mutex; release a mutex; try to acquire a mutex

SYNOPSIS

```
#include <sync/chMutex.h>
int mutexInit(KnMutex * mutex);

int mutexGet(KnMutex * mutex);

int mutexRel(KnMutex * mutex);

int mutexTry(KnMutex * mutex);
```

FEATURES

MUTEX

DESCRIPTION

Mutexes are binary semaphores used to protect shared data from concurrent access: a mutex is a two state-variable - free or locked.

Mutexes are *KnMutex* structures allocated in the user memory.

mutexInit initializes the mutex the address of which is *mutex*. The mutex is initialized as free.

Statically allocated mutexes can be initialized using the *K_KNMUTEX_INITIALIZER* macro, which initializes the mutex as free. This macro is used as follows:

```
KnMutex myMutex = K_KNMUTEX_INITIALIZER ;
```

mutexGet is used to acquire a mutex. If the mutex is free, it becomes locked and the caller continues its execution normally. If the mutex is locked, the caller is blocked until the mutex is released.

mutexRel is used to release a mutex. If threads are blocked behind the mutex, one of them is awakened.

mutexTry is only an attempt to acquire the mutex: it has the same effect as *mutexGet*, except that if the mutex is locked, the thread is not blocked (a return code is provided).

Semaphores initialized with a count value of 1 (see *semInit (2K)*) are functionally equivalent to mutexes. However, there are implementation differences between mutexes and semaphores which make them behave differently. First, mutex operations are implemented so that optimal performance is obtained when no thread state change is needed (i.e. when a thread acquires a free mutex, or when a mutex is released and no thread is waiting for it). A consequence of this optimization is that mutexes do not guarantee fairness.

A blocking *mutexGet* is *NONABORTABLE* (see *threadAbort* (2K)).

RESTRICTIONS

A user application and a supervisor application may not share a mutex.

On the contrary, two user applications may share a mutex by mapping it in both user address spaces. Such shared mutexes must be dynamically allocated mutexes.

RETURN VALUE

mutexTry returns 0 if the mutex was already locked, 1 if it was free. Upon successful completion of *mutexInit*, *mutexGet* and *mutexRel*, 0 is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT] Some of the provided data are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`semInit`(2K)

NAME

mutexInit, mutexGet, mutexRel, mutexTry – initialize a mutex; acquire a mutex; release a mutex; try to acquire a mutex

SYNOPSIS

```
#include <sync/chMutex.h>
int mutexInit(KnMutex * mutex);

int mutexGet(KnMutex * mutex);

int mutexRel(KnMutex * mutex);

int mutexTry(KnMutex * mutex);
```

FEATURES

MUTEX

DESCRIPTION

Mutexes are binary semaphores used to protect shared data from concurrent access: a mutex is a two state-variable - free or locked.

Mutexes are *KnMutex* structures allocated in the user memory.

mutexInit initializes the mutex the address of which is *mutex*. The mutex is initialized as free.

Statically allocated mutexes can be initialized using the *K_KNMUTEX_INITIALIZER* macro, which initializes the mutex as free. This macro is used as follows:

```
KnMutex myMutex = K_KNMUTEX_INITIALIZER ;
```

mutexGet is used to acquire a mutex. If the mutex is free, it becomes locked and the caller continues its execution normally. If the mutex is locked, the caller is blocked until the mutex is released.

mutexRel is used to release a mutex. If threads are blocked behind the mutex, one of them is awakened.

mutexTry is only an attempt to acquire the mutex: it has the same effect as *mutexGet*, except that if the mutex is locked, the thread is not blocked (a return code is provided).

Semaphores initialized with a count value of 1 (see *semInit (2K)*) are functionally equivalent to mutexes. However, there are implementation differences between mutexes and semaphores which make them behave differently. First, mutex operations are implemented so that optimal performance is obtained when no thread state change is needed (i.e. when a thread acquires a free mutex, or when a mutex is released and no thread is waiting for it). A consequence of this optimization is that mutexes do not guarantee fairness.

A blocking *mutexGet* is *NONABORTABLE* (see *threadAbort* (2K)).

RESTRICTIONS

A user application and a supervisor application may not share a mutex.

On the contrary, two user applications may share a mutex by mapping it in both user address spaces. Such shared mutexes must be dynamically allocated mutexes.

RETURN VALUE

mutexTry returns 0 if the mutex was already locked, 1 if it was free. Upon successful completion of *mutexInit*, *mutexGet* and *mutexRel*, 0 is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT] Some of the provided data are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`semInit`(2K)

NAME

mutexInit, mutexGet, mutexRel, mutexTry – initialize a mutex; acquire a mutex; release a mutex; try to acquire a mutex

SYNOPSIS

```
#include <sync/chMutex.h>
int mutexInit(KnMutex * mutex);

int mutexGet(KnMutex * mutex);

int mutexRel(KnMutex * mutex);

int mutexTry(KnMutex * mutex);
```

FEATURES

MUTEX

DESCRIPTION

Mutexes are binary semaphores used to protect shared data from concurrent access: a mutex is a two state-variable - free or locked.

Mutexes are *KnMutex* structures allocated in the user memory.

mutexInit initializes the mutex the address of which is *mutex*. The mutex is initialized as free.

Statically allocated mutexes can be initialized using the *K_KNMUTEX_INITIALIZER* macro, which initializes the mutex as free. This macro is used as follows:

```
KnMutex myMutex = K_KNMUTEX_INITIALIZER ;
```

mutexGet is used to acquire a mutex. If the mutex is free, it becomes locked and the caller continues its execution normally. If the mutex is locked, the caller is blocked until the mutex is released.

mutexRel is used to release a mutex. If threads are blocked behind the mutex, one of them is awakened.

mutexTry is only an attempt to acquire the mutex: it has the same effect as *mutexGet*, except that if the mutex is locked, the thread is not blocked (a return code is provided).

Semaphores initialized with a count value of 1 (see *semInit (2K)*) are functionally equivalent to mutexes. However, there are implementation differences between mutexes and semaphores which make them behave differently. First, mutex operations are implemented so that optimal performance is obtained when no thread state change is needed (i.e. when a thread acquires a free mutex, or when a mutex is released and no thread is waiting for it). A consequence of this optimization is that mutexes do not guarantee fairness.

A blocking *mutexGet* is *NONABORTABLE* (see *threadAbort* (2K)).

RESTRICTIONS

A user application and a supervisor application may not share a mutex.

On the contrary, two user applications may share a mutex by mapping it in both user address spaces. Such shared mutexes must be dynamically allocated mutexes.

RETURN VALUE

mutexTry returns 0 if the mutex was already locked, 1 if it was free. Upon successful completion of *mutexInit*, *mutexGet* and *mutexRel*, 0 is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the provided data are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`semInit`(2K)

NAME

mutexInit, mutexGet, mutexRel, mutexTry – initialize a mutex; acquire a mutex; release a mutex; try to acquire a mutex

SYNOPSIS

```
#include <sync/chMutex.h>
int mutexInit(KnMutex * mutex);

int mutexGet(KnMutex * mutex);

int mutexRel(KnMutex * mutex);

int mutexTry(KnMutex * mutex);
```

FEATURES

MUTEX

DESCRIPTION

Mutexes are binary semaphores used to protect shared data from concurrent access: a mutex is a two state-variable - free or locked.

Mutexes are *KnMutex* structures allocated in the user memory.

mutexInit initializes the mutex the address of which is *mutex*. The mutex is initialized as free.

Statically allocated mutexes can be initialized using the *K_KNMUTEX_INITIALIZER* macro, which initializes the mutex as free. This macro is used as follows:

```
KnMutex myMutex = K_KNMUTEX_INITIALIZER ;
```

mutexGet is used to acquire a mutex. If the mutex is free, it becomes locked and the caller continues its execution normally. If the mutex is locked, the caller is blocked until the mutex is released.

mutexRel is used to release a mutex. If threads are blocked behind the mutex, one of them is awakened.

mutexTry is only an attempt to acquire the mutex: it has the same effect as *mutexGet*, except that if the mutex is locked, the thread is not blocked (a return code is provided).

Semaphores initialized with a count value of 1 (see *semInit (2K)*) are functionally equivalent to mutexes. However, there are implementation differences between mutexes and semaphores which make them behave differently. First, mutex operations are implemented so that optimal performance is obtained when no thread state change is needed (i.e. when a thread acquires a free mutex, or when a mutex is released and no thread is waiting for it). A consequence of this optimization is that mutexes do not guarantee fairness.

A blocking *mutexGet* is *NONABORTABLE* (see *threadAbort* (2K)).

RESTRICTIONS

A user application and a supervisor application may not share a mutex.

On the contrary, two user applications may share a mutex by mapping it in both user address spaces. Such shared mutexes must be dynamically allocated mutexes.

RETURN VALUE

mutexTry returns 0 if the mutex was already locked, 1 if it was free. Upon successful completion of *mutexInit*, *mutexGet* and *mutexRel*, 0 is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the provided data are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`semInit`(2K)

NAME

padGet – return actor-specific values associated with keys

SYNOPSIS

```
#include <pd/chPd.h>
void *padGet(KnCap *actorcap, PdKey key);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *padGet* function returns the value currently bound to *key* for the actor whose capability is given by *actorcap*.

If *actorcap* is K_MYACTOR, the current actor is used.

You can only call *padGet* from a supervisor actor, but *actorcap* can designate any local actor.

Calling *padGet* with a *key* value not obtained from *padKeyCreate* or after *key* has been deleted using *padKeyDelete*can produce unpredictable results.

RETURN VALUE

The *padGet* function returns the actor-specific data value associated with the *key* specified for the relevant actor. If no actor-specific data value has been associated with *key*, a NULL value is returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

padKeyCreate(2K), padKeyDelete(2K), padSet(2K)

NAME

padKeyCreate – create a private key for an actor

SYNOPSIS

```
#include <pd/chPd.h>
int padKeyCreate(PdKey *pkey, KnPdHdl destructor);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *padKeyCreate* function creates an actor-specific data key visible to all actors on a site.

The key created is returned in **pkey*.

You can only call *padKeyCreate* from a supervisor actor.

The keys provided by *padKeyCreate* are opaque indices used to locate actor-specific data. Although the same key may be used by different actors, the values bound to the key by *padSet* (2K) are maintained on a per-actor basis and persist for the life of the designated actor.

Upon key creation, the value NULL is associated with the new key in all active actors. Upon actor creation, the value NULL is associated with all keys defined in the new actor.

An optional *destructor* function may be associated with each key. At actor exit, if a key has a non-NULL destructor pointer, and the actor has a non-NULL value associated with that key, the function `destructor()` is called with the current value as its sole argument.

By default, the Private Data Manager invokes the destructor function through the LAP invocation mechanism (see *IsvLapCreate*(2K)) to ensure that the destructor function has the same execution actor than the thread which called `padKeyCreate()` originally. However, when an actor associates a destructor function to itself, the destructor function cannot be invoked by LAP (as the actor is being deleted) and is executed in the context of the Private Data Manager. As a consequence, an actor self deletion handler should not use any API which assumes that the current execution actor is the actor being deleted.

The order of destructor calls is unspecified if there is more than one destructor for an actor when it exits.

Destructor calls are repeated for a maximum of *PD_DESTRUCTOR_ITERATIONS* (as defined in *pd/chPd.h*) times as necessary to reclaim all actor-specific data in the case that a destructor causes more actor-specific data to be created.

RETURN VALUE

Upon successful completion, the newly created key is stored in **pkey* and a value of 0 is returned. Otherwise, a positive error code is returned.

ERRORS

[PD_ENOKEY]	The system-imposed limit on the total number of keys has been exceeded (see *PAD_KEYS_MAX* in *pd/chPd.h*).

[PD_ENOMEM]	There is insufficient memory to create the key.
[PD_ESERVER]	The Private Data Manager is unreachable.
[PD_EUSER]	This call is reserved for supervisor actors.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`padSet`(2K), `padKeyDelete`(2K)

NAME

padKeyDelete – delete an actor private key

SYNOPSIS

```
#include <pd/chPd.h>
int padKeyDelete(PdKey key);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *padKeyDelete* function deletes an actor-specific data key previously returned by *padKeyCreate (2K).*

This function can only be called from a supervisor actor.

The actor-specific data values associated with *key* need not be NULL at the time *padKeyDelete* is called. The application frees storage and performs any cleanup operations needed for data structures related to the deleted key or to associated actor-specific data. The cleanup can be performed before or after invoking *padKeyDelete*. Any attempt to use *key* after the call to *padKeyDelete* can produce unpredictable results There are no destructor functions invoked by *padKeyDelete.*

At actor destruction, all keys still allocated by an actor will be deleted.

RETURN VALUE

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

ERRORS

[PD_EINVAL]	The *key* value is invalid.
[PD_ESERVER]	The Private Data Manager is unreachable.
[PD_EUSER]	This call is reserved for supervisor actors.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`padKeyCreate`(2K)

NAME

padSet – set the actor's key to a specific value

SYNOPSIS

```
#include <pd/chPd.h>
int padSet(KnCap *actorcap, PdKey key, const void *value);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *padSet* function associates an actor-specific *value* with a *key* (obtained via a previous call to *padKeyCreate (2K)*) for the actor whose capability is given by *actorcap*.

If *actorcap* is K_MYACTOR, the current actor is used.

You can only call *padSet* from a supervisor actor, but *actorcap* can designate any local actor.

Different actors may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by a controlling actor.

Calling *padSet* from a destructor may result in lost storage.

RETURN VALUE

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

ERRORS

[PD_EINVAL]	The *key* value is invalid, or *actorcap* is an inconsistent actor capability.
[PD_ENOMEM]	There is insufficient memory to associate the value with the key.
[PD_ESERVER]	The Private Data Manager is unreachable.
[PD_EUSER]	This call is reserved for SUPERVISOR actors.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`padKeyCreate`(2K)

NAME

portCreate, portDeclare – create a port; declare a port

SYNOPSIS

```
#include <ipc/chIpc.h>
int portCreate(KnCap * actorcap, KnUniqueId * portui);

int portDeclare(KnCap * actorcap, KnUniqueId * portui);
```

FEATURES

IPC

DESCRIPTION

portCreate creates a port which is attached to the actor the capability of which (see *actorCreate* (2K)) is given by *actorcap*. If *actorcap* is K_MYACTOR, the port is attached to the current actor.

A unique identifier is allocated to the port. This identifier is returned in the *KnUniqueId* structure given by *portui*.

The port is created in the *DISABLED* state (see *portEnable* (2K)).

The port message queue capacity is limited to K_CPORTQUEUE.

The maximum number of ports that may be created on a site is limited to a value set at kernel generation.

portDeclare is equivalent to *portCreate*, except that the unique identifier of the port is given by the user (in *portui*) instead of being generated by the kernel. The unique identifier provided by the user must have been previously built using *uiBuild* (2K). As the user is responsible for insuring the unicity of the port identifier, *portDeclare* is restricted to *SUPERVISOR* threads (see *threadCreate* (2K)), or threads belonging to *SYSTEM* actors (see *actorCreate* (2K)).

RETURN VALUE

Upon successful completion, a positive local identifier of the port is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor (*portDeclare* only).
[K_ENOMEM]	The system is out of resources.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `portDelete`(2K), `portEnable`(2K), `portDisable`(2K), `actorCreate`(2K)

NAME

portCreate, portDeclare – create a port; declare a port

SYNOPSIS

```
#include <ipc/chIpc.h>
int portCreate(KnCap * actorcap, KnUniqueId * portui);

int portDeclare(KnCap * actorcap, KnUniqueId * portui);
```

FEATURES

IPC

DESCRIPTION

portCreate creates a port which is attached to the actor the capability of which (see *actorCreate* (2K)) is given by *actorcap*. If *actorcap* is K_MYACTOR, the port is attached to the current actor.

A unique identifier is allocated to the port. This identifier is returned in the *KnUniqueId* structure given by *portui*.

The port is created in the *DISABLED* state (see *portEnable* (2K)).

The port message queue capacity is limited to K_CPORTQUEUE.

The maximum number of ports that may be created on a site is limited to a value set at kernel generation.

portDeclare is equivalent to *portCreate*, except that the unique identifier of the port is given by the user (in *portui*) instead of being generated by the kernel. The unique identifier provided by the user must have been previously built using *uiBuild* (2K). As the user is responsible for insuring the unicity of the port identifier, *portDeclare* is restricted to *SUPERVISOR* threads (see *threadCreate* (2K)), or threads belonging to *SYSTEM* actors (see *actorCreate* (2K)).

RETURN VALUE

Upon successful completion, a positive local identifier of the port is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor (*portDeclare* only).
[K_ENOMEM]	The system is out of resources.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `portDelete`(2K), `portEnable`(2K), `portDisable`(2K), `actorCreate`(2K)

NAME

portDelete – delete a port

SYNOPSIS

```
#include <ipc/chIpc.h>
int portDelete(KnCap *actorcap, int portli);
```

FEATURES

IPC

DESCRIPTION

portDelete deletes the port the local identifier of which is *portli* in the actor the capability of which is specified by the *KnCap* structure given by *actorcap* (see *actorCreate*(2K)). If *actorcap* is K_MYACTOR, the current actor is used.

When the port is deleted, the queued messages are deleted (the *ipcCall*(2K) transactions are aborted), and the waiting threads exit their waiting status with the K_ENOPORT error code.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *portli* is not a valid port identifier within the named actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.

RESTRICTIONS

The port and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

portCreate(2K), actorCreate(2K), ipcCall(2K)

NAME

portEnable – Enable a port portDisable; Disable a port

SYNOPSIS

```
#include <ipc/chIpc.h>
int portEnable(KnCap *actorcap, int portli, int priority);

int portDisable(KnCap *actorcap, int portli);
```

FEATURES

IPC

DESCRIPTION

The *portEnable* utility is used to put ports into the *ENABLED* state. The port's local identifier is *portli* in the actor whose capability is specified by the *KnCap* structure given by *actorcap* (see *actorCreate*(2K)). If *actorcap* is K_MYACTOR, the current actor is used. The *priority* field defines the priority of the port in the set of *ENABLED* ports (see below).

The *portDisable* utility puts the port in the *DISABLED* state.

The state of a port determines whether or not the port is used if a thread performs a multiple receive call (*ipcReceive* (2K) with K_ANYENABLED option set). Only a message delivered to an *ENABLED* port may be received during a multiple receive call. When several ports in the set have messages queued behind them, the *priority* parameter given when the port was *ENABLED* is used (a low value means a high priority). During a multiple receive, the oldest message queued behind the highest priority *ENABLED* port is delivered. If none of the *ENABLED* ports have a queued message, the first message received by any of the *ENABLED* ports is delivered.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*portli* is not a valid port identifier within the current actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

The port and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portCreate`(2K), `ipcReceive`(2K)

NAME

portMigrate, portGetSeqNum – Migrate a port; Get a port sequence number

SYNOPSIS

```
#include <ipc/chIpc.h>
int portMigrate(int options, KnCap * srcactorcap, int portli, KnCap * dstactorcap,
KnEvtNum * seqnum);

int portGetSeqNum(KnCap * srcactorcap, int portli, KnEvtNum * seqnum);
```

FEATURES

IPC

DESCRIPTION

The *portMigrate* utility forces the port whose local identifier is *portli* in the actor whose capability is given by *srcactorcap* to migrate to the actor whose capability is given by *dstactorcap*. After the call, the port is attached to this destination actor. The local identifier of the portis returned by *portMigrate* to the destination actor.

If *srcactorcap* and / or *dstactorcap* is K_MYACTOR, the current actor is used as the source and / or destination of the migration.

The *options* , which describe the way in which the migration is performed may take one of the following values:

K_WITHMSGS	The messages queued behind the port follow the port during the migration.
K_KILLMSGS	The queued messages are deleted (and *ipcCall* (2K) transactions are aborted) before the migration.

The *seqnum* out parameter is a pointer to a count of messages that have been received on the source port, plus one additional increment for the migrate itself. The port message count is reset to 0 on the destination port.

If the port was inserted into port groups (using *grpPortInsert* (2K)) before the migration, the port remains a member of the same groups after the migration.

Ports that had been enabled (using *portEnable* (2K)) before the migration are disabled after the migration.

When a port migrates, any thread waiting for a message on that port exits its waiting state amd returns the K_ENOPORT error code.

A port may not be migrated if a message handler (see *svMsgHandler* (2K)) is attached to it. If such an attempt is made, the K_EBUSYPORT error code is returned.

The *portGetSeqNum* function gets the count of messages that have been received on the port whose local identifier is *portli* in the actor whose capability is given by *srcactorcap*. This count is stored by the kernel in the data pointed to by *seqnum*.

RETURN VALUE

Upon successful completion, *portMigrate* returns the local identifier of the port in the destination actor. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*srcactorcap* or *dstactorcap* is an inconsistent actor capability or *portli* is not a port of the source actor.
[K_EUNKNOWN]	*srcactorcap* or *dstactorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_ENOMEM]	The system is out of resources.
[K_EBUSYPORT]	Attempted to migrate a port to which a message handler is attached.

RESTRICTIONS

Messages that are sent using *ipcSend* are not reliably forwarded to the new location of the port.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portCreate`(2K) , `portEnable`(2K) , `grpPortInsert`(2K) , `ipcCall`(2K) , `svMsgHandler`(2K)

NAME

portUi, portLi – Get the unique identifier of a port, given its local identifier; Get the local identifier of a port, given its unique identifier

SYNOPSIS

```
#include <ipc/chIpc.h>
int portUi(KnUniqueId * portui, int portli);

int portLi(KnUniqueId * portui);
```

FEATURES

IPC

DESCRIPTION

The *portUi* call returns to the *KnUniqueId* structure given by *portui* , the unique identifier of the port whose local identifier is *portli* in the current actor (the port must be attached to the current actor).

The *portLi* call returns the local identifier in the current actor of the port whose unique identifier is pointed by *portui* (the port must be attached to the current actor).

RETURN VALUE

Upon successful completion, *portui* returns 0; *portLi* returns the local identifier of the port. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	For *portUi* : *portli* is not a valid port identifier within the current actor. For *portLi* : *portui* is not the unique identifier of a port attached to the current actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

portCreate(2K)

NAME

portMigrate, portGetSeqNum – Migrate a port; Get a port sequence number

SYNOPSIS

#include <ipc/chIpc.h>
int **portMigrate**(int *options*, KnCap * *srcactorcap*, int *portli*, KnCap * *dstactorcap*, KnEvtNum * *seqnum*);

int **portGetSeqNum**(KnCap * *srcactorcap*, int *portli*, KnEvtNum * *seqnum*);

FEATURES

IPC

DESCRIPTION

The *portMigrate* utility forces the port whose local identifier is *portli* in the actor whose capability is given by *srcactorcap* to migrate to the actor whose capability is given by *dstactorcap*. After the call, the port is attached to this destination actor. The local identifier of the portis returned by *portMigrate* to the destination actor.

If *srcactorcap* and/or *dstactorcap* is K_MYACTOR, the current actor is used as the source and/or destination of the migration.

The *options* , which describe the way in which the migration is performed may take one of the following values:

K_WITHMSGS	The messages queued behind the port follow the port during the migration.
K_KILLMSGS	The queued messages are deleted (and *ipcCall* (2K) transactions are aborted) before the migration.

The *seqnum* out parameter is a pointer to a count of messages that have been received on the source port, plus one additional increment for the migrate itself. The port message count is reset to 0 on the destination port.

If the port was inserted into port groups (using *grpPortInsert* (2K)) before the migration, the port remains a member of the same groups after the migration.

Ports that had been enabled (using *portEnable* (2K)) before the migration are disabled after the migration.

When a port migrates, any thread waiting for a message on that port exits its waiting state amd returns the K_ENOPORT error code.

A port may not be migrated if a message handler (see *svMsgHandler* (2K)) is attached to it. If such an attempt is made, the K_EBUSYPORT error code is returned.

The *portGetSeqNum* function gets the count of messages that have been received on the port whose local identifier is *portli* in the actor whose capability is given by *srcactorcap*. This count is stored by the kernel in the data pointed to by *seqnum*.

RETURN VALUE

Upon successful completion, *portMigrate* returns the local identifier of the port in the destination actor. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*srcactorcap* or *dstactorcap* is an inconsistent actor capability or *portli* is not a port of the source actor.
[K_EUNKNOWN]	*srcactorcap* or *dstactorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_ENOMEM]	The system is out of resources.
[K_EBUSYPORT]	Attempted to migrate a port to which a message handler is attached.

RESTRICTIONS

Messages that are sent using *ipcSend* are not reliably forwarded to the new location of the port.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portCreate`(2K) , `portEnable`(2K) , `grpPortInsert`(2K) , `ipcCall`(2K) , `svMsgHandler`(2K)

NAME

portPi – get and / or set the protection identifier of a port

SYNOPSIS

#include <ipc/chIpc.h>
int **portPi**(KnCap **actorcap*, int *portli*, KnProtId **oldpi*, KnProtId **newpi*);

FEATURES

IPC

DESCRIPTION

The *portPi* call gets and / or sets the protection identifier of the port whose local identifier is *portli* in the actor whose capability is given by *actorcap* (see *actorCreate*(2K)).

If *actorcap* is K_MYACTOR, the operation is applied to the current actor.

If *portli* is K_DEFAULTPORT, the actor's default port is used.

The *oldpi* and *newpi* fields are pointers to KnProtId structures whose members are the following:

```
unsigned short uid ;
unsigned short gid ;
unsigned short did ;
```

The user identifier is *uid*, the group identifier is *gid*, and the domain identifier is *did*.

The corresponding port's protection identifier is copied into the structure pointed to by *oldpi* in the client address space. The new corresponding context is copied from the structure pointed to by *newpi*.

If *oldpi* is a NULL pointer, the port's protection identifier is not copied to the caller's address space. If *newpi* is a NULL pointer, the port's protection identifier is not modified. The caller thread must be a *SUPERVISOR* thread (see *threadCreate*(2K)) or must belong to a *SYSTEM* actor (see *actorCreate*(2K)).

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_ENOPORT]	*portli* is not a valid port identifier in the actor.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `portCreate`(2K), `threadCreate`(2K), `actorCreate`(2K)

NAME

portUi, portLi – Get the unique identifier of a port, given its local identifier; Get the local identifier of a port, given its unique identifier

SYNOPSIS

```
#include <ipc/chIpc.h>
int portUi(KnUniqueId * portui, int portli);

int portLi(KnUniqueId * portui);
```

FEATURES

IPC

DESCRIPTION

The *portUi* call returns to the *KnUniqueId* structure given by *portui* , the unique identifier of the port whose local identifier is *portli* in the current actor (the port must be attached to the current actor).

The *portLi* call returns the local identifier in the current actor of the port whose unique identifier is pointed by *portui* (the port must be attached to the current actor).

RETURN VALUE

Upon successful completion, *portui* returns 0; *portLi* returns the local identifier of the port. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	For *portUi* : *portli* is not a valid port identifier within the current actor. For *portLi* : *portui* is not the unique identifier of a port attached to the current actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

portCreate(2K)

NAME

ptdErrnoAddr – return a thread-specific errno address

SYNOPSIS

```
#include <pd/chPd.h>
int *ptdErrnoAddr(void);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *ptdErrnoAddr* call returns the address of the private errno word of the calling thread.

It may be called either explicitly or implicitly from a thread-specific data destructor function.

This function is implemented as a macro.

A typical usage (in a multi-threaded POSIX environment) of *ptdErrnoAddr* is to define *errno* as *(*ptdErrnoAddr)()*.

RETURN VALUE

The function *ptdErrnoAddr* returns the thread-specific errno address.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ptdGet – return thread-specific value associated with key

SYNOPSIS

```
#include <pd/chPd.h>
void* ptdGet(PdKey key);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

ptdGet returns the value currently bound to *key* in the calling thread.

The effect of calling *ptdGet* with a *key* value not obtained from *ptdKeyCreate* or after *key* has been deleted with *ptdKeyDelete* is undefined.

ptdGet may be called either explicitly or implicitly from a thread-specific data destructor function.

This function is implemented as a macro.

RETURN VALUE

The function *ptdGet* returns the thread-specific data value associated with the given *key*. If no thread-specific data value has been associated with *key*, then the value NULL is returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ptdKeyCreate`(2K), `ptdKeyDelete`(2K), `ptdSet`(2K)

NAME

ptdKeyCreate – create a thread-specific data key

SYNOPSIS

```
#include <pd/chPd.h>
int ptdKeyCreate(PdKey *pkey, KnPdHdl destructor);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

ptdKeyCreate creates a thread-specific data key visible to all threads in the actor.

The created key is returned in **pkey*.

Keys provided by *ptdKeyCreate* are opaque indices used to locate thread-specific data. Although the same key may be used by different threads, the values bound to the key by *ptdSet* (2K) are maintained on a per-thread basis, and persist through the life of the calling thread.

Upon key creation, the value NULL shall be associated with the new key in all active threads. Upon thread creation, the value NULL shall be associated will all defined keys in the new thread.

An optional destructor function *destructor* may be associated with each key. At thread exit, if a key has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the function that is pointed to is called with the current value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

Destructor calls are repeated at as most *PD_DESTRUCTOR_ITERATIONS* (as defined in *pd/chPd.h*) times as necessary to reclaim all thread-specific data, in the case a destructor causes more thread-specific data to be created.

destructor will be called by *ptdThreadDelete*(2K) in user mode or by the Private Data Manager at thread termination in supervisor mode. Calling *ptdThreadDelete*(2K) in supervisor mode is useless (and harmless), since it is defined as an empty function.

RETURN VALUE

Upon successful completion, the newly created key is stored in **pkey*, and a value of 0 is returned. Otherwise, a positive error code is returned.

ERRORS

[PD_ENOKEY]	The system-imposed limit on the total number of keys per actor has been exceeded (see *PTD_KEYS_MAX* in *pd/chPd.h*).
[PD_ENOMEM]	Insufficient memory exists to create the key.
[PD_ESERVER]	The Private Data Manager is unreachable.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `ptdSet`(2K), `ptdKeyDelete`(2K)

NAME

ptdKeyDelete – delete a thread-specific data key

SYNOPSIS

```
#include <pd/chPd.h>
int ptdKeyDelete(PdKey key);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *ptdKeyDelete* call deletes a thread-specific data key previously returned by *ptdKeyCreate (2K).* The thread-specific data values associated with *key* need not be NULL at the time *ptdKeyDelete* is called. The application frees storage and performs any cleanup operations needed for data structures related to the deleted key or to associated thread-specific data. The cleanup can be performed before or after invoking *ptdKeyDelete.* Any attempt to use *key* after the call to *ptdKeyDelete* can produce unpredictable results.. There are no destructor functions invoked by *ptdKeyDelete.*

At actor destruction, all keys still allocated by an actor will silently be deleted.

RETURN VALUE

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

ERRORS

[PD_EINVAL]	The *key* value is invalid.
[PD_ESERVER]	The Private Data Manager is unreachable.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

ptdKeyCreate(2K)

NAME

ptdRemoteGet – return a thread-specific data value for another thread

SYNOPSIS

```
#include <pd/chPd.h>
int ptdRemoteGet(KnCap *actorcap, int threadli, PdKey key, void **pvalue);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *ptdRemoteGet* call gets the value currently bound to the specific *key* for the thread whose local indentifier is *threadli* in the actor whose capability is given by *actorcap*.

If *actorcap* is K_MYACTOR, the target thread must belong to the current actor. In this case, if *threadli* is K_MYSELF, the current thread is used.

The value is returned to **pvalue*. If no thread-specific data value is associated with *key*, then the value NULL is returned to **pvalue*.

In user mode, *actorcap* must represent the current actor.

RETURN VALUE

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

ERRORS

[PD_EINVAL]	The *key* value is invalid, or *threadli* does not belong to the target actor, or *actorcap* is an inconsistent actor capability.
[PD_ESERVER]	The Private Data Manager is unreachable.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ptdKeyCreate`(2K), `ptdGet`(2K), `ptdRemoteSet`(2K)

NAME

ptdRemoteSet – set a thread-specific data value for another thread

SYNOPSIS

```
#include <pd/chPd.h>
int ptdRemoteSet(KnCap *actorcap, int threadli, PdKey key, const void *value);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *ptdRemoteSet* call associates a thread-specific *value* with a *key* (obtained via a previous call to *ptdKeyCreate (2K)*) for the thread whose local identifier is *threadli* in the actor whose capability is given by *actorcap*.

If *actorcap* is K_MYACTOR, the thread used must belong to the current actor. In this case, if *threadli* is K_MYSELF, the current thread is used.

Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

In user mode, *actorcap* must represent the current actor.

RETURN VALUE

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

ERRORS

[PD_EINVAL]	The *key* value is invalid, or *threadli* does not belong to the target actor, or *actorcap* is an inconsistent actor capability.
[PD_ENOMEM]	There is insufficient memory to associate the value with the key.
[PD_ESERVER]	The Private Data Manager is unreachable.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ptdKeyCreate`(2K), `ptdSet`(2K)

NAME

ptdSet – set a thread-specific value

SYNOPSIS

#include <pd/chPd.h>
int **ptdSet**(PdKey *key*, const void **value*);

FEATURES

PRIVATE-DATA

DESCRIPTION

ptdSet associates a thread-specific *value* with a *key* obtained via a previous call to *ptdKeyCreate (2K)*. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

Calling *ptdSet* from a destructor may result in lost storage.

RETURN VALUE

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

ERRORS

[PD_EINVAL]	The *key* value is invalid.
[PD_ENOMEM]	Insufficient memory exists to associate the value with the key.
[PD_ESERVER]	The Private Data Manager is unreachable.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ptdKeyCreate`(2K)

NAME

ptdThreadDelete – delete all thread-specific values and call destructors

SYNOPSIS

```
#include <pd/chPd.h>
void ptdThreadDelete(void);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *ptdThreadDelete* call deletes the thread-specific values associated with all valid keys in the current user actor for the current thread.

For each non-NULL value, *ptdThreadDelete* calls the appropriate destructor before clearing the value.

In user mode, *ptdThreadDelete* should be called before thread deletion to enable destructors to be invoked.

In supervisor mode, *ptdThreadDelete* is an empty function. Destructors will be called at thread termination by the Private Data Manager.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ptdKeyCreate`(2K), `ptdSet`(2K)

NAME

ptdThreadId – return the thread ID

SYNOPSIS

```
#include <pd/chPd.h>
int ptdThreadId(void);
```

FEATURES

PRIVATE-DATA

DESCRIPTION

The *ptdThreadId* function returns the unique thread ID for the calling thread. The thread ID is equal to the thread local identifier as returned by *threadSelf*(2K).

It may be called either explicitly or implicitly from a thread-specific data destructor function.

This function is implemented as a macro.

RETURN VALUE

The *ptdThreadId* function returns the thread-specific unique ID.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadSelf`(2K)

NAME

rgnAllocate – allocate a region in an actor address space

SYNOPSIS

```
#include <mem/chMem.h>
int rgnAllocate(KnCap *actorcap, KnRgnDesc *Rgndesc);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

rgnAllocate creates a region in the target actor address space and maps some newly allocated volatile memory to the region.

The target actor is specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the region is allocated in the supervisor address space and is not attached to any particular supervisor actor; therefore, it cannot be implicitly deallocated by an *actorDelete*(2K) of a supervisor actor.

rgndesc points to a *KnRgnDesc* structure which contains the specification of the required region, and the members of which are the following:

```
VmAddr  startAddr ;
VmSize  size ;
VmFlags options ;
VmAddr  endAddr ;
void*   opaque1 ;
VmFlags opaque2 ;
```

If neither K_ANYWHERE nor K_RESTRICTIVE options (see below) are specified, *startAddr* is the region start address in the target address space. The region start address must be aligned with a virtual page boundary (see *vmPageSize*(2K)).

`size` is the size of the region in bytes. If the size is not page-aligned, the kernel rounds it up to the next virtual page boundary.

endAddr is only used with the K_RESTRICTIVE option (see below).

options is a combination of following flags:

K_ANYWHERE — The input value of *startAddr* is ignored, and the region is allocated wherever possible in the limits of the target address space as returned by *vmStat(2K)*. In this case the region's start address is returned in *startAddr*.

K_RESTRICTIVE — The region is allocated wherever possible between *startAddr* and *endAddr*. If *startAddr* is not page-aligned, the kernel rounds it up to the next virtual page boundary. If *endAddr* is not page-aligned the kernel rounds it down to the preceding virtual page boundary. The created

	region start address is returned in *startAddr*. If this flag is set, the K_ANYWHERE flag is ignored.
K_INHERITCOPY	If a *rgnDup(2K)* is subsequently applied, taking the target actor as the source, a copy of the region will be allocated in the destination address space by the *rgnDup* operation.
K_INHERITSHARE	If a *rgnDup(2K)* is subsequently applied, taking the target actor as the source, the region will be made shared by the *rgnDup* operation.
K_FILLZERO	The memory eventually allocated for the region is zero-filled. Otherwise, the initial state of the memory is undefined.
K_NODEMAND	Guarantees that no page faults occur during subsequent (legal) access to data within the region.
K_NOWAITFORMEMORY	Do not wait for memory to become available. If the K_NOWAITFORMEMORY and K_NODEMAND flags are set, the *rgnAllocate* fails when there is not enough physical memory, and returns the K_ENOMEM error. If only K_NOWAITFORMEMORY is set, possible memory underflow during a future on demand allocation provokes a memory underflow exception. Otherwise memory underflow blocks and waits for memory to become available.
K_NOSWAPOUT	Guarantees that no pageouts occur for the data within the region.
K_RESERVED	The region is a reserved region. If K_RESERVED option is set all other options, except of K_ANYWHERE or K_RESTRICTIVE ones, are ignored. Virtual-to-physical mappings (attributes as well as addresses) of a reserved region are managed from the kernel using *svPhysMap(9DKI)* call. A thread can access to any effectively mapped (valid) address within a reserved region, but any fault provokes an exception processing (see *svExcHandler(2K)*). Mapped portions of reserved regions can also be used as data buffers for IPC messages, *vmCopy*, *sgRead*, etc. A reserved region can be deallocated by a

	regular *rgnFree* or *actorDelete* operation. Any other kernel operation performed on a reserved region either has no effect (eg. *rgnSetProtect(2K)*), or returns K_ENOTIMP (eg. *vmFlush(2K)* or *rgnInitFromActor(2K)*).
K_STACK	Optimize future region extension toward low addresses (by default extension toward high addresses is optimized).
K_READABLE	Read access is permitted. The current implementation always permits read access for backward compatibility reasons.
K_WRITABLE	Write access is permitted.
K_EXECUTABLE	Execute access is permitted.
K_SUPERVISOR	The region is accessible only by supervisor threads.
K_NODUMP	The region will not be dumped if the target actor is core dumped.

Any access to a region conflicting with the region protections can potentially be reported by an exception (see *svExcHandler*(2K)) but the guarantes of such exceptions are machine-dependent. For instance, on any available paged memory hardware the kernel guarantees that a user-level write access to a read only region provokes a protection exception, but only on some of them the kernel guarantees to detect illegal access when a thread executes a region without K_EXECUTABLE attribute.

All flags specified in the *options* field as well as values specified by *opaque1* and *opaque2* fields compose region attributes. The attributes are stored in the kernel state associated with the region and can be consulted using *rgnStat(2K)* kernel call.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the provided arguments are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was given.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EROUND]	*rgndesc->startAddr* is not page-aligned.

[K_ESPACE]	Try to allocate a region outside the valid range for the address space of an actor as returned by *vmStat*(2K).
[K_ESPACE]	*rgndesc->size* is 0.
[K_EOVERLAP]	The K_ANYWHERE option was specified and there is not enough room available in the address space to allocate the region.
[K_EOVERLAP]	The K_RESTRICTIVE option was specified and there is not enough room in the target address range.
[K_EOVERLAP]	The allocated region overlaps an existing region.
[K_ENOMEM]	The system is out of resources.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

The K_RESTRICTIVE option is valid only in the MEM_PROTECTED and MEM_VIRTUAL modules. This option is ignored by the MEM_FLAT module.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnDup`(2K), `rgnFree`(2K), `rgnSetProtect`(2K), `rgnStat`(2K), `svExcHandler`(2K)

NAME

rgnDup – duplicate an actor address space

SYNOPSIS

```
#include <mem/chMem.h>
int rgnDup(KnCap *tgtactorcap, KnCap *srcactorcap, VmFlags flags);
```

FEATURES

MEM_VIRTUAL

DESCRIPTION

The *rgnDup* function duplicates regions from the address space specified by *srcactorcap* into the actor address space specified by *tgtactorcap*. If *srcactorcap* and/or *tgtactorcap* is K_MYACTOR, the address space of the current actor is used as the source and/or the destination, respectively.

For each region from the source actor *rgnDup* duplicates the region in the target actor address space with the same address, the same size and the same attributes (options).

The duplicate region can either share the data with the source region (map to the same segment of volatile memory as the source), or take a private volatile copy. The data is shared if the *flags* argument is K_INHERITSHARE or if *flags* is zero and the source region has the K_INHERITSHARE attribute. The data is copied if the *flags* argument is K_INHERITCOPY or if *flags* is zero and the source region has the K_INHERITCOPY attribute.

The region isn't duplicated if the *flag* argument is zero and the region has enither the K_INHERITSHARE nor the K_INHERITCOPY attribute.

Before the operation, the target actor address space must be empty.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the provided arguments are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability.
[K_EUNKNOWN]	*tgtactorcap* or *srcactorcap* does not specify a reachable actor.
[K_ENOEMPTY]	The target actor address space is not empty.
[K_ENOMEM]	The system is out of resources.
[K_EMAPPER]	The segment mapper doesn't respect the kernel/mapper protocol.

