

CarbonKernel

Real-time Operating System Simulator

Version 1.4

Native Programming Interface

October 2001

Table of Contents

1 CARBONKERNEL PROGRAMMING INTERFACE 3

| | |
|-------------------------------------------|---|
| 1.1 CONCEPTS AND FEATURES | 3 |
| 1.1.1 Monitoring specific memory accesses | 3 |
| 1.1.2 Message ports | 3 |
| 1.1.3 Dataports | 4 |
| 1.1.4 Control panels and magnets | 4 |
| 1.1.5 Synchronization objects | 5 |
| 1.1.6 Virtual terminals | 5 |
| 1.1.7 Time management | 5 |
| 1.1.8 Interrupt management | 6 |
| 1.1.9 Device I/O simulation | 6 |
| 1.1.10 Information services | 7 |
| 1.1.11 Simulation control | 7 |

2 THE MAGNET INTERFACE 70

| | |
|------------------------------|----|
| 2.1 WHAT'S A MAGNET ? | 70 |
| 2.2 USING TCL/TK FOR MAGNETS | 70 |
| 2.3 HOSTING A MAGNET | 70 |
| 2.4 NAMING CONVENTION | 70 |

1 CarbonKernel Programming Interface

This chapter describes the CarbonKernel Programming Interface (aka CKPI) which is part of CarbonKernel, the real-time operating system simulator. The CKPI is an interface which exports a comprehensive set of services implemented by the CarbonKernel's virtual RTOS for use by SDDK modules, FROGS native models and applications.

1.1 Concepts and features

1.1.1 Monitoring specific memory accesses

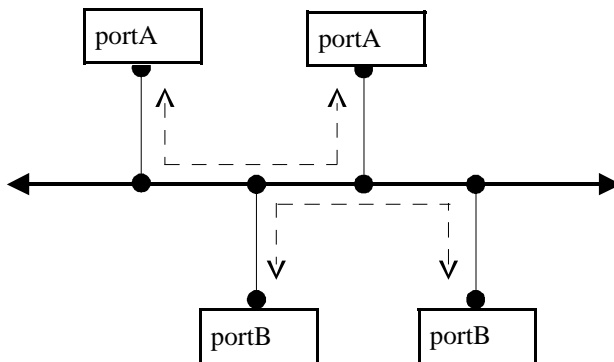
The CKPI exports four services, namely `ckGetMemPortByte()`, `ckGetMemPortWord()`, `ckGetMemPortDword()` and `ckGetMemPortQword()`, which you can invoke to obtain a chunk of access-monitored memory. When the application reads/writes from/to this memory, the simulation kernel schedules the invocation of a user-defined handler in the application code. This handler is always fired on behalf of the same context (thread/isr/dsr/callout) which caused the trigger.

Those services are implemented through the virtual memory management services of the host system. Restrictive access protections are set to the returned memory object, thus making the host operating system send an exception to the simulator when the application refers to it. The simulation kernel then traps the exception, and injects this event in the normal scheduling process.

1.1.2 Message ports

Message ports are access points to a global event bus managed by the VRTOS. Reading and writing from/to message ports is the simplest way to exchange bulks of unformatted binary data between simulation threads. It should be noted that *message ports are not intended to replace the standard data exchange services* implemented by the RTOS model; instead, they provide to simulated real-time threads a convenient mean for exchanging messages with the outer world. For instance, physical read/write routines from a simulated device driver can connect to data sources or sinks managed by native simulation models using message ports.

Message ports are named using an arbitrary long C string identifier. A logical broadcast communication path is established between all ports bound to the same port identifier over the event bus. When data is written to a port, it can be received by listeners of any other ports sharing the same identifier than the sender's.



Message ports are used internally by the simulation kernel to connect the ISE's animated magnets to their corresponding data sources within the simulator. Message ports also underlies the CarbonKernel's *dataport* feature allowing C/C++ variables to automatically broadcast their new value upon change.

1.1.3 Dataports

Dataports are C/C++ variables whose contents are automatically broadcast through an associated message port upon change. This is a powerful feature allowing simulation threads to monitor variable update events. Dataport variables *must* be data vectors, i.e. you may *not* define a *netshared* dataport variable.

The variable is automatically bound to its message port during the simulator initialization phase. The name of the associated message port is composed as follows:

```
/<variableName>@<nodeName>
```

where the first member is the exact name of the variable, and the second is the currently initialized node's name. The message port management routines are dataport-aware, and automatically complete the node specification of the port identifier if it starts with the dataport marker '/', and no '@' delimiter is found in this identifier. In such a case, the running node's name is appended to the port identifier.