RESTRICTIONS

The target actor, the source actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnAllocate`(2K), `rgnInit`(2K), `rgnMap`(2K), `rgnSetInherit`(2K), `rgnSetProtect`(2K), `rgnStat`(2K)

NAME

rgnFree – deallocate regions of an actor address space

SYNOPSIS

```
#include <mem/chMem.h>
int rgnFree(KnCap *actorcap, KnRgnDes *rgndesc);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

rgnFree deallocates all regions from a given address range of the actor address space specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the supervisor address space is used.

rgndesc points to a *KnRgnDesc* structure described in *rgnAllocate (2K)*. If *options* is equal to K_FREEALL, all the actor's allocated regions are deallocated. Otherwise, *option* must be set to 0, and in that case the *startAddr* field specifies the starting address, and `size` specifies the size of the address range to be deallocated. The *endAddr* field is ignored.

If *actorcap* is equal to K_SVACTOR, the K_FREEALL flag cannot be used.

rgndesc->startAddr must be aligned with a page boundary (see *vmPageSize*(2K)). If *rgndesc->size* is not page-aligned, the kernel rounds it up to the next page boundary.

Since *rgndesc->startAddr* and *rgndesc->startAddr + rgndesc->size* do not necessarily match the boundaries of a region, a *rgnFree* can shrink the address ranges of some regions and even, in the case of a *rgnFree* in the middle of a region, create a new one. It explains why the K_ENOMEM error can be returned by a *rgnFree.*

Besides, the deallocated address range can contain holes, i.e. sub-ranges without regions mapped to.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the provided arguments are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was given. The value of *option is* not K_FREEALL or 0.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EROUND]	*rgndesc->startAddr* is not page-aligned.
[K_ENOMEM]	The system is out of resources.
[K_EMAPPER]	The segment mapper does not respect the kernel/mapper protocol

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnAllocate`(2K), `rgnMap`(2K), `rgnInit`(2K), `MpCreate`(2SEG), `MpRelease`(2SEG)

NAME

rgnInitFromActor – allocate a region in an actor address space and initialise it from another region

SYNOPSIS

#include <mem/chMem.h>
int **rgnInitFromActor**(KnCap *tgtactorcap, KnRgnDesc *tgtrgndesc, KnCap *srcactorcap, KnRgnDesc *srcrgndesc);

FEATURES

MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *rgnInitFromActor* function creates regions in a contiguous address range of the target actor address space. It then maps to these regions a copy of the segments and/or volatile memory which are already mapped to an address range of the source actor's address space.

The target and source actors are specified by *tgtactorcap* and *srcactorcap* - pointers to the capabilities of the target and source actors respectively. If *srcactorcap* and/or *tgtactorcap* is K_MYACTOR, the address space of the current actor is used as the source and/or the destination, respectively. If *srcactorcap* and/or *tgtactorcap* is K_SVACTOR, the supervisor address space is used as the source and/or the destination, respectively. Note that, if *tgtactorcap* is K_SVACTOR, the allocated regions aren't attached to any particular supervisor actor; therefore, they can't be implcitly deallocated by performing an *actorDelete*(2K) on a supervisor actor.

The *tgtrgndesc* field points to a *KnRgnDesc* structure containing the specification for the target address range and the attributes (options) for the regions to be created as described in *rgnAllocate (2K)*.

The *srcrgndesc* field points to another *KnRgnDesc* structure where only the *startAddr* and `size` fields are significant. It specifies the source range starting address and size respectively. The *srcrgndesc->size* must to be less than or equal to *tgtrgndesc->size*. The source address range cannot contain holes: any address from the source address range has to belong to a region.

The caller can specify any of the *tgtrgndesc->options* as described in *rgnAllocate (2K)* apart from K_RESERVED, which is prohibited. Note that the K_FILLZERO option is interpreted differently here: only the range from *tgtrgndesc->startAddr + srcrgndesc->size* to *tgtrgndesc->startAddr + tgtrgndesc->size - 1* will be zero-filled. All regions in the target address range will be created with the same attributes and will be differentiated by the objects (copies of particular segments or of particular volatile memory) mapped to them.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the provided arguments are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was given.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.

[K_EROUND]	*tgtrgndesc->startAddr* is not page-aligned.
[K_EROUND]	*srcrgndesc->startAddr* is not page-aligned.
[K_ESPACE]	Tried to create a region outside the valid range for the address space of an actor as returned by *vmStat*(2K).
[K_ESPACE]	*tgtrgndesc->size* is zero.
[K_EOVERLAP]	The K_ANYWHERE option was specified and there is not sufficient room available in the address space to create the regions.
[K_EOVERLAP]	The K_RESTRICTIVE option was specified and there is not sufficient room in the target address range.
[K_EOVERLAP]	The regions created overlap an existing region.
[K_ESPACE]	The source region is outside the valid range for the address space of an actor as returned by *vmStat*(2K).
[K_ESPACE]	*srcrgndesc->size* is zero.
[K_EADDR]	The source address range contains holes.
[K_ESIZE]	*srcrgndesc->size* is greater than *tgtrgndesc->size*.
[K_ENOMEM]	The system is out of resources.
[K_EMAPPER]	A segment mapper doesn't respect the kernel/mapper protocol.

RESTRICTIONS

The target actor, the source actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnAllocate`(2K), `rgnMap`(2K), `rgnStat`(2K)

NAME

rgnMapFromActor – create a region in an actor address space and map another region to it

SYNOPSIS

#include <mem/chMem.h>
int **rgnMapFromActor**(KnCap *_tgtactorcap_, KnRgnDesc *_tgtrgndesc_, KnCap *_srcactorcap_, KnRgnDesc *_srcrgndesc_);

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *rgnMapFromActor* system call creates regions in a contiguous address range of the target actor address space. It then maps the segments and/or volatile memory already mapped to an address range of the source actor address space to the newly created regions.

The target and source actors are specified by *tgtactorcap* and *srcactorcap* these are pointers to the capabilities of the target and source actors, respectively. If *srcactorcap* and/or *tgtactorcap* is K_MYACTOR, the address space of the current actor is used. If *tgtactorcap* is K_SVACTOR, the regions are allocated in the supervisor address space and are not attached to any particular supervisor actor. They cannot therefore be implicitly deallocated by performing an *actorDelete*(2K) of a supervisor actor.

The *tgtrgndesc* pointer points to a *KnRgnDesc* structure containing the specification for the target address range and the attributes (options) for the regions to be created. This is the same as described in *rgnAllocate(2K)*, with the exception that K_FILLZERO is ignored and the K_RESERVED option is prohibited. All regions in the target address range will be created with the same attributes, and will be differentiated by the objects (particular segments or particular volatile memory) mapped to them.

The *srcrgndesc* pointer points to another *KnRgnDesc* structure, where only the *startAddr* field is significant. It specifies the starting address of the source address range. This starting address must be aligned on a virtual page boundary. The size of the source address range is equal to the size of the target address range.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was given.
[K_EUNKNOWN]	*srcactorcap or tgtactorcap* does not specify a reachable actor.
[K_EROUND]	*tgtrgndesc->startAddr* is not page-aligned.
[K_EROUND]	*srcrgndesc->startAddr* is not page-aligned.

[K_ESPACE]	Tried to create a region outside the valid range for the address space of an actor as returned by *vmStat*(2K).
[K_ESPACE]	*tgtrgndesc->size* is 0.
[K_EOVERLAP]	The K_ANYWHERE option was specified and there is not enough room available in the address space to create the regions.
[K_EOVERLAP]	The K_RESTRICTIVE option was specified and there is not enough room in the target address range.
[K_EOVERLAP]	The regions created overlap an existing region.
[K_ESPACE]	The source region is outside the valid range for the address space of an actor as returned by *vmStat*(2K).
[K_ESIZE]	If one of these three options are wrong K_WRITABLE,K_READABLE,K_executable
[K_ESPACE]	*srcrgndesc->size* is 0.
[K_EADDR]	The source address range contains holes.
[K_ENOMEM]	The system is out of resources.
[K_EMAPPER]	A segment mapper does not respect the kernel / mapper protocol.
[K_EMAPPER]	A segment mapper has detected a mapper-level error.

RESTRICTIONS

The target actor, the source actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnAllocate`(2K), `rgnInit`(2K), `rgnMap`(2K), `rgnStat`(2K), `vmStat`(2K), `MpGetAccess`(2SEG)

NAME

rgnPhysMap – create a region in an actor address space and map (on demand) to it physical memory specified by the caller

SYNOPSIS

#include <mem/chMem.h>
int **rgnPhysMap**(KnCap *_actorcap_, KnRgnDesc *_rgndesc_, KnMemoryHandler _handler_, void *_opaque_);

FEATURES

MEM_VIRTUAL

DESCRIPTION

The *rgnPhysMap* function creates a region in the address space of the target actor.

The target actor is specified by *actorcap* - a pointer to the target actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the region is allocated in the supervisor address space and isn't attached to any particular supervisor actor; therefore, it can't be implicitly deallocated by performing an *actorDelete*(2K) on a supervisor actor.

The *rgndesc* field points to a *KnRgnDesc* structure containing the specification of the region to be created as described in *rgnAllocate (2K)*.

The caller can specify any of the *rgndesc->options* as described in *rgnAllocate (2K)*; with the exceptions that the K_FILLZERO flag is ignored, and the K_RESERVED flag is prohibited.

The *handler* argument specifies a memory handler which is attached to the newly created region. When access to a virtual page of the region is required (for example, a fault happens), the kernel invokes the memory handler. The handler returns the physical address to be mapped and the mapping attributes to the required virtual address.

The memory handler interface is machine-dependent. The kernel on i386/i486 hardware defines the following handler interface:

```
#include <mem/chMem.h>
#include <mem/f_chMem.h>

int handler (opaque, offset, paddr, access, cntlBits)
   void*        opaque ;
   VmOffset     offset ;
   PhysAddr     *paddr ;
   VmFlags      access ;
   PteCntlBits  *cntlBits ;
```

Where *opaque* is an input argument specified by the *opaque* argument of *rgnPhysMap*.;*offset* is another input argument which specifies the required page offset within the region. The *paddr* output argument specifies the physical address to be mapped. The last input argument is *access*, which defines the type of access rights required. If the K_WRITABLE bit is set in the *access* argument, write access is required. The second output argument, *cntlBits*specifies the

mapping control bits as defined in the Intel386/Intel486 Programmer's Reference Manual. The valid bits are the following:

PTE_PRESENT (0x01)

PTE_READ_WRITE (0x02)

PTE_USER_SUPERVISOR (0x04)

PTE_WRITE_TRANSPARENT (0x08)

PTE_CACHE_DISABLE (0x10)

As the the memory handler is designed for mapping I/O devices it doesn't include a feature to detect and write back the data modified in the region.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the provided arguments are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was given.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EROUND]	*rgndesc->startAddr* is not page-aligned.
[K_ESPACE]	Triedto create a region outside the valid range for the address space of an actor as returned by *vmStat*(2K).
[K_ESPACE]	*rgndesc->size* is zero.
[K_EOVERLAP]	The K_ANYWHERE option was specified and there is insufficient room available in the address space to create the region.
[K_EOVERLAP]	The K_RESTRICTIVE option was specified and there is insufficient room in the target address range.
[K_EOVERLAP]	The created region overlaps an existing region.
[K_ENOMEM]	The system is out of resources.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnAllocate`(2K), `rgnInit`(2K), `rgnMap`(2K), `rgnStat`(2K)

NAME

rgnSetInherit, rgnSetPaging, rgnSetOpaque – Change inheritance options associated with a region; Change paging options associated with a region; Change inheritance options associated with a region; Change opaque values associated with a region

SYNOPSIS

```
#include <mem/chMem.h>
int rgnSetInherit(KnCap * actorcap, KnRgnDesc * rgndesc);

int rgnSetPaging(KnCap * actorcap, KnRgnDesc * rgndesc);

int rgnSetOpaque(KnCap * actorcap, KnRgnDesc * rgndesc);
```

FEATURES

MEM_VIRTUAL

DESCRIPTION

These calls allow you to change the attributes of all regions from the target address range of the target actor address space.

The target actor is specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the supervisor address space is used.

The *rgndesc* pointer points to a *KnRgnDesc* structure which contains the specification of the target address range and the new values of the attributes. Only the following members of the structure are used:

```
VmAddr    startAddr ;
VmSize    size ;
VmFlags   options ;
void*     opaque1 ;
VmFlags   opaque2 ;
```

The *startAddr* field gives the start of the target address range and the `size` field gives its size; *rgndesc->startAddr* must be aligned with a virtual page boundary (see *vmPageSize* (2K)). If *rgndesc->size* is not page-aligned the kernel rounds it up to the next virtual page boundary.

The *options* field gives the new attributes of the regions from the target address range.

The target address range can contain holes; sub-ranges without regions mapped to them.

If *rgndesc->startAddr* is inside a region, the implementation replaces the original region with two regions. The first region conserves the old attributes and terminates at *rgndesc->startAddr - 1*. The second region takes the new attributes and starts immediately after the first one.

If *rgndesc->startAddr + rgndesc->size* is inside a region, the implementation replaces the original region with three regions. The first one conserves the old attributes and terminates at *rgnDesc->startAddr -1*. The second region takes the

new attributes and terminates at *rgndesc->startAddr + rgndesc->size - 1*. The third region conserves the old attributes and starts immediately after the second one.

An attribute modification can sometimes replace a number of consecutive regions with a single one.

The *rgnSetInherit* command replaces the values of the K_INHERITCOPY and K_INHERITSHARE attributes with the corresponding values from the *options* field of *rgndesc*.

The *rgnSetPaging* command replaces the values of the K_NODEMAND and K_NOWAITFORMEMORY attributes with the corresponding values from the *options* field of *rgndesc*.

The *rgnSetOpaque* command replaces the opaque values associated with the region with the new opaque values of *rgndesc*.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was given.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EROUND]	*rgndesc->startAddr* is not page-aligned.
[K_ENOMEM]	The system is out of resources.
[K_ENOMEM]	Some or all of the memory identified by the operation could not be locked when K_NODEMAND and K_NOWAITFORMEMORY had both been specified.
[K_EMAPPER]	The segment mapper has detected a mapper-level error.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnAllocate`(2K)

NAME

rgnSetInherit, rgnSetPaging, rgnSetOpaque – Change inheritance options associated with a region; Change paging options associated with a region; Change inheritance options associated with a region; Change opaque values associated with a region

SYNOPSIS

```
#include <mem/chMem.h>
int rgnSetInherit(KnCap * actorcap, KnRgnDesc * rgndesc);

int rgnSetPaging(KnCap * actorcap, KnRgnDesc * rgndesc);

int rgnSetOpaque(KnCap * actorcap, KnRgnDesc * rgndesc);
```

FEATURES

MEM_VIRTUAL

DESCRIPTION

These calls allow you to change the attributes of all regions from the target address range of the target actor address space.

The target actor is specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the supervisor address space is used.

The *rgndesc* pointer points to a *KnRgnDesc* structure which contains the specification of the target address range and the new values of the attributes. Only the following members of the structure are used:

```
VmAddr    startAddr ;
VmSize    size ;
VmFlags   options ;
void*     opaque1 ;
VmFlags   opaque2 ;
```

The *startAddr* field gives the start of the target address range and the `size` field gives its size; *rgndesc->startAddr* must be aligned with a virtual page boundary (see *vmPageSize* (2K)). If *rgndesc->size* is not page-aligned the kernel rounds it up to the next virtual page boundary.

The *options* field gives the new attributes of the regions from the target address range.

The target address range can contain holes; sub-ranges without regions mapped to them.

If *rgndesc->startAddr* is inside a region, the implementation replaces the original region with two regions. The first region conserves the old attributes and terminates at *rgndesc->startAddr - 1*. The second region takes the new attributes and starts immediately after the first one.

If *rgndesc->startAddr + rgndesc->size* is inside a region, the implementation replaces the original region with three regions. The first one conserves the old attributes and terminates at *rgnDesc->startAddr -1*. The second region takes the

new attributes and terminates at *rgndesc->startAddr + rgndesc->size - 1*. The third region conserves the old attributes and starts immediately after the second one.

An attribute modification can sometimes replace a number of consecutive regions with a single one.

The *rgnSetInherit* command replaces the values of the K_INHERITCOPY and K_INHERITSHARE attributes with the corresponding values from the *options* field of *rgndesc*.

The *rgnSetPaging* command replaces the values of the K_NODEMAND and K_NOWAITFORMEMORY attributes with the corresponding values from the *options* field of *rgndesc*.

The *rgnSetOpaque* command replaces the opaque values associated with the region with the new opaque values of *rgndesc*.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was given.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EROUND]	*rgndesc->startAddr* is not page-aligned.
[K_ENOMEM]	The system is out of resources.
[K_ENOMEM]	Some or all of the memory identified by the operation could not be locked when K_NODEMAND and K_NOWAITFORMEMORY had both been specified.
[K_EMAPPER]	The segment mapper has detected a mapper-level error.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnAllocate`(2K)

NAME

rgnSetInherit, rgnSetPaging, rgnSetOpaque – Change inheritance options associated with a region; Change paging options associated with a region; Change inheritance options associated with a region; Change opaque values associated with a region

SYNOPSIS

```
#include <mem/chMem.h>
int rgnSetInherit(KnCap * actorcap, KnRgnDesc * rgndesc);

int rgnSetPaging(KnCap * actorcap, KnRgnDesc * rgndesc);

int rgnSetOpaque(KnCap * actorcap, KnRgnDesc * rgndesc);
```

FEATURES

MEM_VIRTUAL

DESCRIPTION

These calls allow you to change the attributes of all regions from the target address range of the target actor address space.

The target actor is specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the supervisor address space is used.

The *rgndesc* pointer points to a *KnRgnDesc* structure which contains the specification of the target address range and the new values of the attributes. Only the following members of the structure are used:

```
VmAddr    startAddr ;
VmSize    size ;
VmFlags   options ;
void*     opaque1 ;
VmFlags   opaque2 ;
```

The *startAddr* field gives the start of the target address range and the `size` field gives its size; *rgndesc->startAddr* must be aligned with a virtual page boundary (see *vmPageSize* (2K)). If *rgndesc->size* is not page-aligned the kernel rounds it up to the next virtual page boundary.

The *options* field gives the new attributes of the regions from the target address range.

The target address range can contain holes; sub-ranges without regions mapped to them.

If *rgndesc->startAddr* is inside a region, the implementation replaces the original region with two regions. The first region conserves the old attributes and terminates at *rgndesc->startAddr - 1*. The second region takes the new attributes and starts immediately after the first one.

If *rgndesc->startAddr + rgndesc->size* is inside a region, the implementation replaces the original region with three regions. The first one conserves the old attributes and terminates at *rgnDesc->startAddr -1*. The second region takes the

new attributes and terminates at *rgndesc->startAddr + rgndesc->size - 1*. The third region conserves the old attributes and starts immediately after the second one.

An attribute modification can sometimes replace a number of consecutive regions with a single one.

The *rgnSetInherit* command replaces the values of the K_INHERITCOPY and K_INHERITSHARE attributes with the corresponding values from the *options* field of *rgndesc*.

The *rgnSetPaging* command replaces the values of the K_NODEMAND and K_NOWAITFORMEMORY attributes with the corresponding values from the *options* field of *rgndesc*.

The *rgnSetOpaque* command replaces the opaque values associated with the region with the new opaque values of *rgndesc*.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was given.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EROUND]	*rgndesc->startAddr* is not page-aligned.
[K_ENOMEM]	The system is out of resources.
[K_ENOMEM]	Some or all of the memory identified by the operation could not be locked when K_NODEMAND and K_NOWAITFORMEMORY had both been specified.
[K_EMAPPER]	The segment mapper has detected a mapper-level error.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnAllocate`(2K)

NAME

rgnSetProtect – change protection options associated with a region

SYNOPSIS

#include <mem/chMem.h>
int **rgnSetProtect**(KnCap **actorcap*, KnRgnDesc **rgndesc*);

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *rgnSetProtect* function allows you to change the protection attributes of all regions of the target address range of the target actor address space.

The target actor is specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the supervisor address space is used.

The *rgndesc* pointer points to a *KnRgnDesc* structure which contains the specification of the target address range and the new values of the attributes. Only the following members of the structure are used:

```
VmAddr   startAddr ;
VmSize   size ;
VmFlags  options ;
```

The *startAddr* field gives the start of the target address range and the `size` field gives its size; *rgndesc->startAddr* must be aligned with a page boundary (see *vmPageSize*(2K)). If *rgndesc->size* is not page-aligned the kernel rounds it up to the next virtual page boundary.

The *options* field gives the new attributes of the regions from the target address range.

The target address range can contain holes; sub-ranges without regions mapped to them.

If *rgndesc->startAddr* is inside a region, the implementation replaces the original region with two regions. The first region conserves the old attributes and terminates at *rgndesc->startAddr - 1*. The second region takes the new attributes and starts immediately after the first one.

If *rgndesc->startAddr + rgndesc->size* is inside a region, the implementation replaces the original region with three regions. The first one conserves the old attributes and terminates at *rgndesc->startAddr -1*. The second region takes the new attributes and terminates at *rgndesc->startAddr + rgndesc->size - 1*. The third region conserves the old attributes and starts immediately after the second one.

An attribute modification can sometimes replace a number of consecutive regions with a single one.

The values for the *options* field of *rgndesc* parameter must be among `K_READABLE`, `K_WRITABLE`, `K_EXECUTABLE` and `K_SUPERVISOR`.

The `rgnSetProtect` command replaces the values of the `K_READABLE`, `K_WRITABLE`, `K_EXECUTABLE` and `K_SUPERVISOR` attributes with the corresponding values from the *options* field of *rgndesc*.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguements provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability or invalid options were given.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EROUND]	*rgndesc->startAddr* is not page-aligned.
[K_ENOMEM]	The system is out of resources.
[K_EMAPPER]	The segment mapper has detected a mapper-level error.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnAllocate`(2K), `MpGetAcess`(2SEG)

NAME

rgnStat – get the statistics of a region of an actor address space

SYNOPSIS

#include <mem/chMem.h>
int **rgnStat**(KnCap **actorcap*, KnRgnDesc **rgndesc*, KnRgnStat **stat*, unsigned *buffsize*);

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *rgnStat* function gets the status of regions from the given address range of the target actor address space.

The target actor is specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If the target actor is a user actor *rgnStat* takes all regions of the target address range into account. If the target actor is a supervisor actor *rgnStat* only takes into account the regions attached to the target supervisor actor. If *actorcap* is K_SVACTOR, *rgnStat* takes all regions of the target supervisor address range into account.

The *rgndesc* pointer points to a *KnRgnDesc* structure which contains the specification of the target address range. Only the following members of the structure are used:

```
VmAddr   startAddr ;
VmSize   size ;
VmFlags  options
```

The *startAddr* field gives the start of the target address range and the `size` field gives its size.

The *stat* argument points to a buffer in which the description of the region will be returned. The *buffsize* argument gives the size of the buffer. If the K_USERSPACE flag is set in *rgndesc->option* the buffer is in the current user address space; otherwise the buffer is in the caller (current user or supervisor) address space.

The statuses are returned as a number of items, each one describing a particular sub-range of the target address range. The *KnRgnStat* structure defines the fields of each item, as follows:

```
VmAddr    startAddr ;
VmSize    size ;
VmFlags   options ;
void*     opaque1 ;
VmFlags   opaque2 ;
VmOffset  offset ;
KnLcStat  lcstat ;
```

The *startAddr* field is the starting address of the sub-range described by the item and `size` is size of the sub-range. There is exactly one item per region which has a non-empty intersection with the target address range. The items are sorted by increasing order of *startAddr*. If *rgndesc->startAddr* is inside a region, the *startAddr*

of the first item is equal to *rgndesc->startAddr*. If *rgndesc->startAddr + rgndesc->size* is inside a region, the `size` of the last item is equal to *(rgndesc->startAddr + rgndesc->size) - rgn_start_addr*, where *rgn_start_addr* is the start address of the last region of the target range. Otherwise, an item describes precisely one region.

The *options* field defines the region options as specified in *rgnAllocate(2K)* .

(The next four fields are only meaninful if virtual memory management is being used)

The values *opaque1* and *opaque2* define the opaque values associated with the region.

The *offset* value defines the offset of the item's *startAddr* within the mapped segment.

If the K_RGNFULLSTAT flag is set in the *rgndesc->options* field, the *lcstat* field contains the following status information about the range of the local cache mapped to the item's adress range:

```
VmSize  physMem ;
VmSize  lockMem ;
KnCap   segcap;
KnCap   lccap ;
```

The *physMem* field contains the size (in bytes) of the physical memory mapped to the address range.

The *lockMem* field contains the size (in bytes) of no-demand physical memory mapped to the address range.

If the target local cache corresponds to an external segment, the *segcap* field contains the capability of the segment; otherwise the field contains zero.

If the target local cache can be designated by a capability, the *lccap* field contains the capability of the local cache; otherwise the field contains zero.

RETURN VALUE

If successful, *rgnStat* returns the number of items needed to describe the target address range, otherwise a negative error code is returned. When the MEM_FLAT module is enabled and the caller is a supervisor actor (K_SVACTOR), this primitive returns 0. In this case no regions are available.

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was given.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `rgnAllocate`(2K)

NAME

rtMutexInit, rtMutexGet, rtMutexRel, rtMutexTry – Initialize a realtime mutex; Acquire a realtime mutex; Release a realtime mutex; Try to acquire a realtime mutex

SYNOPSIS

```
#include <sync/chRtMutex.h>
int rtMutexInit(KnRtMutex * mutex);

int rtMutexGet(KnRtMutex * mutex);

int rtMutexRel(KnRtMutex * mutex);

int rtMutexTry(KnRtMutex * mutex);
```

FEATURES

RTMUTEX

DESCRIPTION

Realtime mutexes are binary semaphores used to protect shared data from concurrent access: a realtime mutex is a two state variable - free or locked. When a thread is blocking higher priority threads because it owns one or more realtime mutexes, it executes either its highest prioirty, or the priority of the highest priority thread waiting on any of the realtime mutexes owned by the thread.

Realtime mutexes are *KnRtMutex* structures allocated in user memory.

The *rtMutexInit* function initializes the realtime mutex whose address is *mutex*. The mutex is initialized as free.

The *rtMutexGet* function is used to acquire a realtime mutex. If the mutex is free, it becomes locked and the caller continues to execute normally. If the mutex is locked, the caller is blocked. The owner thread inherits the priority level of the calling thread as long as it continues to own the mutex. The implementation updates its execution prioirty to the maximum of its assigned priority plus all its inherited priority. Futhermore, if the owner thread itself becomes blocked on another realtime mutex, the same prioirty inheritance effect is propagated recursively to this other owner thread.

The *rtMutexRel* function is used to release a realtime mutex. If threads are blocked behind the mutex, the thread with the highest priority is awakened. The priority of the calling thread can be reduced by using *rtMutexRel* .

The *rtMutexTry* function is an attempt to acquire a realtime mutex: it has the same effect as *rtMutexGet*, except that if the mutex is locked, the thread is not blocked (a return code is provided).

Any priority adjustments caused by priority inheritance are transparent and are not visible through the *threadScheduler* interface. In other words, the priority value returned from *threadScheduler* will be the value specified by the most recent *threadScheduler* or *threadCreate* call.

A blocking *rtMutexGet* is *NONABORTABLE* (see *threadAbort* (2K)).

RETURN VALUE

The *rtMutexTry* function returns 0 if the mutex was already locked, 1 if it was free. Upon successful completion of *rtMutexInit*, *rtMutexGet* and *rtMutexRel*, 0 is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT] Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mutexInit`(2K)

NAME

rtMutexInit, rtMutexGet, rtMutexRel, rtMutexTry – Initialize a realtime mutex; Acquire a realtime mutex; Release a realtime mutex; Try to acquire a realtime mutex

SYNOPSIS

```
#include <sync/chRtMutex.h>
int rtMutexInit(KnRtMutex * mutex);

int rtMutexGet(KnRtMutex * mutex);

int rtMutexRel(KnRtMutex * mutex);

int rtMutexTry(KnRtMutex * mutex);
```

FEATURES

RTMUTEX

DESCRIPTION

Realtime mutexes are binary semaphores used to protect shared data from concurrent access: a realtime mutex is a two state variable - free or locked. When a thread is blocking higher priority threads because it owns one or more realtime mutexes, it executes either its highest prioirty, or the priority of the highest priority thread waiting on any of the realtime mutexes owned by the thread.

Realtime mutexes are *KnRtMutex* structures allocated in user memory.

The *rtMutexInit* function initializes the realtime mutex whose address is *mutex*. The mutex is initialized as free.

The *rtMutexGet* function is used to acquire a realtime mutex. If the mutex is free, it becomes locked and the caller continues to execute normally. If the mutex is locked, the caller is blocked. The owner thread inherits the priority level of the calling thread as long as it continues to own the mutex. The implementation updates its execution prioirty to the maximum of its assigned priority plus all its inherited priority. Futhermore, if the owner thread itself becomes blocked on another realtime mutex, the same prioirty inheritance effect is propagated recursively to this other owner thread.

The *rtMutexRel* function is used to release a realtime mutex. If threads are blocked behind the mutex, the thread with the highest priority is awakened. The priority of the calling thread can be reduced by using *rtMutexRel* .

The *rtMutexTry* function is an attempt to acquire a realtime mutex: it has the same effect as *rtMutexGet*, except that if the mutex is locked, the thread is not blocked (a return code is provided).

Any priority adjustments caused by priority inheritance are transparent and are not visible through the *threadScheduler* interface. In other words, the priority value returned from *threadScheduler* will be the value specified by the most recent *threadScheduler* or *threadCreate* call.

A blocking *rtMutexGet* is *NONABORTABLE* (see *threadAbort* (2K)).

RETURN VALUE

The *rtMutexTry* function returns 0 if the mutex was already locked, 1 if it was free. Upon successful completion of *rtMutexInit*, *rtMutexGet* and *rtMutexRel*, 0 is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT] Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mutexInit`(2K)

NAME

rtMutexInit, rtMutexGet, rtMutexRel, rtMutexTry – Initialize a realtime mutex; Acquire a realtime mutex; Release a realtime mutex; Try to acquire a realtime mutex

SYNOPSIS

#include <sync/chRtMutex.h>
int **rtMutexInit**(KnRtMutex * *mutex*);

int **rtMutexGet**(KnRtMutex * *mutex*);

int **rtMutexRel**(KnRtMutex * *mutex*);

int **rtMutexTry**(KnRtMutex * *mutex*);

FEATURES

RTMUTEX

DESCRIPTION

Realtime mutexes are binary semaphores used to protect shared data from concurrent access: a realtime mutex is a two state variable - free or locked. When a thread is blocking higher priority threads because it owns one or more realtime mutexes, it executes either its highest prioirty, or the priority of the highest priority thread waiting on any of the realtime mutexes owned by the thread.

Realtime mutexes are *KnRtMutex* structures allocated in user memory.

The *rtMutexInit* function initializes the realtime mutex whose address is *mutex*. The mutex is initialized as free.

The *rtMutexGet* function is used to acquire a realtime mutex. If the mutex is free, it becomes locked and the caller continues to execute normally. If the mutex is locked, the caller is blocked. The owner thread inherits the priority level of the calling thread as long as it continues to own the mutex. The implementation updates its execution prioirty to the maximum of its assigned priority plus all its inherited priority. Futhermore, if the owner thread itself becomes blocked on another realtime mutex, the same prioirty inheritance effect is propagated recursively to this other owner thread.

The *rtMutexRel* function is used to release a realtime mutex. If threads are blocked behind the mutex, the thread with the highest priority is awakened. The priority of the calling thread can be reduced by using *rtMutexRel* .

The *rtMutexTry* function is an attempt to acquire a realtime mutex: it has the same effect as *rtMutexGet*, except that if the mutex is locked, the thread is not blocked (a return code is provided).

Any priority adjustments caused by priority inheritance are transparent and are not visible through the *threadScheduler* interface. In other words, the priority value returned from *threadScheduler* will be the value specified by the most recent *threadScheduler* or *threadCreate* call.

A blocking *rtMutexGet* is *NONABORTABLE* (see *threadAbort* (2K)).

RETURN VALUE

The *rtMutexTry* function returns 0 if the mutex was already locked, 1 if it was free. Upon successful completion of *rtMutexInit*, *rtMutexGet* and *rtMutexRel*, 0 is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT] Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mutexInit`(2K)

NAME

rtMutexInit, rtMutexGet, rtMutexRel, rtMutexTry – Initialize a realtime mutex; Acquire a realtime mutex; Release a realtime mutex; Try to acquire a realtime mutex

SYNOPSIS

#include <sync/chRtMutex.h>
int **rtMutexInit**(KnRtMutex * *mutex*);

int **rtMutexGet**(KnRtMutex * *mutex*);

int **rtMutexRel**(KnRtMutex * *mutex*);

int **rtMutexTry**(KnRtMutex * *mutex*);

FEATURES

RTMUTEX

DESCRIPTION

Realtime mutexes are binary semaphores used to protect shared data from concurrent access: a realtime mutex is a two state variable - free or locked. When a thread is blocking higher priority threads because it owns one or more realtime mutexes, it executes either its highest prioirty, or the priority of the highest priority thread waiting on any of the realtime mutexes owned by the thread.

Realtime mutexes are *KnRtMutex* structures allocated in user memory.

The *rtMutexInit* function initializes the realtime mutex whose address is *mutex*. The mutex is initialized as free.

The *rtMutexGet* function is used to acquire a realtime mutex. If the mutex is free, it becomes locked and the caller continues to execute normally. If the mutex is locked, the caller is blocked. The owner thread inherits the priority level of the calling thread as long as it continues to own the mutex. The implementation updates its execution prioirty to the maximum of its assigned priority plus all its inherited priority. Futhermore, if the owner thread itself becomes blocked on another realtime mutex, the same prioirty inheritance effect is propagated recursively to this other owner thread.

The *rtMutexRel* function is used to release a realtime mutex. If threads are blocked behind the mutex, the thread with the highest priority is awakened. The priority of the calling thread can be reduced by using *rtMutexRel* .

The *rtMutexTry* function is an attempt to acquire a realtime mutex: it has the same effect as *rtMutexGet*, except that if the mutex is locked, the thread is not blocked (a return code is provided).

Any priority adjustments caused by priority inheritance are transparent and are not visible through the *threadScheduler* interface. In other words, the priority value returned from *threadScheduler* will be the value specified by the most recent *threadScheduler* or *threadCreate* call.

A blocking *rtMutexGet* is *NONABORTABLE* (see *threadAbort* (2K)).

RETURN VALUE

The *rtMutexTry* function returns 0 if the mutex was already locked, 1 if it was free. Upon successful completion of *rtMutexInit*, *rtMutexGet* and *rtMutexRel*, 0 is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT] Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mutexInit`(2K)

NAME

schedAdmin – scheduling classes administration

SYNOPSIS

```
#include <sched/chSched.h>
#include <sched/chRt.h>
#include <sched/chTs.h>
int schedAdmin(void *adminparam);
```

FEATURES

SCHED_CLASS

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *schedAdmin* function is used to administer the attributes of the scheduling classes CLASS_RT and CLASS_TS, within the SCHED_CLASS scheduler.

It provides the necessary functionality to implement the CHORUS/MiX V.4 Process Manager. It is not intended for general use by other applications.

RETURN VALUE

Upon successful completion, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL] Invalid argument.

RESTRICTION

The *schedAdmin* function is restricted to supervisor threads.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

threadScheduler(2K)

NAME

semInit, semP, semV – initialize a semaphore; wait on a semaphore; signal a semaphore

SYNOPSIS

```
#include <sync/chSem.h>
int semInit(KnSem * sem, unsigned int count);

int semP(KnSem * sem, KnTimeVal * waitLimit);

int semV(KnSem * sem);
```

FEATURES

SEM

DESCRIPTION

Semaphores are *KnSem* structures allocated in the user memory.

semInit initializes the semaphore the address of which is *sem*. *count* is the initial positive value given to the semaphore counter.

Unless they are shared between two or more actors, statically allocated semaphores can be initialized using the *K_KNSEM_INITIALIZER(cnt)* macro, where *cnt* is the initial positive value given to the semaphore counter. This macro is used as follows:

```
KnSem mySem = K_KNSEM_INITIALIZER(cnt) ;
```

semInit is used to initialize a semaphore that is shared between two or more actors.

semP decrements the semaphore counter; if the counter reaches a strictly negative value, the calling thread is blocked, according to the options described by *waitLimit* in *intro(2K)*. *waitLimit* is a pointer to a *KnTimeVal* structure containing a timeout interval as described in *sysTime(2K)*.

semV increments the counter. If the new value is less than or equal to 0, the thread that has been blocked behind the semaphore for the longest time is awakened.

As semaphore structures are allocated in the client memory, the number of semaphores used by an application is not limited. Any modification to the semaphore structure while the semaphore is in use will cause unpredictable synchronization behavior in the application.

A blocking *semP* is *ABORTABLE* (see *threadAbort* (2K)).

RESTRICTIONS

A user application and a supervisor application may not share a semaphore.

On the contrary, two user applications may share a semaphore by mapping it in both user address spaces.

RETURN VALUE

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

ERRORS

[K_EINVAL]	The *waitLimit* is not a valid *KnTimeVal* .

[K_EABORT] *semP* has been aborted.

[K_ETIMEOUT] The timeout occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mutexInit`(2K)

NAME

semInit, semP, semV – initialize a semaphore; wait on a semaphore; signal a semaphore

SYNOPSIS

```
#include <sync/chSem.h>
int semInit(KnSem * sem, unsigned int count);

int semP(KnSem * sem, KnTimeVal * waitLimit);

int semV(KnSem * sem);
```

FEATURES

SEM

DESCRIPTION

Semaphores are *KnSem* structures allocated in the user memory.

semInit initializes the semaphore the address of which is *sem*. *count* is the initial positive value given to the semaphore counter.

Unless they are shared between two or more actors, statically allocated semaphores can be initialized using the *K_KNSEM_INITIALIZER(cnt)* macro, where *cnt* is the initial positive value given to the semaphore counter. This macro is used as follows:

```
KnSem mySem = K_KNSEM_INITIALIZER(cnt) ;
```

semInit is used to initialize a semaphore that is shared between two or more actors.

semP decrements the semaphore counter; if the counter reaches a strictly negative value, the calling thread is blocked, according to the options described by *waitLimit* in *intro(2K)*. *waitLimit* is a pointer to a *KnTimeVal* structure containing a timeout interval as described in *sysTime(2K)*.

semV increments the counter. If the new value is less than or equal to 0, the thread that has been blocked behind the semaphore for the longest time is awakened.

As semaphore structures are allocated in the client memory, the number of semaphores used by an application is not limited. Any modification to the semaphore structure while the semaphore is in use will cause unpredictable synchronization behavior in the application.

A blocking *semP* is *ABORTABLE* (see *threadAbort* (2K)).

RESTRICTIONS

A user application and a supervisor application may not share a semaphore.

On the contrary, two user applications may share a semaphore by mapping it in both user address spaces.

RETURN VALUE

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

ERRORS

[K_EINVAL]	The *waitLimit* is not a valid *KnTimeVal* .

[K_EABORT] *semP* has been aborted.

[K_ETIMEOUT] The timeout occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mutexInit`(2K)

NAME

semInit, semP, semV – initialize a semaphore; wait on a semaphore; signal a semaphore

SYNOPSIS

```
#include <sync/chSem.h>
int semInit(KnSem * sem, unsigned int count);

int semP(KnSem * sem, KnTimeVal * waitLimit);

int semV(KnSem * sem);
```

FEATURES

SEM

DESCRIPTION

Semaphores are *KnSem* structures allocated in the user memory.

semInit initializes the semaphore the address of which is *sem*. *count* is the initial positive value given to the semaphore counter.

Unless they are shared between two or more actors, statically allocated semaphores can be initialized using the *K_KNSEM_INITIALIZER(cnt)* macro, where *cnt* is the initial positive value given to the semaphore counter. This macro is used as follows:

```
KnSem mySem = K_KNSEM_INITIALIZER(cnt) ;
```

semInit is used to initialize a semaphore that is shared between two or more actors.

semP decrements the semaphore counter; if the counter reaches a strictly negative value, the calling thread is blocked, according to the options described by *waitLimit* in *intro(2K)*. *waitLimit* is a pointer to a *KnTimeVal* structure containing a timeout interval as described in *sysTime(2K)*.

semV increments the counter. If the new value is less than or equal to 0, the thread that has been blocked behind the semaphore for the longest time is awakened.

As semaphore structures are allocated in the client memory, the number of semaphores used by an application is not limited. Any modification to the semaphore structure while the semaphore is in use will cause unpredictable synchronization behavior in the application.

A blocking *semP* is *ABORTABLE* (see *threadAbort* (2K)).

RESTRICTIONS

A user application and a supervisor application may not share a semaphore.

On the contrary, two user applications may share a semaphore by mapping it in both user address spaces.

RETURN VALUE

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

ERRORS

[K_EINVAL]	The *waitLimit* is not a valid *KnTimeVal* .

[K_EABORT] *semP* has been aborted.

[K_ETIMEOUT] The timeout occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mutexInit`(2K)

NAME

svExcHandler, svAbortHandler – Define an exception handler; Define an abort handler

SYNOPSIS

#include <exec/chExec.h>
int **svExcHandler**(KnCap * *actorcap*, KnExcHdl *routine*);

int **svAbortHandler**(KnCap * *actorcap*, KnAbortHdl *routine*);

FEATURES

CORE

DESCRIPTION

The *svExcHandler* function defines an exception handler for the actor whose capability is pointed to by *actorcap*. If *actorcap* is K_MYACTOR, the current actor is used. The *routine* field defines a handler that will be called every time a thread whose execution actor matches the specified actor encounters an exception (see *svTrapConnect* (2K) for details on execution actors). An exception handler is a function which takes three arguments:

```
int handler (ctx, excno, ptr)

     KnThreadCtx*    ctx ;
     int             excno ;
     void*           ptr ;
```

The *ctx* pointer is the register context of the faulty thread. It points to a *KnThreadCtx* structure. The fields of this structure are machine-dependent.

The *excno* field is a machine-dependent exception number. It can only be used for exception handlers.

The *ptr* pointer highlight any relevant machine-dependent data that may be needed to process the exception (for example,. the faulty address for a page fault). It may not be used on certain hardware architectures.

The exception handler may modify the faulty thread's register context before returning.

The *svAbortHandler* function defines a handler to be called each time a thread of the specified actor returns from executing within its home actor environment, while in the *ABORTED* state. When the handler is called, the thread exits its *ABORTED* state (just as if *threadAborted* (2K) were called). An abort handler has the same form as an exception handler, except that the exception number is the K_EABORT constant, and no return value is taken into account.

These calls are restricted to *SUPERVISOR* threads. Handlers are executed in *SUPERVISOR* execution mode. The handlers' code and data accessed must be parts of the locked-in-memory regions of a *SUPERVISOR* actor.

These calls are used by subsystem process managers. The *svExcHandler* function is used to catch exceptions, and to translate them into subsystem-specific events

(for example, . UNIX signals); *svAbortHandler* is used to force abort requests to be taken into account by controlled threads.

RETURN VALUE

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

ERRORS

[K_EINVAL] *actorcap* is an inconsistent actor capability.

[K_EUNKNOWN] *actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svTrapConnect`(2K)

NAME

svActorAbortHandler, svActorAbortHandlerConnect, svActorAbortHandlerDisconnect, svActorAbortHandlerGetConnected – Connect an actor abort handler; Disconnect an actor abort handler; Get an actor abort handler

SYNOPSIS

#include <exec/chExec.h>
int **svActorAbortHandlerConnect**(KnCap * *actorcap*, int *vectindex*, KnLapDesc * *newhandler*);

int **svActorAbortHandlerDisconnect**(KnCap * *actorcap*, int *vectindex*, KnLapDesc * *curhandler*);

int **svActorAbortHandlerGetConnected**(KnCap * *actorcap*, int *vectindex*, KnLapDesc * *curhandler*);

FEATURES

CORE

DESCRIPTION

The *svActorAbortHandler* calls are used to manage abort handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

Abort handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control the aborted state of a thread, the Nucleus keeps a vector of K_ACTOR_ABORTVECT_MAX abort handlers per actor.

The *vectindex* parameter identifies which entry of this abort handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is left to the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_ABORTVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_ABORTVECT_MAX - 1.

When a thread returns from executing within its home actor environment while in the *ABORTED* state, the system invokes all abort handlers which have been associated with the thread's home actor (see *lapInvoke(2K)*). The system clears the thread *ABORTED* state before calling the handlers (just as if *threadAborted(2K)* were called). Abort handlers are invoked in sequence, starting from index 0, until one of them succeeds in processingthe abort event (sets the status to K_ABORT_COMPLETED).

The argument of the lap abort handler is a pointer to a `KnActorAbortDesc` data structure. This structure has the following fields:

```
KnAbortStatus *abortStatus ;
KnThreadCtx *threadCtx ;
```

The *abortStatus* field can be used by the abort handler to indicate that the abort has been successfully processed. The abort handler is called with *abortStatus* pointing to a status variable set to K_ABORT_INPROGRESS. The handler can indicate that it has completed the abort processing by setting this status variable to K_ABORT_COMPLETED. In this case no other abort handlers will be invoked.

The *threadCtx* field gives access to the register context of the aborted thread. This context is processor— dependent. Abort handlers can modify the register context of the aborted thread to recover from the abort.

The *svActorAbortHandlerConnect* function duplicates the lap descriptor pointed to by *newhandler* into the specified entry of the abort handlers vector.

The *svActorAbortHandlerDisconnect* function clears the lap descriptor stored in the specified entry of the abort handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svActorAbortHandlerGetConnected* function copies the lap descriptor currently i nstalled in the abort handlers vector at the location pointed to by *curhandler* .

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorAbortHandlerConnect* is called and there is already a valid lap descriptor installed in the specified entry of the abort handlers vector.
[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorAbortHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svActorAbortHandler, svActorAbortHandlerConnect, svActorAbortHandlerDisconnect, svActorAbortHandlerGetConnected – Connect an actor abort handler; Disconnect an actor abort handler; Get an actor abort handler

SYNOPSIS

#include <exec/chExec.h>
int **svActorAbortHandlerConnect**(KnCap * *actorcap*, int *vectindex*, KnLapDesc * *newhandler*);

int **svActorAbortHandlerDisconnect**(KnCap * *actorcap*, int *vectindex*, KnLapDesc * *curhandler*);

int **svActorAbortHandlerGetConnected**(KnCap * *actorcap*, int *vectindex*, KnLapDesc * *curhandler*);

FEATURES

CORE

DESCRIPTION

The *svActorAbortHandler* calls are used to manage abort handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

Abort handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control the aborted state of a thread, the Nucleus keeps a vector of K_ACTOR_ABORTVECT_MAX abort handlers per actor.

The *vectindex* parameter identifies which entry of this abort handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is left to the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_ABORTVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_ABORTVECT_MAX - 1.

When a thread returns from executing within its home actor environment while in the *ABORTED* state, the system invokes all abort handlers which have been associated with the thread's home actor (see *lapInvoke(2K)*). The system clears the thread *ABORTED* state before calling the handlers (just as if *threadAborted(2K)* were called). Abort handlers are invoked in sequence, starting from index 0, until one of them succeeds in processingthe abort event (sets the status to K_ABORT_COMPLETED).

The argument of the lap abort handler is a pointer to a `KnActorAbortDesc` data structure. This structure has the following fields:

```
KnAbortStatus *abortStatus ;
KnThreadCtx *threadCtx ;
```

The *abortStatus* field can be used by the abort handler to indicate that the abort has been successfully processed. The abort handler is called with *abortStatus* pointing to a status variable set to K_ABORT_INPROGRESS. The handler can indicate that it has completed the abort processing by setting this status variable to K_ABORT_COMPLETED. In this case no other abort handlers will be invoked.

The *threadCtx* field gives access to the register context of the aborted thread. This context is processor— dependent. Abort handlers can modify the register context of the aborted thread to recover from the abort.

The *svActorAbortHandlerConnect* function duplicates the lap descriptor pointed to by *newhandler* into the specified entry of the abort handlers vector.

The *svActorAbortHandlerDisconnect* function clears the lap descriptor stored in the specified entry of the abort handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svActorAbortHandlerGetConnected* function copies the lap descriptor currently i nstalled in the abort handlers vector at the location pointed to by *curhandler* .

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorAbortHandlerConnect* is called and there is already a valid lap descriptor installed in the specified entry of the abort handlers vector.
[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorAbortHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svActorAbortHandler, svActorAbortHandlerConnect, svActorAbortHandlerDisconnect, svActorAbortHandlerGetConnected – Connect an actor abort handler; Disconnect an actor abort handler; Get an actor abort handler

SYNOPSIS

#include <exec/chExec.h>
int **svActorAbortHandlerConnect**(KnCap * *actorcap*, int *vectindex*, KnLapDesc * *newhandler*);

int **svActorAbortHandlerDisconnect**(KnCap * *actorcap*, int *vectindex*, KnLapDesc * *curhandler*);

int **svActorAbortHandlerGetConnected**(KnCap * *actorcap*, int *vectindex*, KnLapDesc * *curhandler*);

FEATURES

CORE

DESCRIPTION

The *svActorAbortHandler* calls are used to manage abort handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

Abort handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control the aborted state of a thread, the Nucleus keeps a vector of K_ACTOR_ABORTVECT_MAX abort handlers per actor.

The *vectindex* parameter identifies which entry of this abort handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is left to the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_ABORTVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_ABORTVECT_MAX - 1.

When a thread returns from executing within its home actor environment while in the *ABORTED* state, the system invokes all abort handlers which have been associated with the thread's home actor (see *lapInvoke(2K)*). The system clears the thread *ABORTED* state before calling the handlers (just as if *threadAborted(2K)* were called). Abort handlers are invoked in sequence, starting from index 0, until one of them succeeds in processingthe abort event (sets the status to K_ABORT_COMPLETED).

The argument of the lap abort handler is a pointer to a `KnActorAbortDesc` data structure. This structure has the following fields:

```
KnAbortStatus *abortStatus ;
KnThreadCtx *threadCtx ;
```

The *abortStatus* field can be used by the abort handler to indicate that the abort has been successfully processed. The abort handler is called with *abortStatus* pointing to a status variable set to K_ABORT_INPROGRESS. The handler can indicate that it has completed the abort processing by setting this status variable to K_ABORT_COMPLETED. In this case no other abort handlers will be invoked.

The *threadCtx* field gives access to the register context of the aborted thread. This context is processor— dependent. Abort handlers can modify the register context of the aborted thread to recover from the abort.

The *svActorAbortHandlerConnect* function duplicates the lap descriptor pointed to by *newhandler* into the specified entry of the abort handlers vector.

The *svActorAbortHandlerDisconnect* function clears the lap descriptor stored in the specified entry of the abort handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svActorAbortHandlerGetConnected* function copies the lap descriptor currently i nstalled in the abort handlers vector at the location pointed to by *curhandler* .

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorAbortHandlerConnect* is called and there is already a valid lap descriptor installed in the specified entry of the abort handlers vector.
[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorAbortHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svActorAbortHandler, svActorAbortHandlerConnect, svActorAbortHandlerDisconnect, svActorAbortHandlerGetConnected – Connect an actor abort handler; Disconnect an actor abort handler; Get an actor abort handler

SYNOPSIS

```
#include <exec/chExec.h>
int svActorAbortHandlerConnect(KnCap * actorcap, int vectindex, KnLapDesc *
newhandler);

int svActorAbortHandlerDisconnect(KnCap * actorcap, int vectindex, KnLapDesc *
curhandler);

int svActorAbortHandlerGetConnected(KnCap * actorcap, int vectindex, KnLapDesc *
curhandler);
```

FEATURES

CORE

DESCRIPTION

The *svActorAbortHandler* calls are used to manage abort handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

Abort handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control the aborted state of a thread, the Nucleus keeps a vector of K_ACTOR_ABORTVECT_MAX abort handlers per actor.

The *vectindex* parameter identifies which entry of this abort handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is left to the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_ABORTVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_ABORTVECT_MAX - 1.

When a thread returns from executing within its home actor environment while in the *ABORTED* state, the system invokes all abort handlers which have been associated with the thread's home actor (see *lapInvoke(2K)*). The system clears the thread *ABORTED* state before calling the handlers (just as if *threadAborted(2K)* were called). Abort handlers are invoked in sequence, starting from index 0, until one of them succeeds in processingthe abort event (sets the status to K_ABORT_COMPLETED).

The argument of the lap abort handler is a pointer to a `KnActorAbortDesc` data structure. This structure has the following fields:

```
KnAbortStatus *abortStatus ;
KnThreadCtx *threadCtx ;
```

The *abortStatus* field can be used by the abort handler to indicate that the abort has been successfully processed. The abort handler is called with *abortStatus* pointing to a status variable set to K_ABORT_INPROGRESS. The handler can indicate that it has completed the abort processing by setting this status variable to K_ABORT_COMPLETED. In this case no other abort handlers will be invoked.

The *threadCtx* field gives access to the register context of the aborted thread. This context is processor— dependent. Abort handlers can modify the register context of the aborted thread to recover from the abort.

The *svActorAbortHandlerConnect* function duplicates the lap descriptor pointed to by *newhandler* into the specified entry of the abort handlers vector.

The *svActorAbortHandlerDisconnect* function clears the lap descriptor stored in the specified entry of the abort handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svActorAbortHandlerGetConnected* function copies the lap descriptor currently i nstalled in the abort handlers vector at the location pointed to by *curhandler* .

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorAbortHandlerConnect* is called and there is already a valid lap descriptor installed in the specified entry of the abort handlers vector.
[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorAbortHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svActorExcHandler, svActorExcHandlerConnect, svActorExcHandlerDisconnect, svActorExcHandlerGetConnected – Connect an actor exception handler; Disconnect an actor exception handler; Get an actor exception handler

SYNOPSIS

```
#include <exec/chExec.h>
int svActorExcHandlerConnect(KnCap * actorcap, int vectindex, KnLapDesc * newhandler);

int svActorExcHandlerDisconnect(KnCap * actorcap, int vectindex, KnLapDesc * curhandler);

int svActorExcHandlerGetConnected(KnCap * actorcap, int vectindex, KnLapDesc * curhandler);
```

FEATURES

CORE

DESCRIPTION

The *svActorExcHandler* calls are used to manage exception handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

Exception handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control exceptions which occur while a thread is executing within a given actor, the system keeps a vector of K_ACTOR_EXCVECT_MAX exception handlers per actor.

The *vectindex* parameter identifies which entry of this exception handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_EXCVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_EXCVECT_MAX - 1.

When a thread causes an exception, the system uses the exception handlers vector associated with the current execution actor of the faulting thread to recover from the exception. During this recovery phase, all valid exception handlers are invoked in sequence (see *lapInvoke(2K)*), starting from index 0, until one of them succeeds in recoveringfrom the exception (sets the status to K_EXC_COMPLETED).

If the exception is not recovered after calling all the exception handlers, the system starts the termination phase where all exception handlers are called a second time in reverse order (starting from index K_ACTOR_EXCVECT_MAX - 1) until one of them takes a default action on the faulting thread (sets the K_EXC_COMPLETED status described below).

The argument of the lap exception handler is a pointer to a `KnActorExcDesc` data structure. This structure has the following fields:

```
KnExcPhase excPhase ;
KnExcStatus *excStatus ;
int excNumber ;
KnThreadCtx *threadCtx ;
void *exceptionCtx ;
```

The *excPhase* field indicates the phase in which the handler was invoked, eitherK_EXC_RECOVER (recovery phase) or K_EXC_TERMINATE (termination phase).

The *excStatus* field can be used by the exception handler to indicate that the exception has been processed successfully. The exception handler is called with *excStatus* pointing to a status variable set to K_EXC_INPROGRESS. The handler can indicate that it has completed the exception processing by setting this status variable to K_EXC_COMPLETED. In this case, no other exception handlers will be invoked.

The *excNumber* field is a processor—dependent exception number.

The *threadCtx* field gives access to the register context of the faulting thread. This context is processor—dependent. Exception handlers may modify the register context of the faulting thread to recover from the exception.

The *exceptionCtx* field gives access to additional data specific to the exception (for example, the fault address for a page fault). This context is processor—and exception— dependent.

The *svActorExcHandlerConnect* function duplicates the lap descriptor pointed to by *newhandler* into the entry specified in the exception handlers vector.

The *svActorExcHandlerDisconnect* function clears the lap descriptor stored in the specified entry of the exception handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svActorExcHandlerGetConnected* function copies the lap descriptor currently installed in the exception handlers vector to the location pointed to by *curhandler* .

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorExcHandlerConnect* is called and there is already a valid lap descriptor installed in the entry specified of the exception handlers vector.

[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorExcHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svActorExcHandler, svActorExcHandlerConnect, svActorExcHandlerDisconnect, svActorExcHandlerGetConnected – Connect an actor exception handler; Disconnect an actor exception handler; Get an actor exception handler

SYNOPSIS

```
#include <exec/chExec.h>
int svActorExcHandlerConnect(KnCap * actorcap, int vectindex, KnLapDesc * newhandler);

int svActorExcHandlerDisconnect(KnCap * actorcap, int vectindex, KnLapDesc * curhandler);

int svActorExcHandlerGetConnected(KnCap * actorcap, int vectindex, KnLapDesc * curhandler);
```

FEATURES

CORE

DESCRIPTION

The *svActorExcHandler* calls are used to manage exception handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

Exception handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control exceptions which occur while a thread is executing within a given actor, the system keeps a vector of K_ACTOR_EXCVECT_MAX exception handlers per actor.

The *vectindex* parameter identifies which entry of this exception handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_EXCVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_EXCVECT_MAX - 1.

When a thread causes an exception, the system uses the exception handlers vector associated with the current execution actor of the faulting thread to recover from the exception. During this recovery phase, all valid exception handlers are invoked in sequence (see *lapInvoke(2K)*), starting from index 0, until one of them succeeds in recoveringfrom the exception (sets the status to K_EXC_COMPLETED).

If the exception is not recovered after calling all the exception handlers, the system starts the termination phase where all exception handlers are called a second time in reverse order (starting from index K_ACTOR_EXCVECT_MAX - 1) until one of them takes a default action on the faulting thread (sets the K_EXC_COMPLETED status described below).

The argument of the lap exception handler is a pointer to a `KnActorExcDesc` data structure. This structure has the following fields:

```
KnExcPhase excPhase ;
KnExcStatus *excStatus ;
int excNumber ;
KnThreadCtx *threadCtx ;
void *exceptionCtx ;
```

The *excPhase* field indicates the phase in which the handler was invoked, eitherK_EXC_RECOVER (recovery phase) or K_EXC_TERMINATE (termination phase).

The *excStatus* field can be used by the exception handler to indicate that the exception has been processed successfully. The exception handler is called with *excStatus* pointing to a status variable set to K_EXC_INPROGRESS. The handler can indicate that it has completed the exception processing by setting this status variable to K_EXC_COMPLETED. In this case, no other exception handlers will be invoked.

The *excNumber* field is a processor—dependent exception number.

The *threadCtx* field gives access to the register context of the faulting thread. This context is processor—dependent. Exception handlers may modify the register context of the faulting thread to recover from the exception.

The *exceptionCtx* field gives access to additional data specific to the exception (for example, the fault address for a page fault). This context is processor—and exception— dependent.

The *svActorExcHandlerConnect* function duplicates the lap descriptor pointed to by *newhandler* into the entry specified in the exception handlers vector.

The *svActorExcHandlerDisconnect* function clears the lap descriptor stored in the specified entry of the exception handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svActorExcHandlerGetConnected* function copies the lap descriptor currently installed in the exception handlers vector to the location pointed to by *curhandler* .

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorExcHandlerConnect* is called and there is already a valid lap descriptor installed in the entry specified of the exception handlers vector.

[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorExcHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svActorExcHandler, svActorExcHandlerConnect, svActorExcHandlerDisconnect, svActorExcHandlerGetConnected – Connect an actor exception handler; Disconnect an actor exception handler; Get an actor exception handler

SYNOPSIS

```
#include <exec/chExec.h>
int svActorExcHandlerConnect(KnCap * actorcap, int vectindex, KnLapDesc * newhandler);

int svActorExcHandlerDisconnect(KnCap * actorcap, int vectindex, KnLapDesc * curhandler);

int svActorExcHandlerGetConnected(KnCap * actorcap, int vectindex, KnLapDesc * curhandler);
```

FEATURES

CORE

DESCRIPTION

The *svActorExcHandler* calls are used to manage exception handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

Exception handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control exceptions which occur while a thread is executing within a given actor, the system keeps a vector of K_ACTOR_EXCVECT_MAX exception handlers per actor.

The *vectindex* parameter identifies which entry of this exception handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_EXCVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_EXCVECT_MAX - 1.

When a thread causes an exception, the system uses the exception handlers vector associated with the current execution actor of the faulting thread to recover from the exception. During this recovery phase, all valid exception handlers are invoked in sequence (see *lapInvoke(2K)*), starting from index 0, until one of them succeeds in recoveringfrom the exception (sets the status to K_EXC_COMPLETED).

If the exception is not recovered after calling all the exception handlers, the system starts the termination phase where all exception handlers are called a second time in reverse order (starting from index K_ACTOR_EXCVECT_MAX - 1) until one of them takes a default action on the faulting thread (sets the K_EXC_COMPLETED status described below).

The argument of the lap exception handler is a pointer to a `KnActorExcDesc` data structure. This structure has the following fields:

```
KnExcPhase excPhase ;
KnExcStatus *excStatus ;
int excNumber ;
KnThreadCtx *threadCtx ;
void *exceptionCtx ;
```

The *excPhase* field indicates the phase in which the handler was invoked, eitherK_EXC_RECOVER (recovery phase) or K_EXC_TERMINATE (termination phase).

The *excStatus* field can be used by the exception handler to indicate that the exception has been processed successfully. The exception handler is called with *excStatus* pointing to a status variable set to K_EXC_INPROGRESS. The handler can indicate that it has completed the exception processing by setting this status variable to K_EXC_COMPLETED. In this case, no other exception handlers will be invoked.

The *excNumber* field is a processor—dependent exception number.

The *threadCtx* field gives access to the register context of the faulting thread. This context is processor—dependent. Exception handlers may modify the register context of the faulting thread to recover from the exception.

The *exceptionCtx* field gives access to additional data specific to the exception (for example, the fault address for a page fault). This context is processor—and exception— dependent.

The *svActorExcHandlerConnect* function duplicates the lap descriptor pointed to by *newhandler* into the entry specified in the exception handlers vector.

The *svActorExcHandlerDisconnect* function clears the lap descriptor stored in the specified entry of the exception handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svActorExcHandlerGetConnected* function copies the lap descriptor currently installed in the exception handlers vector to the location pointed to by *curhandler* .

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorExcHandlerConnect* is called and there is already a valid lap descriptor installed in the entry specified of the exception handlers vector.

[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorExcHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svActorExcHandler, svActorExcHandlerConnect, svActorExcHandlerDisconnect, svActorExcHandlerGetConnected – Connect an actor exception handler; Disconnect an actor exception handler; Get an actor exception handler

SYNOPSIS

```
#include <exec/chExec.h>
int svActorExcHandlerConnect(KnCap * actorcap, int vectindex, KnLapDesc * newhandler);

int svActorExcHandlerDisconnect(KnCap * actorcap, int vectindex, KnLapDesc * curhandler);

int svActorExcHandlerGetConnected(KnCap * actorcap, int vectindex, KnLapDesc * curhandler);
```

FEATURES

CORE

DESCRIPTION

The *svActorExcHandler* calls are used to manage exception handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

Exception handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control exceptions which occur while a thread is executing within a given actor, the system keeps a vector of K_ACTOR_EXCVECT_MAX exception handlers per actor.

The *vectindex* parameter identifies which entry of this exception handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_EXCVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_EXCVECT_MAX - 1.

When a thread causes an exception, the system uses the exception handlers vector associated with the current execution actor of the faulting thread to recover from the exception. During this recovery phase, all valid exception handlers are invoked in sequence (see *lapInvoke(2K)*), starting from index 0, until one of them succeeds in recoveringfrom the exception (sets the status to K_EXC_COMPLETED).

If the exception is not recovered after calling all the exception handlers, the system starts the termination phase where all exception handlers are called a second time in reverse order (starting from index K_ACTOR_EXCVECT_MAX - 1) until one of them takes a default action on the faulting thread (sets the K_EXC_COMPLETED status described below).

The argument of the lap exception handler is a pointer to a `KnActorExcDesc` data structure. This structure has the following fields:

```
KnExcPhase excPhase ;
KnExcStatus *excStatus ;
int excNumber ;
KnThreadCtx *threadCtx ;
void *exceptionCtx ;
```

The *excPhase* field indicates the phase in which the handler was invoked, eitherK_EXC_RECOVER (recovery phase) or K_EXC_TERMINATE (termination phase).

The *excStatus* field can be used by the exception handler to indicate that the exception has been processed successfully. The exception handler is called with *excStatus* pointing to a status variable set to K_EXC_INPROGRESS. The handler can indicate that it has completed the exception processing by setting this status variable to K_EXC_COMPLETED. In this case, no other exception handlers will be invoked.

The *excNumber* field is a processor—dependent exception number.

The *threadCtx* field gives access to the register context of the faulting thread. This context is processor—dependent. Exception handlers may modify the register context of the faulting thread to recover from the exception.

The *exceptionCtx* field gives access to additional data specific to the exception (for example, the fault address for a page fault). This context is processor—and exception— dependent.

The *svActorExcHandlerConnect* function duplicates the lap descriptor pointed to by *newhandler* into the entry specified in the exception handlers vector.

The *svActorExcHandlerDisconnect* function clears the lap descriptor stored in the specified entry of the exception handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svActorExcHandlerGetConnected* function copies the lap descriptor currently installed in the exception handlers vector to the location pointed to by *curhandler* .

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorExcHandlerConnect* is called and there is already a valid lap descriptor installed in the entry specified of the exception handlers vector.

[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorExcHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svActorStopHandler, svActorStopHandlerConnect, svActorStopHandlerDisconnect, svActorStopHandlerGetConnected – Actor stop handler management: Connect an actor stop handler; Disconnect an actor stop handler; Get an actor stop handler

SYNOPSIS

```
#include <exec/chExec.h>
int svActorStopHandlerConnect(KnCap * actorcap, unsigned int vectindex, KnLapDesc * newhandler);

int svActorStopHandlerDisconnect(KnCap * actorcap, unsigned int vectindex, KnLapDesc * curhandler);

int svActorStopHandlerGetConnected(KnCap * actorcap, unsigned int vectindex, KnLapDesc * curhandler);
```

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *svActorStopHandler* calls are used to manage stop handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

The stop handlers are invoked as a consequence of a *threadStop* (2K) or an *actorStop* (2K) system call. In the last case, they are called for all the threads that appear in the actor being stopped. If a thread has performed a safe lap in this actor, it has at least one lap frame in this actor; in such conditions, the stop handlers are called for every lap frame that appears in the actor.

Stop handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control the stopped state of a thread, the Nucleus keeps a vector of K_ACTOR_STOPVECT_MAX stop handlers per actor.

The *vectindex* parameter identifies which entry of this stop handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is left to the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_STOPVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_STOPVECT_MAX - 1.

The stop handlers are usually called by the stopped thread just before it actually stops (K_STOP_INCTX). However, if the thread being stopped is blocked in a blocking system call, or if it has left the current stopped actor through a *lapInvoke* (2K) system call, stop handlers are called by the thread which performs the stop action (K_STOP_OUTCTX).

In any case stop handlers are invoked in sequence, starting from index 0, until one of them succeeds to process the stop event (for example, sets the K_STOP_COMPLETED status).

The argument of the lap stop handler is a pointer to a `KnActorStopDesc` data structure. This structure has the following fields:

```
KnStopMode      stopMode ;
KnStopStatus*   stopStatus ;
int             topFrameLevel ;
int             curFrameLevel ;
KnStopThStatus* threadStatus ;
KnThreadLid     threadLid ;
KnThreadCtx*    threadCtx ;
```

The *stopMode* field indicates whether or not the stop handlers are called in the context of the stopping thread (K_STOP_INCTX vs K_STOP_OUTCTX).

The *stopStatus* field may be used by the stop handler to indicate that the stop has been successfully processed. The stop handler is called with *stopStatus* pointing to a status variable set to K_STOP_INPROGRESS. The handler can indicate that it has completed the stop processing by setting this status variable to K_STOP_COMPLETED. In this case no other stop handlers will be invoked.

The *topFrameLevel* and the *curFrameLevel* fields are used to represent respectivelly the top and the current lap frames. The top lap frame is the one identified by the highest lap frame level (see *svLapCreate* (2K)).

The *threadStatus* field is used to specify whether or not the thread must be stopped when returning from the stop handler invocation. The corresponding values are K_STOP_THREADSTART and K_STOP_THREADSTOP.

The *threadLid* field describes the thread for which the stopping action must be performed.

The *threadCtx* field gives access to the register context of the stopped thread. This context is processor dependent. The given context does not depends on the invocation mode (K_STOP_INCTX vs K_STOP_OUTCTX); this is always the valid thread's context in the stopped actor.

svActorStopHandlerConnect duplicates the lap descriptor pointed to by *newhandler* into the specified entry of the stop handlers vector.

svActorStopHandlerDisconnect clears the lap descriptor stored in the specified entry of the stop handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

svActorStopHandlerGetConnected copies at the location pointed to by *curhandler* the lap descriptor currently installed in the stop handlers vector.

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorStopHandlerConnect* is called and there is already a valid lap descriptor installed in the specified entry of the stop handlers vector.
[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorStopHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorStop`(2K) , `lapInvoke`(2K) , `svLapCreate`(2K) , `threadStop`(2K)

NAME

svActorStopHandler, svActorStopHandlerConnect, svActorStopHandlerDisconnect, svActorStopHandlerGetConnected – Actor stop handler management: Connect an actor stop handler; Disconnect an actor stop handler; Get an actor stop handler

SYNOPSIS

#include <exec/chExec.h>

int **svActorStopHandlerConnect**(KnCap * *actorcap*, unsigned int *vectindex*, KnLapDesc * *newhandler*);

int **svActorStopHandlerDisconnect**(KnCap * *actorcap*, unsigned int *vectindex*, KnLapDesc * *curhandler*);

int **svActorStopHandlerGetConnected**(KnCap * *actorcap*, unsigned int *vectindex*, KnLapDesc * *curhandler*);

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *svActorStopHandler* calls are used to manage stop handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

The stop handlers are invoked as a consequence of a *threadStop* (2K) or an *actorStop* (2K) system call. In the last case, they are called for all the threads that appear in the actor being stopped. If a thread has performed a safe lap in this actor, it has at least one lap frame in this actor; in such conditions, the stop handlers are called for every lap frame that appears in the actor.

Stop handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control the stopped state of a thread, the Nucleus keeps a vector of K_ACTOR_STOPVECT_MAX stop handlers per actor.

The *vectindex* parameter identifies which entry of this stop handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is left to the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_STOPVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_STOPVECT_MAX - 1.

The stop handlers are usually called by the stopped thread just before it actually stops (K_STOP_INCTX). However, if the thread being stopped is blocked in a blocking system call, or if it has left the current stopped actor through a *lapInvoke* (2K) system call, stop handlers are called by the thread which performs the stop action (K_STOP_OUTCTX).

In any case stop handlers are invoked in sequence, starting from index 0, until one of them succeeds to process the stop event (for example, sets the K_STOP_COMPLETED status).

The argument of the lap stop handler is a pointer to a `KnActorStopDesc` data structure. This structure has the following fields:

```
KnStopMode      stopMode ;
KnStopStatus*   stopStatus ;
int             topFrameLevel ;
int             curFrameLevel ;
KnStopThStatus* threadStatus ;
KnThreadLid     threadLid ;
KnThreadCtx*    threadCtx ;
```

The *stopMode* field indicates whether or not the stop handlers are called in the context of the stopping thread (K_STOP_INCTX vs K_STOP_OUTCTX).

The *stopStatus* field may be used by the stop handler to indicate that the stop has been successfully processed. The stop handler is called with *stopStatus* pointing to a status variable set to K_STOP_INPROGRESS. The handler can indicate that it has completed the stop processing by setting this status variable to K_STOP_COMPLETED. In this case no other stop handlers will be invoked.

The *topFrameLevel* and the *curFrameLevel* fields are used to represent respectivelly the top and the current lap frames. The top lap frame is the one identified by the highest lap frame level (see *svLapCreate* (2K)).

The *threadStatus* field is used to specify whether or not the thread must be stopped when returning from the stop handler invocation. The corresponding values are K_STOP_THREADSTART and K_STOP_THREADSTOP.

The *threadLid* field describes the thread for which the stopping action must be performed.

The *threadCtx* field gives access to the register context of the stopped thread. This context is processor dependent. The given context does not depends on the invocation mode (K_STOP_INCTX vs K_STOP_OUTCTX); this is always the valid thread's context in the stopped actor.

svActorStopHandlerConnect duplicates the lap descriptor pointed to by *newhandler* into the specified entry of the stop handlers vector.

svActorStopHandlerDisconnect clears the lap descriptor stored in the specified entry of the stop handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

svActorStopHandlerGetConnected copies at the location pointed to by *curhandler* the lap descriptor currently installed in the stop handlers vector.

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorStopHandlerConnect* is called and there is already a valid lap descriptor installed in the specified entry of the stop handlers vector.
[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorStopHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorStop`(2K) , `lapInvoke`(2K) , `svLapCreate`(2K) , `threadStop`(2K)

NAME

svActorStopHandler, svActorStopHandlerConnect, svActorStopHandlerDisconnect, svActorStopHandlerGetConnected – Actor stop handler management: Connect an actor stop handler; Disconnect an actor stop handler; Get an actor stop handler

SYNOPSIS

```
#include <exec/chExec.h>
int svActorStopHandlerConnect(KnCap * actorcap, unsigned int vectindex, KnLapDesc * newhandler);

int svActorStopHandlerDisconnect(KnCap * actorcap, unsigned int vectindex, KnLapDesc * curhandler);

int svActorStopHandlerGetConnected(KnCap * actorcap, unsigned int vectindex, KnLapDesc * curhandler);
```

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *svActorStopHandler* calls are used to manage stop handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

The stop handlers are invoked as a consequence of a *threadStop* (2K) or an *actorStop* (2K) system call. In the last case, they are called for all the threads that appear in the actor being stopped. If a thread has performed a safe lap in this actor, it has at least one lap frame in this actor; in such conditions, the stop handlers are called for every lap frame that appears in the actor.

Stop handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control the stopped state of a thread, the Nucleus keeps a vector of K_ACTOR_STOPVECT_MAX stop handlers per actor.

The *vectindex* parameter identifies which entry of this stop handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is left to the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_STOPVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_STOPVECT_MAX - 1.

The stop handlers are usually called by the stopped thread just before it actually stops (K_STOP_INCTX). However, if the thread being stopped is blocked in a blocking system call, or if it has left the current stopped actor through a *lapInvoke* (2K) system call, stop handlers are called by the thread which performs the stop action (K_STOP_OUTCTX).

In any case stop handlers are invoked in sequence, starting from index 0, until one of them succeeds to process the stop event (for example, sets the K_STOP_COMPLETED status).

The argument of the lap stop handler is a pointer to a `KnActorStopDesc` data structure. This structure has the following fields:

```
KnStopMode      stopMode ;
KnStopStatus*   stopStatus ;
int             topFrameLevel ;
int             curFrameLevel ;
KnStopThStatus* threadStatus ;
KnThreadLid     threadLid ;
KnThreadCtx*    threadCtx ;
```

The *stopMode* field indicates whether or not the stop handlers are called in the context of the stopping thread (K_STOP_INCTX vs K_STOP_OUTCTX).

The *stopStatus* field may be used by the stop handler to indicate that the stop has been successfully processed. The stop handler is called with *stopStatus* pointing to a status variable set to K_STOP_INPROGRESS. The handler can indicate that it has completed the stop processing by setting this status variable to K_STOP_COMPLETED. In this case no other stop handlers will be invoked.

The *topFrameLevel* and the *curFrameLevel* fields are used to represent respectivelly the top and the current lap frames. The top lap frame is the one identified by the highest lap frame level (see *svLapCreate* (2K)).

The *threadStatus* field is used to specify whether or not the thread must be stopped when returning from the stop handler invocation. The corresponding values are K_STOP_THREADSTART and K_STOP_THREADSTOP.

The *threadLid* field describes the thread for which the stopping action must be performed.

The *threadCtx* field gives access to the register context of the stopped thread. This context is processor dependent. The given context does not depends on the invocation mode (K_STOP_INCTX vs K_STOP_OUTCTX); this is always the valid thread's context in the stopped actor.

svActorStopHandlerConnect duplicates the lap descriptor pointed to by *newhandler* into the specified entry of the stop handlers vector.

svActorStopHandlerDisconnect clears the lap descriptor stored in the specified entry of the stop handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

svActorStopHandlerGetConnected copies at the location pointed to by *curhandler* the lap descriptor currently installed in the stop handlers vector.

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorStopHandlerConnect* is called and there is already a valid lap descriptor installed in the specified entry of the stop handlers vector.
[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorStopHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorStop`(2K) , `lapInvoke`(2K) , `svLapCreate`(2K) , `threadStop`(2K)

NAME

svActorStopHandler, svActorStopHandlerConnect, svActorStopHandlerDisconnect, svActorStopHandlerGetConnected – Actor stop handler management: Connect an actor stop handler; Disconnect an actor stop handler; Get an actor stop handler

SYNOPSIS

#include <exec/chExec.h>

int **svActorStopHandlerConnect**(KnCap * *actorcap*, unsigned int *vectindex*, KnLapDesc * *newhandler*);

int **svActorStopHandlerDisconnect**(KnCap * *actorcap*, unsigned int *vectindex*, KnLapDesc * *curhandler*);

int **svActorStopHandlerGetConnected**(KnCap * *actorcap*, unsigned int *vectindex*, KnLapDesc * *curhandler*);

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *svActorStopHandler* calls are used to manage stop handlers associated with the actor whose capability is pointed to by *actorcap* . These calls are restricted to *SUPERVISOR* threads.

The stop handlers are invoked as a consequence of a *threadStop* (2K) or an *actorStop* (2K) system call. In the last case, they are called for all the threads that appear in the actor being stopped. If a thread has performed a safe lap in this actor, it has at least one lap frame in this actor; in such conditions, the stop handlers are called for every lap frame that appears in the actor.

Stop handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

As several subsystem managers may need to control the stopped state of a thread, the Nucleus keeps a vector of K_ACTOR_STOPVECT_MAX stop handlers per actor.

The *vectindex* parameter identifies which entry of this stop handlers vector should be used by the calls. The consistent allocation of vector indexes among subsystem managers is left to the responsibility of the subsystem managers. Depending on the subsystems, the last vector entry may be reserved for use by the application. For that purpose, the constant K_ACTOR_STOPVECT_LAST can be used independently of the system configuration to designate the index K_ACTOR_STOPVECT_MAX - 1.

The stop handlers are usually called by the stopped thread just before it actually stops (K_STOP_INCTX). However, if the thread being stopped is blocked in a blocking system call, or if it has left the current stopped actor through a *lapInvoke* (2K) system call, stop handlers are called by the thread which performs the stop action (K_STOP_OUTCTX).

In any case stop handlers are invoked in sequence, starting from index 0, until one of them succeeds to process the stop event (for example, sets the K_STOP_COMPLETED status).

The argument of the lap stop handler is a pointer to a `KnActorStopDesc` data structure. This structure has the following fields:

```
KnStopMode      stopMode ;
KnStopStatus*   stopStatus ;
int             topFrameLevel ;
int             curFrameLevel ;
KnStopThStatus* threadStatus ;
KnThreadLid     threadLid ;
KnThreadCtx*    threadCtx ;
```

The *stopMode* field indicates whether or not the stop handlers are called in the context of the stopping thread (K_STOP_INCTX vs K_STOP_OUTCTX).

The *stopStatus* field may be used by the stop handler to indicate that the stop has been successfully processed. The stop handler is called with *stopStatus* pointing to a status variable set to K_STOP_INPROGRESS. The handler can indicate that it has completed the stop processing by setting this status variable to K_STOP_COMPLETED. In this case no other stop handlers will be invoked.

The *topFrameLevel* and the *curFrameLevel* fields are used to represent respectivelly the top and the current lap frames. The top lap frame is the one identified by the highest lap frame level (see *svLapCreate* (2K)).

The *threadStatus* field is used to specify whether or not the thread must be stopped when returning from the stop handler invocation. The corresponding values are K_STOP_THREADSTART and K_STOP_THREADSTOP.

The *threadLid* field describes the thread for which the stopping action must be performed.

The *threadCtx* field gives access to the register context of the stopped thread. This context is processor dependent. The given context does not depends on the invocation mode (K_STOP_INCTX vs K_STOP_OUTCTX); this is always the valid thread's context in the stopped actor.

svActorStopHandlerConnect duplicates the lap descriptor pointed to by *newhandler* into the specified entry of the stop handlers vector.

svActorStopHandlerDisconnect clears the lap descriptor stored in the specified entry of the stop handlers vector. If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

svActorStopHandlerGetConnected copies at the location pointed to by *curhandler* the lap descriptor currently installed in the stop handlers vector.

RETURN VALUE

On success, the calls return K_OK otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svActorStopHandlerConnect* is called and there is already a valid lap descriptor installed in the specified entry of the stop handlers vector.
[K_EINVAL]	*actorcap* is an inconsistent actor capability or *vectindex* is invalid. The *curhandler* lap descriptor specified to *svActorStopHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorStop`(2K) , `lapInvoke`(2K) , `svLapCreate`(2K) , `threadStop`(2K)

NAME

svActorVirtualTimeout, svActorVirtualTimeoutSet, svActorVirtualTimeoutCancel – Set an actor's virtual timeout; Cancel an actor's virtual timeout

SYNOPSIS

#include <vtimer/chVtimer.h>

int **svActorVirtualTimeoutSet**(KnCap * *Iactor*, KnVirtTimeout * *vtimeout*, KnTimeVal * *cputime*, int *flag*, KnLapDesc * *lapdesc*);

int **svActorVirtualTimeoutCancel**(KnVirtTimeout * *vtimeout*);

FEATURES

VTIMER

DESCRIPTION

The *svActorVirtualTimeoutSet* function sets a timeout on the total execution time of all threads created within the actor specified by *actorcap* .

Once the threads have consumed *cputime* of additional execution time, the lap handler designated by *lapdesc* is invoked with *vtimeout* as its argument (see *lapInvoke(2K)*).

If *flag* is set to K_VTIME_INTERNAL, only the execution time in the threads' owning actor is taken into account for the timeout.

If *flag* is set to K_VTIME_TOTAL, all execution time is counted, regardless of whether or not the threads are executing in cross-actor invocations.

Note that in case of a cross-actor invocation, thread execution time is charged to the thread's owning actor only, not to the execution actor.

The *lapDesc* argument is a lap descriptor previously created by *svLapCreate* .

The *KnVirtTimeout* object is opaque; it must be pre-allocated by the caller but its fields are inaccessible to the caller. Virtual timeouts are always relative.

Virtual timeout handlers are not invoked at interrupt level. The execution environment is similar to that of an abort handler. The handler is executed by the next thread of that actor to run, and has no restrictions on use of system calls. A thread can be diverted to execute a handler only when executing in its owning actor; during any cross-actor invocation (trap, ipc, or lap call), virtual timeout handler execution is masked.

The *svActorVirtualTimeoutCancel* function attempts to cancel a virtual timeout request. The *vtimeout* argument contains the address of the *KnVirtTimeout* object used in the corresponding call to *svActorVirtualTimeoutSet* . If the timeout request is still pending, it is cancelled and the call returns K_EOK. If the virtual timeout interval has passed and the handler invocation has been posted, the call takes no action and returns K_ETIMEOUT. In the latter case, there is no information available as to whether the handler execution has actually begun, nor whether the handler is still executing.

A timeout cannot be reinstalled. To reinstall a timeout request, the timeout must be previously cancelled, otherwise an unpredictable behaviour may occur.

RETURN VALUES

Upon successful completion, *svActorVirtualTimeoutSet* and *svActorVirtualTimeoutCancel* return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	The actor capability is not valid
[K_EINVAL]	The validity of the LAP descriptor is not checked.
[K_EUNKNOWN]	*actor* is unreachable.
[K_ETIMEOUT]	(*svActorVirtualTimeoutCancel* only) the virtual interval had already expired, and the handler had been queued for execution.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `svLapCreate`(2K) , `svThreadVirtualTimeoutSet`(2K) , `threadTimes`(2K) , `virtualTimeGetRes`(2K)

NAME

svActorVirtualTimeout, svActorVirtualTimeoutSet, svActorVirtualTimeoutCancel – Set an actor's virtual timeout; Cancel an actor's virtual timeout

SYNOPSIS

#include <vtimer/chVtimer.h>

int **svActorVirtualTimeoutSet**(KnCap * *Iactor*, KnVirtTimeout * *vtimeout*, KnTimeVal * *cputime*, int *flag*, KnLapDesc * *lapdesc*);

int **svActorVirtualTimeoutCancel**(KnVirtTimeout * *vtimeout*);

FEATURES

VTIMER

DESCRIPTION

The *svActorVirtualTimeoutSet* function sets a timeout on the total execution time of all threads created within the actor specified by *actorcap* .

Once the threads have consumed *cputime* of additional execution time, the lap handler designated by *lapdesc* is invoked with *vtimeout* as its argument (see *lapInvoke(2K)*).

If *flag* is set to K_VTIME_INTERNAL, only the execution time in the threads' owning actor is taken into account for the timeout.

If *flag* is set to K_VTIME_TOTAL, all execution time is counted, regardless of whether or not the threads are executing in cross-actor invocations.

Note that in case of a cross-actor invocation, thread execution time is charged to the thread's owning actor only, not to the execution actor.

The *lapDesc* argument is a lap descriptor previously created by *svLapCreate* .

The *KnVirtTimeout* object is opaque; it must be pre-allocated by the caller but its fields are inaccessible to the caller. Virtual timeouts are always relative.

Virtual timeout handlers are not invoked at interrupt level. The execution environment is similar to that of an abort handler. The handler is executed by the next thread of that actor to run, and has no restrictions on use of system calls. A thread can be diverted to execute a handler only when executing in its owning actor; during any cross-actor invocation (trap, ipc, or lap call), virtual timeout handler execution is masked.

The *svActorVirtualTimeoutCancel* function attempts to cancel a virtual timeout request. The *vtimeout* argument contains the address of the *KnVirtTimeout* object used in the corresponding call to *svActorVirtualTimeoutSet* . If the timeout request is still pending, it is cancelled and the call returns K_EOK. If the virtual timeout interval has passed and the handler invocation has been posted, the call takes no action and returns K_ETIMEOUT. In the latter case, there is no information available as to whether the handler execution has actually begun, nor whether the handler is still executing.

A timeout cannot be reinstalled. To reinstall a timeout request, the timeout must be previously cancelled, otherwise an unpredictable behaviour may occur.

RETURN VALUES

Upon successful completion, *svActorVirtualTimeoutSet* and *svActorVirtualTimeoutCancel* return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	The actor capability is not valid
[K_EINVAL]	The validity of the LAP descriptor is not checked.
[K_EUNKNOWN]	*actor* is unreachable.
[K_ETIMEOUT]	(*svActorVirtualTimeoutCancel* only) the virtual interval had already expired, and the handler had been queued for execution.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `svLapCreate`(2K) , `svThreadVirtualTimeoutSet`(2K) , `threadTimes`(2K) , `virtualTimeGetRes`(2K)

NAME

svActorVirtualTimeout, svActorVirtualTimeoutSet, svActorVirtualTimeoutCancel – Set an actor's virtual timeout; Cancel an actor's virtual timeout

SYNOPSIS

#include <vtimer/chVtimer.h>

int **svActorVirtualTimeoutSet**(KnCap * *Iactor*, KnVirtTimeout * *vtimeout*, KnTimeVal * *cputime*, int *flag*, KnLapDesc * *lapdesc*);

int **svActorVirtualTimeoutCancel**(KnVirtTimeout * *vtimeout*);

FEATURES

VTIMER

DESCRIPTION

The *svActorVirtualTimeoutSet* function sets a timeout on the total execution time of all threads created within the actor specified by *actorcap* .

Once the threads have consumed *cputime* of additional execution time, the lap handler designated by *lapdesc* is invoked with *vtimeout* as its argument (see *lapInvoke(2K)*).

If *flag* is set to K_VTIME_INTERNAL, only the execution time in the threads' owning actor is taken into account for the timeout.

If *flag* is set to K_VTIME_TOTAL, all execution time is counted, regardless of whether or not the threads are executing in cross-actor invocations.

Note that in case of a cross-actor invocation, thread execution time is charged to the thread's owning actor only, not to the execution actor.

The *lapDesc* argument is a lap descriptor previously created by *svLapCreate* .

The *KnVirtTimeout* object is opaque; it must be pre-allocated by the caller but its fields are inaccessible to the caller. Virtual timeouts are always relative.

Virtual timeout handlers are not invoked at interrupt level. The execution environment is similar to that of an abort handler. The handler is executed by the next thread of that actor to run, and has no restrictions on use of system calls. A thread can be diverted to execute a handler only when executing in its owning actor; during any cross-actor invocation (trap, ipc, or lap call), virtual timeout handler execution is masked.

The *svActorVirtualTimeoutCancel* function attempts to cancel a virtual timeout request. The *vtimeout* argument contains the address of the *KnVirtTimeout* object used in the corresponding call to *svActorVirtualTimeoutSet* . If the timeout request is still pending, it is cancelled and the call returns K_EOK. If the virtual timeout interval has passed and the handler invocation has been posted, the call takes no action and returns K_ETIMEOUT. In the latter case, there is no information available as to whether the handler execution has actually begun, nor whether the handler is still executing.

A timeout cannot be reinstalled. To reinstall a timeout request, the timeout must be previously cancelled, otherwise an unpredictable behaviour may occur.

RETURN VALUES

Upon successful completion, *svActorVirtualTimeoutSet* and *svActorVirtualTimeoutCancel* return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	The actor capability is not valid
[K_EINVAL]	The validity of the LAP descriptor is not checked.
[K_EUNKNOWN]	*actor* is unreachable.
[K_ETIMEOUT]	(*svActorVirtualTimeoutCancel* only) the virtual interval had already expired, and the handler had been queued for execution.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `svLapCreate`(2K) , `svThreadVirtualTimeoutSet`(2K) , `threadTimes`(2K) , `virtualTimeGetRes`(2K)

NAME

svCopyIn, svCopyInString, svCopyOut – Copy from trap caller space; Copy string from trap caller space; Copy to trap caller space

SYNOPSIS

```
#include <mem/chMem.h>
int svCopyIn(VmAddr src, void * dest, int count);

int svCopyInString(VmAddr src, void * dest, int maxCount);

int svCopyOut(void * src, VmAddr dest, int count);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

These calls are only used by trap handlers (see *svTrapConnect* (2K)). They are used to copy the trap caller parameters.

The *svCopyIn* function copies *count* bytes from the address *src* in the caller's address space, to the address *dest* in the kernel address space. If access to the area defined by *src* and *count* in the caller address space produces a memory access fault, *svCopyIn* will return an error code (K_EFAULT).

The *svCopyInString* function copies characters from the address *src* in the caller's address space, to the address *dest* in the kernel address space until a maximum *maxCount* bytes have been transferred (including the NULL terminating character), or a NULL character is transferred. If access to the area defined by *src* and *maxCount* in the caller address space produces a memory access fault, *svCopyInString* will return an error code (K_EFAULT). If *src* string (including the NULL character) is bigger than *maxCount*, *svCopyInString* will return an error code (K_ENOMEM). If no error is found during string copy, *svCopyInString* will return the actual number of characters transferred(including the NULL character).

The *svCopyOut* function copies *count* bytes from the address *src* in the kernel address space, to the address *dest* in the caller address space. If access to the area defined by *dest* and *count* in the caller address space produces a memory access fault, *svCopyOut* will return an error code (K_EFAULT).

RETURN VALUE

If sucessful, *svCopyIn* and *svCopyOut* return 0. Otherwise, they return K_EFAULT. If successful, *svCopyInString* returns the number of characters transferred, otherwise, it returnsK_EFAULT, or K_ENOMEM.

ERRORS

[K_EFAULT]	Access to the provided caller's data produced a memory access fault.
[K_ENOMEM]	String is longer than requested.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svCopyIn, svCopyInString, svCopyOut – Copy from trap caller space; Copy string from trap caller space; Copy to trap caller space

SYNOPSIS

```
#include <mem/chMem.h>
int svCopyIn(VmAddr src, void * dest, int count);

int svCopyInString(VmAddr src, void * dest, int maxCount);

int svCopyOut(void * src, VmAddr dest, int count);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

These calls are only used by trap handlers (see *svTrapConnect* (2K)). They are used to copy the trap caller parameters.

The *svCopyIn* function copies *count* bytes from the address *src* in the caller's address space, to the address *dest* in the kernel address space. If access to the area defined by *src* and *count* in the caller address space produces a memory access fault, *svCopyIn* will return an error code (K_EFAULT).

The *svCopyInString* function copies characters from the address *src* in the caller's address space, to the address *dest* in the kernel address space until a maximum *maxCount* bytes have been transferred (including the NULL terminating character), or a NULL character is transferred. If access to the area defined by *src* and *maxCount* in the caller address space produces a memory access fault, *svCopyInString* will return an error code (K_EFAULT). If *src* string (including the NULL character) is bigger than *maxCount*, *svCopyInString* will return an error code (K_ENOMEM). If no error is found during string copy, *svCopyInString* will return the actual number of characters transferred(including the NULL character).

The *svCopyOut* function copies *count* bytes from the address *src* in the kernel address space, to the address *dest* in the caller address space. If access to the area defined by *dest* and *count* in the caller address space produces a memory access fault, *svCopyOut* will return an error code (K_EFAULT).

RETURN VALUE

If sucessful, *svCopyIn* and *svCopyOut* return 0. Otherwise, they return K_EFAULT. If successful, *svCopyInString* returns the number of characters transferred, otherwise, it returnsK_EFAULT, or K_ENOMEM.

ERRORS

[K_EFAULT]	Access to the provided caller's data produced a memory access fault.
[K_ENOMEM]	String is longer than requested.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svCopyIn, svCopyInString, svCopyOut – Copy from trap caller space; Copy string from trap caller space; Copy to trap caller space

SYNOPSIS

```
#include <mem/chMem.h>
int svCopyIn(VmAddr src, void * dest, int count);

int svCopyInString(VmAddr src, void * dest, int maxCount);

int svCopyOut(void * src, VmAddr dest, int count);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

These calls are only used by trap handlers (see *svTrapConnect* (2K)). They are used to copy the trap caller parameters.

The *svCopyIn* function copies *count* bytes from the address *src* in the caller's address space, to the address *dest* in the kernel address space. If access to the area defined by *src* and *count* in the caller address space produces a memory access fault, *svCopyIn* will return an error code (K_EFAULT).

The *svCopyInString* function copies characters from the address *src* in the caller's address space, to the address *dest* in the kernel address space until a maximum *maxCount* bytes have been transferred (including the NULL terminating character), or a NULL character is transferred. If access to the area defined by *src* and *maxCount* in the caller address space produces a memory access fault, *svCopyInString* will return an error code (K_EFAULT). If *src* string (including the NULL character) is bigger than *maxCount*, *svCopyInString* will return an error code (K_ENOMEM). If no error is found during string copy, *svCopyInString* will return the actual number of characters transferred(including the NULL character).

The *svCopyOut* function copies *count* bytes from the address *src* in the kernel address space, to the address *dest* in the caller address space. If access to the area defined by *dest* and *count* in the caller address space produces a memory access fault, *svCopyOut* will return an error code (K_EFAULT).

RETURN VALUE

If sucessful, *svCopyIn* and *svCopyOut* return 0. Otherwise, they return K_EFAULT. If successful, *svCopyInString* returns the number of characters transferred, otherwise, it returnsK_EFAULT, or K_ENOMEM.

ERRORS

[K_EFAULT]	Access to the provided caller's data produced a memory access fault.
[K_ENOMEM]	String is longer than requested.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svExcHandler, svAbortHandler – Define an exception handler; Define an abort handler

SYNOPSIS

#include <exec/chExec.h>
int **svExcHandler**(KnCap * *actorcap*, KnExcHdl *routine*);

int **svAbortHandler**(KnCap * *actorcap*, KnAbortHdl *routine*);

FEATURES

CORE

DESCRIPTION

The *svExcHandler* function defines an exception handler for the actor whose capability is pointed to by *actorcap*. If *actorcap* is K_MYACTOR, the current actor is used. The *routine* field defines a handler that will be called every time a thread whose execution actor matches the specified actor encounters an exception (see *svTrapConnect* (2K) for details on execution actors). An exception handler is a function which takes three arguments:

```
int handler (ctx, excno, ptr)

     KnThreadCtx*    ctx ;
     int             excno ;
     void*           ptr ;
```

The *ctx* pointer is the register context of the faulty thread. It points to a *KnThreadCtx* structure. The fields of this structure are machine-dependent.

The *excno* field is a machine-dependent exception number. It can only be used for exception handlers.

The *ptr* pointer highlight any relevant machine-dependent data that may be needed to process the exception (for example,. the faulty address for a page fault). It may not be used on certain hardware architectures.

The exception handler may modify the faulty thread's register context before returning.

The *svAbortHandler* function defines a handler to be called each time a thread of the specified actor returns from executing within its home actor environment, while in the *ABORTED* state. When the handler is called, the thread exits its *ABORTED* state (just as if *threadAborted* (2K) were called). An abort handler has the same form as an exception handler, except that the exception number is the K_EABORT constant, and no return value is taken into account.

These calls are restricted to *SUPERVISOR* threads. Handlers are executed in *SUPERVISOR* execution mode. The handlers' code and data accessed must be parts of the locked-in-memory regions of a *SUPERVISOR* actor.