A variable of this kind must be specifically declared with the *dataport* qualifier in the C/C++ source file. For instance, the following fragment defines a broadcast integer variable whose value is monitored from a dedicated sub-routine:

```
dataport int WatchMe = 1;

void updateWatchCount ()
{
    /* the new value of "WatchMe" is broadcast
       through the message port named /WatchMe@<nodeName>
       right after the following statement... */
    WatchMe++;
}

void monitorWatchCount ()
{
    ckhandle_t port;
    int watchCount;
    ckclock_t stime;

    if (ckBindPort("/WatchMe",0,&port) < 0)
        oops();

    for (;;)
    {
        ckReadPort(port,watchCount,CK_INFINITE,&stime);
        printf("WatchMe's      new      value      since      %f      usc      is:
%d\n",stime,watchCount);
    }
}
```

Dataports are first choice candidates for use with ISE's graphical magnets you just need to configure an animated magnet to listen to the dataport identifier of your choice to have a graphical monitoring of the associated variable state.

1.1.4 Control panels and magnets

Control panels are graphical windows managed by the ISE displaying magnets. A magnet is a graphical widget displaying the value of a simulation variable. Thus, the current value and/or state exported by the variable can be monitored graphically, and may be changed interactively by the user provided the magnet allows this operation.

New magnets can be written in Tcl and integrated to the ISE provided that they follow the proper interface convention. This interface is documented in the chapter “*Writing animated magnets*”.

A magnet is logically bound to a message port to send and/or receive data to/from the application. Because *dataports* are specialized message ports broadcasting the contents of variables, they are first choice candidates for exposure in magnets. But almost any basic message ports can be connected to a magnet, provided that both sides (i.e. the application and the magnet code in Tcl) agree upon the format of the exchanged data.

Data are exchanged as Tcl lists between the simulation kernel and the magnet code written in Tcl. The simulation kernel is responsible for converting the contents of a binary object (e.g. An integer variable or a struct) to a Tcl list representation for the purpose of exporting the value to the magnet hosted by the ISE, and conversely, from a Tcl list to a binary form when the magnet requests the simulation to change the object's value.

A control panel is aimed at gathering a set of related magnets in a single display window. There is virtually no limit to the number of magnets which can be attached to a given panel. Likewise, you may define as many control panels as you need, and attach them on a per-node basis using the CarbonKernel librarian.

1.1.5 Synchronization objects

The CKPI provides the following set of synchronization objects one may use to write reusable simulation components:

spin locks allow inter-node and intra-node locking of critical sections, enforcing “busy waiting of threads pending for a unique resource.

mutexes are dedicated to serialize execution of threads pertaining to the same node in critical sections. The CKPI mutexes support the *priority inheritance protocol* to prevent the priority inversion phenomenon.

condition variables provide a convenient mean to synchronize threads from a given node with event signaling.

A specific service, namely `ckPendSynch()`, can be used to synchronize RTOS threads with internal events available from the FROGS/SIMEX layer. For instance, a RTOS thread can pend on a SIMEX queue or semaphore using this service. This is especially useful in implementing FROGS native simulation models providing extended services to the application.

1.1.6 Virtual terminals

The VRTOS supports console i/o from multiple concurrent instances of a GUI application emulating a virtual ASCII-based terminal. A set of dedicated services allows simulation threads to read from and write to them.

The GUI application presenting the terminal interface accepts replay files to execute automated input session with or without simulation time awareness.

1.1.7 Time management

Time management services exist that give (read-only) access to the global simulation clock, and to individual RTC values on a per-node basis. The global simulation clock is always expressed in absolute time and allows fractional values (of micro-seconds), the RTC values are expressed in *ticks*, whose duration (in absolute time) depends on the individual node configuration.

A pair of services, namely `ckLockTime()` and `ckUnlockTime()`, allowing to suspend then resume the global simulation clock can be used to execute some portion of the application code at no time charge. However, due to their impact on the overall simulation process, these calls should be used with extreme care.

1.1.8 Interrupt management

The CKPI interface provides means to create, destroy, raise, mask or unmask simulated interrupts. Interrupt masking can be done individually for a given interrupt object, or globally for a specific interrupt level.

Event-generation laws applied to interrupt sources can be programmatically defined and/or changed.

Interrupt objects can also be graphically defined by the ISE's librarian tool during the simulation configuration phase.

An interrupt object has three major properties:

- ▣ A vector number, which is an integer value ranging from 0 to 255 (inclusive).
- ▣ A level, which is an integer value which may range from 1 to the maximum interrupt level allowed by the simulated RTOS personality.
- ▣ An interrupt service routine, which is a C routine the VRTOS will call each time the interrupt is taken. Interrupts of equal or lower level are masked during a given ISR execution.
- ▣ A deferred service routine, which is a C routine the VRTOS will call each time the interrupt service routine defers the interrupt handling to it. The simulation kernel does not mask interrupts before starting DSRs.