These calls are used by subsystem process managers. The *svExcHandler* function is used to catch exceptions, and to translate them into subsystem-specific events

(for example, . UNIX signals); *svAbortHandler* is used to force abort requests to be taken into account by controlled threads.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorcap* does not specify a local actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svTrapConnect`(2K)

NAME

svGetInvoker – get handler invoker

SYNOPSIS

```
#include <exec/chExec.h>
int svGetInvoker(KnCap *actorcap);
```

FEATURES

CORE

DESCRIPTION

The *svGetInvoker* function returns the capability of the execution actor prior to the execution of the current handler to the value pointed to by *actorcap*.

When called outside the execution of a handler (connected by *svTrapConnect*(2K), *svExcHandler*(2K), *svAbortHandler*(2K), *svMsgHandler*(2K), *svLapCreate*(2K)), or during execution of an IPC message handler for a remote *ipcCall*(2K), *svGetInvoker* will return K_EINVAL.

The *svGetInvoker* function should not be called from within an interrupt handler.

This call is restricted to *SUPERVISOR* threads.

RETURN VALUE

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

ERRORS

[K_EINVAL] The current thread is not executing a handler.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svTrapConnect(2K), svExcHandler(2K), svAbortHandler(2K), svMsgHandler(2K), svLapCreate(2K), ipcCall(2K), lapInvoke(2K)

NAME

svLapBind, svLapUnbind, lapResolve – bind a symbolic name to a lap descriptor; unbind the symbolic name bound to a lap descriptor; get a lap descriptor from a lap symbolic name

SYNOPSIS

#include <lap/chLap.h>
int **svLapBind**(KnLapDesc * *lapdesc*, char * *name*, unsigned int *options*);

int **svLapUnbind**(char * *name*);

int **lapResolve**(KnLapDesc * *lapdesc*, char * *name*, unsigned int *options*);

FEATURES

LAPBIND

DESCRIPTION

The *svLapBind* (2K) system call binds the lap descriptor pointed to by *lapdesc* with the symbolic name pointed to by *name* .

name points to a null-terminated string of K_LAPNAMEMAX characters at most (not including the null-terminating character).

If the K_LAP_PROTECTED option is set in the *options* parameter, the binding will only be visible to trusted threads (e.g. threads executing in user actors will get an error from *lapResolve* (2K) on this name).

The *svLapUnbind* (2K) system call removes the lap binding associated to *name* .

The *lapResolve* (2K) system call initializes the lap descriptor pointed to by *lapdesc* with the lap descriptor bound to *name* . *lapResolve* (2K) will block until a lap descriptor is bound to *name* , except if the K_LAP_NOBLOCK option is set in *options* , in which case the call returns immediatly with an error.

svLapBind and *svLapUnbind* are restricted to *SUPERVISOR* threads.

RETURN VALUE

On success, these calls return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EABORT]	The calling thread has been aborted in *lapResolve* (2K).
[K_EBUSY]	The name given to *svLapBind* (2K) is already in use.
[K_EFAULT]	The *lapdesc* or *name* arguments of *lapResolve* points to the outside of the caller's address space.
[K_EINVAL]	The *name* given to *svLapUnbind* (2K) is not bound to any lap, or the *name* given to *lapResolve* (2K) is not bound to any lap, and the K_LAP_NOBLOCK option is set.
[K_ENOMEM]	The *name* given to *svLapBind* (2K), *svLapUnbind* (2K) or *lapResolve* (2K) is too long. [K_ENOMEM] The system is out of resources.

[K_EPRIV] *lapResolve* (2K) is called by a non trusted thread to resolve a name which has been bound with the K_LAP_PROTECTED option.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K)

NAME

svLapCreate, lapDescZero, lapDescIsZero, lapDescDup – create a lap; reset a lap descriptor; test if a lap descriptor has been initialized; duplicate a lap descriptor

SYNOPSIS

#include <lap/chLap.h>
int **svLapCreate**(KnCap * *actorcap*, KnLapHdl *laphdl*, void * *cookie*, unsigned int *options*, KnLapDesc * *lapdesc*);

void **lapDescZero**(KnLapDesc * *lapdesc*);

int **lapDescIsZero**(KnLapDesc * *lapdesc*);

void **lapDescDup**(KnLapDesc * *olddesc*, KnLapDesc * *newdesc*);

FEATURES

CORE

DESCRIPTION

The *svLapCreate* (2K) system call creates a new local access point (lap) for the actor designated by the *actorcap* capability. If *actorcap* is equal to K_MYACTOR, the home actor of the calling thread is considered. If the call succeeds, the lap descriptor pointed to by *lapdesc* represents this new lap and can be passed to client threads that will use it to invoke the lap handler.

On lap invocation (see *lapInvoke* (2K)), the *laphdl* handler is called. This handler is a function which takes two arguments:

```
void handler (arg, cookie)
void         *arg ;
void         *cookie ;
```

Where *arg* is the argument specified by *lapInvoke* (2K), and *cookie* is the value of the cookie specified by *svLapCreate* (2K).

By default as a consequence of a lap invocation, the execution actor of the calling thread are set to the lap owning actor for the duration of the lap handler execution.

Additional actions may take place during the lap invocation, depending on the combination of flags specified in the *options* parameter:

K_LAP_SAFE — A " *lap frame* " descriptor is allocated to register the calling thread as a temporary resource of the invoked actor. Each " *lap frame* " has an associated level (lap frame level) which represents the number of lap frames for the considered thread.

This option is only valid if the LAPSAFE feature has been specified into the system. It will enforce a stronger checking during lap invocation. This guarantees in particular that the kernel will synchronize the *svLapDelete* (2K) operation with concurrent lap invocations.

Futhermore, full context of the calling thread is saved prior to the lap invocation. This guarantees that the calling thread can return from its invocation even if a failure (exception, deletion of the lap owning actor, etc.) occurs during the execution of the lap handler.

This option is mandatory for laps to be called from user mode.

The two utility routines *lapDescZero* (2K) and *lapDescIsZero* (2K) are available to manipulate the state of a lap descriptor with the following semantics: if a lap descriptor has been initialized with *lapDescZero* (2K) or is implemented in a zero-filled memory region, *lapDescIsZero* (2K) returns a non-zero value until the lap descriptor has been successfully initialized by *svLapCreate* (2K) or *lapResolve* (2K).

lapDescDup (2K) must be used to duplicate the contents of a lap descriptor.

RETURN VALUE

On success, *svLapCreate* (2K) returns K_OK. Otherwise, a negative error code is returned.

lapDescIsZero (2K) returns a non-zero value if the lap descriptor is not initialized.

lapDescZero (2K) and *lapDescDup* (2K) have no return values.

ERRORS

[K_EINVAL] *actorcap* is an inconsistent actor capability.

[K_ENOMEM] The system is out of resources.

RESTRICTIONS

The current implementation does not support the K_LAP_SETJMP option, which is silently ignored.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `lapResolve`(2K) , `svLapBind`(2K) , `svLapDelete`(2K) , `svLapUnbind`(2K) , `threadStat`(2K)

NAME

svLapDelete – delete a lap

SYNOPSIS

```
#include <lap/chLap.h>
int svLapDelete(KnLapDesc *lapdesc);
```

DESCRIPTION

The *svLapDelete*(2K) system call deletes the local access point the descriptor of which is pointed to by *lapdesc*.

If the corresponding lap has been created with the K_LAP_SAFE option set (see *svLapCreate*(2K)), *svLapDelete*(2K) will return only when all invocations currently in progress on this lap have completed. When a safe lap is deleted, the threads currently executing in the lap are aborted. These threads return with error from the invocation of the lap as described in (see *lapInvoke*(2K)).

Symbolic name bindings associated with this lap (see *svLapBind*(2K)) are not affected by *svLapDelete*(2K) and must be explicitly deleted (see *svLapUnbind*(2K)).

RETURN VALUE

On success, *svLapDelete*(2K) returns K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL] *lapdesc* points to an invalid lap descriptor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K), `svLapBind`(2K), `svLapCreate`(2K), `svLapUnbind`(2K)

NAME

svLapBind, svLapUnbind, lapResolve – bind a symbolic name to a lap descriptor; unbind the symbolic name bound to a lap descriptor; get a lap descriptor from a lap symbolic name

SYNOPSIS

#include <lap/chLap.h>
int **svLapBind**(KnLapDesc * *lapdesc*, char * *name*, unsigned int *options*);

int **svLapUnbind**(char * *name*);

int **lapResolve**(KnLapDesc * *lapdesc*, char * *name*, unsigned int *options*);

FEATURES

LAPBIND

DESCRIPTION

The *svLapBind* (2K) system call binds the lap descriptor pointed to by *lapdesc* with the symbolic name pointed to by *name* .

name points to a null-terminated string of K_LAPNAMEMAX characters at most (not including the null-terminating character).

If the K_LAP_PROTECTED option is set in the *options* parameter, the binding will only be visible to trusted threads (e.g. threads executing in user actors will get an error from *lapResolve* (2K) on this name).

The *svLapUnbind* (2K) system call removes the lap binding associated to *name* .

The *lapResolve* (2K) system call initializes the lap descriptor pointed to by *lapdesc* with the lap descriptor bound to *name* . *lapResolve* (2K) will block until a lap descriptor is bound to *name* , except if the K_LAP_NOBLOCK option is set in *options* , in which case the call returns immediatly with an error.

svLapBind and *svLapUnbind* are restricted to *SUPERVISOR* threads.

RETURN VALUE

On success, these calls return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EABORT]	The calling thread has been aborted in *lapResolve* (2K).
[K_EBUSY]	The name given to *svLapBind* (2K) is already in use.
[K_EFAULT]	The *lapdesc* or *name* arguments of *lapResolve* points to the outside of the caller's address space.
[K_EINVAL]	The *name* given to *svLapUnbind* (2K) is not bound to any lap, or the *name* given to *lapResolve* (2K) is not bound to any lap, and the K_LAP_NOBLOCK option is set.
[K_ENOMEM]	The *name* given to *svLapBind* (2K), *svLapUnbind* (2K) or *lapResolve* (2K) is too long. [K_ENOMEM] The system is out of resources.

[K_EPRIV]	*lapResolve* (2K) is called by a non trusted thread to resolve a name which has been bound with the K_LAP_PROTECTED option.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K)

NAME

svMaskAll, svUnmaskAll, svUnmask – Disable interrupt processing; Enable interrupt processing; Reenable interrupt processing

SYNOPSIS

```
#include <sync/chSpinLock.h>
int svMaskAll(void);

void svUnmaskAll(void);

void svUnmask(int oldval);
```

DESCRIPTION

The *svMaskAll* function disables interrupts on the current processor by setting the processor interrupt mask to its maximum value. There is no effect on other processors. It returns the previous interrupt mask.

The *svUnmaskAll* function enables interrupts on the current processor by setting the processor interrupt mask to its minimum value. There is no effect on other processors.

The *svUnmask* function resets the interrupt mask to a state represented by *oldval.* The *oldval* value must have been returned by a previous invocation of *svMaskAll.* Pairs of *svMaskAll* and *svUnmask* may be nested. The interrupt mask is part of a thread's execution context, it is saved as part of the thread's status when the thread is suspended. If a thread performs a blocking call after calling *svMaskAll,* interrupt processing may be enabled by another scheduled thread. It will be disabled again when the thread is restarted. Such situations should be avoided.

The *svMaskAll, svUnmaskAll* and *svUnmask* functions only relate to the interrupt mask from processor standpoint. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

These calls are restricted to *SUPERVISOR* threads. Their use must be limited to a very small number of instructions. They should not be used to ensure mutual exclusion. The routines *svMaskedLockGet* and *svMaskedLockRel* should be used for this purpose.

RETURN VALUE

svMaskAll returns the current interrupt masking status.

ERRORS

No error messages are returned .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svMaskedLockInit`(2K)

NAME

svMaskedLockInit, svMaskedLockGet, svMaskedLockTry, svMaskedLockRel, svSpinLockInit, svSpinLockGet, svSpinLockTry, svSpinLockRel – Initialize a spin lock; Disable interrupts and acquire a spin lock; Try to disable interrupts and acquire a spin lock; Release a spin lock and enable interrupts; Initialize a spin lock; Acquire a spin lock; Try to acquire a spin lock; Release a spin lock

SYNOPSIS

```
#include <sync/chSpinLock.h>
void svMaskedLockInit(KnMaskedSpinLock * lockaddr, KnMaskedSpinLockInfo * info);

void svMaskedLockGet(KnMaskedSpinLock * lockaddr);

int svMaskedLockTry(KnMaskedSpinLock * lockaddr);

void svMaskedLockRel(KnMaskedSpinLock * lockaddr);

void svSpinLockInit(KnSpinLock * lockaddr, KnSpinLockInfo * info);

void svSpinLockGet(KnSpinLock * lockaddr);

int svSpinLockTry(KnSpinLock * lockaddr);

void svSpinLockRel(KnSpinLock * lockaddr);

int svPreemptable(void);

int svIntrLevel(void);
```

DESCRIPTION

Spin locks are mutual exclusion objects, used to protect very short critical sections. These critical sections must not include kernel calls and the data they manipulate must be locked in memory to avoid page faults.

In multiprocessor systems, threads loop actively when resources are not available. A spin lock is an object in memory, typically modified and checked through indivisible machine operations (such as test-and-set).

In monoprocessor systems, spin locks are implemented by disabling preemption or interrupts.

Note that while a thread is holding a spin lock, a masked spin lock, or has disabled preemption, it is no longer allowed to block. In this case, the list of system calls available to the thread is restricted to the following:

```
semV
threadSemPost
eventPost
msgPut
svSpinLockGet
svSpinLockRel
svMaskedLockGet
svMaskedLockRel
svMaskAll
svUnmaskAll
svUnmask
```

```
svPreemptable
svIntrLevel
svTimeoutSet
svTimeoutCancel
svPagesAllocate
```

The system not force the thread to restrict itself to the permitted calls. If the thread issues non-permitted calls, the system will hang.

The *svMaskedLockInit* and *svSpinLockInit* functions initialize a spin lock in the free state. The *info* argument allows the user to enforce lock hierarchy and / or to gather lock use / contention statistics in the future.

In multiprocessor systems, *svMaskedLockGet* atomically disables interrupts on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with interrupts enabled to reduce interrupt latency. This function can be called from within an interrupt handler.

The *svMaskedLockTry* function attempts to acquire a spin lock, but returns immediately without spinning if there is none available. This function can be called from an interrupt handler.

The *svMaskedLockRel* function releases a spin lock and re-enables interrupts on the current processor. This function can be called from within an interrupt handler.

The *svMaskedLockGet*, *svMaskedLockTry* and *svMaskedLockRel* functions only relate to the interrupt mask of the current processor, and do not affect interrupt processing on other processors. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

In multiprocessor systems, *svSpinLockGet* atomically disables preemption on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with preemption enabled to reduce scheduling latency. On a monoprocessor, *svSpinLockGet* disables preemption on the current processor. This function cannot be called from within an interrupt handler.

The *svSpinLockTry* try attempts to acquire a spin lock, returning immediately without spinning if there is none available. This function cannot be called from within an interrupt handler.

The *svSpinLockRel* function releases a spin lock. This function ca not be called from within an interrupt handler.

If the current thread is preemptable, *svPreemptable* returns a non-zero value, otherwise it returns zero.

If called from an interrupt level, *svIntrLevel* returns a non-zero value, otherwise it returns zero.

RETURN VALUE

If the lock is successfully acquired, *svSpinLockTry* returns the previous state of the lock (K_AVAIL). The *svMaskedLockTry* function returns K_AVAIL. The *svPreemptable* and *svIntrLevel* calls return a value which depends on the current execution context. Other calls return 0.

ERRORS

No error codes are returned. In DEBUG mode, the kernel debugger is invoked if the lock level hierarchy is violated, or if a spin lock cannot be acquired after a very large number of iterations, indicating a probable deadlock.

RESTRICTIONS

Hierarchy level checks are not performed in this version.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMaskedLockInit, svMaskedLockGet, svMaskedLockTry, svMaskedLockRel, svSpinLockInit, svSpinLockGet, svSpinLockTry, svSpinLockRel – Initialize a spin lock; Disable interrupts and acquire a spin lock; Try to disable interrupts and acquire a spin lock; Release a spin lock and enable interrupts; Initialize a spin lock; Acquire a spin lock; Try to acquire a spin lock; Release a spin lock

SYNOPSIS

```
#include <sync/chSpinLock.h>
void svMaskedLockInit(KnMaskedSpinLock * lockaddr, KnMaskedSpinLockInfo * info);

void svMaskedLockGet(KnMaskedSpinLock * lockaddr);

int svMaskedLockTry(KnMaskedSpinLock * lockaddr);

void svMaskedLockRel(KnMaskedSpinLock * lockaddr);

void svSpinLockInit(KnSpinLock * lockaddr, KnSpinLockInfo * info);

void svSpinLockGet(KnSpinLock * lockaddr);

int svSpinLockTry(KnSpinLock * lockaddr);

void svSpinLockRel(KnSpinLock * lockaddr);

int svPreemptable(void);

int svIntrLevel(void);
```

DESCRIPTION

Spin locks are mutual exclusion objects, used to protect very short critical sections. These critical sections must not include kernel calls and the data they manipulate must be locked in memory to avoid page faults.

In multiprocessor systems, threads loop actively when resources are not available. A spin lock is an object in memory, typically modified and checked through indivisible machine operations (such as test-and-set).

In monoprocessor systems, spin locks are implemented by disabling preemption or interrupts.

Note that while a thread is holding a spin lock, a masked spin lock, or has disabled preemption, it is no longer allowed to block. In this case, the list of system calls available to the thread is restricted to the following:

```
semV
threadSemPost
eventPost
msgPut
svSpinLockGet
svSpinLockRel
svMaskedLockGet
svMaskedLockRel
svMaskAll
svUnmaskAll
svUnmask
```

```
svPreemptable
svIntrLevel
svTimeoutSet
svTimeoutCancel
svPagesAllocate
```

The system not force the thread to restrict itself to the permitted calls. If the thread issues non-permitted calls, the system will hang.

The *svMaskedLockInit* and *svSpinLockInit* functions initialize a spin lock in the free state. The *info* argument allows the user to enforce lock hierarchy and / or to gather lock use / contention statistics in the future.

In multiprocessor systems, *svMaskedLockGet* atomically disables interrupts on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with interrupts enabled to reduce interrupt latency. This function can be called from within an interrupt handler.

The *svMaskedLockTry* function attempts to acquire a spin lock, but returns immediately without spinning if there is none available. This function can be called from an interrupt handler.

The *svMaskedLockRel* function releases a spin lock and re-enables interrupts on the current processor. This function can be called from within an interrupt handler.

The *svMaskedLockGet*, *svMaskedLockTry* and *svMaskedLockRel* functions only relate to the interrupt mask of the current processor, and do not affect interrupt processing on other processors. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

In multiprocessor systems, *svSpinLockGet* atomically disables preemption on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with preemption enabled to reduce scheduling latency. On a monoprocessor, *svSpinLockGet* disables preemption on the current processor. This function cannot be called from within an interrupt handler.

The *svSpinLockTry* try attempts to acquire a spin lock, returning immediately without spinning if there is none available. This function cannot be called from within an interrupt handler.

The *svSpinLockRel* function releases a spin lock. This function ca not be called from within an interrupt handler.

If the current thread is preemptable, *svPreemptable* returns a non-zero value, otherwise it returns zero.

If called from an interrupt level, *svIntrLevel* returns a non-zero value, otherwise it returns zero.

RETURN VALUE

If the lock is successfully acquired, *svSpinLockTry* returns the previous state of the lock (K_AVAIL). The *svMaskedLockTry* function returns K_AVAIL. The *svPreemptable* and *svIntrLevel* calls return a value which depends on the current execution context. Other calls return 0.

ERRORS

No error codes are returned. In DEBUG mode, the kernel debugger is invoked if the lock level hierarchy is violated, or if a spin lock cannot be acquired after a very large number of iterations, indicating a probable deadlock.

RESTRICTIONS

Hierarchy level checks are not performed in this version.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMaskedLockInit, svMaskedLockGet, svMaskedLockTry, svMaskedLockRel, svSpinLockInit, svSpinLockGet, svSpinLockTry, svSpinLockRel – Initialize a spin lock; Disable interrupts and acquire a spin lock; Try to disable interrupts and acquire a spin lock; Release a spin lock and enable interrupts; Initialize a spin lock; Acquire a spin lock; Try to acquire a spin lock; Release a spin lock

SYNOPSIS

#include <sync/chSpinLock.h>

void **svMaskedLockInit**(KnMaskedSpinLock * *lockaddr*, KnMaskedSpinLockInfo * *info*);

void **svMaskedLockGet**(KnMaskedSpinLock * *lockaddr*);

int **svMaskedLockTry**(KnMaskedSpinLock * *lockaddr*);

void **svMaskedLockRel**(KnMaskedSpinLock * *lockaddr*);

void **svSpinLockInit**(KnSpinLock * *lockaddr*, KnSpinLockInfo * *info*);

void **svSpinLockGet**(KnSpinLock * *lockaddr*);

int **svSpinLockTry**(KnSpinLock * *lockaddr*);

void **svSpinLockRel**(KnSpinLock * *lockaddr*);

int **svPreemptable**(void);

int **svIntrLevel**(void);

DESCRIPTION

Spin locks are mutual exclusion objects, used to protect very short critical sections. These critical sections must not include kernel calls and the data they manipulate must be locked in memory to avoid page faults.

In multiprocessor systems, threads loop actively when resources are not available. A spin lock is an object in memory, typically modified and checked through indivisible machine operations (such as test-and-set).

In monoprocessor systems, spin locks are implemented by disabling preemption or interrupts.

Note that while a thread is holding a spin lock, a masked spin lock, or has disabled preemption, it is no longer allowed to block. In this case, the list of system calls available to the thread is restricted to the following:

```
semV
threadSemPost
eventPost
msgPut
svSpinLockGet
svSpinLockRel
svMaskedLockGet
svMaskedLockRel
svMaskAll
svUnmaskAll
svUnmask
```

```
svPreemptable
svIntrLevel
svTimeoutSet
svTimeoutCancel
svPagesAllocate
```

The system not force the thread to restrict itself to the permitted calls. If the thread issues non-permitted calls, the system will hang.

The *svMaskedLockInit* and *svSpinLockInit* functions initialize a spin lock in the free state. The *info* argument allows the user to enforce lock hierarchy and / or to gather lock use / contention statistics in the future.

In multiprocessor systems, *svMaskedLockGet* atomically disables interrupts on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with interrupts enabled to reduce interrupt latency. This function can be called from within an interrupt handler.

The *svMaskedLockTry* function attempts to acquire a spin lock, but returns immediately without spinning if there is none available. This function can be called from an interrupt handler.

The *svMaskedLockRel* function releases a spin lock and re-enables interrupts on the current processor. This function can be called from within an interrupt handler.

The *svMaskedLockGet*, *svMaskedLockTry* and *svMaskedLockRel* functions only relate to the interrupt mask of the current processor, and do not affect interrupt processing on other processors. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

In multiprocessor systems, *svSpinLockGet* atomically disables preemption on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with preemption enabled to reduce scheduling latency. On a monoprocessor, *svSpinLockGet* disables preemption on the current processor. This function cannot be called from within an interrupt handler.

The *svSpinLockTry* try attempts to acquire a spin lock, returning immediately without spinning if there is none available. This function cannot be called from within an interrupt handler.

The *svSpinLockRel* function releases a spin lock. This function ca not be called from within an interrupt handler.

If the current thread is preemptable, *svPreemptable* returns a non-zero value, otherwise it returns zero.

If called from an interrupt level, *svIntrLevel* returns a non-zero value, otherwise it returns zero.

RETURN VALUE

If the lock is successfully acquired, *svSpinLockTry* returns the previous state of the lock (K_AVAIL). The *svMaskedLockTry* function returns K_AVAIL. The *svPreemptable* and *svIntrLevel* calls return a value which depends on the current execution context. Other calls return 0.

ERRORS

No error codes are returned. In DEBUG mode, the kernel debugger is invoked if the lock level hierarchy is violated, or if a spin lock cannot be acquired after a very large number of iterations, indicating a probable deadlock.

RESTRICTIONS

Hierarchy level checks are not performed in this version.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMaskedLockInit, svMaskedLockGet, svMaskedLockTry, svMaskedLockRel, svSpinLockInit, svSpinLockGet, svSpinLockTry, svSpinLockRel – Initialize a spin lock; Disable interrupts and acquire a spin lock; Try to disable interrupts and acquire a spin lock; Release a spin lock and enable interrupts; Initialize a spin lock; Acquire a spin lock; Try to acquire a spin lock; Release a spin lock

SYNOPSIS

```
#include <sync/chSpinLock.h>
void svMaskedLockInit(KnMaskedSpinLock * lockaddr, KnMaskedSpinLockInfo * info);

void svMaskedLockGet(KnMaskedSpinLock * lockaddr);

int svMaskedLockTry(KnMaskedSpinLock * lockaddr);

void svMaskedLockRel(KnMaskedSpinLock * lockaddr);

void svSpinLockInit(KnSpinLock * lockaddr, KnSpinLockInfo * info);

void svSpinLockGet(KnSpinLock * lockaddr);

int svSpinLockTry(KnSpinLock * lockaddr);

void svSpinLockRel(KnSpinLock * lockaddr);

int svPreemptable(void);

int svIntrLevel(void);
```

DESCRIPTION

Spin locks are mutual exclusion objects, used to protect very short critical sections. These critical sections must not include kernel calls and the data they manipulate must be locked in memory to avoid page faults.

In multiprocessor systems, threads loop actively when resources are not available. A spin lock is an object in memory, typically modified and checked through indivisible machine operations (such as test-and-set).

In monoprocessor systems, spin locks are implemented by disabling preemption or interrupts.

Note that while a thread is holding a spin lock, a masked spin lock, or has disabled preemption, it is no longer allowed to block. In this case, the list of system calls available to the thread is restricted to the following:

```
semV
threadSemPost
eventPost
msgPut
svSpinLockGet
svSpinLockRel
svMaskedLockGet
svMaskedLockRel
svMaskAll
svUnmaskAll
svUnmask
```

```
svPreemptable
svIntrLevel
svTimeoutSet
svTimeoutCancel
svPagesAllocate
```

The system not force the thread to restrict itself to the permitted calls. If the thread issues non-permitted calls, the system will hang.

The *svMaskedLockInit* and *svSpinLockInit* functions initialize a spin lock in the free state. The *info* argument allows the user to enforce lock hierarchy and / or to gather lock use / contention statistics in the future.

In multiprocessor systems, *svMaskedLockGet* atomically disables interrupts on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with interrupts enabled to reduce interrupt latency. This function can be called from within an interrupt handler.

The *svMaskedLockTry* function attempts to acquire a spin lock, but returns immediately without spinning if there is none available. This function can be called from an interrupt handler.

The *svMaskedLockRel* function releases a spin lock and re-enables interrupts on the current processor. This function can be called from within an interrupt handler.

The *svMaskedLockGet*, *svMaskedLockTry* and *svMaskedLockRel* functions only relate to the interrupt mask of the current processor, and do not affect interrupt processing on other processors. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

In multiprocessor systems, *svSpinLockGet* atomically disables preemption on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with preemption enabled to reduce scheduling latency. On a monoprocessor, *svSpinLockGet* disables preemption on the current processor. This function cannot be called from within an interrupt handler.

The *svSpinLockTry* try attempts to acquire a spin lock, returning immediately without spinning if there is none available. This function cannot be called from within an interrupt handler.

The *svSpinLockRel* function releases a spin lock. This function ca not be called from within an interrupt handler.

If the current thread is preemptable, *svPreemptable* returns a non-zero value, otherwise it returns zero.

If called from an interrupt level, *svIntrLevel* returns a non-zero value, otherwise it returns zero.

RETURN VALUE

If the lock is successfully acquired, *svSpinLockTry* returns the previous state of the lock (K_AVAIL). The *svMaskedLockTry* function returns K_AVAIL. The *svPreemptable* and *svIntrLevel* calls return a value which depends on the current execution context. Other calls return 0.

ERRORS

No error codes are returned. In DEBUG mode, the kernel debugger is invoked if the lock level hierarchy is violated, or if a spin lock cannot be acquired after a very large number of iterations, indicating a probable deadlock.

RESTRICTIONS

Hierarchy level checks are not performed in this version.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMemRead, svMemWrite – Copy from supervisor caller space; Copy to supervisor caller space

SYNOPSIS

```
#include <mem/chMem.h>
int svMemRead(void * src, void * dest, int count);

int svMemWrite(void * src, void * dest, int count);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

These calls are used by programs to access supervisor address space memory safely.

The *svMemRead* functioncopies *count* bytes from the address *src* in the supervisor address space, to the address *dest* in the supervisor address space. If access to the area defined by *src* and *count* in the supervisor address space produces a memory access fault, *svMemRead* will return an error code (K_EFAULT).

The *svMemWrite* call copies *count* bytes from the address *src* in the supervisor address space, to the address *dest* in the supervisor address space. If access to the area defined by *dest* and *count* in the supervisor address space produces a memory access fault, *svMemWrite* will return an error code (K_EFAULT).

RETURN VALUE

If successful, *svMemRead* and *svMemWrite* return 0. Otherwise, they return K_EFAULT.

ERRORS

[K_EFAULT]	Access to the provided supervisor data produced a memory access fault.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMemRead, svMemWrite – Copy from supervisor caller space; Copy to supervisor caller space

SYNOPSIS

```
#include <mem/chMem.h>
int svMemRead(void * src, void * dest, int count);

int svMemWrite(void * src, void * dest, int count);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

These calls are used by programs to access supervisor address space memory safely.

The *svMemRead* functioncopies *count* bytes from the address *src* in the supervisor address space, to the address *dest* in the supervisor address space. If access to the area defined by *src* and *count* in the supervisor address space produces a memory access fault, *svMemRead* will return an error code (K_EFAULT).

The *svMemWrite* call copies *count* bytes from the address *src* in the supervisor address space, to the address *dest* in the supervisor address space. If access to the area defined by *dest* and *count* in the supervisor address space produces a memory access fault, *svMemWrite* will return an error code (K_EFAULT).

RETURN VALUE

If successful, *svMemRead* and *svMemWrite* return 0. Otherwise, they return K_EFAULT.

ERRORS

[K_EFAULT]	Access to the provided supervisor data produced a memory access fault.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMsgHandler, svMsgHdlReply – Connect / disconnect a message handler; Prepare a reply to a handled message

SYNOPSIS

#include <ipc/chIpc.h>
int **svMsgHandler**(KnCap * *act*, int *portli*, KnMsgHdl *handler*, unsigned int *stacksize*, void * *cookie*);

int **svMsgHdlReply**(KnMsgDesc * *msg*);

FEATURES

IPC

DESCRIPTION

The *svMsgHandler* function connects a handler to the arrival of messages on the port whose local identifier is *portli* in the actor whose capability is given by *act*. If *act* is K_MYACTOR, the current actor is used.

If *handler* is NULL, *svMsgHandler* disconnects the handler connected previously..

The message handler will be called each time a message is posted to the port. If messages are queued behind the port at the time the handler is connected, the handler will be invoked for these messages.

The *cookie* argument is user-defined and is passed as the last argument to the handler. This allows the application to identify the handler easily when multiple ports are using the same handler.

When a message handler has been connected for a given port, *ipcReceive* (2K) calls on that port are refused (*ipcReceive* (2K) returns the K_EINVAL error code).

A port to which a message handler is connected may not be migrated (see *portMigrate* (2K)).

The stack size required by the handler execution is defined by *stacksize* . In the current version, this argument is ignored. The system only insures that when the handler is called, the system stack of the thread is a constant (defined at kernel generation time) number of free bytes.

The handler is a pointer to a function whose arguments are the following:

```
int handler (request, src, dest, port, reply, cookie)
      KnMsgDesc*     request ;
      KnUniqueId*    src ;
      KnIpcDest*     dest ;
      int            port ;
      KnMsgDesc*     reply ;
      void*          cookie ;
```

The *request* message descriptor describes the message to be handled (the *KnMsgDesc* structure is described in *ipcSend* (2K)). The *seqNum* member of this structure represents the count of messages that have been received behind the port up to and including the message being handled (see also *ipcReceive* (2K)). If the K_USERBODY or K_USERANNEX flag is present in the *flags* field of

this descriptor, the corresponding data must be obtained using *svCopyIn* (2K). Otherwise, the data may be read directly from the system address space.

The *src* pointer points to the UI describing the port from which the message has been sent.

The *dest* pointer points to the destination to which the message has been sent.

The *port* pointer is the local identifier of the port to which the handler is attached.

The location for a reply to the message may already have been allocated by the system or the sender. In this case, *reply* is a non NULL pointer to a message descriptor. Otherwise, *reply* is NULL. Note that the K_USERBODY and K_USERANNEX flags may be present in the *reply* message descriptor. In this case, the handler should use *svCopyOut* (2K) to update the reply message annex and / or body.

To have the system update the handler's reply data, or if *reply* is NULL, the handler can invoke the *svMsgHdlReply* function. This function updates the reply data according to the *msg* message description. This function does not cause the destinator of the reply to receive it: this operation is delayed until the handler exits. The value passed to *svMsgHandler* is defined by *cookie* .

The handler is executed by a system thread. This thread is a restricted form of a thread, a *handler thread.* The handler thread is temporarily attached to the actor which ownsthe port to which the message handler has been attached. Any reference to the "current actor" (K_MYACTOR) within the context of the handler thread execution is interpreted as a reference to the actor which owns the port to which the handler is connected.

The handler thread possesses a local identifier within its actor, it may be obtained using *threadSelf* (2K). However, this thread is a restricted type of thread, its local identifier may not be used for external management of the thread (*threadAbort* (2K), *threadSuspend* (2K), *threadResume* (2K), *threadContext* (2K), *threadTimes* (2K)). Applying these services to a handler thread results in the K_EINVAL error code being returned (invalid thread)..

The only way to delete the handler thread is to return from the handler.

When entering the handler, the handler thread has no current message. This means that *ipcSave* (2K), *ipcReply* (2K) and *ipcSysInfo* (2K) return the K_EINVAL error code. However, the handler thread can perform *ipcReceive* (2K) and get a current message. If the handler returns while the handler thread has a current message, this message is canceled.

When the handler exits, the handler return value defines the format of the reply to the sender's request, as follows:

If *svMsgHdlReply* was not called during handler execution, and the handler returns a positive or null value, the message body and annex are updated by the handler (using the *reply* argument). In this case, the handler returns the message body size. If the handler return value is a negative error code, the caller does not receive a reply; however, if the message was sent using *ipcCall* (2K), the negative error code returned by the handler is returned to the caller.

If *svMsgHdlReply* was called during handler execution, the handler return code is ignored and the caller receives zero if the call was successful or a negative error code (see below).

Both *svMsgHandler* and *svMsgHdlReply* are restricted to supervisor threads. The handler code must be located in the system address space.

RETURN VALUE

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

ERRORS

[K_EINVAL]	(*svMsgHandler*) *actorcap* is an inconsistent actor capability, or *portli* is not a valid port identifier within the actor whose capability is given by *actorcap*. (*svMsgHdlReply*) the current thread is not a handler thread.
[K_EUNKNOWN]	(*svMsgHandler* only) *actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

Currently, if a handler processes an asynchronous message (*ipcSend*) or a message sent remotely, the invocation of *svMsgHdlReply* sends the reply to the caller, instead of preparing it and delaying it until the handler returns .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ipcCall`(2K) , `ipcSend`(2K) , `threadAbort`(2K) , `svCopyIn`(2K) , `svCopyOut`(2K)

NAME

svMsgHandler, svMsgHdlReply – Connect / disconnect a message handler; Prepare a reply to a handled message

SYNOPSIS

#include <ipc/chIpc.h>
int **svMsgHandler**(KnCap * *act*, int *portli*, KnMsgHdl *handler*, unsigned int *stacksize*, void * *cookie*);

int **svMsgHdlReply**(KnMsgDesc * *msg*);

FEATURES

IPC

DESCRIPTION

The *svMsgHandler* function connects a handler to the arrival of messages on the port whose local identifier is *portli* in the actor whose capability is given by *act*. If *act* is K_MYACTOR, the current actor is used.

If *handler* is NULL, *svMsgHandler* disconnects the handler connected previously..

The message handler will be called each time a message is posted to the port. If messages are queued behind the port at the time the handler is connected, the handler will be invoked for these messages.

The *cookie* argument is user-defined and is passed as the last argument to the handler. This allows the application to identify the handler easily when multiple ports are using the same handler.

When a message handler has been connected for a given port, *ipcReceive* (2K) calls on that port are refused (*ipcReceive* (2K) returns the K_EINVAL error code).

A port to which a message handler is connected may not be migrated (see *portMigrate* (2K)).

The stack size required by the handler execution is defined by *stacksize* . In the current version, this argument is ignored. The system only insures that when the handler is called, the system stack of the thread is a constant (defined at kernel generation time) number of free bytes.

The handler is a pointer to a function whose arguments are the following:

```
int handler (request, src, dest, port, reply, cookie)
      KnMsgDesc*     request ;
      KnUniqueId*    src ;
      KnIpcDest*     dest ;
      int            port ;
      KnMsgDesc*     reply ;
      void*          cookie ;
```

The *request* message descriptor describes the message to be handled (the *KnMsgDesc* structure is described in *ipcSend* (2K)). The *seqNum* member of this structure represents the count of messages that have been received behind the port up to and including the message being handled (see also *ipcReceive* (2K)). If the K_USERBODY or K_USERANNEX flag is present in the *flags* field of

this descriptor, the corresponding data must be obtained using *svCopyIn* (2K). Otherwise, the data may be read directly from the system address space.

The *src* pointer points to the UI describing the port from which the message has been sent.

The *dest* pointer points to the destination to which the message has been sent.

The *port* pointer is the local identifier of the port to which the handler is attached.

The location for a reply to the message may already have been allocated by the system or the sender. In this case, *reply* is a non NULL pointer to a message descriptor. Otherwise, *reply* is NULL. Note that the K_USERBODY and K_USERANNEX flags may be present in the *reply* message descriptor. In this case, the handler should use *svCopyOut* (2K) to update the reply message annex and / or body.

To have the system update the handler's reply data, or if *reply* is NULL, the handler can invoke the *svMsgHdlReply* function. This function updates the reply data according to the *msg* message description. This function does not cause the destinator of the reply to receive it: this operation is delayed until the handler exits. The value passed to *svMsgHandler* is defined by *cookie* .

The handler is executed by a system thread. This thread is a restricted form of a thread, a *handler thread.* The handler thread is temporarily attached to the actor which ownsthe port to which the message handler has been attached. Any reference to the "current actor" (K_MYACTOR) within the context of the handler thread execution is interpreted as a reference to the actor which owns the port to which the handler is connected.

The handler thread possesses a local identifier within its actor, it may be obtained using *threadSelf* (2K). However, this thread is a restricted type of thread, its local identifier may not be used for external management of the thread (*threadAbort* (2K), *threadSuspend* (2K), *threadResume* (2K), *threadContext* (2K), *threadTimes* (2K)). Applying these services to a handler thread results in the K_EINVAL error code being returned (invalid thread)..

The only way to delete the handler thread is to return from the handler.

When entering the handler, the handler thread has no current message. This means that *ipcSave* (2K), *ipcReply* (2K) and *ipcSysInfo* (2K) return the K_EINVAL error code. However, the handler thread can perform *ipcReceive* (2K) and get a current message. If the handler returns while the handler thread has a current message, this message is canceled.

When the handler exits, the handler return value defines the format of the reply to the sender's request, as follows:

If *svMsgHdlReply* was not called during handler execution, and the handler returns a positive or null value, the message body and annex are updated by the handler (using the *reply* argument). In this case, the handler returns the message body size. If the handler return value is a negative error code, the caller does not receive a reply; however, if the message was sent using *ipcCall* (2K), the negative error code returned by the handler is returned to the caller.

If *svMsgHdlReply* was called during handler execution, the handler return code is ignored and the caller receives zero if the call was successful or a negative error code (see below).

Both *svMsgHandler* and *svMsgHdlReply* are restricted to supervisor threads. The handler code must be located in the system address space.

RETURN VALUE

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

ERRORS

[K_EINVAL]	(*svMsgHandler*) *actorcap* is an inconsistent actor capability, or *portli* is not a valid port identifier within the actor whose capability is given by *actorcap*. (*svMsgHdlReply*) the current thread is not a handler thread.
[K_EUNKNOWN]	(*svMsgHandler* only) *actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

Currently, if a handler processes an asynchronous message (*ipcSend*) or a message sent remotely, the invocation of *svMsgHdlReply* sends the reply to the caller, instead of preparing it and delaying it until the handler returns .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ipcCall`(2K) , `ipcSend`(2K) , `threadAbort`(2K) , `svCopyIn`(2K) , `svCopyOut`(2K)

NAME

svPagesAllocate, svPagesFree – supervisor address space memory allocator

SYNOPSIS

#include <mem/chMem.h>
int **svPagesAllocate**(VmAddr * *address*, VmSize *size*, VmFlags *flags*, KnCap * *actcap*);

int **svPagesFree**(VmAddr *address*, VmSize *size*, KnCap * *actcap*);

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *svPagesAllocate* function allocates an amount of (usually discontiguous) physical memory and a range of virtual addresses within the supervisor address space. It then maps the physical memory to this address range.

The `size` argument specifies the size (in bytes) of the memory to be allocated. There are no alignment requirements on the `size` value, but *svPagesAllocate* always aligns the size of allocated memory to the next page boundary.

The starting address of the allocated virtual memory range is returned in the variable pointed to by the *address* argument.

The *flags* argument is a combination of the following flags:

K_NOWAITFORMEMORY	If the K_NOWAITFORMEMORY flag is set, *svPagesAllocate* fails when there is not enough physical memory. In this case it returns the K_ENOMEM error; otherwise, *svPagesAllocate* waits for memory to become available as the result of memory swap out activity. If the swap outs is not available *svPagesAllocate* will never wait.
K_REDALLOC	This flag is for internal operating system usage only.

The *actcap* argument specifies an actor with which the allocated memory will be associated. It is required for various functions, such as actor dumps. Note that allocated memory is not automatically freed when the corresponding actor is deleted using *actorDelete* (2K). If the actor association is not needed, K_SVACTOR can be specified. In this case the memory will be anonymous (not linked to any actor).

The *svPagesFree* call releases a virtual address range and the physical memory mapped to it. The *address* argument specifies the range's starting address and the `size` argument specifies its size in bytes.

The address range must be a sub-range of an address range previously allocated by a *svPagesAllocate* call. The range start address must be page aligned but there

are no alignment requirements on the range size. Nevertheless *svPagesFree* always aligns the size of deallocated memory to the next page boundary.

The *svPagesAllocate* and *svPagesFree* calls may called by an interrupt handler. In this case *svPagesAllocate* must be called with K_NOWAITFORMEMORY.

The *actcap* argument specifies the actor from which the pages must be detached. It must match the actor that was specified when the pages were allocated.

RETURN VALUES

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

ERRORS

[K_EFAULT]	The *address* argument points to the outside of the caller's address space.
[K_EINVAL]	*svPagesAllocate* was called by an interrupt handler and K_NOWAITFORMEMORY flag was not specified.
[K_EROUND]	The *address* argument of *svPagesFree* is not page aligned.
[K_ESIZE]	The `size` argument is equal to 0.
[K_ENOMEM]	The system is out of memory.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`vmSetPar`(2K)

NAME

svPagesAllocate, svPagesFree – supervisor address space memory allocator

SYNOPSIS

#include <mem/chMem.h>
int **svPagesAllocate**(VmAddr * *address*, VmSize *size*, VmFlags *flags*, KnCap * *actcap*);

int **svPagesFree**(VmAddr *address*, VmSize *size*, KnCap * *actcap*);

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *svPagesAllocate* function allocates an amount of (usually discontiguous) physical memory and a range of virtual addresses within the supervisor address space. It then maps the physical memory to this address range.

The `size` argument specifies the size (in bytes) of the memory to be allocated. There are no alignment requirements on the `size` value, but *svPagesAllocate* always aligns the size of allocated memory to the next page boundary.

The starting address of the allocated virtual memory range is returned in the variable pointed to by the *address* argument.

The *flags* argument is a combination of the following flags:

K_NOWAITFORMEMORY	If the K_NOWAITFORMEMORY flag is set, *svPagesAllocate* fails when there is not enough physical memory. In this case it returns the K_ENOMEM error; otherwise, *svPagesAllocate* waits for memory to become available as the result of memory swap out activity. If the swap outs is not available *svPagesAllocate* will never wait.
K_REDALLOC	This flag is for internal operating system usage only.

The *actcap* argument specifies an actor with which the allocated memory will be associated. It is required for various functions, such as actor dumps. Note that allocated memory is not automatically freed when the corresponding actor is deleted using *actorDelete* (2K). If the actor association is not needed, K_SVACTOR can be specified. In this case the memory will be anonymous (not linked to any actor).

The *svPagesFree* call releases a virtual address range and the physical memory mapped to it. The *address* argument specifies the range's starting address and the `size` argument specifies its size in bytes.

The address range must be a sub-range of an address range previously allocated by a *svPagesAllocate* call. The range start address must be page aligned but there

are no alignment requirements on the range size. Nevertheless *svPagesFree* always aligns the size of deallocated memory to the next page boundary.

The *svPagesAllocate* and *svPagesFree* calls may called by an interrupt handler. In this case *svPagesAllocate* must be called with K_NOWAITFORMEMORY.

The *actcap* argument specifies the actor from which the pages must be detached. It must match the actor that was specified when the pages were allocated.

RETURN VALUES

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

ERRORS

[K_EFAULT]	The *address* argument points to the outside of the caller's address space.
[K_EINVAL]	*svPagesAllocate* was called by an interrupt handler and K_NOWAITFORMEMORY flag was not specified.
[K_EROUND]	The *address* argument of *svPagesFree* is not page aligned.
[K_ESIZE]	The `size` argument is equal to 0.
[K_ENOMEM]	The system is out of memory.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`vmSetPar`(2K)

NAME

svMaskedLockInit, svMaskedLockGet, svMaskedLockTry, svMaskedLockRel, svSpinLockInit, svSpinLockGet, svSpinLockTry, svSpinLockRel – Initialize a spin lock; Disable interrupts and acquire a spin lock; Try to disable interrupts and acquire a spin lock; Release a spin lock and enable interrupts; Initialize a spin lock; Acquire a spin lock; Try to acquire a spin lock; Release a spin lock

SYNOPSIS

#include <sync/chSpinLock.h>

void **svMaskedLockInit**(KnMaskedSpinLock * *lockaddr*, KnMaskedSpinLockInfo * *info*);

void **svMaskedLockGet**(KnMaskedSpinLock * *lockaddr*);

int **svMaskedLockTry**(KnMaskedSpinLock * *lockaddr*);

void **svMaskedLockRel**(KnMaskedSpinLock * *lockaddr*);

void **svSpinLockInit**(KnSpinLock * *lockaddr*, KnSpinLockInfo * *info*);

void **svSpinLockGet**(KnSpinLock * *lockaddr*);

int **svSpinLockTry**(KnSpinLock * *lockaddr*);

void **svSpinLockRel**(KnSpinLock * *lockaddr*);

int **svPreemptable**(void);

int **svIntrLevel**(void);

DESCRIPTION

Spin locks are mutual exclusion objects, used to protect very short critical sections. These critical sections must not include kernel calls and the data they manipulate must be locked in memory to avoid page faults.