It can happen that multiple interrupt events are scheduled at the same simulation time (i.e. FROGS/SIMEX's clock time). In such a case, interrupts with higher levels are prioritary. If two interrupt objects have the same level, the one with the lowest vector is prioritary. If both interrupt objects have identical levels and vectors, FIFO ordering is applied.

An RTOS model usually installs a real-time clock timer on each emerging node (e.g. The CarbonKernel's `eCos` model does it when constructing an `eCos` node). In such a case, a handle to the RTC interrupt can be fetched by searching for the object named "SystemTimer" using the `ckGetObjectHandle()` service. However, setting the parameter which specifies the number of timer ticks per second to zero in the node's configuration window should prevent this installation.

1.1.9 Device I/O simulation

CarbonKernel gives you the ability to create simulated device drivers using the SDDK interface. These drivers can in turn establish a dialog with FROGS native simulation models, so as to simulate hardware peripherals for instance.

The main goal of a simulated device driver is to implement the simulation counter-part of a "real" driver accessing "real" hardware for the application, by providing a normalized way of sending and receiving data to/from a pseudo-device faking the real hardware during the simulation process.

The reasons you may want to use **SDDK** drivers in your simulation environment can be:

- ensuring your system exports a clean and seamless interface between the mere application code and the low-level code, facilitating the final migration from the simulation host to the real target environment.
- porting target-based driver code to the simulation environment.
- applying test harness to your code, such as filter modules you can use to dynamically monitor and/or perturbate the data stream between the simulated hardware and the application.

For all these reasons you need to access all driver services through a concise, well-defined and generic interface. The **CKPI** implements this interface between the application and the drivers, which is based on services from the VRTOS kernel.

1.1.10 Information services

A comprehensive set of routines are provided by the **CKPI** to pass the user information back concerning the current context and publicly accessible objects defined by the simulation kernel.

The simulator deals with simulation objects it creates, schedules and destroys during its lifetime. Each major programmatic concept is represented by a simulation object class, such as interrupts, message ports, threads and so on.

Simulation objects are dually named resources. A character string is usually passed by the user or otherwise defined by the simulation kernel at creation and stands for the object's external name. The simulation kernel additionally maintains an internal 32bits unique opaque handle for each simulation object. This handle is *globally* unique at the whole simulation level (i.e. unique across node boundaries). `ckGetObjectHandle()` can be used to retrieve an object's handle from its external name. Conversely, `ckGetObjectName()` retrieves an object's external name from its internal handle.

Finally, simulation objects can be tagged for the purpose of discriminating them. You may use `ckSetTag()` and `ckGetTag()` routines to manipulate object tags.

1.1.11 Simulation control

Miscellaneous services are available to control the simulation as a whole, such as suspending then resuming it, emitting traces, changing the host simulation speed or the node's perceived processing speed (i.e. *Target Warp* factor), or hooking user-defined handlers on pre-defined simulation events.

ckSuspendSimulation - – suspend simulation

SYNOPSIS

```
#include <ck/scontrol.h>
void ckSuspendSimulation()
```

DESCRIPTION

ckSuspendSimulation() immediately suspends the simulation process, transferring control to the ISE. This service has no effect if the ISE is not active.

RETURN VALUES

None.

SEE ALSO

ckResumeSimulation()

ckResumeSimulation - resume simulation**SYNOPSIS**

```
#include <ck/scontrol.h>
void ckResumeSimulation()
```

DESCRIPTION

ckResumeSimulation() resumes the simulation process immediately. The simulation thread which was executing when the process was suspended regains control. This service has no effect if the ISE is not active, or if the simulation was not suspended.

RETURN VALUES

None.

SEE ALSO

ckSuspendSimulation()

ckFinishSimulation - terminate simulation**SYNOPSIS**

```
#include <ck/scontrol.h>
void ckFinishSimulation(int xcode)
```

DESCRIPTION

ckFinishSimulation() terminates the simulation process, returning **xcode** to the environment. The ISE takes control over the simulator before it exits, allowing post-mortem inspection of the simulation objects. If the ISE is not active, the simulator exits immediately with the specified return code.

RETURN VALUES

None.

ckWarning - send a warning message

SYNOPSIS

```
#include <ck/scontrol.h>

void ckWarning(const char *format, ...)
```

DESCRIPTION

ckWarning() formats then emits a warning message to the simulation manager. If the ISE is active, this message is automatically logged in the "Error Log" window. Otherwise, it is written to the simulator's standard error stream. **format** is a format string conforming to the printf(3) specifications. An appropriate variable argument list should follow this parameter.

RETURN VALUES

None.

SEE ALSO

ckFatal()

ckFatal - abort simulation

SYNOPSIS

```
#include <ck/scontrol.h>

void ckFatal(const char *format, ...)
```

DESCRIPTION

ckFatal() formats then emits an abort message to the simulation manager. If the ISE is active, this message is automatically logged in the "Error Log" window. Otherwise, it is written to the simulator's standard error stream. **format** is a format string conforming to the printf(3) specifications. An appropriate variable argument list should follow this parameter.

The simulation process is aborted and the simulator exits with a non-zero error code after the message is displayed.

RETURN VALUES

None.

SEE ALSO

ckWarning()

ckTrace - send a trace message

SYNOPSIS

```
#include <ck/scontrol.h>

void ckTrace(int flags, const char *format, ...)
```

DESCRIPTION

ckTrace() formats then emits a trace message to the trace manager. If the ISE is active, this message is automatically logged in the "Trace" window. Otherwise, it is written to the simulator's standard error stream, only if the CK_TRACE_ALERT flag is set in the **flags** parameter. Otherwise, non-displayed messages are simply discarded. **format** is a format string conforming to the printf(3) specifications. An appropriate variable argument list should follow this parameter.

This service adds user-defined trace messages to the regular information emitted by the pre-defined trace points. All of these messages are logged in the "Trace" window of the ISE.

The **flags** parameter gives the message attribute. It can take one of the following values :

CK_TRACE_ALERT, causes the message to be displayed using a special alert background color in the "Trace" window if the ISE is active. It may also cause the simulation to stop if the «Break on trace alerts» option is active. The default alert color is red.

CK_TRACE_HIGHLIGHT causes the message to be highlighted using a special background color in the "Trace" window if the ISE is active. The default color is yellow.

CK_TRACE_NORMAL causes the message to be displayed with no special attribute.

The special background color used to display CK_TRACE_ALERT and CK_TRACE_HIGHLIGHT messages can be specified by adding one of the following values to the attribute mask, whether CK_TRACE_RED, CK_TRACE_YELLOW, CK_TRACE_BLUE or CK_TRACE_GREEN. These bits are ignored for CK_TRACE_NORMAL messages.

RETURN VALUES

None.

SEE ALSO

ckWarning()

ckLockTime/ckUnlockTime - time-locked code section**SYNOPSIS**

```
#include <ck/clock.h>
unsigned ckLockTime()
unsigned ckUnlockTime()
```

DESCRIPTION

ckLockTime() prevents the simulation clock from being charged for the execution of the subsequent instructions, until ckLockTime() reverts the effect of this service. This means the simulation clock value will not change while ckLockTime() is in effect.

A lock count is maintained by these primitives, ensuring the section exists until ckUnlockTime() is called the same number of times ckLockTime() was invoked.

The macros CK_LOCK_TIME() and CK_UNLOCK_TIME() are aliases to these

RETURN VALUES

The new locking level is returned.

ckTick - RTC tick announce

SYNOPSIS

```
#include <ck/clock.h>

void ckTick()
```

DESCRIPTION

ckTick() announces a new clock tick to the simulator. When the RTOS model does not provide a standard entry point for this kind of service, ckTick() can be called on behalf of an interrupt context to inform the VRTOS of such event. When a RTOS model configures a tick timer internally, it is expected to provide an appropriate tick handler, which will call in turn the proper VRTOS service to announce the outstanding clock tick. In such a case, there is no need for calling ckTick() from the application code.

The outstanding tick is charged to the running node's real-time clock.

ckTick() must always be called on behalf of an interrupt context (i.e. ISR or DSR), otherwise, a fatal error is raised.

RETURN VALUES

None.

EXAMPLE

```
int timerIsr (ckvector_t vector, void *cookie)
{
    ckTick(); /* Signal real-time clock tick */
    return CK_ISR_HANDLED;
}

void installTimer (void)
{
    ckhandle_t handle;
    /* Create a timer interrupt object */
    ckCreateIntr("Timer",0,1,timerIsr,NULL,NULL,&handle);
    /* Make the timer tick each 10th of a second */
    ckProgramIntr(handle,CK_INTR_PERIODICAL,"100 msc");
}
```

ckGetTime - get simulation clock value

SYNOPSIS

```
#include <ck/clock.h>
ckclock_t ckGetTime()
```

DESCRIPTION

ckGetTime() returns the current value of the simulation clock maintained by the underlying event-driven simulator (i.e. FROGS/SIMEX). This clock is distinct from the real-time clock each node maintains. This is the absolute point in time reached by the simulation engine.

RETURN VALUES

A floating-point value representing the simulated time elapsed since the simulation kernel was started. This value is expressed in micro-seconds.