In multiprocessor systems, threads loop actively when resources are not available. A spin lock is an object in memory, typically modified and checked through indivisible machine operations (such as test-and-set).

In monoprocessor systems, spin locks are implemented by disabling preemption or interrupts.

Note that while a thread is holding a spin lock, a masked spin lock, or has disabled preemption, it is no longer allowed to block. In this case, the list of system calls available to the thread is restricted to the following:

```
semV
threadSemPost
eventPost
msgPut
svSpinLockGet
svSpinLockRel
svMaskedLockGet
svMaskedLockRel
svMaskAll
svUnmaskAll
svUnmask
```

```
svPreemptable
svIntrLevel
svTimeoutSet
svTimeoutCancel
svPagesAllocate
```

The system not force the thread to restrict itself to the permitted calls. If the thread issues non-permitted calls, the system will hang.

The *svMaskedLockInit* and *svSpinLockInit* functions initialize a spin lock in the free state. The *info* argument allows the user to enforce lock hierarchy and / or to gather lock use / contention statistics in the future.

In multiprocessor systems, *svMaskedLockGet* atomically disables interrupts on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with interrupts enabled to reduce interrupt latency. This function can be called from within an interrupt handler.

The *svMaskedLockTry* function attempts to acquire a spin lock, but returns immediately without spinning if there is none available. This function can be called from an interrupt handler.

The *svMaskedLockRel* function releases a spin lock and re-enables interrupts on the current processor. This function can be called from within an interrupt handler.

The *svMaskedLockGet*, *svMaskedLockTry* and *svMaskedLockRel* functions only relate to the interrupt mask of the current processor, and do not affect interrupt processing on other processors. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

In multiprocessor systems, *svSpinLockGet* atomically disables preemption on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with preemption enabled to reduce scheduling latency. On a monoprocessor, *svSpinLockGet* disables preemption on the current processor. This function cannot be called from within an interrupt handler.

The *svSpinLockTry* try attempts to acquire a spin lock, returning immediately without spinning if there is none available. This function cannot be called from within an interrupt handler.

The *svSpinLockRel* function releases a spin lock. This function ca not be called from within an interrupt handler.

If the current thread is preemptable, *svPreemptable* returns a non-zero value, otherwise it returns zero.

If called from an interrupt level, *svIntrLevel* returns a non-zero value, otherwise it returns zero.

RETURN VALUE

If the lock is successfully acquired, *svSpinLockTry* returns the previous state of the lock (K_AVAIL). The *svMaskedLockTry* function returns K_AVAIL. The *svPreemptable* and *svIntrLevel* calls return a value which depends on the current execution context. Other calls return 0.

ERRORS

No error codes are returned. In DEBUG mode, the kernel debugger is invoked if the lock level hierarchy is violated, or if a spin lock cannot be acquired after a very large number of iterations, indicating a probable deadlock.

RESTRICTIONS

Hierarchy level checks are not performed in this version.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMaskedLockInit, svMaskedLockGet, svMaskedLockTry, svMaskedLockRel, svSpinLockInit, svSpinLockGet, svSpinLockTry, svSpinLockRel – Initialize a spin lock; Disable interrupts and acquire a spin lock; Try to disable interrupts and acquire a spin lock; Release a spin lock and enable interrupts; Initialize a spin lock; Acquire a spin lock; Try to acquire a spin lock; Release a spin lock

SYNOPSIS

#include <sync/chSpinLock.h>

void **svMaskedLockInit**(KnMaskedSpinLock * *lockaddr*, KnMaskedSpinLockInfo * *info*);

void **svMaskedLockGet**(KnMaskedSpinLock * *lockaddr*);

int **svMaskedLockTry**(KnMaskedSpinLock * *lockaddr*);

void **svMaskedLockRel**(KnMaskedSpinLock * *lockaddr*);

void **svSpinLockInit**(KnSpinLock * *lockaddr*, KnSpinLockInfo * *info*);

void **svSpinLockGet**(KnSpinLock * *lockaddr*);

int **svSpinLockTry**(KnSpinLock * *lockaddr*);

void **svSpinLockRel**(KnSpinLock * *lockaddr*);

int **svPreemptable**(void);

int **svIntrLevel**(void);

DESCRIPTION

Spin locks are mutual exclusion objects, used to protect very short critical sections. These critical sections must not include kernel calls and the data they manipulate must be locked in memory to avoid page faults.

In multiprocessor systems, threads loop actively when resources are not available. A spin lock is an object in memory, typically modified and checked through indivisible machine operations (such as test-and-set).

In monoprocessor systems, spin locks are implemented by disabling preemption or interrupts.

Note that while a thread is holding a spin lock, a masked spin lock, or has disabled preemption, it is no longer allowed to block. In this case, the list of system calls available to the thread is restricted to the following:

```
semV
threadSemPost
eventPost
msgPut
svSpinLockGet
svSpinLockRel
svMaskedLockGet
svMaskedLockRel
svMaskAll
svUnmaskAll
svUnmask
```

```
svPreemptable
svIntrLevel
svTimeoutSet
svTimeoutCancel
svPagesAllocate
```

The system not force the thread to restrict itself to the permitted calls. If the thread issues non-permitted calls, the system will hang.

The *svMaskedLockInit* and *svSpinLockInit* functions initialize a spin lock in the free state. The *info* argument allows the user to enforce lock hierarchy and / or to gather lock use / contention statistics in the future.

In multiprocessor systems, *svMaskedLockGet* atomically disables interrupts on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with interrupts enabled to reduce interrupt latency. This function can be called from within an interrupt handler.

The *svMaskedLockTry* function attempts to acquire a spin lock, but returns immediately without spinning if there is none available. This function can be called from an interrupt handler.

The *svMaskedLockRel* function releases a spin lock and re-enables interrupts on the current processor. This function can be called from within an interrupt handler.

The *svMaskedLockGet*, *svMaskedLockTry* and *svMaskedLockRel* functions only relate to the interrupt mask of the current processor, and do not affect interrupt processing on other processors. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

In multiprocessor systems, *svSpinLockGet* atomically disables preemption on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with preemption enabled to reduce scheduling latency. On a monoprocessor, *svSpinLockGet* disables preemption on the current processor. This function cannot be called from within an interrupt handler.

The *svSpinLockTry* try attempts to acquire a spin lock, returning immediately without spinning if there is none available. This function cannot be called from within an interrupt handler.

The *svSpinLockRel* function releases a spin lock. This function ca not be called from within an interrupt handler.

If the current thread is preemptable, *svPreemptable* returns a non-zero value, otherwise it returns zero.

If called from an interrupt level, *svIntrLevel* returns a non-zero value, otherwise it returns zero.

RETURN VALUE

If the lock is successfully acquired, *svSpinLockTry* returns the previous state of the lock (K_AVAIL). The *svMaskedLockTry* function returns K_AVAIL. The *svPreemptable* and *svIntrLevel* calls return a value which depends on the current execution context. Other calls return 0.

ERRORS

No error codes are returned. In DEBUG mode, the kernel debugger is invoked if the lock level hierarchy is violated, or if a spin lock cannot be acquired after a very large number of iterations, indicating a probable deadlock.

RESTRICTIONS

Hierarchy level checks are not performed in this version.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMaskedLockInit, svMaskedLockGet, svMaskedLockTry, svMaskedLockRel, svSpinLockInit, svSpinLockGet, svSpinLockTry, svSpinLockRel – Initialize a spin lock; Disable interrupts and acquire a spin lock; Try to disable interrupts and acquire a spin lock; Release a spin lock and enable interrupts; Initialize a spin lock; Acquire a spin lock; Try to acquire a spin lock; Release a spin lock

SYNOPSIS

```
#include <sync/chSpinLock.h>
void svMaskedLockInit(KnMaskedSpinLock * lockaddr, KnMaskedSpinLockInfo * info);

void svMaskedLockGet(KnMaskedSpinLock * lockaddr);

int svMaskedLockTry(KnMaskedSpinLock * lockaddr);

void svMaskedLockRel(KnMaskedSpinLock * lockaddr);

void svSpinLockInit(KnSpinLock * lockaddr, KnSpinLockInfo * info);

void svSpinLockGet(KnSpinLock * lockaddr);

int svSpinLockTry(KnSpinLock * lockaddr);

void svSpinLockRel(KnSpinLock * lockaddr);

int svPreemptable(void);

int svIntrLevel(void);
```

DESCRIPTION

Spin locks are mutual exclusion objects, used to protect very short critical sections. These critical sections must not include kernel calls and the data they manipulate must be locked in memory to avoid page faults.

In multiprocessor systems, threads loop actively when resources are not available. A spin lock is an object in memory, typically modified and checked through indivisible machine operations (such as test-and-set).

In monoprocessor systems, spin locks are implemented by disabling preemption or interrupts.

Note that while a thread is holding a spin lock, a masked spin lock, or has disabled preemption, it is no longer allowed to block. In this case, the list of system calls available to the thread is restricted to the following:

```
semV
threadSemPost
eventPost
msgPut
svSpinLockGet
svSpinLockRel
svMaskedLockGet
svMaskedLockRel
svMaskAll
svUnmaskAll
svUnmask
```

```
svPreemptable
svIntrLevel
svTimeoutSet
svTimeoutCancel
svPagesAllocate
```

The system not force the thread to restrict itself to the permitted calls. If the thread issues non-permitted calls, the system will hang.

The *svMaskedLockInit* and *svSpinLockInit* functions initialize a spin lock in the free state. The *info* argument allows the user to enforce lock hierarchy and / or to gather lock use / contention statistics in the future.

In multiprocessor systems, *svMaskedLockGet* atomically disables interrupts on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with interrupts enabled to reduce interrupt latency. This function can be called from within an interrupt handler.

The *svMaskedLockTry* function attempts to acquire a spin lock, but returns immediately without spinning if there is none available. This function can be called from an interrupt handler.

The *svMaskedLockRel* function releases a spin lock and re-enables interrupts on the current processor. This function can be called from within an interrupt handler.

The *svMaskedLockGet*, *svMaskedLockTry* and *svMaskedLockRel* functions only relate to the interrupt mask of the current processor, and do not affect interrupt processing on other processors. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

In multiprocessor systems, *svSpinLockGet* atomically disables preemption on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with preemption enabled to reduce scheduling latency. On a monoprocessor, *svSpinLockGet* disables preemption on the current processor. This function cannot be called from within an interrupt handler.

The *svSpinLockTry* try attempts to acquire a spin lock, returning immediately without spinning if there is none available. This function cannot be called from within an interrupt handler.

The *svSpinLockRel* function releases a spin lock. This function ca not be called from within an interrupt handler.

If the current thread is preemptable, *svPreemptable* returns a non-zero value, otherwise it returns zero.

If called from an interrupt level, *svIntrLevel* returns a non-zero value, otherwise it returns zero.

RETURN VALUE

If the lock is successfully acquired, *svSpinLockTry* returns the previous state of the lock (K_AVAIL). The *svMaskedLockTry* function returns K_AVAIL. The *svPreemptable* and *svIntrLevel* calls return a value which depends on the current execution context. Other calls return 0.

ERRORS

No error codes are returned. In DEBUG mode, the kernel debugger is invoked if the lock level hierarchy is violated, or if a spin lock cannot be acquired after a very large number of iterations, indicating a probable deadlock.

RESTRICTIONS

Hierarchy level checks are not performed in this version.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMaskedLockInit, svMaskedLockGet, svMaskedLockTry, svMaskedLockRel, svSpinLockInit, svSpinLockGet, svSpinLockTry, svSpinLockRel – Initialize a spin lock; Disable interrupts and acquire a spin lock; Try to disable interrupts and acquire a spin lock; Release a spin lock and enable interrupts; Initialize a spin lock; Acquire a spin lock; Try to acquire a spin lock; Release a spin lock

SYNOPSIS

```
#include <sync/chSpinLock.h>
void svMaskedLockInit(KnMaskedSpinLock * lockaddr, KnMaskedSpinLockInfo * info);

void svMaskedLockGet(KnMaskedSpinLock * lockaddr);

int svMaskedLockTry(KnMaskedSpinLock * lockaddr);

void svMaskedLockRel(KnMaskedSpinLock * lockaddr);

void svSpinLockInit(KnSpinLock * lockaddr, KnSpinLockInfo * info);

void svSpinLockGet(KnSpinLock * lockaddr);

int svSpinLockTry(KnSpinLock * lockaddr);

void svSpinLockRel(KnSpinLock * lockaddr);

int svPreemptable(void);

int svIntrLevel(void);
```

DESCRIPTION

Spin locks are mutual exclusion objects, used to protect very short critical sections. These critical sections must not include kernel calls and the data they manipulate must be locked in memory to avoid page faults.

In multiprocessor systems, threads loop actively when resources are not available. A spin lock is an object in memory, typically modified and checked through indivisible machine operations (such as test-and-set).

In monoprocessor systems, spin locks are implemented by disabling preemption or interrupts.

Note that while a thread is holding a spin lock, a masked spin lock, or has disabled preemption, it is no longer allowed to block. In this case, the list of system calls available to the thread is restricted to the following:

```
semV
threadSemPost
eventPost
msgPut
svSpinLockGet
svSpinLockRel
svMaskedLockGet
svMaskedLockRel
svMaskAll
svUnmaskAll
svUnmask
```

```
svPreemptable
svIntrLevel
svTimeoutSet
svTimeoutCancel
svPagesAllocate
```

The system not force the thread to restrict itself to the permitted calls. If the thread issues non-permitted calls, the system will hang.

The *svMaskedLockInit* and *svSpinLockInit* functions initialize a spin lock in the free state. The *info* argument allows the user to enforce lock hierarchy and / or to gather lock use / contention statistics in the future.

In multiprocessor systems, *svMaskedLockGet* atomically disables interrupts on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with interrupts enabled to reduce interrupt latency. This function can be called from within an interrupt handler.

The *svMaskedLockTry* function attempts to acquire a spin lock, but returns immediately without spinning if there is none available. This function can be called from an interrupt handler.

The *svMaskedLockRel* function releases a spin lock and re-enables interrupts on the current processor. This function can be called from within an interrupt handler.

The *svMaskedLockGet*, *svMaskedLockTry* and *svMaskedLockRel* functions only relate to the interrupt mask of the current processor, and do not affect interrupt processing on other processors. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

In multiprocessor systems, *svSpinLockGet* atomically disables preemption on the current processor and acquires a spin lock. This routine spins actively on the lock value until it becomes free. Note that spinning is performed with preemption enabled to reduce scheduling latency. On a monoprocessor, *svSpinLockGet* disables preemption on the current processor. This function cannot be called from within an interrupt handler.

The *svSpinLockTry* try attempts to acquire a spin lock, returning immediately without spinning if there is none available. This function cannot be called from within an interrupt handler.

The *svSpinLockRel* function releases a spin lock. This function ca not be called from within an interrupt handler.

If the current thread is preemptable, *svPreemptable* returns a non-zero value, otherwise it returns zero.

If called from an interrupt level, *svIntrLevel* returns a non-zero value, otherwise it returns zero.

RETURN VALUE

If the lock is successfully acquired, *svSpinLockTry* returns the previous state of the lock (K_AVAIL). The *svMaskedLockTry* function returns K_AVAIL. The *svPreemptable* and *svIntrLevel* calls return a value which depends on the current execution context. Other calls return 0.

ERRORS

No error codes are returned. In DEBUG mode, the kernel debugger is invoked if the lock level hierarchy is violated, or if a spin lock cannot be acquired after a very large number of iterations, indicating a probable deadlock.

RESTRICTIONS

Hierarchy level checks are not performed in this version.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svSysCtx – get system context table address

SYNOPSIS

```
#include <exec/chContext.h>
KnChorusContext *svSysCtx(void);
```

DESCRIPTION

The *svSysCtx* function returns the address of the system context table which structure is defined in *exec/chContext.h* .

RETURN VALUE

Address of the system context table.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svSysPanic – trigger the invocation of the panic handler

SYNOPSIS

#include <exec/chPanic.h>
int **svSysPanic**(unsigned *panicNo /* panic number */*, void **args /* panic argument */*, unsigned *argsSize /* panic argument size */*, char **mess /* panic message */*, char **fileName /* file name */*, int *lineNumber /* line number */*);

FEATURES

CORE

DESCRIPTION

The *svSysPanic* system call allows Chorus applications to trigger the execution of the kernel panic handling sequence. When *svSysPanic* is invoked by an application, the kernel executes the panic handling strategy, just as if the panic had been detected by the kernel itself. If a panic handler has been connected by the BSP the kernel invokes this handler, passing the *svSysPanic* arguments unchanged. If no panic handler has been connected, the kernel performs default panic handling (usually by entering the kernel debugger).

The *panicNo* field identifies the panic. It should normally take a value not already assigned to Chorus components, except in the case of generic panics such as debug assertions, for which a single generic number has been assigned.

The *panicArgs* member is a pointer to a panic-specific data structure. The *panicArgsSize* member indicates the size of the panic-specific structure.

This kernel service is only available to supervisor threads.

RETURN VALUE

The *svSysPanic* system call never returns.

RESTRICTIONS

This call is restricted to *SUPERVISOR* threads.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svSysTimeout, svSysTimeoutSet, svSysTimeoutCancel, svTimeoutGetRes – Request a timeout; Cancel a timeout; Get timeout resolution

SYNOPSIS

```
#include <exec/chTimeout.h>
int svSysTimeoutSet(KnTimeout * timeout, KnTimeVal * waitLimit, int flag, KnLapDesc * lapdesc);

int svSysTimeoutCancel(KnTimeout * timeout);

int svTimeoutGetRes(KnTimeVal * resolution);
```

FEATURES

TIMEOUT

DESCRIPTION

The `svSysTimeoutSet()` system call declares *lapdesc* as the descriptor of a raw lap (see `svLapCreate`(2K)), which must be invoked when the interval of time specified by **waitLimit* (see `intro`(2K)) has elapsed.

The *waitLimit* pointer argument refers to a `KnTimeVal` structure whose members are defined in `sysTime`(2K) . If the `KnTimeVal` structure indicated by *waitLimit* is not equal to the *resolution* , then the time value composed by the `tmSec` and `tmNSec` fields is rounded. The fields are rounded down to the nearest multiple of the values in *resolution* unless the time is zero, in which case they are rounded up to the values in *resolution* .

The lap handler of the *lapdesc* descriptor is a special kind of interrupt handler which is executed solely in `SUPERVISOR` execution mode, its code and accessed data must be within the locked-in-memory regions of a `SUPERVISOR` actor. The set of kernel calls that can be used in this timeout handler is limited, as for any interrupt handler.

The argument passed to the lap handler is the *timeout* parameter specified by `svSysTimeoutSet()` . Fields within the `KnTimeout` structure are initialized and modified solely within the nucleus and are inaccessible to the application.

The *flag* argument is reserved for future use and should therefore be set to `0` to ensure compatibility with future releases of the system.

The `svSysTimeoutCancel()` system call attempts to cancel a timeout request. It takes as an argument the address of the `KnTimeout` object used in the call to `svSysTimeoutSet()` . If the timeout request is still pending, it is immediately canceled and a value is returned showing that the timer had not yet expired. Otherwise, the timeout interval had expired and the timeout handler was called, in which case nothing occurs and the value returned indicates this.

The `svTimeoutGetRes()` system call obtains the resolution of the **waitLimit* argument of `svSysTimeoutSet()` . The time value returned in *resolution* represents the smallest possible difference between two distinct **waitLimit* values.

The `svSysTimeoutSet()`, `svTimeoutGetRes()` and `svSysTimeoutCancel()` system calls are restricted to `SUPERVISOR` threads.

The `svSysTimeoutSet()` system call must not be called using a `KnTimeout` which has an existing timeout pending (use `svSysTimeoutCancel()` to cancel the request first).

RETURN VALUES

Upon completion, `svSysTimeoutSet()` returns `K_OK` if the timeout was successfully entered. Otherwise, a negative error code is returned specifying the reason it was not successful. Upon completion, `svSysTimeoutCancel()` returns a boolean value set to `1` if the request was still pending, or `0` if the timeout handler was called.

ERRORS

[K_EINVAL] The **waitLimit* value specified is invalid, or *lapdesc* is not a valid raw lap descriptor.

The value indicated by **waitLimit* is invalid if either the `tmSec` or `tmNSec` fields of the structure are negative, or if the structure is poorly constructed (for example, if the number of nanoseconds in `tmNSec` makes `tmNSec` greater than one second).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svIntrConnect`(2K), `svLapCreate`(2K), `sysTime`(2K)

NAME

svSysTimeout, svSysTimeoutSet, svSysTimeoutCancel, svTimeoutGetRes – Request a timeout; Cancel a timeout; Get timeout resolution

SYNOPSIS

#include <exec/chTimeout.h>
int **svSysTimeoutSet**(KnTimeout * *timeout*, KnTimeVal * *waitLimit*, int *flag*, KnLapDesc * *lapdesc*);

int **svSysTimeoutCancel**(KnTimeout * *timeout*);

int **svTimeoutGetRes**(KnTimeVal * *resolution*);

FEATURES

TIMEOUT

DESCRIPTION

The `svSysTimeoutSet()` system call declares *lapdesc* as the descriptor of a raw lap (see `svLapCreate`(2K)), which must be invoked when the interval of time specified by **waitLimit* (see `intro`(2K)) has elapsed.

The *waitLimit* pointer argument refers to a `KnTimeVal` structure whose members are defined in `sysTime`(2K) . If the `KnTimeVal` structure indicated by *waitLimit* is not equal to the *resolution* , then the time value composed by the `tmSec` and `tmNSec` fields is rounded. The fields are rounded down to the nearest multiple of the values in *resolution* unless the time is zero, in which case they are rounded up to the values in *resolution* .

The lap handler of the *lapdesc* descriptor is a special kind of interrupt handler which is executed solely in `SUPERVISOR` execution mode, its code and accessed data must be within the locked-in-memory regions of a `SUPERVISOR` actor. The set of kernel calls that can be used in this timeout handler is limited, as for any interrupt handler.

The argument passed to the lap handler is the *timeout* parameter specified by `svSysTimeoutSet()` . Fields within the `KnTimeout` structure are initialized and modified solely within the nucleus and are inaccessible to the application.

The *flag* argument is reserved for future use and should therefore be set to `0` to ensure compatibility with future releases of the system.

The `svSysTimeoutCancel()` system call attempts to cancel a timeout request. It takes as an argument the address of the `KnTimeout` object used in the call to `svSysTimeoutSet()` . If the timeout request is still pending, it is immediately canceled and a value is returned showing that the timer had not yet expired. Otherwise, the timeout interval had expired and the timeout handler was called, in which case nothing occurs and the value returned indicates this.

The `svTimeoutGetRes()` system call obtains the resolution of the **waitLimit* argument of `svSysTimeoutSet()` . The time value returned in *resolution* represents the smallest possible difference between two distinct **waitLimit* values.

The `svSysTimeoutSet( )` , `svTimeoutGetRes( )` and `svSysTimeoutCancel( )` system calls are restricted to `SUPERVISOR` threads.

The `svSysTimeoutSet( )` system call must not be called using a `KnTimeout` which has an existing timeout pending (use `svSysTimeoutCancel( )` to cancel the request first).

RETURN VALUES

Upon completion, `svSysTimeoutSet( )` returns `K_OK` if the timeout was successfully entered. Otherwise, a negative error code is returned specifying the reason it was not successful. Upon completion, `svSysTimeoutCancel( )` returns a boolean value set to `1` if the request was still pending, or `0` if the timeout handler was called.

ERRORS

[K_EINVAL] The **waitLimit* value specified is invalid, or *lapdesc* is not a valid raw lap descriptor.

The value indicated by **waitLimit* is invalid if either the `tmSec` or `tmNSec` fields of the structure are negative, or if the structure is poorly constructed (for example, if the number of nanoseconds in `tmNSec` makes `tmNSec` greater than one second).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svIntrConnect`(2K) , `svLapCreate`(2K) , `sysTime`(2K)

NAME

svSysTimeout, svSysTimeoutSet, svSysTimeoutCancel, svTimeoutGetRes – Request a timeout; Cancel a timeout; Get timeout resolution

SYNOPSIS

#include <exec/chTimeout.h>
int **svSysTimeoutSet**(KnTimeout * *timeout*, KnTimeVal * *waitLimit*, int *flag*, KnLapDesc * *lapdesc*);

int **svSysTimeoutCancel**(KnTimeout * *timeout*);

int **svTimeoutGetRes**(KnTimeVal * *resolution*);

FEATURES

TIMEOUT

DESCRIPTION

The `svSysTimeoutSet()` system call declares *lapdesc* as the descriptor of a raw lap (see `svLapCreate`(2K)), which must be invoked when the interval of time specified by **waitLimit* (see `intro`(2K)) has elapsed.

The *waitLimit* pointer argument refers to a `KnTimeVal` structure whose members are defined in `sysTime`(2K) . If the `KnTimeVal` structure indicated by *waitLimit* is not equal to the *resolution* , then the time value composed by the `tmSec` and `tmNSec` fields is rounded. The fields are rounded down to the nearest multiple of the values in *resolution* unless the time is zero, in which case they are rounded up to the values in *resolution* .

The lap handler of the *lapdesc* descriptor is a special kind of interrupt handler which is executed solely in `SUPERVISOR` execution mode, its code and accessed data must be within the locked-in-memory regions of a `SUPERVISOR` actor. The set of kernel calls that can be used in this timeout handler is limited, as for any interrupt handler.

The argument passed to the lap handler is the *timeout* parameter specified by `svSysTimeoutSet()` . Fields within the `KnTimeout` structure are initialized and modified solely within the nucleus and are inaccessible to the application.

The *flag* argument is reserved for future use and should therefore be set to `0` to ensure compatibility with future releases of the system.

The `svSysTimeoutCancel()` system call attempts to cancel a timeout request. It takes as an argument the address of the `KnTimeout` object used in the call to `svSysTimeoutSet()` . If the timeout request is still pending, it is immediately canceled and a value is returned showing that the timer had not yet expired. Otherwise, the timeout interval had expired and the timeout handler was called, in which case nothing occurs and the value returned indicates this.

The `svTimeoutGetRes()` system call obtains the resolution of the **waitLimit* argument of `svSysTimeoutSet()` . The time value returned in *resolution* represents the smallest possible difference between two distinct **waitLimit* values.

The `svSysTimeoutSet( )` , `svTimeoutGetRes( )` and `svSysTimeoutCancel( )` system calls are restricted to `SUPERVISOR` threads.

The `svSysTimeoutSet( )` system call must not be called using a `KnTimeout` which has an existing timeout pending (use `svSysTimeoutCancel( )` to cancel the request first).

RETURN VALUES

Upon completion, `svSysTimeoutSet( )` returns `K_OK` if the timeout was successfully entered. Otherwise, a negative error code is returned specifying the reason it was not successful. Upon completion, `svSysTimeoutCancel( )` returns a boolean value set to `1` if the request was still pending, or `0` if the timeout handler was called.

ERRORS

[K_EINVAL] The **waitLimit* value specified is invalid, or *lapdesc* is not a valid raw lap descriptor.

The value indicated by **waitLimit* is invalid if either the `tmSec` or `tmNSec` fields of the structure are negative, or if the structure is poorly constructed (for example, if the number of nanoseconds in `tmNSec` makes `tmNSec` greater than one second).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svIntrConnect`(2K) , `svLapCreate`(2K) , `sysTime`(2K)

NAME

svSysTrapHandler, svSysTrapHandlerConnect, svSysTrapHandlerDisconnect, svSysTrapHandlerGetConnected – Connect a trap handler; Disconnect a trap handler; Get a trap handler

SYNOPSIS

#include <exec/chTrap.h>
int **svSysTrapHandlerConnect**(unsigned int *trapnumber*, KnLapDesc * *newhandler*);

int **svSysTrapHandlerDisconnect**(unsigned int *trapnumber*, KnLapDesc * *curhandler*);

int **svSysTrapHandlerGetConnected**(unsigned int *trapnumber*, KnLapDesc * *curhandler*);

FEATURES

CORE

DESCRIPTION

The *svSysTrapHandler* calls are used to manage system trap handlers. These calls are restricted to *SUPERVISOR* threads.

The *trapnumber* parameter identifies the trap number to which the handler must be connected.

Trap handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

When a thread executes a hardware trap instruction, the system invokes (see *lapInvoke(2K)*) the trap handler connected to the trap number specified by the trap instruction.

The argument of the lap trap handler is a pointer to a `KnSysTrapDesc` data structure. This structure has the following fields:

```
KnThreadCtx *threadCtx ;
unsigned int trapNumber ;
```

The *threadCtx* field gives access to the register context of the thread saved immediatly after it performed the hardware trap instruction. This context is processor—dependent and may be modified by the trap handler.

The *trapNumber* field is a processor—dependent trap number.

The *svSysTrapHandlerConnect* system call duplicates the lap descriptor pointed to by *newhandler* into the lap descriptor associated with *trapNumber* .

The *svSysTrapHandlerDisconnect* system call clears the lap descriptor associated with *trapNumber* . If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svSysTrapHandlerGetConnected* system call copies the lap descriptor associated with *trapNumber* at the location pointed to by *curhandler* ..

RETURN VALUE

Upon successful completion, the calls return K_OK, otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svSysTrapHandlerConnect* is called and there is already a valid lap descriptor installed for the trap *trapnumber* .
[K_EINVAL]	*trapnumber* is invalid. The *curhandler* lap descriptor specified to *svSysTrapHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svSysTrapHandler, svSysTrapHandlerConnect, svSysTrapHandlerDisconnect, svSysTrapHandlerGetConnected – Connect a trap handler; Disconnect a trap handler; Get a trap handler

SYNOPSIS

#include <exec/chTrap.h>
int **svSysTrapHandlerConnect**(unsigned int *trapnumber*, KnLapDesc * *newhandler*);

int **svSysTrapHandlerDisconnect**(unsigned int *trapnumber*, KnLapDesc * *curhandler*);

int **svSysTrapHandlerGetConnected**(unsigned int *trapnumber*, KnLapDesc * *curhandler*);

FEATURES

CORE

DESCRIPTION

The *svSysTrapHandler* calls are used to manage system trap handlers. These calls are restricted to *SUPERVISOR* threads.

The *trapnumber* parameter identifies the trap number to which the handler must be connected.

Trap handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

When a thread executes a hardware trap instruction, the system invokes (see *lapInvoke(2K)*) the trap handler connected to the trap number specified by the trap instruction.

The argument of the lap trap handler is a pointer to a `KnSysTrapDesc` data structure. This structure has the following fields:

```
KnThreadCtx *threadCtx ;
unsigned int trapNumber ;
```

The *threadCtx* field gives access to the register context of the thread saved immediatly after it performed the hardware trap instruction. This context is processor—dependent and may be modified by the trap handler.

The *trapNumber* field is a processor—dependent trap number.

The *svSysTrapHandlerConnect* system call duplicates the lap descriptor pointed to by *newhandler* into the lap descriptor associated with *trapNumber* .

The *svSysTrapHandlerDisconnect* system call clears the lap descriptor associated with *trapNumber* . If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svSysTrapHandlerGetConnected* system call copies the lap descriptor associated with *trapNumber* at the location pointed to by *curhandler* ..

RETURN VALUE

Upon successful completion, the calls return K_OK, otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svSysTrapHandlerConnect* is called and there is already a valid lap descriptor installed for the trap *trapnumber* .
[K_EINVAL]	*trapnumber* is invalid. The *curhandler* lap descriptor specified to *svSysTrapHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svSysTrapHandler, svSysTrapHandlerConnect, svSysTrapHandlerDisconnect, svSysTrapHandlerGetConnected – Connect a trap handler; Disconnect a trap handler; Get a trap handler

SYNOPSIS

#include <exec/chTrap.h>
int **svSysTrapHandlerConnect**(unsigned int *trapnumber*, KnLapDesc * *newhandler*);

int **svSysTrapHandlerDisconnect**(unsigned int *trapnumber*, KnLapDesc * *curhandler*);

int **svSysTrapHandlerGetConnected**(unsigned int *trapnumber*, KnLapDesc * *curhandler*);

FEATURES

CORE

DESCRIPTION

The *svSysTrapHandler* calls are used to manage system trap handlers. These calls are restricted to *SUPERVISOR* threads.

The *trapnumber* parameter identifies the trap number to which the handler must be connected.

Trap handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

When a thread executes a hardware trap instruction, the system invokes (see *lapInvoke(2K)*) the trap handler connected to the trap number specified by the trap instruction.

The argument of the lap trap handler is a pointer to a `KnSysTrapDesc` data structure. This structure has the following fields:

```
KnThreadCtx *threadCtx ;
unsigned int trapNumber ;
```

The *threadCtx* field gives access to the register context of the thread saved immediatly after it performed the hardware trap instruction. This context is processor—dependent and may be modified by the trap handler.

The *trapNumber* field is a processor—dependent trap number.

The *svSysTrapHandlerConnect* system call duplicates the lap descriptor pointed to by *newhandler* into the lap descriptor associated with *trapNumber* .

The *svSysTrapHandlerDisconnect* system call clears the lap descriptor associated with *trapNumber* . If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svSysTrapHandlerGetConnected* system call copies the lap descriptor associated with *trapNumber* at the location pointed to by *curhandler* ..

RETURN VALUE

Upon successful completion, the calls return K_OK, otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svSysTrapHandlerConnect* is called and there is already a valid lap descriptor installed for the trap *trapnumber* .
[K_EINVAL]	*trapnumber* is invalid. The *curhandler* lap descriptor specified to *svSysTrapHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svSysTrapHandler, svSysTrapHandlerConnect, svSysTrapHandlerDisconnect, svSysTrapHandlerGetConnected – Connect a trap handler; Disconnect a trap handler; Get a trap handler

SYNOPSIS

#include <exec/chTrap.h>
int **svSysTrapHandlerConnect**(unsigned int *trapnumber*, KnLapDesc * *newhandler*);

int **svSysTrapHandlerDisconnect**(unsigned int *trapnumber*, KnLapDesc * *curhandler*);

int **svSysTrapHandlerGetConnected**(unsigned int *trapnumber*, KnLapDesc * *curhandler*);

FEATURES

CORE

DESCRIPTION

The *svSysTrapHandler* calls are used to manage system trap handlers. These calls are restricted to *SUPERVISOR* threads.

The *trapnumber* parameter identifies the trap number to which the handler must be connected.

Trap handlers are specified in the form of lap descriptors (see *svLapCreate(2K)).*

When a thread executes a hardware trap instruction, the system invokes (see *lapInvoke(2K)*) the trap handler connected to the trap number specified by the trap instruction.

The argument of the lap trap handler is a pointer to a `KnSysTrapDesc` data structure. This structure has the following fields:

```
KnThreadCtx *threadCtx ;
unsigned int trapNumber ;
```

The *threadCtx* field gives access to the register context of the thread saved immediatly after it performed the hardware trap instruction. This context is processor—dependent and may be modified by the trap handler.

The *trapNumber* field is a processor—dependent trap number.

The *svSysTrapHandlerConnect* system call duplicates the lap descriptor pointed to by *newhandler* into the lap descriptor associated with *trapNumber* .

The *svSysTrapHandlerDisconnect* system call clears the lap descriptor associated with *trapNumber* . If *curhandler* is not equal to K_CONNECTED_LAP, it must point to a lap descriptor identical to the lap descriptor currently installed.

The *svSysTrapHandlerGetConnected* system call copies the lap descriptor associated with *trapNumber* at the location pointed to by *curhandler* ..

RETURN VALUE

Upon successful completion, the calls return K_OK, otherwise a negative error code is returned.

ERRORS

[K_EBUSY]	*svSysTrapHandlerConnect* is called and there is already a valid lap descriptor installed for the trap *trapnumber* .
[K_EINVAL]	*trapnumber* is invalid. The *curhandler* lap descriptor specified to *svSysTrapHandlerDisconnect* is not equal to K_CONNECTED_LAP and does not match the lap descriptor currently installed.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svLapCreate`(2K) , `lapInvoke`(2K)

NAME

svThreadVirtualTimeout, svThreadVirtualTimeoutSet, svThreadVirtualTimeoutCancel – Set a thread's virtual timeout; Cancel a thread's virtual timeout

SYNOPSIS

#include <vtimer/chVtimer.h>
int **svThreadVirtualTimeoutSet**(KnCap * *actor*, KnThreadLid *threadli*, KnVirtTimeout * *vtimeout*, KnTimeVal * *cputime*, int *flag*, KnLapDesc * *lapdesc*);

int **svThreadVirtualTimeoutCancel**(KnVirtTimeout * *vtimeout*);

FEATURES

VTIMER

DESCRIPTION

The *svThreadVirtualTimeoutSet* system call sets a timeout on the execution time of the thread whose local identifier is *threadLi* within the actor specified by *actorcap* .

Once the designated thread has consumed *cputime* of additional execution time, the lap handler designated by *lapdesc* is invoked with *vtimeout* as its argument (see *lapInvoke(2K)*).

If *flag* is set to K_VTIME_INTERNAL, only execution time in the threads' owning actor is counted toward the timeout.

If *flag* is set to K_VTIME_TOTAL, all execution time is counted, regardless of whether or not the thread is executing in a cross-actor invocation.

The *lapDesc* argument is a lap descriptor previously created by *svLapCreate* .

The *KnVirtTimeout* object is opaque; it must be pre-allocated by the caller, but its fields are inaccessible to the caller. Virtual timeouts are always relative.

Virtual timeout handlers are not invoked at interrupt level. The execution environment is similar to that of an abort handler. The handler is executed by the affected thread, and has no restrictions on use of system calls. A thread can be diverted to execute a handler only when executing in its owning actor; during any cross-actor invocation (trap, ipc, or lap call), virtual timeout handler execution is masked.

When a thread is deleted, all the timeouts posted for this thread are implicitly cancelled.

The *svThreadVirtualTimeoutCancel* system call attempts to cancel a virtual timeout request. The *vtimeout* argument contains the address of the *KnVirtTimeout* object used in the corresponding call to *svThreadVirtualTimeoutSet* . If the timeout request is still pending, it is cancelled and the call returns K_EOK. If the virtual timeout interval had passed and the handler invocation has been posted, the call takes no action and returns K_ETIMEOUT. In the latter case, no information is available as to whether the handler execution had actually begun, nor whether the handler is still executing. A given timeout must not be

cancelled a second time. This implies as well, that the timeouts of a thread which has been previously deleted, must not be cancelled.

A timeout cannot be reinstalled. To reinstall a timeout request, the timeout must be previously cancelled, otherwise an unpredictable behaviour may occur).

RETURN VALUES

Upon successful completion, *svThreadVirtualTimeoutSet* and *svThreadVirtualTimeoutCancel* return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	The actor capability is not valid.
[K_EUNKNOWN]	*actor* is unreachable.
[K_ETIMEOUT]	(*svThreadVirtualTimeoutCancel* only) the virtual interval had already expired, and the handler had been queued for execution.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `svActorVirtualTimeoutSet`(2K) , `svLapCreate`(2K) , `threadTimes`(2K) , `virtualTimeGetRes`(2K)

NAME

svThreadVirtualTimeout, svThreadVirtualTimeoutSet, svThreadVirtualTimeoutCancel – Set a thread's virtual timeout; Cancel a thread's virtual timeout

SYNOPSIS

#include <vtimer/chVtimer.h>
int **svThreadVirtualTimeoutSet**(KnCap * *actor*, KnThreadLid *threadli*, KnVirtTimeout * *vtimeout*, KnTimeVal * *cputime*, int *flag*, KnLapDesc * *lapdesc*);

int **svThreadVirtualTimeoutCancel**(KnVirtTimeout * *vtimeout*);

FEATURES

VTIMER

DESCRIPTION

The *svThreadVirtualTimeoutSet* system call sets a timeout on the execution time of the thread whose local identifier is *threadLi* within the actor specified by *actorcap* .

Once the designated thread has consumed *cputime* of additional execution time, the lap handler designated by *lapdesc* is invoked with *vtimeout* as its argument (see *lapInvoke(2K)*).

If *flag* is set to K_VTIME_INTERNAL, only execution time in the threads' owning actor is counted toward the timeout.

If *flag* is set to K_VTIME_TOTAL, all execution time is counted, regardless of whether or not the thread is executing in a cross-actor invocation.

The *lapDesc* argument is a lap descriptor previously created by *svLapCreate* .

The *KnVirtTimeout* object is opaque; it must be pre-allocated by the caller, but its fields are inaccessible to the caller. Virtual timeouts are always relative.

Virtual timeout handlers are not invoked at interrupt level. The execution environment is similar to that of an abort handler. The handler is executed by the affected thread, and has no restrictions on use of system calls. A thread can be diverted to execute a handler only when executing in its owning actor; during any cross-actor invocation (trap, ipc, or lap call), virtual timeout handler execution is masked.

When a thread is deleted, all the timeouts posted for this thread are implicitly cancelled.

The *svThreadVirtualTimeoutCancel* system call attempts to cancel a virtual timeout request. The *vtimeout* argument contains the address of the *KnVirtTimeout* object used in the corresponding call to *svThreadVirtualTimeoutSet* . If the timeout request is still pending, it is cancelled and the call returns K_EOK. If the virtual timeout interval had passed and the handler invocation has been posted, the call takes no action and returns K_ETIMEOUT. In the latter case, no information is available as to whether the handler execution had actually begun, nor whether the handler is still executing. A given timeout must not be

cancelled a second time. This implies as well, that the timeouts of a thread which has been previously deleted, must not be cancelled.

A timeout cannot be reinstalled. To reinstall a timeout request, the timeout must be previously cancelled, otherwise an unpredictable behaviour may occur).

RETURN VALUES

Upon successful completion, *svThreadVirtualTimeoutSet* and *svThreadVirtualTimeoutCancel* return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	The actor capability is not valid.
[K_EUNKNOWN]	*actor* is unreachable.
[K_ETIMEOUT]	(*svThreadVirtualTimeoutCancel* only) the virtual interval had already expired, and the handler had been queued for execution.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `svActorVirtualTimeoutSet`(2K) , `svLapCreate`(2K) , `threadTimes`(2K) , `virtualTimeGetRes`(2K)

NAME

svThreadVirtualTimeout, svThreadVirtualTimeoutSet, svThreadVirtualTimeoutCancel – Set a thread's virtual timeout; Cancel a thread's virtual timeout

SYNOPSIS

#include <vtimer/chVtimer.h>
int **svThreadVirtualTimeoutSet**(KnCap * *actor*, KnThreadLid *threadli*, KnVirtTimeout * *vtimeout*, KnTimeVal * *cputime*, int *flag*, KnLapDesc * *lapdesc*);

int **svThreadVirtualTimeoutCancel**(KnVirtTimeout * *vtimeout*);

FEATURES

VTIMER

DESCRIPTION

The *svThreadVirtualTimeoutSet* system call sets a timeout on the execution time of the thread whose local identifier is *threadLi* within the actor specified by *actorcap* .

Once the designated thread has consumed *cputime* of additional execution time, the lap handler designated by *lapdesc* is invoked with *vtimeout* as its argument (see *lapInvoke(2K)*).

If *flag* is set to K_VTIME_INTERNAL, only execution time in the threads' owning actor is counted toward the timeout.

If *flag* is set to K_VTIME_TOTAL, all execution time is counted, regardless of whether or not the thread is executing in a cross-actor invocation.

The *lapDesc* argument is a lap descriptor previously created by *svLapCreate* .

The *KnVirtTimeout* object is opaque; it must be pre-allocated by the caller, but its fields are inaccessible to the caller. Virtual timeouts are always relative.

Virtual timeout handlers are not invoked at interrupt level. The execution environment is similar to that of an abort handler. The handler is executed by the affected thread, and has no restrictions on use of system calls. A thread can be diverted to execute a handler only when executing in its owning actor; during any cross-actor invocation (trap, ipc, or lap call), virtual timeout handler execution is masked.

When a thread is deleted, all the timeouts posted for this thread are implicitly cancelled.

The *svThreadVirtualTimeoutCancel* system call attempts to cancel a virtual timeout request. The *vtimeout* argument contains the address of the *KnVirtTimeout* object used in the corresponding call to *svThreadVirtualTimeoutSet* . If the timeout request is still pending, it is cancelled and the call returns K_EOK. If the virtual timeout interval had passed and the handler invocation has been posted, the call takes no action and returns K_ETIMEOUT. In the latter case, no information is available as to whether the handler execution had actually begun, nor whether the handler is still executing. A given timeout must not be

cancelled a second time. This implies as well, that the timeouts of a thread which has been previously deleted, must not be cancelled.

A timeout cannot be reinstalled. To reinstall a timeout request, the timeout must be previously cancelled, otherwise an unpredictable behaviour may occur).

RETURN VALUES

Upon successful completion, *svThreadVirtualTimeoutSet* and *svThreadVirtualTimeoutCancel* return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	The actor capability is not valid.
[K_EUNKNOWN]	*actor* is unreachable.
[K_ETIMEOUT]	(*svThreadVirtualTimeoutCancel* only) the virtual interval had already expired, and the handler had been queued for execution.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`lapInvoke`(2K) , `svActorVirtualTimeoutSet`(2K) , `svLapCreate`(2K) , `threadTimes`(2K) , `virtualTimeGetRes`(2K)

NAME

svSysTimeout, svSysTimeoutSet, svSysTimeoutCancel, svTimeoutGetRes – Request a timeout; Cancel a timeout; Get timeout resolution

SYNOPSIS

```
#include <exec/chTimeout.h>
int svSysTimeoutSet(KnTimeout * timeout, KnTimeVal * waitLimit, int flag, KnLapDesc * lapdesc);

int svSysTimeoutCancel(KnTimeout * timeout);

int svTimeoutGetRes(KnTimeVal * resolution);
```

FEATURES

TIMEOUT

DESCRIPTION

The `svSysTimeoutSet()` system call declares *lapdesc* as the descriptor of a raw lap (see `svLapCreate`(2K)), which must be invoked when the interval of time specified by **waitLimit* (see `intro`(2K)) has elapsed.