SEE ALSO

ckGetTicks()

EXAMPLE

```
{
    ckclock_t now = ckGetTime();
    printf("ABSOLUTE TIME: %.3f usc\n",now);
}
```


ckGetTicks - get node's clock value

SYNOPSIS

```
#include <ck/clock.h>
ckticks_t ckGetTicks()
```

DESCRIPTION

ckGetTicks() returns the count of elapsed ticks for the running node. A new tick is usually announced to the VRTOS by the software handler of a periodical interrupt generated by an event source.

Concurrent nodes may have distinct tick values, depending on their respective RTC configuration (i.e. the number of ticks per second may be different among nodes).

RETURN VALUES

A (long-)long integer value is returned to the caller representing the total number of elapsed ticks since the running node was started.

SEE ALSO

ckTick(), ckGetTime()

EXAMPLE

```
{
    ckticks_t now = ckGetTicks();
    printf("ELAPSED TICKS: %llu real-time ticks\n",now);
}
```

ckPendSynch - low-level synchronization with SIMEX

SYNOPSIS

```
#include <ck/ksynch.h>

int ckPendSynch(ckopaque_t synch, ckticks_t timeout)
```

DESCRIPTION

ckPendSynch() blocks the calling RTOS thread until the FROGS/SIMEX synchronization object identified by **synch** is signaled. This object's class *must* extend the SIMEX's **SxSynchro** superclass. Such synchronization object is usually defined by a FROGS native simulation model running concurrently to the simulated RTOS personality.

A watchdog can be set to unblock the calling RTOS thread using the **timeout** parameter if the object is not signaled within the allotted amount of time. Its value is a number of ticks to wait before the request is aborted. The tick duration is defined by the running node. If **timeout** is zero, the calling RTOS thread waits indefinitely.

This service is useful to FROGS add-ins for making CarbonKernel's RTOS threads wait for a condition they manage internally.

If **synch** is null, no object is monitored but the watchdog is started. The RTOS thread will resume execution after the timeout has elapsed. Consequently, the thread is unconditionally and indefinitely suspended if both **synch** and **timeout** are null.

RETURN VALUES

ckPendSynch() returns 0 if the object was signaled within the allotted time. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

ETIMEDOUT the timeout has elapsed before the synchronization object is signaled.

SEE ALSO

ckDelay()

ckDelay - delay a RTOS thread

SYNOPTIQUE

```
#include <ck/ksynch.h>
void ckDelay(ckticks_t timeout)
```

DESCRIPTION

ckDelay() blocks the calling RTOS thread for the specified number of ticks given by **timeout**. The thread is resumed after the timeout expires.

The tick duration is defined by the running node. If **timeout** is zero, the thread waits indefinitely.

RETURN VALUES

None.

SEE ALSO

ckPendSynch()

ckPoll, ckSelect - synchronous i/o multiplexing

SYNOPSIS

```
#include <ck/ksynch.h>

int ckPoll(fd_set *readyset);

int ckSelect(fd_set *waitset, struct timeval *tv);
```

DESCRIPTION

ckSelect() examines a set of file descriptors whose address is passed in **waitset**, to see if some of its descriptors are ready for reading. The total width of the set as defined by the host implementation of the *fd_set* type is checked.

CkSelect() is a service enabler to the select(2) system call for use on behalf of a RTOS thread. However, only the read condition on file descriptors can be checked by this service. The standard macros FD_SET(), FD_CLR(), FD_ISSET() and FD_ZERO() should be used to manipulate the set.

ckPoll() immediately returns to its caller after the set of ready descriptors is updated, whether there are some or not.

ckSelect() immediately returns to its caller if some of the specified descriptors are ready on entry. Otherwise, one of the following actions is taken:

- ▣ if **tv** is NULL, the calling RTOS thread is suspended until data become available for reading on some descriptors. In this case, the simulation kernel checks the descriptors periodically, according to the best trade-off between the CPU cost involved in probing the descriptors and the responsiveness of the application. Although this behaviour is not deterministic, the implementation ensures that the calling RTOS thread will be resumed a reasonably short time after some data are available on the descriptors.
- ▣ if **tv** points to a valid *timeval* structure, and the fields **tv_sec** and **tv_usec** are zero, ckSelect() behaves exactly like ckPoll().
- ▣ if **tv** points to a valid *timeval* structure, and **tv_sec** and/or **tv_usec** are non-zero, the entire simulation process is blocked until data are available for reading on some descriptors. In this case, **tv** is interpreted as a *host-based time value* (i.e. "newtonian" time), and *not* as a simulated time value.