The *waitLimit* pointer argument refers to a `KnTimeVal` structure whose members are defined in `sysTime`(2K) . If the `KnTimeVal` structure indicated by *waitLimit* is not equal to the *resolution* , then the time value composed by the `tmSec` and `tmNSec` fields is rounded. The fields are rounded down to the nearest multiple of the values in *resolution* unless the time is zero, in which case they are rounded up to the values in *resolution* .

The lap handler of the *lapdesc* descriptor is a special kind of interrupt handler which is executed solely in `SUPERVISOR` execution mode, its code and accessed data must be within the locked-in-memory regions of a `SUPERVISOR` actor. The set of kernel calls that can be used in this timeout handler is limited, as for any interrupt handler.

The argument passed to the lap handler is the *timeout* parameter specified by `svSysTimeoutSet()` . Fields within the `KnTimeout` structure are initialized and modified solely within the nucleus and are inaccessible to the application.

The *flag* argument is reserved for future use and should therefore be set to `0` to ensure compatibility with future releases of the system.

The `svSysTimeoutCancel()` system call attempts to cancel a timeout request. It takes as an argument the address of the `KnTimeout` object used in the call to `svSysTimeoutSet()` . If the timeout request is still pending, it is immediately canceled and a value is returned showing that the timer had not yet expired. Otherwise, the timeout interval had expired and the timeout handler was called, in which case nothing occurs and the value returned indicates this.

The `svTimeoutGetRes()` system call obtains the resolution of the **waitLimit* argument of `svSysTimeoutSet()` . The time value returned in *resolution* represents the smallest possible difference between two distinct **waitLimit* values.

The `svSysTimeoutSet( )` , `svTimeoutGetRes( )` and `svSysTimeoutCancel( )` system calls are restricted to `SUPERVISOR` threads.

The `svSysTimeoutSet( )` system call must not be called using a `KnTimeout` which has an existing timeout pending (use `svSysTimeoutCancel( )` to cancel the request first).

RETURN VALUES

Upon completion, `svSysTimeoutSet( )` returns `K_OK` if the timeout was successfully entered. Otherwise, a negative error code is returned specifying the reason it was not successful. Upon completion, `svSysTimeoutCancel( )` returns a boolean value set to `1` if the request was still pending, or `0` if the timeout handler was called.

ERRORS

[K_EINVAL] The **waitLimit* value specified is invalid, or *lapdesc* is not a valid raw lap descriptor.

The value indicated by **waitLimit* is invalid if either the `tmSec` or `tmNSec` fields of the structure are negative, or if the structure is poorly constructed (for example, if the number of nanoseconds in `tmNSec` makes `tmNSec` greater than one second).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svIntrConnect`(2K) , `svLapCreate`(2K) , `sysTime`(2K)

NAME

svTrapConnect, svTrapDisConnect – Connect a trap handler; Disconnect a trap handler

SYNOPSIS

#include <exec/chTrap.h>
int **svTrapConnect**(unsigned int *trapno*, KnTrapHdl *routine*);

int **svTrapDisConnect**(unsigned int *trapno*);

FEATURES

CORE

DESCRIPTION

The *svTrapConnect* system call connects a trap handler to the hardware trap specified by *trapno*. The *trapno* value is machine-dependent. The *routine* pointer points to the handler. The handler is a function which takes two arguments:

```
int handler (ctx, itno)

 KnThreadCtx* ctx ;
 unsigned int itno ;
```

After being connected, a handler will be activated upon invocation of the trap *trapno* by a thread. The current thread register context is saved. The handler takes as arguments a pointer to the saved context (*ctx*) and the trap number (*trapno).*

Only one handler may be connected to a given trap number at a time. The *svTrapDisConnect* system call disconnects the handler *routine* previously associated with the *trapno* trap number. These calls are restricted to *SUPERVISOR* threads. Trap handlers are executed in *SUPERVISOR* execution mode. The handler'ss code and data accessed must be parts of locked-in-memory regions of a *SUPERVISOR* actor.

When a thread executes a trap handler (via a hardware trap invocation), its execution actor is changed to match the actor which declared the handler. It is still attached to the actor within which it was created, although the executed code is part of another actor's address space (a *SUPERVISOR* actor).

The execution actor of a thread determines the actor which must be used for the invocation of exception handlers (see *svExcHandler* (2K)) if the thread encounters an exception. It also determines when abort handlers are invoked (see *svAbortHandler* (2K)).

The execution actor does not determine the interpretation of actor references, the thread's home actor is always used. Consequently, care must be taken when invoking *actorSelf* (2K) or using the K_MYACTOR value in actor—related calls (as in *threadCreate* (2K) for example) within a trap handler; they refer to the thread's home actor. The execution actor must be explicitly named by its capability, if needed (for example if the handler is to manage this actor's address space).

When the trap handler returns, the previous thread's execution actor is restored.

ERRORS

[K_EBUSY]	There is already a valid descriptor installed for the trap number.

[K_EINVAL] Incorrect trap number.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svExcHandler`(2K)

NAME

svTrapConnect, svTrapDisConnect – Connect a trap handler; Disconnect a trap handler

SYNOPSIS

#include <exec/chTrap.h>
int **svTrapConnect**(unsigned int *trapno*, KnTrapHdl *routine*);

int **svTrapDisConnect**(unsigned int *trapno*);

FEATURES

CORE

DESCRIPTION

The *svTrapConnect* system call connects a trap handler to the hardware trap specified by *trapno*. The *trapno* value is machine-dependent. The *routine* pointer points to the handler. The handler is a function which takes two arguments:

```
int handler (ctx, itno)

 KnThreadCtx* ctx ;
 unsigned int itno ;
```

After being connected, a handler will be activated upon invocation of the trap *trapno* by a thread. The current thread register context is saved. The handler takes as arguments a pointer to the saved context (*ctx*) and the trap number (*trapno).*

Only one handler may be connected to a given trap number at a time. The *svTrapDisConnect* system call disconnects the handler *routine* previously associated with the *trapno* trap number. These calls are restricted to *SUPERVISOR* threads. Trap handlers are executed in *SUPERVISOR* execution mode. The handler'ss code and data accessed must be parts of locked-in-memory regions of a *SUPERVISOR* actor.

When a thread executes a trap handler (via a hardware trap invocation), its execution actor is changed to match the actor which declared the handler. It is still attached to the actor within which it was created, although the executed code is part of another actor's address space (a *SUPERVISOR* actor).

The execution actor of a thread determines the actor which must be used for the invocation of exception handlers (see *svExcHandler* (2K)) if the thread encounters an exception. It also determines when abort handlers are invoked (see *svAbortHandler* (2K)).

The execution actor does not determine the interpretation of actor references, the thread's home actor is always used. Consequently, care must be taken when invoking *actorSelf* (2K) or using the K_MYACTOR value in actor—related calls (as in *threadCreate* (2K) for example) within a trap handler; they refer to the thread's home actor. The execution actor must be explicitly named by its capability, if needed (for example if the handler is to manage this actor's address space).

When the trap handler returns, the previous thread's execution actor is restored.

ERRORS

[K_EBUSY]	There is already a valid descriptor installed for the trap number.

[K_EINVAL] Incorrect trap number.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svExcHandler`(2K)

NAME

svMaskAll, svUnmaskAll, svUnmask – Disable interrupt processing; Enable interrupt processing; Reenable interrupt processing

SYNOPSIS

```
#include <sync/chSpinLock.h>
int svMaskAll(void);

void svUnmaskAll(void);

void svUnmask(int oldval);
```

DESCRIPTION

The *svMaskAll* function disables interrupts on the current processor by setting the processor interrupt mask to its maximum value. There is no effect on other processors. It returns the previous interrupt mask.

The *svUnmaskAll* function enables interrupts on the current processor by setting the processor interrupt mask to its minimum value. There is no effect on other processors.

The *svUnmask* function resets the interrupt mask to a state represented by *oldval*. The *oldval* value must have been returned by a previous invocation of *svMaskAll*. Pairs of *svMaskAll* and *svUnmask* may be nested. The interrupt mask is part of a thread's execution context, it is saved as part of the thread's status when the thread is suspended. If a thread performs a blocking call after calling *svMaskAll*, interrupt processing may be enabled by another scheduled thread. It will be disabled again when the thread is restarted. Such situations should be avoided.

The *svMaskAll*, *svUnmaskAll* and *svUnmask* functions only relate to the interrupt mask from processor standpoint. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

These calls are restricted to *SUPERVISOR* threads. Their use must be limited to a very small number of instructions. They should not be used to ensure mutual exclusion. The routines *svMaskedLockGet* and *svMaskedLockRel* should be used for this purpose.

RETURN VALUE

svMaskAll returns the current interrupt masking status.

ERRORS

No error messages are returned .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svMaskedLockInit`(2K)

NAME

svMaskAll, svUnmaskAll, svUnmask – Disable interrupt processing; Enable interrupt processing; Reenable interrupt processing

SYNOPSIS

```
#include <sync/chSpinLock.h>
int svMaskAll(void);

void svUnmaskAll(void);

void svUnmask(int oldval);
```

DESCRIPTION

The *svMaskAll* function disables interrupts on the current processor by setting the processor interrupt mask to its maximum value. There is no effect on other processors. It returns the previous interrupt mask.

The *svUnmaskAll* function enables interrupts on the current processor by setting the processor interrupt mask to its minimum value. There is no effect on other processors.

The *svUnmask* function resets the interrupt mask to a state represented by *oldval.* The *oldval* value must have been returned by a previous invocation of *svMaskAll.* Pairs of *svMaskAll* and *svUnmask* may be nested. The interrupt mask is part of a thread's execution context, it is saved as part of the thread's status when the thread is suspended. If a thread performs a blocking call after calling *svMaskAll,* interrupt processing may be enabled by another scheduled thread. It will be disabled again when the thread is restarted. Such situations should be avoided.

The *svMaskAll, svUnmaskAll* and *svUnmask* functions only relate to the interrupt mask from processor standpoint. If the machine is equipped with an external interrupt controller, these calls have no effect on the controller status.

These calls are restricted to *SUPERVISOR* threads. Their use must be limited to a very small number of instructions. They should not be used to ensure mutual exclusion. The routines *svMaskedLockGet* and *svMaskedLockRel* should be used for this purpose.

RETURN VALUE

svMaskAll returns the current interrupt masking status.

ERRORS

No error messages are returned .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svMaskedLockInit(2K)

NAME

svVirtualTimeoutSet, svVirtualTimeoutCancel – Set a virtual timeout; Cancel a virtual timeout

SYNOPSIS

#include <vtimer/chVtimer.h>
int **svVirtualTimeoutSet**(KnCap * *actor*, KnThreadLid *threadli*, KnVirtTimeout * *vtimeout*, KnVirtToHdl *handler*, KnTimeVal * *cputime*, int *flag*);

int **svVirtualTimeoutCancel**(KnVirtTimeout * *vtimeout*);

FEATURES

VTIMER

DESCRIPTION

The *svVirtualTimeoutSet* system call sets a timeout on the execution time of the thread whose local identifier is *threadLi* within the actor specified by *actorcap.*

If *threadli* is set to K_ALLACTORTHREADS, the timeout applies to the total execution time of all threads in the actor designated by *actorcap* . Once the designated thread or threads have consumed *cputime* of additional execution time, the handler designated by *handler* is entered with *vtimeout* as its sole argument. (Virtual timeouts are always relative.)

If *flag* is set to K_VTIME_INTERNAL, only execution time in the threads' home actor is counted toward the timeout.

If *flag* is set to K_VTIME_TOTAL, all execution time is counted, regardless of whether or not the thread is executing in a cross-actor invocation.

In the K_ALLACTORTHREADS case, thread execution time is charged to the thread's owning actor only, not to the execution actor.

The *KnVirtTimeout* object is opaque; it must be pre-allocated by the caller, but its fields are inaccessible to the caller.

Virtual timeout handlers are not invoked at interrupt level. The execution environment is similar to that of an abort handler. The handler is executed by the affected thread (or the next affected thread to run, if an actor-level timeout), and has no restrictions on use of system calls. A thread can be diverted to execute a handler only when executing in its owning actor; during any cross-actor invocation (trap, ipc, or lap call), handler execution is masked.

The *svVirtualTimeoutCancel* system call attempts to cancel a virtual timeout request. The *vtimeout* argument contains the address of the *KnVirtTimeout* object used in the corresponding call to *svVirtTimeoutSet.* If the timeout request is still pending, it is cancelled and the call returns K_EOK. If the virtual timeout interval expired and the handler invocation has been posted, the call takes no action and returns K_ETIMEOUT. In the latter case, no information is available as to whether the handler execution has actually begun, nor whether the handler is still executing.

A timeout cannot be reinstalled. To reinstall a timeout request, the timeout must be previously cancelled, otherwise an unpredictable behaviour may occur.

RETURN VALUES

Upon successful completion, *svVirtualTimeoutSet* and *svTimeoutCancel* return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*threadli* does not designate a valid thread, or the time value referenced by *cputime* is invalid, or *flag* contains an invalid value, or actor capability is not valid.
[K_ENOMEM]	The system is ou t of resources.
[K_EUNKNOWN]	*actor* is unreachable.
[K_ETIMEOUT]	(*svVirtualTimeoutCancel* only) the virtual interval had already expired, and the handler had been queued for execution.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadTimes`(2K) , `virtualTimeGetRes`(2K)

NAME

svVirtualTimeoutSet, svVirtualTimeoutCancel – Set a virtual timeout; Cancel a virtual timeout

SYNOPSIS

#include <vtimer/chVtimer.h>
int **svVirtualTimeoutSet**(KnCap * *actor*, KnThreadLid *threadli*, KnVirtTimeout * *vtimeout*, KnVirtToHdl *handler*, KnTimeVal * *cputime*, int *flag*);

int **svVirtualTimeoutCancel**(KnVirtTimeout * *vtimeout*);

FEATURES

VTIMER

DESCRIPTION

The *svVirtualTimeoutSet* system call sets a timeout on the execution time of the thread whose local identifier is *threadLi* within the actor specified by *actorcap.*

If *threadli* is set to K_ALLACTORTHREADS, the timeout applies to the total execution time of all threads in the actor designated by *actorcap* . Once the designated thread or threads have consumed *cputime* of additional execution time, the handler designated by *handler* is entered with *vtimeout* as its sole argument. (Virtual timeouts are always relative.)

If *flag* is set to K_VTIME_INTERNAL, only execution time in the threads' home actor is counted toward the timeout.

If *flag* is set to K_VTIME_TOTAL, all execution time is counted, regardless of whether or not the thread is executing in a cross-actor invocation.

In the K_ALLACTORTHREADS case, thread execution time is charged to the thread's owning actor only, not to the execution actor.

The *KnVirtTimeout* object is opaque; it must be pre-allocated by the caller, but its fields are inaccessible to the caller.

Virtual timeout handlers are not invoked at interrupt level. The execution environment is similar to that of an abort handler. The handler is executed by the affected thread (or the next affected thread to run, if an actor-level timeout), and has no restrictions on use of system calls. A thread can be diverted to execute a handler only when executing in its owning actor; during any cross-actor invocation (trap, ipc, or lap call), handler execution is masked.

The *svVirtualTimeoutCancel* system call attempts to cancel a virtual timeout request. The *vtimeout* argument contains the address of the *KnVirtTimeout* object used in the corresponding call to *svVirtTimeoutSet.* If the timeout request is still pending, it is cancelled and the call returns K_EOK. If the virtual timeout interval expired and the handler invocation has been posted, the call takes no action and returns K_ETIMEOUT. In the latter case, no information is available as to whether the handler execution has actually begun, nor whether the handler is still executing.

A timeout cannot be reinstalled. To reinstall a timeout request, the timeout must be previously cancelled, otherwise an unpredictable behaviour may occur.

RETURN VALUES

Upon successful completion, *svVirtualTimeoutSet* and *svTimeoutCancel* return K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*threadli* does not designate a valid thread, or the time value referenced by *cputime* is invalid, or *flag* contains an invalid value, or actor capability is not valid.
[K_ENOMEM]	The system is ou t of resources.
[K_EUNKNOWN]	*actor* is unreachable.
[K_ETIMEOUT]	(*svVirtualTimeoutCancel* only) the virtual interval had already expired, and the handler had been queued for execution.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadTimes`(2K) , `virtualTimeGetRes`(2K)

NAME

sysBench – kernel benchmark utility

SYNOPSIS

#include <timer/chBench.h>
int **sysBench**(int *option*, int **uTime*);

FEATURES

PERF

DESCRIPTION

The *sysBench* system call returns a microsecond time value suitable for benchmarking. The *option* argument specifies whether the benchmark time value is for the beginning or end point of the benchmark interval. The argument *utime* is the address of an integer which *sysBench* uses to store a time value in microseconds.

It should be noted that the time value returned by *sysBench* does not necessarily correspond to any actual time, it could simply be an instantaneous value from a clock chip. On some platforms, time values can "wrap-around" after a certain amount of time, causing a subsequent bench point to return a lower time value, making the benchmark interval invalid. For example, on x86 architectures the maximum interval measurable using sysBench is 5 milliseconds.

Three options are currently supported:

K_BSTART — This option starts a benchmark interval by enabling benchmarking, disabling interrupts, and returns an instantaneous microsecond time value in *uTime*.

K_BPOINT — This option returns an instantaneous microsecond time value in *uTime*. This option should be called between a K_BSTART and a K_BSTOP call.

K_BSTOP — This disables benchmarking, re-enables interrupts appropriately, and returns another instantaneous microsecond time value in *uTime*.

RETURN VALUE

Upon successful completion K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL] An invalid option was passed as an argument, or the benchmark commands were not issued in the correct order (for example, K_BSTART was not issued before B_STOP or two K_BSTART commands were issued one after another).

EXAMPLES

This small code fragment illustrates a typical usage of *sysBench*.

```
int startTime, stopTime, intervalTime;
if ((res = sysBench(K_BSTART, &startTime)) != K_OK) {
      /* A benchmark was pending */
 return res;
}
```

```
 /* ... code to be measured ... */
if ((res = sysBench(K_BSTOP, &stopTime)) != K_OK) {
 /* K_BSTOP was already issued */
 return res;
}
intervalTime = stopTime - startTime;
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sysGetConf – Get Chorus module configuration value

SYNOPSIS

#include <exec/chModules.h>
int **sysGetConf**(const char **modName*, int *var*, int **pvalue*);

FEATURES

CORE

DESCRIPTION

The *sysGetConf* system call copies the value from the Chorus module corresponding to the module name *modName*, as defined in *chModules.h* into the integer pointed to by *pvalue*. The *modName* pointer must point to a null terminated string containing the name of the module whose configuration value is required. The *pvalue* pointer points to an integer which, on successful completion, will contain the value corresponding to the *var* configuration parameter of *modName*.

The var value may be either *K_GETCONF_VERSION* to get the version of a specified module, or a value defined in a specific module header (for example, *K_GETCONF_ACTOR_MAX* in *exec/chExec.h* for the maximum number of actors on a site, or *K_GETCONF_PORT_MAX* in *ipc/chIpc.h* for the maximum number of ports on a site).

RETURN VALUE

Upon successful completion, if the module specified is present, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_ENOTAVAILABLE] *modName*; the module is not present in the current Chorus run—time configuration. This is an easy way of testing for the presence of a module at run—time.

[K_EFAULT] Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sysGetEnv – Get a value from the Chorus configuration environment

SYNOPSIS

```
#include <env/chEnv.h>
int sysGetEnv(const char *envName, char *envValue, unsigned int *size);
```

FEATURES

ENV

DESCRIPTION

The *sysGetEnv* system call copies the value from the Chorus configuration environment defined by the variable *envName*, into the buffer pointed to by *envValue*. The *envName* configuration variable must point to a null terminated string containing the name of the configuration variable whose value is required. The *envValue* pointer points to a buffer which, on successful completion, will contain the value corresponding to *envName*.

On input,*size*points to the size in bytes of the buffer *envName*. The *sysGetEnv* call uses this value to determine whether there is sufficient space to copy the value of the variable.

On output, `size` points to the number of bytes in the value, including the terminating null character.

RETURN VALUE

Upon successful completion K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAIL]	*envName* the configuration variable does not exist in the current Chorus environment.
[K_ENOMEM]	For both user and supervisor access, the value pointed to by `size` was insufficient to store the entire value of *envName* in *envValue*. In this case, `size` points to the number of bytes necessary to contain the entire value. For user mode access, the string size of either *envName* or *envValue* exceeded the internal size limits imposed by the user system call implementation. The default values of these limits are 32 bytes for *envName*, and 256 bytes for *envValue*.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysSetEnv`(2K), `sysUnsetEnv`(2K)

NAME

sysLog – log a message in the kernel's cyclical buffer

SYNOPSIS

```
#include <log/chLog.h>
void sysLog(char *format, ... /* arg */);
```

FEATURES

LOG

DESCRIPTION

The `sysLog()` call logs a message in the kernel's cyclical buffer. The syntax of `sysLog()` is similar to that of `printf()` with the restriction that the only conversion specifications supported are "%s", "%d", and "%c".

The formatted string is truncated to `SYSLOG_MAX_LINE` characters, as defined in `<log/chLog.h>`.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NOTES

Important actors related to system administration, for example `pppstart.r`, `slattach.r` and `chat.r`, use `sysLog()`.

The system administrator can view lines in the system log using `cs`(1CC) and `chls`(1CC), as shown in the following examples:

```
host% chls -ll nblines
host% rsh target arun /bin/cs -ll nblines
```

where *nblines* is the number of lines to display counting from the end of the system log, and *target* is the target system hostname.

SEE ALSO

`cs`(1CC), `chls`(1CC)

NAME

sysRead, sysWrite, sysPoll – Read characters from the system console; Write characters to the system console; Poll characters from the system console

SYNOPSIS

```
#include <exec/chIo.h>
int sysRead(void * buf, unsigned int nchar);

int sysWrite(void * buf, unsigned int nchar);

int sysPoll(void * buf);
```

DESCRIPTION

The *sysRead* system call reads characters from the system console into the buffer pointed to by *buf* , until either *nchar* -1 characters are read, or a newline character is read and not transferred to *buf* . The characters will be echoed on the system console and the string is terminated with a null character. If *nchar* is equal to 1, one character is read, without echo.

The *sysWrite* system call attempts to write *nchar* characters to the system console from the buffer pointed to by *buf* .

The *sysPoll* system call attempts to read one character from the system console into the buffer pointed to by *buf* . The character will be echoed on the system console. The *sysPoll* system call is non-blocking.

RETURN VALUE

If successful, *sysRead* and *sysPoll* return the number of characters actually read, and *sysWrite* returns the number of characters actually written. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sysRead, sysWrite, sysPoll – Read characters from the system console; Write characters to the system console; Poll characters from the system console

SYNOPSIS

```
#include <exec/chIo.h>
int sysRead(void * buf, unsigned int nchar);

int sysWrite(void * buf, unsigned int nchar);

int sysPoll(void * buf);
```

DESCRIPTION

The *sysRead* system call reads characters from the system console into the buffer pointed to by *buf* , until either *nchar* -1 characters are read, or a newline character is read and not transferred to *buf* . The characters will be echoed on the system console and the string is terminated with a null character. If *nchar* is equal to 1, one character is read, without echo.

The *sysWrite* system call attempts to write *nchar* characters to the system console from the buffer pointed to by *buf* .

The *sysPoll* system call attempts to read one character from the system console into the buffer pointed to by *buf* . The character will be echoed on the system console. The *sysPoll* system call is non-blocking.

RETURN VALUE

If successful, *sysRead* and *sysPoll* return the number of characters actually read, and *sysWrite* returns the number of characters actually written. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sysReboot – request a reboot of the local site

SYNOPSIS

```
#include <restart/chRestart.h>
int sysReboot(KnRebootDesc*rebootdesc);
```

FEATURES

CORE

DESCRIPTION

The *sysReboot* system call requests a reboot of the local site.

The caller thread must be a supervisor thread (see *threadCreate(2K)*) or must belong to a system actor (see *actorCreate(2K)*).

The *rebootdesc* pointer must be either 0 or a pointer to a *KnRebootDesc* structure:

```
KnRebootMode mode ;
```

The *mode* field specifies the reboot request and can be one of the following values:

K_REBOOT_COLD	stops all kernel drivers and initiates a BSP-dependent cold reboot procedure. Note that *sysReboot* doesn't perform any shutdown operation on the POSIX personality.
K_REBOOT_HOT	is not available for applications and must be used by the operating system only
K_REBOOT_NEW	is not available for applications and must be used by the operating system only

sysReboot called with the argument 0 performs the above described K_REBOOT_COLD reboot request.

RETURN VALUE

If successful, *sysReboot* does not return, otherwise a negative error code is returned.

ERRORS

[K_EPRIV]	The caller thread is not a SUPERVISOR thread or does not belong to a SYSTEM actor.
[K_ENOTIMP]	The BSP doesn't implement cold reboot procedure.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysShutdown`(2K)

NAME

sysSetEnv – Set a value in the ChorusOS configuration environment

SYNOPSIS

#include <env/chEnv.h>
int **sysSetEnv**(const char **envName*, const char **envValue*);

DESCRIPTION

The sysSetEnv( ) system call inserts an environment variable and its value into the the ChorusOS configuration environment. The *envName* pointer points to a null terminated string containing the environment variable to be set. The *envValue* pointer points to a null-terminated string containing the corresponding value of the environment variable.

Any existing value for *envName* in the configuration environment is removed and replaced by *envValue*.

RETURN VALUES

Upon successful completion K_OK is returned and the environment variable name and value are inserted into the configuration environment. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL] *envName* was zero length, or an invalide name for a ChorusOS configuration environment variable.

[K_ENOMEM] For both user and supervisor access, there was insufficient space in the ChorusOS configuration environment to store *envName* and *envValue*. For user mode access, the string size of either *envName* or *envValue* exceeded the internal size limits imposed by the user system call implementation. The default values of these limits are 32 bytes for *envName*, and 256 bytes for *envValue*.

[K_EFAULT] Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

sysGetEnv(2K), sysUnsetEnv(2K)

NAME

sysShutdown – shut down or restart the system

SYNOPSIS

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

DESCRIPTION

`sysShutdown()` is executed by trusted user actors or supervisor actors to shut down the OS. It is a high-level interface intended for use in multi-command actors such as `C_INIT`(1M).

PARAMETERS

`sysShutdown()` parameters are similar to the options for `shutdown`(1M). If the first parameter is `-i`, then the subsequent digit indicates the system state and actions to perform as follows:

0	Shut down the entire system safely and reboot.
1	Perform a site restart. A site restart means that the entire system is shut down safely and then restarted from the system image, without accessing stable storage, such that direct hot restartable actors are automatically restarted from persistent memory. This state requires the `HOT_RESTART` feature. See the manual entry for `restart`(1M) for a more detailed description of site restart.

RETURN VALUES

Upon successful completetion, `0` is returned and the system is no longer available. Otherwise, a negative error code is returned if a nucleus error is encountered, or a positive error code is returned if an OS error is encountered.

ERRORS

Nucleus errors include any errors reported by `threadCreate`(2K) or `threadScheduler`(2K).

[EFAULT]	Actor capability of the invoker cannot be obtained.
[EINVAL]	Invalid argument.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`shutdown`(1M), `restart`(1M)

NAME

sysTime, sysTimeGetRes – get system time; get system time resolution

SYNOPSIS

```
#include <exec/chTime.h>
int sysTime(KnTimeVal * time);

int sysTimeGetRes(KnTimeVal * resolution);
```

FEATURES

SYSTIME

DESCRIPTION

The kernel maintains its time using a *KnTimeVal* structure, the members of which are the following:

```
long tmSec ; /* seconds */
long tmNSec ; /* nanoseconds */
```

The *tmSec* field represents the number of seconds, and the *tmNSec* field represents the number of additional nanoseconds.

This time is set to 0 when the kernel is initialized, and incremented by the timer period at each timer interrupt.

sysTime allows the user to get the value of the kernel time. *sysTime* returns the kernel time in the *KnTimeVal* structure pointed to by *time.*

sysTimeGetRes obtains the resolution of the *sysTime* system call. The time value returned in *resolution* represents the smallest possible difference between two distinct values of the system time.

RETURN VALUE

Upon successful completion K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

univTime(2K)

NAME

sysTime, sysTimeGetRes – get system time; get system time resolution

SYNOPSIS

```
#include <exec/chTime.h>
int sysTime(KnTimeVal * time);

int sysTimeGetRes(KnTimeVal * resolution);
```

FEATURES

SYSTIME

DESCRIPTION

The kernel maintains its time using a *KnTimeVal* structure, the members of which are the following:

```
long tmSec ; /* seconds */
long tmNSec ; /* nanoseconds */
```

The *tmSec* field represents the number of seconds, and the *tmNSec* field represents the number of additional nanoseconds.

This time is set to 0 when the kernel is initialized, and incremented by the timer period at each timer interrupt.

sysTime allows the user to get the value of the kernel time. *sysTime* returns the kernel time in the *KnTimeVal* structure pointed to by *time.*

sysTimeGetRes obtains the resolution of the *sysTime* system call. The time value returned in *resolution* represents the smallest possible difference between two distinct values of the system time.

RETURN VALUE

Upon successful completion K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

univTime(2K)

NAME

sysTimer, sysTimerStartPeriodic, sysTimerStartFreerun, sysTimerReadCounter, sysTimerGetCounterPeriod, sysTimerGetCounterFrequency, sysTimerStop – system timer management

SYNOPSIS

#include <perf/chBench.h>
KnError **sysTimerStartPeriodic**(KnTimeVal * *period*, TimerTickHandler *handler*, void * *cookie*);

KnError **sysTimerStartFreerun**(KnTimeVal * *minperiod*);

unsigned long **sysTimerReadCounter**(void);

unsigned long **sysTimerGetCounterPeriod**(void);

unsigned long **sysTimerGetCounterFrequency**(void);

void **sysTimerStop**(void);

FEATURES

PERF

DESCRIPTION

The system timer management services are provided by the microkernel to share the system timer with the application. They are intended mainly for benchmarking and profiling.

Typically, bench programs require free running timers and profiling handlers require to be invoked periodically.

`sysTimerStartPeriodic` must be issued to share the system timer between the system and the application. This starts the timer in periodic mode.

The `sysTimerStartPeriodic` is restricted to SUPERVISOR actors.

In case of failure, an error code is returned as described below:

`K_EBUSY`	The system timer is already in use by another application.
`K_EINVAL`	The given period *KnTimeVal *period* specified an illegal nanosecond value: lesser than 0 or greater than 1,000,000,000, or a non zero seconds value.
`K_EFAULT`	*KnTimeVal *period* points to an illegal address.
`K_EFAIL`	The timer can't be initialized with the required period.

The *handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level. The handler is invoked passing *cookie* as the only argument.

`sysTimerStartFreerun` disconnects the system periodic timer. This starts the timer in freerun mode. The *minperiod* specifies the minimum period required for the free-run timer. The implementation is free to program the timer with any period higher than the required one.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already shared by another application.

`K_EFAIL` The timer can't be initialized with the required interval.

The `sysTimerReadCounter` routine returns the current value of the timer counter. The value can be used to measure the elapsed time between two calls to the read_counter routine. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterPeriod` returns the difference between the maximum and minimum values of the timer counter. The routine may be called when the timer operates in any mode. The get_counter_period routine may be called from interrupt level. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterFrequency` returns the frequency of the timer counter in Hz. The routine may be called when the timer is operating in any mode. The get_counter_frequency routine may be called from interrupt level.

Invocation of the above three calls without a previous call to sysTimerStartPeriodic or sysTimerStartFreerun returns an undocumented value.

`sysTimerStop` returns the timer to the system. The timer is reprogrammed to the system frequency.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer`(9DDI)

NAME

sysTimer, sysTimerStartPeriodic, sysTimerStartFreerun, sysTimerReadCounter, sysTimerGetCounterPeriod, sysTimerGetCounterFrequency, sysTimerStop – system timer management

SYNOPSIS

```
#include <perf/chBench.h>
KnError sysTimerStartPeriodic(KnTimeVal * period, TimerTickHandler handler,
void * cookie);

KnError sysTimerStartFreerun(KnTimeVal * minperiod);

unsigned long sysTimerReadCounter(void);

unsigned long sysTimerGetCounterPeriod(void);

unsigned long sysTimerGetCounterFrequency(void);

void sysTimerStop(void);
```

FEATURES

PERF

DESCRIPTION

The system timer management services are provided by the microkernel to share the system timer with the application. They are intended mainly for benchmarking and profiling.

Typically, bench programs require free running timers and profiling handlers require to be invoked periodically.

`sysTimerStartPeriodic` must be issued to share the system timer between the system and the application. This starts the timer in periodic mode.

The `sysTimerStartPeriodic` is restricted to SUPERVISOR actors.

In case of failure, an error code is returned as described below:

`K_EBUSY`	The system timer is already in use by another application.
`K_EINVAL`	The given period *KnTimeVal *period* specified an illegal nanosecond value: lesser than 0 or greater than 1,000,000,000, or a non zero seconds value.
`K_EFAULT`	*KnTimeVal *period* points to an illegal address.
`K_EFAIL`	The timer can't be initialized with the required period.

The *handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level. The handler is invoked passing *cookie* as the only argument.

`sysTimerStartFreerun` disconnects the system periodic timer. This starts the timer in freerun mode. The *minperiod* specifies the minimum period required for the free-run timer. The implementation is free to program the timer with any period higher than the required one.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already shared by another application.

`K_EFAIL` The timer can't be initialized with the required interval.

The `sysTimerReadCounter` routine returns the current value of the timer counter. The value can be used to measure the elapsed time between two calls to the read_counter routine. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterPeriod` returns the difference between the maximum and minimum values of the timer counter. The routine may be called when the timer operates in any mode. The get_counter_period routine may be called from interrupt level. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterFrequency` returns the frequency of the timer counter in Hz. The routine may be called when the timer is operating in any mode. The get_counter_frequency routine may be called from interrupt level.

Invocation of the above three calls without a previous call to sysTimerStartPeriodic or sysTimerStartFreerun returns an undocumented value.

`sysTimerStop` returns the timer to the system. The timer is reprogrammed to the system frequency.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer`(9DDI)

NAME

sysTimer, sysTimerStartPeriodic, sysTimerStartFreerun, sysTimerReadCounter, sysTimerGetCounterPeriod, sysTimerGetCounterFrequency, sysTimerStop – system timer management

SYNOPSIS

#include <perf/chBench.h>
KnError **sysTimerStartPeriodic**(KnTimeVal * *period*, TimerTickHandler *handler*, void * *cookie*);

KnError **sysTimerStartFreerun**(KnTimeVal * *minperiod*);

unsigned long **sysTimerReadCounter**(void);

unsigned long **sysTimerGetCounterPeriod**(void);

unsigned long **sysTimerGetCounterFrequency**(void);

void **sysTimerStop**(void);

FEATURES

PERF

DESCRIPTION

The system timer management services are provided by the microkernel to share the system timer with the application. They are intended mainly for benchmarking and profiling.

Typically, bench programs require free running timers and profiling handlers require to be invoked periodically.

`sysTimerStartPeriodic` must be issued to share the system timer between the system and the application. This starts the timer in periodic mode.

The `sysTimerStartPeriodic` is restricted to SUPERVISOR actors.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already in use by another application.

`K_EINVAL` The given period *KnTimeVal *period* specified an illegal nanosecond value: lesser than 0 or greater than 1,000,000,000, or a non zero seconds value.

`K_EFAULT` *KnTimeVal *period* points to an illegal address.

`K_EFAIL` The timer can't be initialized with the required period.

The *handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level. The handler is invoked passing *cookie* as the only argument.

`sysTimerStartFreerun` disconnects the system periodic timer. This starts the timer in freerun mode. The *minperiod* specifies the minimum period required for the free-run timer. The implementation is free to program the timer with any period higher than the required one.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already shared by another application.

`K_EFAIL` The timer can't be initialized with the required interval.

The `sysTimerReadCounter` routine returns the current value of the timer counter. The value can be used to measure the elapsed time between two calls to the read_counter routine. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterPeriod` returns the difference between the maximum and minimum values of the timer counter. The routine may be called when the timer operates in any mode. The get_counter_period routine may be called from interrupt level. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterFrequency` returns the frequency of the timer counter in Hz. The routine may be called when the timer is operating in any mode. The get_counter_frequency routine may be called from interrupt level.

Invocation of the above three calls without a previous call to sysTimerStartPeriodic or sysTimerStartFreerun returns an undocumented value.

`sysTimerStop` returns the timer to the system. The timer is reprogrammed to the system frequency.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer`(9DDI)

NAME

sysTimer, sysTimerStartPeriodic, sysTimerStartFreerun, sysTimerReadCounter, sysTimerGetCounterPeriod, sysTimerGetCounterFrequency, sysTimerStop – system timer management

SYNOPSIS

#include <perf/chBench.h>
KnError **sysTimerStartPeriodic**(KnTimeVal * *period*, TimerTickHandler *handler*, void * *cookie*);

KnError **sysTimerStartFreerun**(KnTimeVal * *minperiod*);

unsigned long **sysTimerReadCounter**(void);

unsigned long **sysTimerGetCounterPeriod**(void);

unsigned long **sysTimerGetCounterFrequency**(void);

void **sysTimerStop**(void);

FEATURES

PERF

DESCRIPTION

The system timer management services are provided by the microkernel to share the system timer with the application. They are intended mainly for benchmarking and profiling.

Typically, bench programs require free running timers and profiling handlers require to be invoked periodically.

`sysTimerStartPeriodic` must be issued to share the system timer between the system and the application. This starts the timer in periodic mode.

The `sysTimerStartPeriodic` is restricted to SUPERVISOR actors.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already in use by another application.

`K_EINVAL` The given period *KnTimeVal *period* specified an illegal nanosecond value: lesser than 0 or greater than 1,000,000,000, or a non zero seconds value.

`K_EFAULT` *KnTimeVal *period* points to an illegal address.

`K_EFAIL` The timer can't be initialized with the required period.

The *handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level. The handler is invoked passing *cookie* as the only argument.

`sysTimerStartFreerun` disconnects the system periodic timer. This starts the timer in freerun mode. The *minperiod* specifies the minimum period required for the free-run timer. The implementation is free to program the timer with any period higher than the required one.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already shared by another application.

`K_EFAIL` The timer can't be initialized with the required interval.

The `sysTimerReadCounter` routine returns the current value of the timer counter. The value can be used to measure the elapsed time between two calls to the read_counter routine. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterPeriod` returns the difference between the maximum and minimum values of the timer counter. The routine may be called when the timer operates in any mode. The get_counter_period routine may be called from interrupt level. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterFrequency` returns the frequency of the timer counter in Hz. The routine may be called when the timer is operating in any mode. The get_counter_frequency routine may be called from interrupt level.

Invocation of the above three calls without a previous call to sysTimerStartPeriodic or sysTimerStartFreerun returns an undocumented value.

`sysTimerStop` returns the timer to the system. The timer is reprogrammed to the system frequency.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer`(9DDI)

NAME

sysTimer, sysTimerStartPeriodic, sysTimerStartFreerun, sysTimerReadCounter, sysTimerGetCounterPeriod, sysTimerGetCounterFrequency, sysTimerStop – system timer management

SYNOPSIS

#include <perf/chBench.h>
KnError **sysTimerStartPeriodic**(KnTimeVal * *period*, TimerTickHandler *handler*, void * *cookie*);

KnError **sysTimerStartFreerun**(KnTimeVal * *minperiod*);

unsigned long **sysTimerReadCounter**(void);

unsigned long **sysTimerGetCounterPeriod**(void);

unsigned long **sysTimerGetCounterFrequency**(void);

void **sysTimerStop**(void);

FEATURES

PERF

DESCRIPTION

The system timer management services are provided by the microkernel to share the system timer with the application. They are intended mainly for benchmarking and profiling.

Typically, bench programs require free running timers and profiling handlers require to be invoked periodically.

`sysTimerStartPeriodic` must be issued to share the system timer between the system and the application. This starts the timer in periodic mode.

The `sysTimerStartPeriodic` is restricted to SUPERVISOR actors.

In case of failure, an error code is returned as described below:

`K_EBUSY`	The system timer is already in use by another application.
`K_EINVAL`	The given period *KnTimeVal *period* specified an illegal nanosecond value: lesser than 0 or greater than 1,000,000,000, or a non zero seconds value.
`K_EFAULT`	*KnTimeVal *period* points to an illegal address.
`K_EFAIL`	The timer can't be initialized with the required period.

The *handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level. The handler is invoked passing *cookie* as the only argument.

`sysTimerStartFreerun` disconnects the system periodic timer. This starts the timer in freerun mode. The *minperiod* specifies the minimum period required for the free-run timer. The implementation is free to program the timer with any period higher than the required one.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already shared by another application.

`K_EFAIL` The timer can't be initialized with the required interval.

The `sysTimerReadCounter` routine returns the current value of the timer counter. The value can be used to measure the elapsed time between two calls to the read_counter routine. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterPeriod` returns the difference between the maximum and minimum values of the timer counter. The routine may be called when the timer operates in any mode. The get_counter_period routine may be called from interrupt level. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterFrequency` returns the frequency of the timer counter in Hz. The routine may be called when the timer is operating in any mode. The get_counter_frequency routine may be called from interrupt level.

Invocation of the above three calls without a previous call to sysTimerStartPeriodic or sysTimerStartFreerun returns an undocumented value.

`sysTimerStop` returns the timer to the system. The timer is reprogrammed to the system frequency.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer`(9DDI)

NAME

sysTimer, sysTimerStartPeriodic, sysTimerStartFreerun, sysTimerReadCounter, sysTimerGetCounterPeriod, sysTimerGetCounterFrequency, sysTimerStop – system timer management

SYNOPSIS

```
#include <perf/chBench.h>
KnError sysTimerStartPeriodic(KnTimeVal * period, TimerTickHandler handler,
void * cookie);

KnError sysTimerStartFreerun(KnTimeVal * minperiod);

unsigned long sysTimerReadCounter(void);

unsigned long sysTimerGetCounterPeriod(void);

unsigned long sysTimerGetCounterFrequency(void);

void sysTimerStop(void);
```

FEATURES

PERF

DESCRIPTION

The system timer management services are provided by the microkernel to share the system timer with the application. They are intended mainly for benchmarking and profiling.

Typically, bench programs require free running timers and profiling handlers require to be invoked periodically.

`sysTimerStartPeriodic` must be issued to share the system timer between the system and the application. This starts the timer in periodic mode.

The `sysTimerStartPeriodic` is restricted to SUPERVISOR actors.

In case of failure, an error code is returned as described below:

`K_EBUSY`	The system timer is already in use by another application.
`K_EINVAL`	The given period *KnTimeVal *period* specified an illegal nanosecond value: lesser than 0 or greater than 1,000,000,000, or a non zero seconds value.
`K_EFAULT`	*KnTimeVal *period* points to an illegal address.
`K_EFAIL`	The timer can't be initialized with the required period.

The *handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level. The handler is invoked passing *cookie* as the only argument.

`sysTimerStartFreerun` disconnects the system periodic timer. This starts the timer in freerun mode. The *minperiod* specifies the minimum period required for the free-run timer. The implementation is free to program the timer with any period higher than the required one.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already shared by another application.

`K_EFAIL` The timer can't be initialized with the required interval.

The `sysTimerReadCounter` routine returns the current value of the timer counter. The value can be used to measure the elapsed time between two calls to the read_counter routine. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterPeriod` returns the difference between the maximum and minimum values of the timer counter. The routine may be called when the timer operates in any mode. The get_counter_period routine may be called from interrupt level. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterFrequency` returns the frequency of the timer counter in Hz. The routine may be called when the timer is operating in any mode. The get_counter_frequency routine may be called from interrupt level.

Invocation of the above three calls without a previous call to sysTimerStartPeriodic or sysTimerStartFreerun returns an undocumented value.

`sysTimerStop` returns the timer to the system. The timer is reprogrammed to the system frequency.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer`(9DDI)

NAME

sysTimer, sysTimerStartPeriodic, sysTimerStartFreerun, sysTimerReadCounter, sysTimerGetCounterPeriod, sysTimerGetCounterFrequency, sysTimerStop – system timer management

SYNOPSIS

#include <perf/chBench.h>
KnError **sysTimerStartPeriodic**(KnTimeVal * *period*, TimerTickHandler *handler*, void * *cookie*);

KnError **sysTimerStartFreerun**(KnTimeVal * *minperiod*);

unsigned long **sysTimerReadCounter**(void);

unsigned long **sysTimerGetCounterPeriod**(void);

unsigned long **sysTimerGetCounterFrequency**(void);

void **sysTimerStop**(void);

FEATURES

PERF

DESCRIPTION

The system timer management services are provided by the microkernel to share the system timer with the application. They are intended mainly for benchmarking and profiling.

Typically, bench programs require free running timers and profiling handlers require to be invoked periodically.

`sysTimerStartPeriodic` must be issued to share the system timer between the system and the application. This starts the timer in periodic mode.

The `sysTimerStartPeriodic` is restricted to SUPERVISOR actors.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already in use by another application.

`K_EINVAL` The given period *KnTimeVal *period* specified an illegal nanosecond value: lesser than 0 or greater than 1,000,000,000, or a non zero seconds value.

`K_EFAULT` *KnTimeVal *period* points to an illegal address.

`K_EFAIL` The timer can't be initialized with the required period.

The *handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level. The handler is invoked passing *cookie* as the only argument.

`sysTimerStartFreerun` disconnects the system periodic timer. This starts the timer in freerun mode. The *minperiod* specifies the minimum period required for the free-run timer. The implementation is free to program the timer with any period higher than the required one.

In case of failure, an error code is returned as described below:

`K_EBUSY` The system timer is already shared by another application.

`K_EFAIL` The timer can't be initialized with the required interval.

The `sysTimerReadCounter` routine returns the current value of the timer counter. The value can be used to measure the elapsed time between two calls to the read_counter routine. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterPeriod` returns the difference between the maximum and minimum values of the timer counter. The routine may be called when the timer operates in any mode. The get_counter_period routine may be called from interrupt level. Invocation of this call without a previous call to share the system timer returns an undocumented value.

`sysTimerGetCounterFrequency` returns the frequency of the timer counter in Hz. The routine may be called when the timer is operating in any mode. The get_counter_frequency routine may be called from interrupt level.

Invocation of the above three calls without a previous call to sysTimerStartPeriodic or sysTimerStartFreerun returns an undocumented value.

`sysTimerStop` returns the timer to the system. The timer is reprogrammed to the system frequency.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer`(9DDI)

NAME

sysUnsetEnv – delete a value from the ChorusOS configuration environment

SYNOPSIS

#include <env/chEnv.h>
int **sysUnsetEnv**(const char **envName*);

DESCRIPTION

The sysUnsetEnv( ) system call deletes the environment variable pointed to by *envName*. This is a null-terminated string.

RETURN VALUES

Upon successful completion K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAIL] *envName* was not an existing configuration variable in the current ChorusOS environment.

[K_EFAULT] Some of the data provided are outside the current actor's address space.

[K_ENOMEM] For user mode access, the string size of *envName* exceeds the internal size limit imposed by the user system call implementation. The default value of this limit is 32 bytes.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

sysSetEnv(2K), sysGetEnv(2K)

NAME

sysRead, sysWrite, sysPoll – Read characters from the system console; Write characters to the system console; Poll characters from the system console

SYNOPSIS

```
#include <exec/chIo.h>
int sysRead(void * buf, unsigned int nchar);

int sysWrite(void * buf, unsigned int nchar);

int sysPoll(void * buf);
```

DESCRIPTION

The *sysRead* system call reads characters from the system console into the buffer pointed to by *buf* , until either *nchar* -1 characters are read, or a newline character is read and not transferred to *buf* . The characters will be echoed on the system console and the string is terminated with a null character. If *nchar* is equal to 1, one character is read, without echo.

The *sysWrite* system call attempts to write *nchar* characters to the system console from the buffer pointed to by *buf* .

The *sysPoll* system call attempts to read one character from the system console into the buffer pointed to by *buf* . The character will be echoed on the system console. The *sysPoll* system call is non-blocking.

RETURN VALUE

If successful, *sysRead* and *sysPoll* return the number of characters actually read, and *sysWrite* returns the number of characters actually written. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

threadAbort, threadAborted – Abort a thread; Check whether the current thread has been aborted

SYNOPSIS

```
#include <exec/chExec.h>
int threadAborted(void);

int threadAbort(KnCap * actorcap, int threadli);
```

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *threadAbort* system call aborts the thread whose local identifier is *threadli* in the actor whose capability is given by *actorcap* .

If *actorcap* is K_MYACTOR, the aborted thread will be in the current actor.

Aborting a thread has different effects depending on the thread's state:

- If the thread is blocked by an ABORTABLE call, (*threadDelay* (2K)), the thread exits its blocked state and the blocking kernel call returns a specific error code (K_EABORT); the abortion has been processed by the thread.
- Otherwise (the thread is running or blocked in a NON ABORTABLE call), the thread enters the ABORTED state; the abortion still has to be processed. The abortion processing can take three distinct forms:
  - When a running thread in the ABORTED state issues an ABORTABLE blocking call (*semP* (2K)), it does not enter the blocked state; the blocking call returns the specific error code K_EABORT immediately and the thread quits the ABORTED state (the abort has been processed by the thread).
  - A running thread can invoke the *threadAborted* system call at any time, in order to test whether it is in the ABORTED state. If the thread is in the ABORTED state *threadAborted* resets this state, and returns 1 (the abort has been processed by the thread). Subsequent calls to *threadAborted* will return 0 (if the thread is not aborted again).
  - Finally, SUPERVISOR actors may also be informed via a (asynchronous) handler call, when threads enter the ABORTED state, using *svAbortHandler* (2K). In this case, when the thread in the ABORTED state returns to execute within its home actor execution environment (see *svTrapConnect* (2K)), the actor's abort handler is invoked (if any), and the thread quits the ABORTED state (the abort has been processed by the actor). Consequently, calls to *threadAborted* from the thread will return 0 (if the thread is not aborted again).

RETURN VALUES

Upon successful completion, *threadAbort* returns 0. Otherwise, a negative error code is returned.

The *threadAborted* call returns 1 if the current thread is in the *ABORTED* state, 0 if not.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`semP`(2K) , `svAbortHandler`(2K) , `svTrapConnect`(2K) , `threadDelay`(2K)

NAME

threadAbort, threadAborted – Abort a thread; Check whether the current thread has been aborted

SYNOPSIS

```
#include <exec/chExec.h>
int threadAborted(void);

int threadAbort(KnCap * actorcap, int threadli);
```

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *threadAbort* system call aborts the thread whose local identifier is *threadli* in the actor whose capability is given by *actorcap* .

If *actorcap* is K_MYACTOR, the aborted thread will be in the current actor.

Aborting a thread has different effects depending on the thread's state:

- If the thread is blocked by an ABORTABLE call, (*threadDelay* (2K)), the thread exits its blocked state and the blocking kernel call returns a specific error code (K_EABORT); the abortion has been processed by the thread.
- Otherwise (the thread is running or blocked in a NON ABORTABLE call), the thread enters the ABORTED state; the abortion still has to be processed. The abortion processing can take three distinct forms:
  - When a running thread in the ABORTED state issues an ABORTABLE blocking call (*semP* (2K)), it does not enter the blocked state; the blocking call returns the specific error code K_EABORT immediately and the thread quits the ABORTED state (the abort has been processed by the thread).
  - A running thread can invoke the *threadAborted* system call at any time, in order to test whether it is in the ABORTED state. If the thread is in the ABORTED state *threadAborted* resets this state, and returns 1 (the abort has been processed by the thread). Subsequent calls to *threadAborted* will return 0 (if the thread is not aborted again).
  - Finally, SUPERVISOR actors may also be informed via a (asynchronous) handler call, when threads enter the ABORTED state, using *svAbortHandler* (2K). In this case, when the thread in the ABORTED state returns to execute within its home actor execution environment (see *svTrapConnect* (2K)), the actor's abort handler is invoked (if any), and the thread quits the ABORTED state (the abort has been processed by the actor). Consequently, calls to *threadAborted* from the thread will return 0 (if the thread is not aborted again).

RETURN VALUES

Upon successful completion, *threadAbort* returns 0. Otherwise, a negative error code is returned.

The *threadAborted* call returns 1 if the current thread is in the *ABORTED* state, 0 if not.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`semP`(2K) , `svAbortHandler`(2K) , `svTrapConnect`(2K) , `threadDelay`(2K)

NAME

threadActivate – make a thread active

SYNOPSIS

#include <exec/chExec.h>
int **threadActivate**(KnCap **actorcap*, KnThreadLid *threadli*);

FEATURES

CORE

DESCRIPTION

The *threadActivate* system call activates thread *threadli* in the actor *actorcap*. If the thread was created in *K_INACTIVE*, *threadActivate* will make it eligible to execute.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

threadCreate(2K)

NAME

threadBind – bind a thread to a processor

SYNOPSIS

#include <sched/chSched.h>
int **threadBind**(KnCap **actorcap*, KnThreadLid *threadli*, KnCpuId *newcpunb*, KnCpuId **oldcpunb*, KnBindId *bindid*);

FEATURES

SCHED_FIFO, SCHED_CLASS

DESCRIPTION

On a Shared Memory Processor based Architecture, a thread is bound by default to the processor on which *threadCreate* was performed. The thread binding can be changed using *threadBind.*

The *threadBind* system call binds the thread whose local identifier is *threadli*, in the actor whose capability is given by *actorcap*, to the processor designated by *newcpunb.*

If the thread is currently running on a different processor, it is cancelled and scheduled for execution on the processor specified.

If *actorcap* is K_MYACTOR, the thread is a thread of the current actor. In this case, if *threadli* is K_MYSELF, the current thread is used.

If *newcpunb* is K_SAMECPU, the thread keeps its current binding and the function returns the current binding if *oldcpunb* is not a null pointer.

The *bindid* parameter defines the type of binding applied to *threadli.* In the current release, the K_NONEXCLUSIVE binding type is mandatory.

RETURN VALUE

Upon successful completion K_OK is returned, otherwise a negative error code is returned.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor, or *newcpunb* is not a valid processor number, or *bindid* is not a valid binding type.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_ENOTIMP]	For a given thread, *threadBind* is not performed from the CPU of the current (old) binding or *threadBind* is called on a site running a kernel configured for uniprocessor architecture only.
[K_EBUSY]	*bindId* is not a valid binding type.

RESTRICTIONS

The target thread and the current thread must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadCreate`(2K)

NAME

threadContext – get and / or set the context of a thread

SYNOPSIS

#include <exec/chExec.h>
int **threadContext**(KnCap **actorcap*, KnThreadLid *threadli*, unsigned int *type*, void **oldcontext*, void **newcontext*);

FEATURES

CORE

DESCRIPTION

The *threadContext* system call gets and / or sets the execution context of the thread whose local identifier is *threadli*, in the actor whose capability is given by *actorcap*. If *actorcap* is K_MYACTOR, the thread is a thread of the current actor. In this case, if *threadli* is K_MYSELF, the current thread is used.

The *type* parameter selects the component of the execution context which is affected. If *type* is K_CURRCTX, the thread's *hardware* context is affected. In this case, *oldcontext* and *newcontext* must be pointers to *KnThreadCtx* structures defined in the *exec/chThCtx_f.h* header file. The fields of this structure are machine-dependent. A thread's *hardware* context is defined as the set of general—purpose machine register values (including stack pointer, program counter, and so on) put on the stack when an initial (non-nested) trap, exception, or preemption occurred. Any subsequent nested events (in the case of a trap or exception) have no effect on the thread's hardware context as accessed by K_CURRCTX.

If *type* is K_SUPPRIV, the thread's supervisor software register context is affected (see *threadLoadR*(2K)). In this case, oldcontext and newcontext must be pointers to an array of `void*` whose size is K_NBPRIVDATA.

If *type* is K_USERPRIV, the thread's user software register context is affected (see *threadLoadR*(2K)). In this case, oldcontext and newcontext must be pointers to an array of `void*` whose size is K_NBPRIVDATA. This type is valid only for threads of user privilege.

If *type* is K_SOFTCTX, the thread's software register context is affected (see *threadLoadR*(2K)). In this case, *oldcontext* and *newcontext* must be pointers to KnThreadSoftCtx structures, whose members are the following:

```
long  userReg ; /* user privilege register */
long  supReg ; /* supervisor privilege register */
```

Note that the caller thread must be a supervisor thread (see *threadCreate*(2K)) or must belong to a system actor (see *actorCreate*(2K)) in order to modify the *supReg* value.

In either case, the thread's current execution context is copied into the structure pointed to by *oldcontext* (if not NULL) in the client address space. The thread's new context will be taken from the structure pointed to by *newcontext* (if not NULL).

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided is out of the current actor's address space.
[K_EPRIV]	The caller thread is not allowed to perform the operation.
[K_ENOTAVAILABLE]	The hardware execution context of the target thread is not available.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`actorCreate`(2K), `threadCreate`(2K), `threadLoadR`(2K)

NAME

threadCreate – create a thread

SYNOPSIS

#include <exec/chExec.h>
int **threadCreate**(KnCap **actorcap*, KnThreadLid **threadli*, KnThreadStatus *status*, void **schedparam*, void **startinfo*);

DESCRIPTION

threadCreate creates a new thread in the actor the capability of which is given by *actorcap*. If *actorcap* is K_MYACTOR, the thread is created within the current actor.

threadCreate returns the local identifier of the new thread in *threadli*.

status is the thread's initial status: if *status* is K_INACTIVE, the thread will not execute until it is explicitly started using a *threadActivate*(2K). If *status* is K_ACTIVE, the thread can run immediately (assuming that the actor is in the *ACTIVE* state (see *actorStart*(2K)).

schedparam points to a descriptor of thread scheduling attributes (see *threadScheduler(2K)). If schedparm* is 0, the thread inherits the scheduling attributes of the parent thread.

startinfo points to a structure specifying the initial state of the new thread. The current implementation supports two types of *startinfo* structures: *KnDefaultStartInfo_f* and *KnStartInfo*

These structures are machine-dependent (*i.e.* may contain machine-dependent fields).

For most conventional processors (eg. x86/Pentium, SPARC, etc.), *KnDefaultStartInfo_f* contains the following fields:

```
KnStartInfoType   dsType ;
unsigned          dsSystemStackSize ;
KnPc              dsEntry ;
void*             dsUserStackPointer ;
KnThreadPrivilege dsPrivilege ;
```

The *dsType* field must be set equal to K_DEFAULT_START_INFO.

The *dsSystemStackSize* field provides a hint to the kernel of the supervisor stack size required by the thread. If *dsSystemStackSize* is equal to K_DEFAULT_STACK_SIZE, the default supervisor stack size will be used.

The *dsEntry* and *dsUserStackPointer* fields specify initial values of the program counter and the user stack pointer of the thread when activated. The program counter and the stack pointer are the only machine registers the values of which may be directly provided upon thread creation. Other register values may be initialized by creating a thread in *K_INACTIVE* state and calling *threadContext* before resuming it (see *threadActivate*(2K)). Otherwise, these registers are initialized with arbitrary values.

If the *dsPrivilege* field is equal to K_USERTHREAD, the thread is activated as a user mode thread with user privileges. If the *dsPrivilege* field is equal to K_SUPTHREAD, the thread is activated at supervisor privilege level and has access to restricted machine instructions and kernel calls. In the latter case, the *dsUserStackPointer* field is ignored.

Similarly, the *KnStartInfo* contains the following fields:

```
KnStartInfoType   dsType ;
unsigned          dsSystemStackSize ;
KnPc              dsEntry ;
void*             dsUserStackPointer ;
KnThreadPrivilege dsPrivilege ;
unsigned int   dsSoftReg ;
void*             dsSystemStackTop ;
```

The *dsType* field must be K_START_INFO possibly or'ed with K_START_INFO_SOFTREG and / or K_START_INFO_SYS_STACK bits.

The meaning of *dsSystemStackSize*, *dsEntry*, *dsUserStackPointer* and *dsPrivilege* fields remains the same as in *KnDefaultStartInfo_f* structure.

The value *dsSoftReg* field is only meaningful if K_START_INFO_SOFTREG bit is set in the *dsType* filed. It specifies the initial value of the thread's software register corresponding to its privilege level. This is used to pass an argument to the created thread, which gets this argument using *threadLoadR*(2K).

The value *dsSystemStackTop* field is only meaningful if K_START_INFO_SYS_STACK bit is set in the *dsType* field. It specifies the starting address of the memory area to be used as the created thread's system stack (while *dsSystemStackSize* gives, as usually, the size in bytes of this area).

WARNINGS

Note that in the current version, K_SUSPENDED is equivalent to K_INACTIVE and *threadResume*(2K) is equivalent to *threadActivate*(2K). K_SUSPENDED and *threadResume* are supported for compatibility reasons.

MEM_PROTECTED for ARM and i386 as well as MEM_VIRTUAL on i386 and ppc60x implementations do not support supervisor stacks greater than a tunable which is defined per platform. To get the default value of the system stack size, it is necessary to invoke the configuration tool (*configurator*).

The supervisor stack must be always allocated by a supervisor thread only. Consequently only supervisor threads are allowed to change the system stack.

RETURN VALUE

Upon successful completion, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.

[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_EPRIV]	The current thread is not allowed to create a *SUPERVISOR* thread.
[K_ENOTIMP]	*schedparam* requires a non-supported scheduling policy.
[K_ENOMEM]	The system is out of resources.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadActivate`(2K), `threadResume`(2K), `actorCreate`(2K), `threadContext`(2K), `threadScheduler`(2K)

NAME

threadDelay – delay the current thread

SYNOPSIS

#include <exec/chExec.h>
int **threadDelay**(KnTimeVal *_waitLimit_);

FEATURES

CORE

DESCRIPTION

threadDelay delays the execution of the current thread according to the value of **waitLimit*.

waitLimit is a pointer to a *KnTimeVal* structure containing a timeout interval as described in *sysTime(2K)*. Semantics and constant option values of *waitLimit* argument are fully described in *intro(2K)*. A *waitLimit* value of K_NOBLOCK (or zero) means that the thread is still ready to run after the call. In this case, *threadDelay* may have the effect of simply yielding the processor to another thread that is ready to run. The precise actions in this case depend on the scheduler module configured.

ERRORS

[K_EINVAL]	The *waitLimit* structure is not a valid *KnTimeVal*.
[K_ETIMEOUT]	The time out occurred.
[K_EABORT]	The thread has been aborted while delayed.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadAbort`(2K)

NAME

threadDelete – delete a thread

SYNOPSIS

```
#include <exec/chExec.h>
int threadDelete(KnCap *actorcap, KnThreadLid threadli);
```

DESCRIPTION

threadDelete deletes the thread the local identifier of which is *threadli* in the actor the capability of which is given by *actorcap.*

If *actorcap* is K_MYACTOR, the deleted thread must belong to the current actor.

In this case, if *threadli* is K_MYSELF, the current thread is deleted.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.
[K_BUSY]	The thread is already in the process of being deleted.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

threadCreate(2K)

NAME

threadLoadR, threadStoreR – Get the current thread's valid soft register value; Reset the current thread's valid soft register value

SYNOPSIS

#include <exec/chSoftCtx.h>
long **threadLoadR**(void);

long **threadStoreR**(long *reg*);

DESCRIPTION

The execution context of a thread (its various machine registers) is extended by two software registers, one for each of the possible privilege levels of a thread (*USER* or *SUPERVISOR*). Like hardware registers, these two registers are saved and restored at each thread switch. These registers provide the means for maintaining per-thread data efficiently.

The *threadLoadR* system call returns the value of the current thread's software register corresponding to the current privilege level of the thread.

The *threadStoreR* system call sets the current thread's software register corresponding to the current privilege level of the thread to the value specified by *reg* , and returns its new value.

The values of these registers may also be affected using *threadContext* (2K).

RETURN VALUE

The *threadLoadR* call returns the current register value. The *threadStoreR* call returns the new register value.

EXAMPLES

The software register facility can be used to store the thread's local identifier in its software register. However, this only impacts on performance, as *threadLoadR* does not imply a kernel call on most architectures, this is fully resolved in libraries which access a globally shared system context. However, *threadSelf* always implies a kernel call. In order to do this, call:

```
threadStoreR (threadSelf());
```

at the beginning of the thread's code; then invoke:

```
threadLoadR ;
```

instead of:

```
threadSelfC ;
```

A more common way to use this facility is to associate a per-thread structure with each thread of an actor, and to store the address of this structure in the software register. This will avoid systematically transferring this address as a parameter across the layers of the actor's code (the use of a global variable is not possible, especially on multiprocessor implementations). In this case, each actor's thread will perform a

```
threadStoreR (myArea);
```

at the beginning of the thread's code, where myArea is the address of the per thread's structure. Note that there are several ways for the thread to acquire the

```
(myArea);
```

value: it may be provided to the thread's routine as an argument (by initializing its stack correctly before creation); it may also be computed by the created thread as the result of a hash function (the hash function taking the thread's local identifier as input, and returning an entry in the array of per-thread records). An alternative to this initialization process is for the creator to call *threadContext* (2K) before resuming the thread. In each case, the per-thread record is subsequently accessed by:

```
threadLoadR ();
```

Finally, note that the software register may be used differently by different code layers: the sequence S in:

```
oldVal = threadLoadR ();
threadStoreR (newVal);
        /* S */
threadStoreR (oldVal);
```

may make private use of the software register value.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadContext`(2K)

NAME

threadName – get and / or set the symbolic name of a thread

SYNOPSIS

#include <exec/chExec.h>
int **threadName**(KnCap **actorcap*, int *threadli*, char **oldname*, char **newname*);

FEATURES

CORE

DESCRIPTION

The *threadName* system call gets and / or sets the symbolic name of the thread whose local identifier is *threadli*, in the actor whose capability is given by *actorcap*. If *actorcap* is K_MYACTOR, the target thread must be in the current actor. In this case, if *threadli* is K_MYSELF, the current thread is affected.

If *oldname* is not a NULL pointer, the thread's symbolic name is copied to the caller address space at the location specified by *oldname*. A thread's symbolic name has a maximum size of K_THREADNAMEMAX (including the NULL character).

If *newname* is not a NULL pointer, the thread's symbolic name is set to the new name pointed to by *newname* in the caller address space. The new name is truncated to a maximum size of K_THREADNAMEMAX (including the NULL character).

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadCreate`(2K)

NAME

threadSuspend, threadResume – Suspend a thread; Resume a thread

SYNOPSIS

```
#include <exec/chExec.h>
int threadSuspend(KnCap * actorcap, int threadli);

int threadResume(KnCap * actorcap, int threadli);
```

FEATURES

CORE

DESCRIPTION

NOTE: These calls are obsolete. Although they are maintained for compatibility reasons, they should be avoided wherever possible. See *threadActivate(2K)*, *threadStart(2K)*, and *threadSemInit(2K)* for documentation on other calls that can be used instead of *threadSuspend(2K)* and threadResume(2K). The *threadSuspend* system call puts the thread whose local identifier is *threadli* , in the actor whose capability is given by *actorcap* , into the *SUSPENDED* state.

A suspended thread may resume execution (enter the *ACTIVE* state) only if the thread is explicitly resumed using *threadResume(2K)*.

The *threadSuspend* and *threadResume* system calls are based on a per-thread suspend counter, initialized to 0 at thread creation, incremented by *threadSuspend* and decremented by *threadResume*.

The *threadResume* system call decrements the suspend counter of the thread whose local identifier is *threadli* , in the actor whose capability is given by *actorcap*. It only awakens the thread by setting its state to *ACTIVE* , if the suspend counter has become 0.

If a thread which is in a *BLOCKED* state is suspended using *threadSuspend*, the thread will suspend just after it exits its *BLOCKED* state. If it is resumed using *threadResume* while still in the *BLOCKED* state, it will continue its execution normally after it exits its *BLOCKED* state (if its suspend counter has become 0). The same rule applies for a thread whose actor is in the *STOPPED* state.

If *actorcap* is K_MYACTOR, the thread must be a thread of the current actor.

In this case, if *threadli* is K_MYSELF, the current thread is used.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadCreate`(2K)

NAME

threadScheduler – get and / or set thread scheduling parameters

SYNOPSIS

```
#include <exec/chExec.h>
#include <sched/chSched.h>
#include <sched/chFifo.h>
#include <sched/chRr.h>
#include <sched/chRt.h>
int threadScheduler(KnCap *actorcap, KnThreadLid threadli, void *oldpara, void
*newparam);
```

FEATURES

SCHED_FIFO, SCHED_CLASS

DESCRIPTION

threadScheduler returns and optionally modifies the scheduling parameters of the thread the local identifier of which is *threadli* in the actor the capability of which is given by *actorcap*. If *actorcap* is K_MYACTOR, the thread must belong to the current actor. In this case, if *threadli* is K_MYSELF, the current thread is affected.

oldparam and *newparam* point to descriptors of old and new scheduling parameters respectively. The type of the descriptor depends on the scheduler. Generally, there is one type of scheduling parameters per scheduling policy (i.e. scheduling class) implemented by the scheduler.

Two priority-based schedulers are currently supported: SCHED_FIFO and SCHED_CLASS. The SCHED_FIFO scheduler implements the CLASS_FIFO scheduling policy, which manages parameters of type *KnFifoThParms.* The SCHED_CLASS scheduler implements the CLASS_FIFO, CLASS_RR and CLASS_RT scheduling policies, which respectively manage parameters of type *KnFifoThParms, KnRrThParms and KnRtThParms.* The SCHED_CLASS also implements the CLASS_TS that is reserved for the implementation of the Chorus / MiX SVR4 subsystem, and therefore should not be used. These policies, as well as the corresponding parameter structures, are described within specific sections below.

If *oldparam* is not NULL, the thread's current scheduling parameters are stored in the area pointed to by *oldparam* in the caller's address space.

If *newparam* is not NULL, it points to an area in the caller's address space containing the new scheduling parameters.

In addition, *threadScheduler,* as well as the two available schedulers, support the notion of *default scheduling class*, which defines a normalized scheduling parameters structure, *KnThreadDefaultSched. newparms* can point to a variable of this type. *KnThreadDefaultSched* is a structure containing the following fields:

```
KnSchedClass            tdClass ;
KnThreadPriority        tdPriority ;
```

where *tdClass* must be equal to K_SCHED_DEFAULT. *tdPriority* establishes the priority of the thread, which may vary between K_PRIOMAX (highest priority) and K_PRIOMIN (lowest priority). The two available schedulers currently map this (virtual) default scheduling class to the CLASS_FIFO policy. This means in particular that they interprete the *KnThreadDefaultSched* priority field in the same way the CLASS_FIFO interpretes the priority field of a *KnFifoThParms* structure.

CLASS_FIFO

The CLASS_FIFO scheduling class implements a pure priority-based, preemptive, fifo policy. Thread's scheduling parameters are defined by the *KnFifoThParms* structure, the members of which are the following:

```
KnSchedClass    fifoClass ;
KnFifoPriority  fifoPriority ;
```

fifoClass must be equal to K_SCHED_FIFO. *fifoPriority* defines the priority of the thread, which may vary between K_FIFO_PRIOMAX (0, highest priority) and K_FIFO_PRIOMIN (255, lowest priority).

CLASS_FIFO uses the full range of priorities (256) in both the SCHED_FIFO and SCHED_CLASS schedulers.

A thread which becomes ready to run after being blocked is always inserted last at the end of its priority ready queue (fifo policy). This means that a running thread is only preempted if a thread of strictly highest priority becomes ready to run. A preempted thread is placed at the head of its priority ready queue. This means that it will be elected first when the preempting thread completes or is blocked.

CLASS_RR

The CLASS_RR scheduling class is only available within the SCHED_CLASS scheduler. It implements a priority-based, preemptive, round-robin policy. Thread's scheduling parameters are defined by the *KnRrThParms* structure, the members of which are the following:

```
KnSchedClass    rrClass ;
KnRrPriority    rrPriority ;
```

rrClass must be equal to K_SCHED_RR. *rrPriority* defines the priority of the thread, which may vary between K_RR_PRIOMAX (0, highest priority) and K_RR_PRIOMIN (255, lowest priority).

CLASS_RR uses the full range of priorities (256) of the SCHED_CLASS scheduler.

The SCHED_RR policy is similar to the SCHED_FIFO policy, except that an elected thread is given a fixed time quantum. If the thread is still running at quantum expiration, it is placed at the end of its priority ready queue, and then may be preempted by threads of equal priority. The thread's quantum is reset when the thread becomes ready after being blocked; it is not reset when the thread is elected again after a preemption.

The time quantum is the same for all priority levels and all threads. It is defined by the K_RR_QUANTUM value (100 milliseconds). It will be turned into a system tunable in a future release.

CLASS_RT

The CLASS_RT scheduling class is only available within the SCHED_CLASS scheduler. It implements the same policy as the real-time class of UNIX SVR4.0. Refer to the `priocntl`(2) manual entry of UNIX SVR4.0 for a complete description.

This policy is basically a round-robin policy, with per thread time quanta.

Thread's scheduling parameters are defined by the *KnRtThParms* structure, the members of which are the following:

```
KnSchedClass    rtClass ;
KnRtParms       *rtParms ;
```

rtClass must be equal to K_SCHED_RT.

The actual parameters are defined by the *KnRtParms* structure, the members of which are the following:

```
KnRtPriority    rtPriority ;
unsigned long   rtQSecs ;
long            rtQNSecs ;
```

The fields of this structure map the fields of the *rtparms_t* of UNIX SVR4. *rtPriority* defines the priority of the thread, which may vary between K_RT_PRIOMAX (159, highest priority) and K_RT_PRIOMIN (100, lowest priority). Note that the order of priorities is inverted compared to the CLASS_FIFO priorities. CLASS_RT uses only a sub-range of the SCHED_CLASS priorities. This sub-range corresponds to the range [96, 155] of CLASS_FIFO and CLASS_RR.

The *rtQSec* and *rtQNsec* define the thread's time quantum, respectively in seconds and nanoseconds.

The *rtQNsec* field can take the following special values:

RT_SAMEQUANTUM	The thread's quantum must not be modified, or if the thread enters the CLASS_RT class, the thread's quantum is set to the RT_INFINITE quantum.
RT_INFINITE	The thread's time quantum is infinite.
RT_DEFAULT	The thread's time quantum is set to the default for the thread' s priority, as follows:

K_RT_PRIOMIN to K_RT_PRIOMIN+9	100 ticks
K_RT_PRIOMIN+10 to K_RT_PRIOMIN+19	80 ticks
K_RT_PRIOMIN+20 to K_RT_PRIOMIN+29	60 ticks
K_RT_PRIOMIN+30 to K_RT_PRIOMIN+39	40 ticks
K_RT_PRIOMIN+40 to K_RT_PRIOMIN+49	20 ticks
K_RT_PRIOMIN+50 to K_RT_PRIOMIN+59	10 ticks

The default tick period is 10 ms. In this case, the values in the table should be multiplied by ten in order to get the quantum in ms.

The CLASS_RT manages per configurable priority default time quanta.

CLASS_TS

The CLASS_TS scheduling class is reserved and implements the same policy as the time-sharing class of UNIX SVR4.0.

RETURN VALUE

Upon successful completion, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EINVAL]	*newparam* points to a structure containing invalid scheduling parameters.
[K_ENOMEM]	The system is out of resources.
[K_ENOTIMP]	*newparm* specifies a non-supported scheduling policy.
[K_EFAULT]	Some of the provided data are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `threadCreate`(2K)

NAME

threadSelf – get the current thread local identifier

SYNOPSIS

```
#include <exec/chExec.h>
KnThreadLid threadSelf(void);
```

DESCRIPTION

threadSelf returns the current thread's local identifier in its actor.

RETURN VALUE

The current thread's local identifier is returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

threadCreate(2K)

NAME

threadSemInit, threadSemPost, threadSemWait – Initialize a thread semaphore; Signal a thread semaphore; Wait on a thread semaphore

SYNOPSIS

```
#include <exec/chExec.h>
int threadSemInit(KnThSem * sem);

int threadSemPost(KnThSem * sem);

int threadSemWait(KnThSem * sem, KnTimeVal * waitLimit);
```

DESCRIPTION

Thread semaphores are *KnThSem* structures allocated in user memory. A thread semaphore is a binary semaphore on which one, and only one, thread can wait, and which can be signalled to by multiple threads and/or interrupt handlers.

A thread semaphore can be in two states: POSTED and UNPOSTED.

The *threadSemInit* system call initializes the thread semaphore whose address is *sem* in the UNPOSTED state. The thread which performs *threadSemInit* on a given thread semaphore is the only thread which can wait on the semaphore (using *threadSemWait*).

The *threadSemWait* system call makes the current thread wait conditionally on the thread semaphore *sem*. If the thread semaphore is in the POSTED state, *threadSemWait* returns immediately and atomically changes the state of the thread semaphore to UNPOSTED. If the thread semaphore is in the UNPOSTED state, the thread is blocked according to the options described by *waitLimit* in *intro(2K)*. The *waitLimit* pointer points to a *KnTimeVal* structure containing a timeout interval as described in *sysTime(2K)*.

The *threadSemPost* system call signals the thread semaphore *sem*. If the thread that initialized the thread semaphore is blocked behind the semaphore, *threadSemPost* awakens that thread without modifying the state of *sem*. If the thread is not blocked, *threadSemPost* sets the state of *sem* to POSTED. If the thread is not blocked and *sem* is already in the POSTED state, *threadSemPost* has no effect.

The *threadSemPost* system call may be called within an interrupt handler, or with preemption disabled.

RETURN VALUE

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

RESTRICTIONS

A user application and a supervisor application may not share a semaphore.

However, two user applications may share a semaphore by mapping it in both user address spaces.

ERRORS

[K_EABORT]	*threadSemWait* has been aborted.
[K_EFAULT]	*sem* points outside the current actor's address space.

[K_EINVAL]	The thread semaphore structure has not been correctly initialized, or the *waitLimit* structure is not a valid *KnTimeVal.*
[K_ETIMEOUT]	The timeout occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

threadSemInit, threadSemPost, threadSemWait – Initialize a thread semaphore; Signal a thread semaphore; Wait on a thread semaphore

SYNOPSIS

```
#include <exec/chExec.h>
int threadSemInit(KnThSem * sem);

int threadSemPost(KnThSem * sem);

int threadSemWait(KnThSem * sem, KnTimeVal * waitLimit);
```

DESCRIPTION

Thread semaphores are *KnThSem* structures allocated in user memory. A thread semaphore is a binary semaphore on which one, and only one, thread can wait, and which can be signalled to by multiple threads and/or interrupt handlers.

A thread semaphore can be in two states: POSTED and UNPOSTED.

The *threadSemInit* system call initializes the thread semaphore whose address is *sem* in the UNPOSTED state. The thread which performs *threadSemInit* on a given thread semaphore is the only thread which can wait on the semaphore (using *threadSemWait*).

The *threadSemWait* system call makes the current thread wait conditionally on the thread semaphore *sem*. If the thread semaphore is in the POSTED state, *threadSemWait* returns immediately and atomically changes the state of the thread semaphore to UNPOSTED. If the thread semaphore is in the UNPOSTED state, the thread is blocked according to the options described by *waitLimit* in *intro(2K)*. The *waitLimit* pointer points to a *KnTimeVal* structure containing a timeout interval as described in *sysTime(2K)*.

The *threadSemPost* system call signals the thread semaphore *sem*. If the thread that initialized the thread semaphore is blocked behind the semaphore, *threadSemPost* awakens that thread without modifying the state of *sem*. If the thread is not blocked, *threadSemPost* sets the state of *sem* to POSTED. If the thread is not blocked and *sem* is already in the POSTED state, *threadSemPost* has no effect.

The *threadSemPost* system call may be called within an interrupt handler, or with preemption disabled.

RETURN VALUE

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

RESTRICTIONS

A user application and a supervisor application may not share a semaphore.

However, two user applications may share a semaphore by mapping it in both user address spaces.

ERRORS

[K_EABORT]	*threadSemWait* has been aborted.
[K_EFAULT]	*sem* points outside the current actor's address space.

[K_EINVAL]	The thread semaphore structure has not been correctly initialized, or the *waitLimit* structure is not a valid *KnTimeVal.*
[K_ETIMEOUT]	The timeout occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

threadSemInit, threadSemPost, threadSemWait – Initialize a thread semaphore; Signal a thread semaphore; Wait on a thread semaphore

SYNOPSIS

```
#include <exec/chExec.h>
int threadSemInit(KnThSem * sem);

int threadSemPost(KnThSem * sem);

int threadSemWait(KnThSem * sem, KnTimeVal * waitLimit);
```

DESCRIPTION

Thread semaphores are *KnThSem* structures allocated in user memory. A thread semaphore is a binary semaphore on which one, and only one, thread can wait, and which can be signalled to by multiple threads and/or interrupt handlers.

A thread semaphore can be in two states: POSTED and UNPOSTED.

The *threadSemInit* system call initializes the thread semaphore whose address is *sem* in the UNPOSTED state. The thread which performs *threadSemInit* on a given thread semaphore is the only thread which can wait on the semaphore (using *threadSemWait*).

The *threadSemWait* system call makes the current thread wait conditionally on the thread semaphore *sem*. If the thread semaphore is in the POSTED state, *threadSemWait* returns immediately and atomically changes the state of the thread semaphore to UNPOSTED. If the thread semaphore is in the UNPOSTED state, the thread is blocked according to the options described by *waitLimit* in *intro(2K)*. The *waitLimit* pointer points to a *KnTimeVal* structure containing a timeout interval as described in *sysTime(2K)*.

The *threadSemPost* system call signals the thread semaphore *sem*. If the thread that initialized the thread semaphore is blocked behind the semaphore, *threadSemPost* awakens that thread without modifying the state of *sem*. If the thread is not blocked, *threadSemPost* sets the state of *sem* to POSTED. If the thread is not blocked and *sem* is already in the POSTED state, *threadSemPost* has no effect.

The *threadSemPost* system call may be called within an interrupt handler, or with preemption disabled.

RETURN VALUE

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

RESTRICTIONS

A user application and a supervisor application may not share a semaphore.

However, two user applications may share a semaphore by mapping it in both user address spaces.

ERRORS

[K_EABORT]	*threadSemWait* has been aborted.
[K_EFAULT]	*sem* points outside the current actor's address space.

[K_EINVAL]	The thread semaphore structure has not been correctly initialized, or the *waitLimit* structure is not a valid *KnTimeVal*.
[K_ETIMEOUT]	The timeout occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

threadStart, threadStop – Stop a thread; Start a thread

SYNOPSIS

```
#include <exec/chExec.h>
int threadStop(KnCap * actorcap, int threadli);

int threadStart(KnCap * actorcap, int threadli);
```

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *threadStop* system call stops the thread *threadli* , in the actor whose capability is given by *actorcap.*

The thread must either be owned by the actor or have entered the actor through a safe LAP invocation (see *svLapCreate* (2K)).

The effect of *threadStop* is to prevent the thread from running until the thread is restarted using *threadStart* (see *svLapCreate* (2K)) and becomes runnable (not suspended, and scheduled).

The effect of *threadStop* is not instantaneous on threads executing system calls implemented via trap or cross-actor invocation, but it is guaranteed that a thread performing such a system call will not return from that call.

The *threadStart* system call restarts the thread *threadli* , in the actor whose capability is given by *actorcap.*

The thread must either be owned by the actor or have entered the actor through a safe LAP invocation (see *svLapCreate* (2K)).

It may only be applied to a thread previously stopped by *threadStop* , or by *actorStop* .

The *threadStop* and *threadStart* system calls are reserved for the usage of system or application debuggers, and should not be used inside applications.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svLapCreate`(2K) , `actorStop`(2K)

RESTRICTIONS The target actor and the current actor must be located on the same site.

NAME

threadStat – obtain the descriptions of the threads running in an actor

SYNOPSIS

#include <exec/chExec.h>
int **threadStat**(KnCap **actorcap*, unsigned int *options*, KnThreadStat **stat*, unsigned int *buffsize*);

FEATURES

CORE

DESCRIPTION

The *threadStat* system call obtains the descriptions of threads running in the actor whose capability is given by *actorcap*. If *actorcap* is K_MYACTOR, threads of the current actor are used.

Only threads belonging to the actor or having entered it through a safe LAP invocation are returned. The system does not record information about threads entering actors through non-safe LAPs.

The *options* argument is reserved for future use and must be set to 0.

The *stat* argument points to a buffer in which the thread descriptions will be returned. The buffer is considered part of the caller (current user or supervisor) address space.

The *buffsize* argument corresponds to the size of the stat buffer. It should be a multiple of the size of the `KnThreadStat` struct.

On successful return the buffer will contain an array of descriptors, each one describing a single thread. For one thread, it is possible to retrieve several entries, each one describing a particular lap frame of the same thread executing in the actor (see *svLapCreate*(2K)).

Each descriptor is a *KnThreadStat* structure with the following fields:

```
KnThreadLid     tsLid ;
KnThreadStatus  tsStatus ;
int             tsFrame ;
VmAddr          tsSvStack ;
VmSize          tsSvStackSize ;
```

The *tsLid* field is the thread local identifier.

The *tsStatus* status is the thread status (K_INACTIVE or K_ACTIVE, see *threadCreate*(2K) for details).

tsFrame is the lap frame level that appears in this actor for the considered thread (see *svLapCreate*(2K)). Level 0 means that the thread is owned by the actor

The *tsSvStack* field is the address of the space in which the supervisor stack has been allocated. It is usually the lowest boundary of the stack area.

tsSvStackSize is the number of bytes allocated by the kernel for the thread's supervisor stack.

RETURN VALUE

If successful, *threadStat* returns the number of threads running in the actor, otherwise a negative error code is returned.

As mentionned above, if several lap frames appear in the considered actor for one particular thread, all the lap frames are taken into account in the returned value.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *options* specifies invalid options.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	the *stat* argument is outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadCreate`(2K), `svLapInvoke`(2K)

NAME

threadStart, threadStop – Stop a thread; Start a thread

SYNOPSIS

```
#include <exec/chExec.h>
int threadStop(KnCap * actorcap, int threadli);

int threadStart(KnCap * actorcap, int threadli);
```

FEATURES

CORE

DESCRIPTION

Caution - This system call is strictly reserved for internal use only. It MUST NOT be used by any application.

The *threadStop* system call stops the thread *threadli* , in the actor whose capability is given by *actorcap.*

The thread must either be owned by the actor or have entered the actor through a safe LAP invocation (see *svLapCreate* (2K)).

The effect of *threadStop* is to prevent the thread from running until the thread is restarted using *threadStart* (see *svLapCreate* (2K)) and becomes runnable (not suspended, and scheduled).

The effect of *threadStop* is not instantaneous on threads executing system calls implemented via trap or cross-actor invocation, but it is guaranteed that a thread performing such a system call will not return from that call.

The *threadStart* system call restarts the thread *threadli* , in the actor whose capability is given by *actorcap.*

The thread must either be owned by the actor or have entered the actor through a safe LAP invocation (see *svLapCreate* (2K)).

It may only be applied to a thread previously stopped by *threadStop* , or by *actorStop* .

The *threadStop* and *threadStart* system calls are reserved for the usage of system or application debuggers, and should not be used inside applications.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svLapCreate`(2K) , `actorStop`(2K)

RESTRICTIONS The target actor and the current actor must be located on the same site.

NAME

threadLoadR, threadStoreR – Get the current thread's valid soft register value; Reset the current thread's valid soft register value

SYNOPSIS

#include <exec/chSoftCtx.h>
long **threadLoadR**(void);

long **threadStoreR**(long *reg*);

DESCRIPTION

The execution context of a thread (its various machine registers) is extended by two software registers, one for each of the possible privilege levels of a thread (*USER* or *SUPERVISOR*). Like hardware registers, these two registers are saved and restored at each thread switch. These registers provide the means for maintaining per-thread data efficiently.

The *threadLoadR* system call returns the value of the current thread's software register corresponding to the current privilege level of the thread.

The *threadStoreR* system call sets the current thread's software register corresponding to the current privilege level of the thread to the value specified by *reg* , and returns its new value.

The values of these registers may also be affected using *threadContext* (2K).

RETURN VALUE

The *threadLoadR* call returns the current register value. The *threadStoreR* call returns the new register value.

EXAMPLES

The software register facility can be used to store the thread's local identifier in its software register. However, this only impacts on performance, as *threadLoadR* does not imply a kernel call on most architectures, this is fully resolved in libraries which access a globally shared system context. However, *threadSelf* always implies a kernel call. In order to do this, call:

```
threadStoreR (threadSelf());
```

at the beginning of the thread's code; then invoke:

```
threadLoadR ;
```

instead of:

```
threadSelfC ;
```

A more common way to use this facility is to associate a per-thread structure with each thread of an actor, and to store the address of this structure in the software register. This will avoid systematically transferring this address as a parameter across the layers of the actor's code (the use of a global variable is not possible, especially on multiprocessor implementations). In this case, each actor's thread will perform a

```
threadStoreR (myArea);
```

at the beginning of the thread's code, where myArea is the address of the per thread's structure. Note that there are several ways for the thread to acquire the

```
(myArea);
```

value: it may be provided to the thread's routine as an argument (by initializing its stack correctly before creation); it may also be computed by the created thread as the result of a hash function (the hash function taking the thread's local identifier as input, and returning an entry in the array of per-thread records). An alternative to this initialization process is for the creator to call *threadContext* (2K) before resuming the thread. In each case, the per-thread record is subsequently accessed by:

```
threadLoadR ();
```

Finally, note that the software register may be used differently by different code layers: the sequence S in:

```
oldVal = threadLoadR ();
threadStoreR (newVal);
        /* S */
threadStoreR (oldVal);
```

may make private use of the software register value.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadContext`(2K)

NAME

threadSuspend, threadResume – Suspend a thread; Resume a thread

SYNOPSIS

```
#include <exec/chExec.h>
int threadSuspend(KnCap * actorcap, int threadli);

int threadResume(KnCap * actorcap, int threadli);
```

FEATURES

CORE

DESCRIPTION

NOTE: These calls are obsolete. Although they are maintained for compatibility reasons, they should be avoided wherever possible. See *threadActivate(2K)*, *threadStart(2K)*, and *threadSemInit(2K)* for documentation on other calls that can be used instead of *threadSuspend(2K)* and threadResume(2K). The *threadSuspend* system call puts the thread whose local identifier is *threadli* , in the actor whose capability is given by *actorcap* , into the *SUSPENDED* state.

A suspended thread may resume execution (enter the *ACTIVE* state) only if the thread is explicitly resumed using *threadResume(2K)*.

The *threadSuspend* and *threadResume* system calls are based on a per-thread suspend counter, initialized to 0 at thread creation, incremented by *threadSuspend* and decremented by *threadResume*.

The *threadResume* system call decrements the suspend counter of the thread whose local identifier is *threadli* , in the actor whose capability is given by *actorcap*. It only awakens the thread by setting its state to *ACTIVE* , if the suspend counter has become 0.

If a thread which is in a *BLOCKED* state is suspended using *threadSuspend*, the thread will suspend just after it exits its *BLOCKED* state. If it is resumed using *threadResume* while still in the *BLOCKED* state, it will continue its execution normally after it exits its *BLOCKED* state (if its suspend counter has become 0). The same rule applies for a thread whose actor is in the *STOPPED* state.

If *actorcap* is K_MYACTOR, the thread must be a thread of the current actor.

In this case, if *threadli* is K_MYSELF, the current thread is used.

RETURN VALUE

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

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`threadCreate`(2K)

NAME

threadTimes – get thread execution time

SYNOPSIS

#include <vtimer/chVtimer.h>
int **threadTimes**(KnCap **actorcap*, int *threadli*, KnTimeVal **internal*, KnTimeVal **external*);

FEATURES

VTIMER

DESCRIPTION

The *threadTimes* system call returns the execution time of the thread whose local identifier is *threadli*, within the actor specified by *actorcap*.

If *threadLi* is equal to K_ALLACTORTHREADS, *threadTimes* returns the total execution time of all the threads within the actor specified by *actorcap*.

The *internal* time is the time spent executing in the actor in which the thread(s) was (were) originally created. The *external* time includes all execution time spent with the thread's *execution actor* set to another actor; in other words, while the thread(s) is (are) executing in a cross-actor invocation.

The time spent in a kernel system call is added to the current execution time (internal or external) of the thread (a kernel system call isn't considered as a cross-actor invocation).

RESTRICTIONS

The target actor and the current actor must be located on the same site.

RETURN VALUES

Upon successful completion, *threadTimes* returns K_OK. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*actorcap* is an inconsistent actor capability, or *threadli* is not a valid thread identifier in the specified actor.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

timerCreate – create a timer

SYNOPSIS

#include <etimer/chEtimer.h>
int **timerCreate**(KnCap **actor*, int *clock*, KnThreadPool **threadPool*, void **cookie*, int **timerLi*);

FEATURES

TIMER

DESCRIPTION

timerCreate creates a timer relative to he clock type *clock* in the actor specified by *actor*. If *actor* has the value K_MYACTOR, the timer is created in the current actor. *timerCreate* returns the local id of the new timer in *timerLi*. The timer may then be armed using *timerSet*(2K).

The only clock type currently supported is K_CLOCK_REALTIME. This clock corresponds to the system time as returned by *sysTime*(2K).

threadPool must point to an initialized *KnThreadPool* object (see *timerThreadPoolInit*(2K)). At each timer expiration for the newly created timer, a thread blocked on the designated *KnThreadPool* object will be awakened so as to return from its *timerThreadPoolWait*(2K) system call.

The *cookie* value will be provided to that thread on return from *timerThreadPoolWait*(2K). Library or application code may create and use these type of threads to execute timer notification handlers. The *cookie* value can be used to address a library object corresponding to the particular timer.

RETURN VALUE

Upon successful completion, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*clock* is not equal to K_CLOCK_REALTIME, or *threadPool* has not been initialized.
[K_ENOMEM]	The maximum number of timers has been reached.
[K_EFAULT]	An address argument falls outside the caller's address space.
[K_EUNKNOWN]	*actor* does not specify a reachable actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timerDelete`(2K), `timerSet`(2K), `timerThreadPoolInit`(2K), `timerThreadPoolWait`(2K)

NAME

timerDelete – delete a timer

SYNOPSIS