COMMENTS

The simulation engine underlying CarbonKernel implements a multi-tasking kernel using co-routines, each of them having their own execution stack and running in a pseudo-parallel schedule. Whilst this characteristic gives a desirable control over the simulation clock and allows the simulation events to be repeatable and accurate across sessions, it has the following drawback: simulation threads cannot call blocking host-based services, unless having all other simulation threads hung during the wait is not an issue.

ckSelect() has been designed to circumvent this problem, by blocking the calling thread at the VRTOS level, until it is resumed after an i/o polling routine periodically fired by the SIMEX thread scheduler detects the expected condition on the descriptors. A thread blocked by a call to ckSelect() has the *ck select* status in the inspector's display.

ckSelect() should be used to manage the input channel a RTOS thread may have established with an external process.

RETURN VALUES

On success, ckPoll() and ckSelect() update the set with the value of a sub-set defining the readable descriptors, and return a positive integer indicating the number of descriptors in the set.

Zero indicates that the time limit referred to by **tv** expired.

EXAMPLE

```
void probeThread (int s)
{
    fd_set waitset;

    FD_ZERO(&waitset);
    /* "s" could be a socket connected to a GUI frontend */
    FD_SET(s,&waitset);

    for (;;)
    {
        if (ckSelect(&waitset,NULL) <= 0)
            oops("failed to select() socket");

        /* Now we can process the available message on "s" */
        ...
    }
}
```

ckCreateIntr - create an interrupt object

SYNOPSIS

```
#include <ck/intr.h>

int ckCreateIntr(const char *name, ckvector_t vector,
cklevel_t level, ckisr_t *isr, ckdsr_t *dsr, void *cookie,
ckhandle_t *handlep);
```

DESCRIPTION

This service creates a new interrupt object on the current node. The interrupt object can be given an external **name**, or remain anonymous to the ckGetObjectHandle() service if NULL is passed. A valid **vector** ranging from 0 to 255 (inclusive) and interrupt **level** ranging from 1 to the node's limit (inclusive) should be passed to define the object's priority.

For a given scheduling time, interrupt prioritization is done as follows:

- If two (or more) interrupts share the same level, lower the vector, priority the interrupt.
- Two (or more) interrupts having the same vector and priority level are fired in FIFO creation order.

COMMENTS

CarbonKernel's VRTOS uses a split interrupt model. Interrupt handlers are actually a pair of functions, one of which (the interrupt service routine, or ISR) is executed immediately and runs with the interrupts from the same or lower levels disabled. Since less priority interrupts are disabled for the duration of the ISR, the ISR should be very brief.

After the ISR exits, but before the VRTOS starts the rescheduling procedure, a deferred service routine (DSR) can be invoked, if the ISR requested this chaining by using a specific return code. The DSR then executes with scheduling disabled, but with interrupts enabled, so that further invocations of the same DSR can be queued. The DSR can't use any system calls that might put its underlying thread to sleep.

When the interrupt is taken, the **isr** is called with the current **vector** number and the **cookie** value. The interrupt service routine should return a status to its caller, indicating whether its associated deferred service routine should be chained. Returning CK_ISR_CALL_DSR causes the **dsr** to be invoked after the last pending ISR has returned, and before the VRTOS starts the rescheduling procedure. Otherwise, returning CK_ISR_HANDLED prevents the DSR from being called for the current interrupt. As a result of this, **dsr** can be passed as NULL if no DSR is associated to the interrupt, otherwise a valid DSR address should be passed. At the opposite, a valid **isr** must be passed to service the interrupt.

If the interrupt source is "bursty", it may be acceptable for several services of a given interrupt and appropriate calls to its ISR to occur before a requested DSR has been executed. The VRTOS maintains counts for posted DSRs, and in such a case the DSR will eventually be called once for the burst with a **count** value that tells it how many ISRs requested that the DSR be called.

The DSR is passed three arguments, namely the current **vector** number, the **count** of DSR triggers from the bottom-half ISR, and the caller's **cookie**.

RETURN VALUES

Zero is returned on success, and **handlep** points to the new interrupt object handle. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The `ckCreateIntr()` service fails if:

EINVAL **vector** is not a value in the range [0-255] inclusive.

EINVAL **level** is not in the range [1-*maxlvl*] inclusive, where *maxlvl* is specific to the current RTOS personality. Its value can be retrieved by a call to the `ckGetConf()` service with a **name** parameter of *_CK_NODE_MAX_ILVL*.

SEE ALSO

`ckDestroyIntr()`

EXAMPLE

```
int theIsr (ckvector_t vector, void *cookie)
{
    ckWarning("Interrupt caught on vector %u\n",vector);
    return CK_ISR_HANDLED;
}
...
{
    ckhandle_t handle;
    /* Create an interrupt object */
    ckCreateIntr("MyIntr",0,7,theIsr,NULL,NULL,&handle);
    /* Program the interrupt at 192.7 milli-seconds */
    ckProgramIntr(handle,CK_INTR_TIMER,"192.7 msc");
}
```

ckDestroyIntr - **destroy an interrupt object**

SYNOPSIS

```
#include <ck/intr.h>

int ckDestroyIntr(ckhandle_t handle);
```

DESCRIPTION

This service destroys an interrupt object which must exist on the current node. Any event source associated to this object is automatically removed. The destroyed interrupt must belong to the running node.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The ckDestroyIntr() service fails if:

ESRCH **handle** does not refer to a valid interrupt object defined on the current node.

SEE ALSO

ckCreateIntr()

ckRaiseIntr - raise an interrupt

SYNOPSIS

```
#include <ck/intr.h>

void ckRaiseIntr(cknid_t nid, ckvector_t vector, cklevel_t
level, ckisr_t *isr, void *cookie)
```

DESCRIPTION

ckRaiseIntr() schedules an interrupt of priority **level** to occur immediately on the node identified by **nid**. **isr** is the address of an interrupt service routine which is going to be called to service this event. There can't be any DSR attached to this kind of one-shot interrupt. Hence, CK_ISR_HANDLED should be returned by the ISR.

The current node can be designated by the special node identifier CK_CURRENT_NID.

The service routine is executed on behalf of a regular interrupt context, just as if it were triggered by an automatic event source. The VRTOS passes the current **vector** number and the **cookie** as the ISR's parameters.

RETURN VALUES

None.

SEE ALSO

ckMaskIntr(), ckGetIntrLevel(), ckGetContext(), ckControlIntr()

EXAMPLE

```
int theIsr (ckvector_t vector, void *cookie)
{
    /* This is a level #1 interrupt on vector #0 */
    return CK_ISR_HANDLED;
}
...
{
    /* Call theIsr() on behalf of an interrupt context */
    ckRaiseIntr(CK_CURRENT_NID,0,1,theIsr,NULL);

    ...

    /* ckRaiseIntr() behaves identically to: */
    ckhandle_t handle;
    ckCreateIntr("MyIntr",0,1,theIsr,NULL,NULL,&handle);
    ckControlIntr(handle,CK_INTR_RAISE);
    ckDestroyIntr(handle);
}
```

ckControlIntr - apply command to an interrupt

SYNOPSIS

```
#include <ck/intr.h>

int ckControlIntr(ckhandle_t handle, ckintrop_t cmd);
```

DESCRIPTION

This service applies a command to an interrupt object identified by **handle**, which may belong to any node. The command is specified by **cmd** which may take one of the following values:

CK_INTR_MASK individually masks the designated interrupt. The associated ISR will not be called until it is explicitly unmasked.

CK_INTR_UNMASK reverts the effect of an individual interrupt masking.

CK_INTR_RAISE schedules the designated interrupt for immediate activation.

CK_INTR_GET_LEVEL returns the level of the interrupt object.

CK_INTR_GET_VECTOR returns the vector of the interrupt object.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The ckControlIntr() service fails if:

ESRCH **handle** does not refer to a valid interrupt object.

EINVAL **cmd** is not a valid command word.

SEE ALSO

ckCreateIntr(), ckRaiseIntr()

EXAMPLE

```
{
    ckhandle_t handle;

    /* Fetch the handle of the "uartIntr" object, then
       raise this interrupt by hand */

    if (!ckGetObjectHandle(CK_CURRENT_NID,
                           CK_INTR_OBJECT,
                           "uartIntr",
                           &handle))
        ckControlIntr(handle, CK_INTR_RAISE);
}
```

ckProgramIntr - program an interrupt

SYNOPSIS

```
#include <ck/intr.h>

int ckProgramIntr(ckhandle_t handle, ckintrlaw_t law, const
char *param);
```

DESCRIPTION

This service attaches an event source to the interrupt object identified by **handle**. Each event generated by the source according to a given generation law will raise the associated interrupt automatically.

An event source **law** and its **parameter** must be passed to the routine. The **law** parameter may take one of the following values:

CK_INTR_PERIODICAL causes a periodical event source to be attached to the interrupt object. The parameter string describes the activation time frame and the

CK_INTR_EXPONENTIAL causes an exponential event source to be attached to the interrupt object. The parameter string describes the activation time frame and the mean.

CK_INTR_UNIFORM causes a uniform event source to be attached to the interrupt object. The parameter string describes the activation time frame and the numerical bounds.

CK_INTR_FILE causes a filed event source to be attached to the interrupt object. The parameter string gives the path of the file containing the event dates. The path can contain environment variables which will be expanded when the source is created.

CK_INTR_TIMER causes a one-shot timer event to be attached to the interrupt object. The parameter string gives the absolute event date.

CK_INTR_NULL detaches the current source from the interrupt object. If no event source was attached to the interrupt, this action has no effect.

Re-programming an interrupt object is a valid operation. The previous event source is simply discarded before the new one is attached to the interrupt object.