```
#include <etimer/chEtimer.h>
int timerDelete(KnCap *actor, int timerLi);
```

FEATURES

TIMER

DESCRIPTION

The *timerDelete* system call deletes, and, if necessary, disarms the timer *timerLi* in the actor *actor*. If *actor* is equal to K_MYACTOR, the current actor is used

RETURN VALUE

Upon successful completion, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*timerLi* is not a valid timer identifier within the current host actor.
[K_EFAULT]	An address argument falls outside the caller's address space.
[K_EUNKNOWN]	*actor* does not specify a reachable actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

timerCreate(2K), timerSet(2K), timerThreadPoolInit(2K), timerThreadPoolWait(2K)

NAME

timerGetRes – get the timer resolution

SYNOPSIS

#include <etimer/chEtimer.h>
int **timerGetRes**(int *clock*, KnTimeVal **resolution*);

FEATURES

TIMER

DESCRIPTION

The *timerGetRes* system call obtains the resolution of the clock type *clock*. The value stored in *resolution* represents the smallest possible difference between two distinct time specifications related to *clock*.

The only clock type currently supported is K_CLOCK_REALTIME. This clock corresponds to the system time as returned by *sysTime*(2K).

RETURN VALUE

Upon successful completion, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*clock* is not equal to K_CLOCK_REALTIME.
[K_EFAULT]	An address argument falls outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timerCreate`(2K), `timerSet`(2K)

NAME

timerSet – start, cancel or query a timer

SYNOPSIS

#include <etimer/chEtimer.h>
int **timerSet**(KnCap **actor*, int *timerLi*, int *flags*, KnITimer **newit*, KnITimer **oldit*);

FEATURES

TIMER

DESCRIPTION

timerSet starts, cancels, or queries the state of timer *timerli* in actor *actor*. If *actor* has the value K_MYACTOR, the current actor is assumed.

The *KnITimer* structure is defined as follows:

```
KnTimeVal ITmReload ; /* Reload value (periodic timer) */
KnTimeVal ITmValue ; /* Initial value                 */
```

If *newit* is non-NULL, the timer is armed accordingly. *ITmValue* contains the initial timeout value; *ITmReload* contains the subsequent interval for a periodic timer, or zero for a one-shot timer. If the K_TIMER_ABSOLUTE bit is set in *flags*, *ITmValue* is interpreted as an absolute time value (in terms of the values returned by *sysTime*). Otherwise, *ITmValue* is interpreted as a relative time interval.

If the timer was already set, the previous setting is cancelled and replaced with the new specification. If *ITmValue* contains a zero value, the timer is disarmed.

If *newit* is NULL, the current timer setting is unchanged.

If *oldit* is non-NULL, the current setting of the timer (or zero if not set) is stored at the referenced location. A relative interval is always stored even if the timer was armed by absolute time.

RETURN VALUE

Upon successful completion, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*timerLi* is not a valid timer identifier in the actor *actor*, or *ITmReload* or *ITmValue* are not valid KnTimeVal values.
[K_EFAULT]	An address argument falls outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timerCreate`(2K), `timerDelete`(2K), `timerThreadPoolInit`(2K), `timerThreadPoolWait`(2K)

NAME

timerThreadPoolInit – initialize a timer thread pool

SYNOPSIS

#include <etimer/chEtimer.h>
int **timerThreadPoolInit**(KnThreadPool **threadPool*);

FEATURES

TIMER

DESCRIPTION

timerThreadPoolInit initializes the timer thread pool *threadPool*.

The *threadPool* object is opaque; it must be pre-allocated by the caller but its fields are inaccessible to the caller. A *threadPool* object must be initialized before it can be accessed by a *timerCreate*.

RETURN VALUE

Upon successful completion, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	An address argument falls outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timerCreate`(2K), `timerThreadPoolWait`(2K)

NAME

timerThreadPoolWait – wait for a timer expiration event

SYNOPSIS

```
#include <etimer/chEtimer.h>
int timerThreadPoolWait(KnThreadPool *threadPool, void **cookie, int *overrun,
KnTimeVal *waitLimit);
```

FEATURES

TIMER

DESCRIPTION

timerThreadPoolWait waits for a subsequent timer expiration in a timer that was created using the thread pool *threadPool* (see *timerCreate*(2K)). When an expiration event occurs, and the current thread is selected among those waiting in the thread pool, it will return from *timerThreadPoolWait*. On return, *cookie* will contain the cookie value provided at the creation of the corresponding timer, and *overrun* will contain the overrun count corresponding to the current expiration.

Until this type of timer event occurs, the current thread is blocked according to the *waitLimit* specification (see `intro`(2K)).

threadPool must have been previously initialized using *timerThreadPoolInit*(2K).

Only one thread at a time may be active on behalf of a single timer. Thus, during the interval between a thread's return from *timerThreadPoolWait* and its next invocation of *timerThreadPoolWait*, no other thread will be awakened for an expiration of the same timer; the expiration will be counted as a timer overrun.

A timer expiration which occurs when there are no threads currently waiting on the associated thread pool is also counted as an overrun. A positive overrun count indicates that at least one timeout notification for the timer was delayed. An overrun count greater than one indicates that at least one timeout notification was lost, that is, no waiting threads could be awakened by the expiration event.

Each time a thread returns from *timerThreadPoolWait*, the current overrun count for the corresponding timer is stored in *overrun*, and the timer's internal overrun count is reset to zero. Thus the overrun count delivered with each event notification represents only the number of delayed or lost expirations since the last notification.

RETURN VALUE

Upon successful completion, K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EINVAL]	*threadPool* is not an initialized thread pool object.
[K_EFAULT]	An address argument falls outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `timerCreate`(2K), `timerThreadPoolInit`(2K)

NAME

uiBuild, uiClear, uiEqual, uiGetSite, uiIsLocal, uiSite, uiValid – Build a user-defined unique identifier; Clear a unique identifer; Compare two unique identifers; Extract the site number from a unique identifier; Check a unique identifier's locality; Get a pre-defined site unique identifier; Check whether an unique identifier has been cleared

SYNOPSIS

```
#include <ipc/chId.h>
int uiBuild(KnUniqueId * ui, unsigned int type, unsigned int site, unsigned int hea,
unsigned int tail);

int uiClear(KnUniqueId * Ui);

int uiEqual(KnUniqueId * ui1, KnUniqueId * ui2);

int uiGetSite(KnUniqueId * ui);

int uiIsLocal(KnUniqueId * ui);

int uiSite(KnUniqueId * ui, unsigned long site);

int uiValid(KnUniqueId * ui);
```

FEATURES

IPC

DESCRIPTION

The *uiBuild* system call builds a unique identifier from user-provided values. The built UI value is returned in the *KnUniqueId* structure pointed to by *ui.*

The `type` field defines the type of the UI as follows:

K_UIPORT	The UI will be usable as a port UI for the IPC module. The UI can then be usedin a call to *portDeclare(2K)).*
K_UIGROUP	The UI will be usable as a group UI for the IPC module. The UI can then be used in a call to *grpPortInsert(2K)).*
K_UISITE	The UI will be a site UI.
K_UIMSGQUEUE	The UI will be usable as a message queue UI for the MIPC_G module. The UI can then be used in a call to *msgQueueBind(2K)).*

The *site* field is a site number. If a non-zero value is given, this field will be interpreted by the kernel as a location hint for the UI.

The `head` and `tail` fields represent the stamp of the UI. `tail` is a 32 bits value. `head` is is a 13 bits value.

The *uiIsLocal* function checks whether the UI given by *ui* is recognized by the local site, and returns 1 if the UI is recognized, 0 otherwise.

On every site, the kernel declares a pre-defined site UI by default. The *uiSite* function returns the value of this pre-defined UI for the site whose site number is *site* (see *uiLocalSite* (2K)). Note that *uiSite (ui, site)* is equivalent to *uiBuild (ui, K_UISITE, site, 0, 0).*

The *uiClear* function initializes *ui* with a standard null value.

The *uiEqual* function returns 1 if the two UI's passed as arguments are equal, 0 otherwise.

The *uiGetSite* function returns the site information contained within *ui.*

The *uiValid* function returns 0 if *ui* is equal to the null UI, 1 otherwise.

RETURN VALUE

The *uiIsLocal* function returns 1 if the UI checked is local, 0 otherwise. The *uiValid* function returns 1 if the checked UI is different from the null value, 0 otherwise. The *uiEqual* function returns 1 if the UI's are identical, 0 otherwise. Other calls return 0 upon successful completion, a negative error code otherwise.

The *uiClear, uiValid, uiEqual* and *uiGetSite* never return K_EFAULT. If they are invoked with an invalid UI pointer, an exception occurs within the caller's context.

ERRORS

[K_EFAULT] Some of the provided data are outside the current actor's address space (*uiBuild* and *uiLocalSite*).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portDeclare`(2K) , `msgQueueBind`(2K) , `uiLocalSite`(2K)

NAME

uiBuild, uiClear, uiEqual, uiGetSite, uiIsLocal, uiSite, uiValid – Build a user-defined unique identifier; Clear a unique identifer; Compare two unique identifers; Extract the site number from a unique identifier; Check a unique identifier's locality; Get a pre-defined site unique identifier; Check whether an unique identifier has been cleared

SYNOPSIS

#include <ipc/chId.h>

int **uiBuild**(KnUniqueId * *ui*, unsigned int *type*, unsigned int *site*, unsigned int *hea*, unsigned int *tail*);

int **uiClear**(KnUniqueId * *Ui*);

int **uiEqual**(KnUniqueId * *ui1*, KnUniqueId * *ui2*);

int **uiGetSite**(KnUniqueId * *ui*);

int **uiIsLocal**(KnUniqueId * *ui*);

int **uiSite**(KnUniqueId * *ui*, unsigned long *site*);

int **uiValid**(KnUniqueId * *ui*);

FEATURES

IPC

DESCRIPTION

The *uiBuild* system call builds a unique identifier from user-provided values. The built UI value is returned in the *KnUniqueId* structure pointed to by *ui*.

The `type` field defines the type of the UI as follows:

K_UIPORT	The UI will be usable as a port UI for the IPC module. The UI can then be usedin a call to *portDeclare(2K))*.
K_UIGROUP	The UI will be usable as a group UI for the IPC module. The UI can then be used in a call to *grpPortInsert(2K))*.
K_UISITE	The UI will be a site UI.
K_UIMSGQUEUE	The UI will be usable as a message queue UI for the MIPC_G module. The UI can then be used in a call to *msgQueueBind(2K))*.

The *site* field is a site number. If a non-zero value is given, this field will be interpreted by the kernel as a location hint for the UI.

The `head` and `tail` fields represent the stamp of the UI. `tail` is a 32 bits value. `head` is is a 13 bits value.

The *uiIsLocal* function checks whether the UI given by *ui* is recognized by the local site, and returns 1 if the UI is recognized, 0 otherwise.

On every site, the kernel declares a pre-defined site UI by default. The *uiSite* function returns the value of this pre-defined UI for the site whose site number is *site* (see *uiLocalSite* (2K)). Note that *uiSite (ui, site)* is equivalent to *uiBuild (ui, K_UISITE, site, 0, 0).*

The *uiClear* function initializes *ui* with a standard null value.

The *uiEqual* function returns 1 if the two UI's passed as arguments are equal, 0 otherwise.

The *uiGetSite* function returns the site information contained within *ui.*

The *uiValid* function returns 0 if *ui* is equal to the null UI, 1 otherwise.

RETURN VALUE

The *uiIsLocal* function returns 1 if the UI checked is local, 0 otherwise. The *uiValid* function returns 1 if the checked UI is different from the null value, 0 otherwise. The *uiEqual* function returns 1 if the UI's are identical, 0 otherwise. Other calls return 0 upon successful completion, a negative error code otherwise.

The *uiClear, uiValid, uiEqual* and *uiGetSite* never return K_EFAULT. If they are invoked with an invalid UI pointer, an exception occurs within the caller's context.

ERRORS

[K_EFAULT] Some of the provided data are outside the current actor's address space (*uiBuild* and *uiLocalSite*).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portDeclare`(2K) , `msgQueueBind`(2K) , `uiLocalSite`(2K)

NAME

uiBuild, uiClear, uiEqual, uiGetSite, uiIsLocal, uiSite, uiValid – Build a user-defined unique identifier; Clear a unique identifer; Compare two unique identifers; Extract the site number from a unique identifier; Check a unique identifier's locality; Get a pre-defined site unique identifier; Check whether an unique identifier has been cleared

SYNOPSIS

```
#include <ipc/chId.h>
int uiBuild(KnUniqueId * ui, unsigned int type, unsigned int site, unsigned int hea,
unsigned int tail);

int uiClear(KnUniqueId * Ui);

int uiEqual(KnUniqueId * ui1, KnUniqueId * ui2);

int uiGetSite(KnUniqueId * ui);

int uiIsLocal(KnUniqueId * ui);

int uiSite(KnUniqueId * ui, unsigned long site);

int uiValid(KnUniqueId * ui);
```

FEATURES

IPC

DESCRIPTION

The *uiBuild* system call builds a unique identifier from user-provided values. The built UI value is returned in the *KnUniqueId* structure pointed to by *ui.*

The `type` field defines the type of the UI as follows:

K_UIPORT	The UI will be usable as a port UI for the IPC module. The UI can then be usedin a call to *portDeclare(2K)).*
K_UIGROUP	The UI will be usable as a group UI for the IPC module. The UI can then be used in a call to *grpPortInsert(2K)).*
K_UISITE	The UI will be a site UI.
K_UIMSGQUEUE	The UI will be usable as a message queue UI for the MIPC_G module. The UI can then be used in a call to *msgQueueBind(2K)).*

The *site* field is a site number. If a non-zero value is given, this field will be interpreted by the kernel as a location hint for the UI.

The `head` and `tail` fields represent the stamp of the UI. `tail` is a 32 bits value. `head` is is a 13 bits value.

The *uiIsLocal* function checks whether the UI given by *ui* is recognized by the local site, and returns 1 if the UI is recognized, 0 otherwise.

On every site, the kernel declares a pre-defined site UI by default. The *uiSite* function returns the value of this pre-defined UI for the site whose site number is *site* (see *uiLocalSite* (2K)). Note that *uiSite (ui, site)* is equivalent to *uiBuild (ui, K_UISITE, site, 0, 0).*

The *uiClear* function initializes *ui* with a standard null value.

The *uiEqual* function returns 1 if the two UI's passed as arguments are equal, 0 otherwise.

The *uiGetSite* function returns the site information contained within *ui.*

The *uiValid* function returns 0 if *ui* is equal to the null UI, 1 otherwise.

RETURN VALUE

The *uiIsLocal* function returns 1 if the UI checked is local, 0 otherwise. The *uiValid* function returns 1 if the checked UI is different from the null value, 0 otherwise. The *uiEqual* function returns 1 if the UI's are identical, 0 otherwise. Other calls return 0 upon successful completion, a negative error code otherwise.

The *uiClear, uiValid, uiEqual* and *uiGetSite* never return K_EFAULT. If they are invoked with an invalid UI pointer, an exception occurs within the caller's context.

ERRORS

[K_EFAULT] Some of the provided data are outside the current actor's address space (*uiBuild* and *uiLocalSite*).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portDeclare`(2K) , `msgQueueBind`(2K) , `uiLocalSite`(2K)

NAME

uiBuild, uiClear, uiEqual, uiGetSite, uiIsLocal, uiSite, uiValid – Build a user-defined unique identifier; Clear a unique identifer; Compare two unique identifers; Extract the site number from a unique identifier; Check a unique identifier's locality; Get a pre-defined site unique identifier; Check whether an unique identifier has been cleared

SYNOPSIS

#include <ipc/chId.h>

int **uiBuild**(KnUniqueId * *ui*, unsigned int *type*, unsigned int *site*, unsigned int *hea*, unsigned int *tail*);

int **uiClear**(KnUniqueId * *Ui*);

int **uiEqual**(KnUniqueId * *ui1*, KnUniqueId * *ui2*);

int **uiGetSite**(KnUniqueId * *ui*);

int **uiIsLocal**(KnUniqueId * *ui*);

int **uiSite**(KnUniqueId * *ui*, unsigned long *site*);

int **uiValid**(KnUniqueId * *ui*);

FEATURES

IPC

DESCRIPTION

The *uiBuild* system call builds a unique identifier from user-provided values. The built UI value is returned in the *KnUniqueId* structure pointed to by *ui*.

The `type` field defines the type of the UI as follows:

K_UIPORT	The UI will be usable as a port UI for the IPC module. The UI can then be usedin a call to *portDeclare(2K))*.
K_UIGROUP	The UI will be usable as a group UI for the IPC module. The UI can then be used in a call to *grpPortInsert(2K))*.
K_UISITE	The UI will be a site UI.
K_UIMSGQUEUE	The UI will be usable as a message queue UI for the MIPC_G module. The UI can then be used in a call to *msgQueueBind(2K))*.

The *site* field is a site number. If a non-zero value is given, this field will be interpreted by the kernel as a location hint for the UI.

The `head` and `tail` fields represent the stamp of the UI. `tail` is a 32 bits value. `head` is is a 13 bits value.

The *uiIsLocal* function checks whether the UI given by *ui* is recognized by the local site, and returns 1 if the UI is recognized, 0 otherwise.

On every site, the kernel declares a pre-defined site UI by default. The *uiSite* function returns the value of this pre-defined UI for the site whose site number is *site* (see *uiLocalSite* (2K)). Note that *uiSite (ui, site)* is equivalent to *uiBuild (ui, K_UISITE, site, 0, 0).*

The *uiClear* function initializes *ui* with a standard null value.

The *uiEqual* function returns 1 if the two UI's passed as arguments are equal, 0 otherwise.

The *uiGetSite* function returns the site information contained within *ui.*

The *uiValid* function returns 0 if *ui* is equal to the null UI, 1 otherwise.

RETURN VALUE

The *uiIsLocal* function returns 1 if the UI checked is local, 0 otherwise. The *uiValid* function returns 1 if the checked UI is different from the null value, 0 otherwise. The *uiEqual* function returns 1 if the UI's are identical, 0 otherwise. Other calls return 0 upon successful completion, a negative error code otherwise.

The *uiClear, uiValid, uiEqual* and *uiGetSite* never return K_EFAULT. If they are invoked with an invalid UI pointer, an exception occurs within the caller's context.

ERRORS

[K_EFAULT] Some of the provided data are outside the current actor's address space (*uiBuild* and *uiLocalSite*).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portDeclare`(2K) , `msgQueueBind`(2K) , `uiLocalSite`(2K)

NAME

uiBuild, uiClear, uiEqual, uiGetSite, uiIsLocal, uiSite, uiValid – Build a user-defined unique identifier; Clear a unique identifer; Compare two unique identifers; Extract the site number from a unique identifier; Check a unique identifier's locality; Get a pre-defined site unique identifier; Check whether an unique identifier has been cleared

SYNOPSIS

```
#include <ipc/chId.h>
int uiBuild(KnUniqueId * ui, unsigned int type, unsigned int site, unsigned int hea,
unsigned int tail);

int uiClear(KnUniqueId * Ui);

int uiEqual(KnUniqueId * ui1, KnUniqueId * ui2);

int uiGetSite(KnUniqueId * ui);

int uiIsLocal(KnUniqueId * ui);

int uiSite(KnUniqueId * ui, unsigned long site);

int uiValid(KnUniqueId * ui);
```

FEATURES

IPC

DESCRIPTION

The *uiBuild* system call builds a unique identifier from user-provided values. The built UI value is returned in the *KnUniqueId* structure pointed to by *ui.*

The `type` field defines the type of the UI as follows:

K_UIPORT	The UI will be usable as a port UI for the IPC module. The UI can then be usedin a call to *portDeclare(2K)).*
K_UIGROUP	The UI will be usable as a group UI for the IPC module. The UI can then be used in a call to *grpPortInsert(2K)).*
K_UISITE	The UI will be a site UI.
K_UIMSGQUEUE	The UI will be usable as a message queue UI for the MIPC_G module. The UI can then be used in a call to *msgQueueBind(2K)).*

The *site* field is a site number. If a non-zero value is given, this field will be interpreted by the kernel as a location hint for the UI.

The `head` and `tail` fields represent the stamp of the UI. `tail` is a 32 bits value. `head` is is a 13 bits value.

The *uiIsLocal* function checks whether the UI given by *ui* is recognized by the local site, and returns 1 if the UI is recognized, 0 otherwise.

On every site, the kernel declares a pre-defined site UI by default. The *uiSite* function returns the value of this pre-defined UI for the site whose site number is *site* (see *uiLocalSite* (2K)). Note that *uiSite (ui, site)* is equivalent to *uiBuild (ui, K_UISITE, site, 0, 0).*

The *uiClear* function initializes *ui* with a standard null value.

The *uiEqual* function returns 1 if the two UI's passed as arguments are equal, 0 otherwise.

The *uiGetSite* function returns the site information contained within *ui.*

The *uiValid* function returns 0 if *ui* is equal to the null UI, 1 otherwise.

RETURN VALUE

The *uiIsLocal* function returns 1 if the UI checked is local, 0 otherwise. The *uiValid* function returns 1 if the checked UI is different from the null value, 0 otherwise. The *uiEqual* function returns 1 if the UI's are identical, 0 otherwise. Other calls return 0 upon successful completion, a negative error code otherwise.

The *uiClear, uiValid, uiEqual* and *uiGetSite* never return K_EFAULT. If they are invoked with an invalid UI pointer, an exception occurs within the caller's context.

ERRORS

[K_EFAULT] Some of the provided data are outside the current actor's address space (*uiBuild* and *uiLocalSite*).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portDeclare`(2K) , `msgQueueBind`(2K) , `uiLocalSite`(2K)

NAME

uiLocalSite – get the local site number

SYNOPSIS

```
#include <ipc/chId.h>
int uiLocalSite(int *site);
```

DESCRIPTION

The *uiLocalSite* system call returns the local site number to the value pointed to by *site*.

NOTES

A site number is a 32 bit unsigned integer, but *uiLocalSite* returns the value of the local site number as a signed integer.

RETURN VALUES

Upon successful completion K_OK is returned. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the data provided are outside the current actor's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

uiBuild, uiClear, uiEqual, uiGetSite, uiIsLocal, uiSite, uiValid – Build a user-defined unique identifier; Clear a unique identifer; Compare two unique identifers; Extract the site number from a unique identifier; Check a unique identifier's locality; Get a pre-defined site unique identifier; Check whether an unique identifier has been cleared

SYNOPSIS

```
#include <ipc/chId.h>
int uiBuild(KnUniqueId * ui, unsigned int type, unsigned int site, unsigned int hea,
unsigned int tail);

int uiClear(KnUniqueId * Ui);

int uiEqual(KnUniqueId * ui1, KnUniqueId * ui2);

int uiGetSite(KnUniqueId * ui);

int uiIsLocal(KnUniqueId * ui);

int uiSite(KnUniqueId * ui, unsigned long site);

int uiValid(KnUniqueId * ui);
```

FEATURES

IPC

DESCRIPTION

The *uiBuild* system call builds a unique identifier from user-provided values. The built UI value is returned in the *KnUniqueId* structure pointed to by *ui*.

The `type` field defines the type of the UI as follows:

K_UIPORT	The UI will be usable as a port UI for the IPC module. The UI can then be usedin a call to *portDeclare(2K))*.
K_UIGROUP	The UI will be usable as a group UI for the IPC module. The UI can then be used in a call to *grpPortInsert(2K))*.
K_UISITE	The UI will be a site UI.
K_UIMSGQUEUE	The UI will be usable as a message queue UI for the MIPC_G module. The UI can then be used in a call to *msgQueueBind(2K))*.

The *site* field is a site number. If a non-zero value is given, this field will be interpreted by the kernel as a location hint for the UI.

The `head` and `tail` fields represent the stamp of the UI. `tail` is a 32 bits value. `head` is is a 13 bits value.

The *uiIsLocal* function checks whether the UI given by *ui* is recognized by the local site, and returns 1 if the UI is recognized, 0 otherwise.

On every site, the kernel declares a pre-defined site UI by default. The *uiSite* function returns the value of this pre-defined UI for the site whose site number is *site* (see *uiLocalSite* (2K)). Note that *uiSite (ui, site)* is equivalent to *uiBuild (ui, K_UISITE, site, 0, 0).*

The *uiClear* function initializes *ui* with a standard null value.

The *uiEqual* function returns 1 if the two UI's passed as arguments are equal, 0 otherwise.

The *uiGetSite* function returns the site information contained within *ui.*

The *uiValid* function returns 0 if *ui* is equal to the null UI, 1 otherwise.

RETURN VALUE

The *uiIsLocal* function returns 1 if the UI checked is local, 0 otherwise. The *uiValid* function returns 1 if the checked UI is different from the null value, 0 otherwise. The *uiEqual* function returns 1 if the UI's are identical, 0 otherwise. Other calls return 0 upon successful completion, a negative error code otherwise.

The *uiClear, uiValid, uiEqual* and *uiGetSite* never return K_EFAULT. If they are invoked with an invalid UI pointer, an exception occurs within the caller's context.

ERRORS

[K_EFAULT] Some of the provided data are outside the current actor's address space (*uiBuild* and *uiLocalSite*).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portDeclare`(2K) , `msgQueueBind`(2K) , `uiLocalSite`(2K)

NAME

uiBuild, uiClear, uiEqual, uiGetSite, uiIsLocal, uiSite, uiValid – Build a user-defined unique identifier; Clear a unique identifer; Compare two unique identifers; Extract the site number from a unique identifier; Check a unique identifier's locality; Get a pre-defined site unique identifier; Check whether an unique identifier has been cleared

SYNOPSIS

```
#include <ipc/chId.h>
int uiBuild(KnUniqueId * ui, unsigned int type, unsigned int site, unsigned int hea,
unsigned int tail);

int uiClear(KnUniqueId * Ui);

int uiEqual(KnUniqueId * ui1, KnUniqueId * ui2);

int uiGetSite(KnUniqueId * ui);

int uiIsLocal(KnUniqueId * ui);

int uiSite(KnUniqueId * ui, unsigned long site);

int uiValid(KnUniqueId * ui);
```

FEATURES

IPC

DESCRIPTION

The *uiBuild* system call builds a unique identifier from user-provided values. The built UI value is returned in the *KnUniqueId* structure pointed to by *ui*.

The `type` field defines the type of the UI as follows:

K_UIPORT	The UI will be usable as a port UI for the IPC module. The UI can then be usedin a call to *portDeclare(2K))*.
K_UIGROUP	The UI will be usable as a group UI for the IPC module. The UI can then be used in a call to *grpPortInsert(2K))*.
K_UISITE	The UI will be a site UI.
K_UIMSGQUEUE	The UI will be usable as a message queue UI for the MIPC_G module. The UI can then be used in a call to *msgQueueBind(2K))*.

The *site* field is a site number. If a non-zero value is given, this field will be interpreted by the kernel as a location hint for the UI.

The `head` and `tail` fields represent the stamp of the UI. `tail` is a 32 bits value. `head` is is a 13 bits value.

The *uiIsLocal* function checks whether the UI given by *ui* is recognized by the local site, and returns 1 if the UI is recognized, 0 otherwise.

On every site, the kernel declares a pre-defined site UI by default. The *uiSite* function returns the value of this pre-defined UI for the site whose site number is *site* (see *uiLocalSite* (2K)). Note that *uiSite (ui, site)* is equivalent to *uiBuild (ui, K_UISITE, site, 0, 0).*

The *uiClear* function initializes *ui* with a standard null value.

The *uiEqual* function returns 1 if the two UI's passed as arguments are equal, 0 otherwise.

The *uiGetSite* function returns the site information contained within *ui.*

The *uiValid* function returns 0 if *ui* is equal to the null UI, 1 otherwise.

RETURN VALUE

The *uiIsLocal* function returns 1 if the UI checked is local, 0 otherwise. The *uiValid* function returns 1 if the checked UI is different from the null value, 0 otherwise. The *uiEqual* function returns 1 if the UI's are identical, 0 otherwise. Other calls return 0 upon successful completion, a negative error code otherwise.

The *uiClear, uiValid, uiEqual* and *uiGetSite* never return K_EFAULT. If they are invoked with an invalid UI pointer, an exception occurs within the caller's context.

ERRORS

[K_EFAULT] Some of the provided data are outside the current actor's address space (*uiBuild* and *uiLocalSite*).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`portDeclare`(2K) , `msgQueueBind`(2K) , `uiLocalSite`(2K)

NAME

univTime, univTimeSet, univTimeAdjust – Get time-of-day; Set time-of-day; Adjust time-of-day univTimeGetRes; Get time-of-day resolution

SYNOPSIS

#include <date/chDate.h>
int **univTime**(KnTimeVal * *curtime*);

int **univTimeSet**(KnTimeVal * *oldtime*, KnTimeVal * *newtime*);

int **univTimeAdjust**(KnTimeVal * *olddelta*, KnTimeVal * *newdelta*);

int **univTimeGetRes**(KnTimeVal * *resolution*);

FEATURES

DATE

DESCRIPTION

The kernel maintains a Universal Time (time-of-day) variable using a *KnTimeVal* structure whose members are defined in *sysTime(2K)*.

If not initialized from a hardware time-of-day chip during system initialization, this variable will hold an invalid value until *univTimeSet* is called.

The *univTime* function returns the time of day in Universal Time to *curtime* . If the time of day is not yet set, *curtime* will be set to an invalid value (*tmSec = 0*, *tmNSec = K_INVALID_NSEC*).

The *univTimeSet* and *univTimeAdjust* functions allow you to get and/or set the kernel time-of-day value, or to request a progressive adjustment of that value.

The *univTimeGetRes* function obtains the resolution of the Universal Time system calls. The time value returned in *resolution* represents the smallest possible difference between two distinct values of the universal time.

All the *univTime* arguments are pointers to *KnTimeVal* structures. The input arguments are *newtime* and *newdelta* . The output arguments sasre *curtime*, *oldtime*, *olddelta* and *resolution* . If *newtime* is not 0, the kernel variable is assigned with the contents of the structure pointed to by *newtime*. If *oldtime* is not 0, the current value of the kernel variable is given in the structure pointed to by *oldtime* (if *newtime* is not 0, the returned value corresponds to the value of the kernel variable before its modification). If *newdelta* is not 0, the kernel will add (if the structure pointed to by *newdelta* represents a positive value) or substract (if *newdelta* represents a negative value) the number of (seconds, nanoseconds) represented by *newdelta*. This adjustment is done progressively: a small portion of the adjustment which is a fraction of the system clock resolution is performed at each clock interrupt until the full adjustment is made. The time value always increases. If *olddelta* is not 0, the value of the remaining adjustment is given in the structure pointed to by *olddelta*.

Only *SUPERVISOR* threads (see *threadCreate* (2K)) or threads belonging to *SYSTEM* actors (see *actorCreate* (2K)) are allowed to set the time-of-day or to request an adjustment.

RETURN VALUE

Upon successful completion, the Universal Time system calls return zero. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_EINVAL]	The time value referenced by *newtime* or *newdelta* is invalid.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysTime`(2K)

NAME

univTime, univTimeSet, univTimeAdjust – Get time-of-day; Set time-of-day; Adjust time-of-day univTimeGetRes; Get time-of-day resolution

SYNOPSIS

```
#include <date/chDate.h>
int univTime(KnTimeVal * curtime);

int univTimeSet(KnTimeVal * oldtime, KnTimeVal * newtime);

int univTimeAdjust(KnTimeVal * olddelta, KnTimeVal * newdelta);

int univTimeGetRes(KnTimeVal * resolution);
```

FEATURES

DATE

DESCRIPTION

The kernel maintains a Universal Time (time-of-day) variable using a *KnTimeVal* structure whose members are defined in *sysTime(2K)*.

If not initialized from a hardware time-of-day chip during system initialization, this variable will hold an invalid value until *univTimeSet* is called.

The *univTime* function returns the time of day in Universal Time to *curtime* . If the time of day is not yet set, *curtime* will be set to an invalid value (*tmSec = 0*, *tmNSec = K_INVALID_NSEC*).

The *univTimeSet* and *univTimeAdjust* functions allow you to get and/or set the kernel time-of-day value, or to request a progressive adjustment of that value.

The *univTimeGetRes* function obtains the resolution of the Universal Time system calls. The time value returned in *resolution* represents the smallest possible difference between two distinct values of the universal time.

All the *univTime* arguments are pointers to *KnTimeVal* structures. The input arguments are *newtime* and *newdelta* . The output arguments sasre *curtime, oldtime, olddelta* and *resolution* . If *newtime* is not 0, the kernel variable is assigned with the contents of the structure pointed to by *newtime.* If *oldtime* is not 0, the current value of the kernel variable is given in the structure pointed to by *oldtime* (if *newtime* is not 0, the returned value corresponds to the value of the kernel variable before its modification). If *newdelta* is not 0, the kernel will add (if the structure pointed to by *newdelta* represents a positive value) or substract (if *newdelta* represents a negative value) the number of (seconds, nanoseconds) represented by *newdelta.* This adjustment is done progressively: a small portion of the adjustment which is a fraction of the system clock resolution is performed at each clock interrupt until the full adjustment is made. The time value always increases. If *olddelta* is not 0, the value of the remaining adjustment is given in the structure pointed to by *olddelta.*

Only *SUPERVISOR* threads (see *threadCreate* (2K)) or threads belonging to *SYSTEM* actors (see *actorCreate* (2K)) are allowed to set the time-of-day or to request an adjustment.

RETURN VALUE

Upon successful completion, the Universal Time system calls return zero. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_EINVAL]	The time value referenced by *newtime* or *newdelta* is invalid.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysTime`(2K)

NAME

univTime, univTimeSet, univTimeAdjust – Get time-of-day; Set time-of-day; Adjust time-of-day univTimeGetRes; Get time-of-day resolution

SYNOPSIS

```
#include <date/chDate.h>
int univTime(KnTimeVal * curtime);

int univTimeSet(KnTimeVal * oldtime, KnTimeVal * newtime);

int univTimeAdjust(KnTimeVal * olddelta, KnTimeVal * newdelta);

int univTimeGetRes(KnTimeVal * resolution);
```

FEATURES

DATE

DESCRIPTION

The kernel maintains a Universal Time (time-of-day) variable using a *KnTimeVal* structure whose members are defined in *sysTime(2K)*.

If not initialized from a hardware time-of-day chip during system initialization, this variable will hold an invalid value until *univTimeSet* is called.

The *univTime* function returns the time of day in Universal Time to *curtime* . If the time of day is not yet set, *curtime* will be set to an invalid value (*tmSec = 0*, *tmNSec = K_INVALID_NSEC*).

The *univTimeSet* and *univTimeAdjust* functions allow you to get and/or set the kernel time-of-day value, or to request a progressive adjustment of that value.

The *univTimeGetRes* function obtains the resolution of the Universal Time system calls. The time value returned in *resolution* represents the smallest possible difference between two distinct values of the universal time.

All the *univTime* arguments are pointers to *KnTimeVal* structures. The input arguments are *newtime* and *newdelta* . The output arguments sasre *curtime, oldtime, olddelta* and *resolution* . If *newtime* is not 0, the kernel variable is assigned with the contents of the structure pointed to by *newtime.* If *oldtime* is not 0, the current value of the kernel variable is given in the structure pointed to by *oldtime* (if *newtime* is not 0, the returned value corresponds to the value of the kernel variable before its modification). If *newdelta* is not 0, the kernel will add (if the structure pointed to by *newdelta* represents a positive value) or substract (if *newdelta* represents a negative value) the number of (seconds, nanoseconds) represented by *newdelta.* This adjustment is done progressively: a small portion of the adjustment which is a fraction of the system clock resolution is performed at each clock interrupt until the full adjustment is made. The time value always increases. If *olddelta* is not 0, the value of the remaining adjustment is given in the structure pointed to by *olddelta.*

Only *SUPERVISOR* threads (see *threadCreate* (2K)) or threads belonging to *SYSTEM* actors (see *actorCreate* (2K)) are allowed to set the time-of-day or to request an adjustment.

RETURN VALUE

Upon successful completion, the Universal Time system calls return zero. Otherwise, a negative error code is returned.

ERRORS

[K_EFAULT]	Some of the data provided are outside the current actor's address space.
[K_EINVAL]	The time value referenced by *newtime* or *newdelta* is invalid.
[K_EPRIV]	The current thread is neither a supervisor thread nor a thread of a system actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysTime`(2K)

NAME

virtualTimeGetRes – get virtual time resolution

SYNOPSIS

#include <vtimer/chVtimer.h>
int **virtualTimeGetRes**(KnTimeVal **resolution*);

FEATURES

VTIMER

DESCRIPTION

The *virtualTimeGetRes* function obtains the resolution of virtual time specifications and returned values in the *threadTimes*(2K) and *svVirtualTimeoutSet*(2K). The time value returned in *resolution* represents the smallest possible difference between the two distinct virtual time values.

RETURN VALUE

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

ERRORS

[K_EFAULT] *resolution* contains an invalid memory address.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svVirtualTimeoutSet`(2K), `threadTimes`(2K)

NAME

vmCopy – copy data between actor address spaces

SYNOPSIS

#include <mem/chMem.h>
int **vmCopy**(KnCap **srcactorcap*, VmAddr *srcaddress*, KnCap **dstactorcap*, VmAddr *dstaddress*, VmSize *size*, VmFlags *flags*);

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *vmCopy* function copies data from a source actor address space to a target actor address space. The source data is specified by *srcactorcap* - the source actor capability, *srcaddress* - the data start address in the source actor address space and `size` - the data size. The target is described by *dstactorcap* - the target actor capability and *dstaddress* - the start address in the target actor address space.

If *srcactorcap* and/or *dstactorcap* is K_MYACTOR, the current actor is used. If *srcactorcap* and/or *dstactorcap* is K_SVACTOR, the supervisor address space is used.

The *flags* argument has to be zero and will be used for future interface extensions.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability.
[K_EUNKNOWN]	*tgtactorcap* or *srcactorcap* does not specify a reachable actor.
[K_EFAULT]	Some or all of the addresses in the source or the target address range are invalid.
[K_EFAIL]	A remote transaction has failed during a remote *vmCopy*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

vmFree – free memory

SYNOPSIS

#include <mem/chMem.h>
int **vmFree**(KnCap **actorcap*, VmAddr *address*, VmSize *size*);

FEATURES

MEM_VIRTUAL

DESCRIPTION

The *vmFree* function frees memory in an actor address space from *address* to *address* + `size` -*1*. The actor is specified by its capability *actorcap*. On—demand physical memory corresponding to the specified address space range is deallocated without region destruction. The *address* and `size` fields must be page-aligned.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was provided.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EADDR]	Some or all the addresses from the target address range are invalid .
[K_EROUND]	*address* isn't page aligned.
[K_EROUND]	`size` isn't page-aligned.
[K_EBUSY]	Tried to free no—demand (mapped to a region with the K_NODEMAND attribute) physical memory.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

vmLock, vmUnLock – Fix data in memory; Data in memory

SYNOPSIS

#include <mem/chMem.h>
int **vmLock**(KnCap * *actorcap*, VmAddr *address*, VmSize *size*, VmFlags *flags*);

int **vmUnLock**(KnCap * *actorcap*, VmAddr *address*, VmSize *size*);

FEATURES

MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *vmLock* function locks the translations of the target virtual address range of the target actor address space. The *(vmUnLock)* function unlocks the translations of the target virtual address range of the target actor address space.

The target actor is specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the supervisor address space is used.

The target address range is specified using its starting *address* and its *size*. Both *address* and `size` must be page aligned.

The *vmLock* function causes each virtual address from the target address range to be mapped to physical memory with the required protections. If the K_WRITABLE flag is set in the *flags* argument, write access is required; otherwise, read access is required. Then *vmLock* gets a reader lock on the translations from the target address range. The reader lock guarantees that when a thread tries to modify either the physical address or the protections of a locked mapping (using *rgnFree, rgnSetProt* or *lcFlush* , for example), it blocks until the corresponding *vmUnLock* is encountered.

If the K_WRITABLE flag is set in the *flags* argument, *vmLock* marks the locked pages as modified. This could be necessary if, for example after locking, the pages are modified by a driver using the corresponding physical addresses.

An address range locked by a number of non-overlapped *vmLock* operations may be unlocked by any number of non-overlapped *vmUnLock* operations.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability has been provided.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EADDR]	Some or all addresses from the target address range are invalid.

[K_EPROT]	Write access is required but some or all the addresses from the target address range are write—p rotected.
[K_EROUND]	*address* isn't page aligned.
[K_EROUND]	`size` isn't page-aligned.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnFree`(2K) , `lcFlush`(2K)

NAME

vmPageSize – get the minimum allocatable memory block size

SYNOPSIS

```
#include <mem/chMem.h>
int vmPageSize(void);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *vmPageSize* system call returns the minimum allocatable memory block size. In case of virtual memory management, this size corresponds to the virtual page size.

RETURN VALUE

The minimum allocatable memory block size, in bytes.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svPagesAllocate`(2K)

NAME

vmPhysAddr – get the physical address corresponding to a virtual address

SYNOPSIS

#include <mem/chMem.h>
int **vmPhysAddr**(KnCap **actorcap*, VmAddr *virtaddress*, PhAddr **physaddr*);

FEATURES

MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *vmPhysAddr* system call returns to *physaddr* the physical address mapped to the virtual address *virtaddress* in the target actor address space.

The target actor is specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the supervisor address space is used.

The caller must insure that a physical address was effectively mapped to the target virtual address before invoking *vmPhysAddr*, and that the same physical address will always be mapped to the target virtual address during the invocation.

The *vmPhysAddr* call can be called from interrupt level.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability was provided.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

vmSetPar – set the memory management parameters

SYNOPSIS

```
#include <mem/chMem.h>
int vmSetPar(KnVmPar *par);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

vmSetPar sets the virtual memory management parameters on the current site.

The KnVmPar structure describes the site memory management parameters:

```
KnUniqueId  defaultMapper ;
VmSize      redMark ;
VmSize      orangeMark ;
VmSize      greenMark ;
VmAddr      usrStartAddr ;
VmAddr      usrEndAddr ;
VmAddr      svStartAddr ;
VmAddr      svEndAddr ;
VmOffset    minOffset ;
VmOffset    maxOffset ;
```

defaultMapper field specifies the default mapper UI.

When the size of free memory on the site goes over the orangeMark the memory management system starts the swap out daemon(s).

When the size of free memory on the site comes back to the greenMark, the swap out daemons stop.

When the size of free memory on the site goes over the redMark the memory management system tries to stop all threads not involved in swap out activity.

usrStartAddr and usrEndAddr specify the boundary of user address space.

svStartAddr and svEndAddr specify the boundary of the system address space.

minOffset and maxOffset specifies the valid range of segment offsets.

RETURN VALUE

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

ERRORS

[K_EINVAL] The values of marks are inconsistent.

[K_EFAULT] Some of the provided arguments are outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `vmStat`(2K)

NAME

vmStat – get memory management statistics

SYNOPSIS

```
#include <mem/chMem.h>
int vmStat(KnVmStat *stat);
```

FEATURES

MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *vmStat* system call gets memory management statistics on the current site.

The *KnVmStat* contains the following fields:

```
KnVmPar par ;
VmSize  hostMem ;
VmSize  freeMem ;
VmSize  lockMem ;
```

The *hostMem* field gives the site's physical memory size.

The *freeMem* field gives the amount of currently unallocated physical memory.

The *lockMem* field gives the amount of currently allocated and non-swappable physical memory.

The *par* field gives the memory management parameters as described in *vmSetPar*(2K).

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

vmSetPar(2K)

NAME

vmLock, vmUnLock – Fix data in memory; Data in memory

SYNOPSIS

```
#include <mem/chMem.h>
int vmLock(KnCap * actorcap, VmAddr address, VmSize size, VmFlags flags);

int vmUnLock(KnCap * actorcap, VmAddr address, VmSize size);
```

FEATURES

MEM_PROTECTED, MEM_VIRTUAL

DESCRIPTION

The *vmLock* function locks the translations of the target virtual address range of the target actor address space. The *(vmUnLock)* function unlocks the translations of the target virtual address range of the target actor address space.

The target actor is specified by *actorcap* - a pointer to the actor capability. If *actorcap* is K_MYACTOR, the address space of the current actor is used. If *actorcap* is K_SVACTOR, the supervisor address space is used.

The target address range is specified using its starting *address* and its *size*. Both *address* and `size` must be page aligned.

The *vmLock* function causes each virtual address from the target address range to be mapped to physical memory with the required protections. If the K_WRITABLE flag is set in the *flags* argument, write access is required; otherwise, read access is required. Then *vmLock* gets a reader lock on the translations from the target address range. The reader lock guarantees that when a thread tries to modify either the physical address or the protections of a locked mapping (using *rgnFree*, *rgnSetProt* or *lcFlush* , for example), it blocks until the corresponding *vmUnLock* is encountered.

If the K_WRITABLE flag is set in the *flags* argument, *vmLock* marks the locked pages as modified. This could be necessary if, for example after locking, the pages are modified by a driver using the corresponding physical addresses.

An address range locked by a number of non-overlapped *vmLock* operations may be unlocked by any number of non-overlapped *vmUnLock* operations.

RETURN VALUE

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

ERRORS

[K_EFAULT]	Some of the arguments provided are outside the caller's address space.
[K_EINVAL]	An inconsistent actor capability has been provided.
[K_EUNKNOWN]	*actorcap* does not specify a reachable actor.
[K_EADDR]	Some or all addresses from the target address range are invalid.

[K_EPROT]	Write access is required but some or all the addresses from the target address range are write—p rotected.
[K_EROUND]	*address* isn't page aligned.
[K_EROUND]	`size` isn't page-aligned.

RESTRICTIONS

The target actor and the current actor must be located on the same site.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rgnFree`(2K) , `lcFlush`(2K)

Index

G

I

L

M

P

R

S

T

U

V