Interrupt generation support is built on top of CarbonKernel's event sources. This interface's programming syntax is identical to the one described for the ISE's programmed event sources.

The expected format of the parameter string depends on the generation law. The following formats are recognized:

| CK_INTR_PERIODICAL | <i>[tmin-tmax/]period</i> |
|---------------------|------------------------------|
| CK_INTR_EXPONENTIAL | <i>[tmin-tmax/]mean</i> |
| CK_INTR_UNIFORM | <i>[tmin-tmax/]dmin-dmax</i> |
| CK_INTR_FILE | <i>source file</i> |
| CK_INTR_TIMER | <i>absolute time</i> |
| CK_INTR_NULL | n/a |

Numerical laws:

Event sources controlled by numerical generation laws (e.g. periodical, exponential or uniform) have an activation time frame during which the events can be triggered. The source is automatically activated by the simulation kernel when the minimum time bound is reached, then shut after the maximum time bound. No events are generated outside this time frame.

If no activation time frame is given in the parameter string, an infinite time frame starting immediately is assumed. Otherwise, the time bounds can be specified using floating-point values, possibly suffixed with a time unit which may be **usc** (micro-seconds), **msec** (milli-seconds) or **sec** (seconds). A unit name can be shortened to its initial letter. If no unit is given, micro-seconds are assumed. If the time unit used for the maximum time bound is not specified, the same unit applied to the minimum time bound is assumed.

Bounds are joined by a dash character (-), and separated from other information using a slash, a colon or a comma. The following special cases are supported:

| | |
|-----------|---------------------------|
| tmin/... | <i>tmin-infinite</i> /... |
| -tmax/... | <i>0-tmax</i> /... |
| tmin-/... | <i>tmin-infinite</i> /... |
| -/... | <i>0-infinite</i> /... |

File law:

A file source reads the contents of a text file to get the time-table it should follow to generate the events appropriately. Each line should be whether an event date, a comment, or an empty line. Lines which cannot be parsed are ignored.

The very first line of the file *must* start with the following special marker, identifying a time log file: « # \$@**timelog** ».

A line starting with a pound sign is a comment. Empty lines are silently ignored.

Other non-empty lines are processed according to the same rules as a time bound of a numerical law. Event dates must be specified in ascending order, preposterous time values will be ignored.

Timer specification:

A timer source parameter should specify a single event date. Its syntax is identical to a time bound of a numerical law.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The `ckProgramIntr()` service fails if:

ESRCH **handle** does not refer to a valid interrupt object.

EINVAL **law** is not a valid generation law.

EINVAL an error occurred while parsing **param**.

SEE ALSO

ckControlIntr(), ckRaiseIntr()

EXAMPLE

```
int diehardInterruptHandler (ckvector_t vector, void *cookie)
{
    /* we should probably do some nasty things here */
    return CK_ISR_HANDLED;
}
...
{
    /* Build our diehard interrupt */
    ckhandle_t handle;
    ckCreateIntr("diehardIntr",0,5,diehardInterrupt,NULL,NULL,&handle);
    /* Plan for this interrupt to shake our code a bit ;o) */
    ckProgramIntr(handle,CK_INTR_EXPONENTIAL,"10s-35s,55u");
}
```

ckMaskIntr - mask interrupts

SYNOPSIS

```
#include <ck/intr.h>

cklevel_t ckMaskIntr(cklevel_t level)
```

DESCRIPTION

ckMaskIntr() masks all interrupts whose priorities are lower or equal to **level**.

CK_MASK_ALL can be passed to lock out all interrupts. If 0 is passed, interrupts global masking will be cleared.

It should be noted that globally unmasking an interrupt level does not revalidate interrupts individually masked using the ckControlIntr(CK_INTR_MASK) command.

RETURN VALUES

This service returns the previous masking level. 0 means "not masked".

SEE ALSO

ckRaiseIntr()

EXAMPLE

```
{
    cklevel_t oldLevel = ckMaskIntr(CK_MASK_ALL);
    /* This section is uninterruptible */
    ...
    /* Exit the section */
    ckMaskIntr(oldLevel);
}
```

ckGetMemPortByte, ckGetMemPortWord, ckGetMemPortDword, ckGetMemPortQword - create memory ports

SYNOPSIS

```
#include <ck/memport.h>

caddr_t ckGetMemPortByte(int mode, ckmphandler_t *handler,
void *cookie);

caddr_t ckGetMemPortWord(int mode, ckmphandler_t *handler,
void *cookie);

caddr_t ckGetMemPortDword(int mode, ckmphandler_t *handler,
void *cookie);

caddr_t ckGetMemPortQword(int mode, ckmphandler_t *handler,
void *cookie);
```

DESCRIPTION

These services return chunks of memory with special access protection causing a user-defined handler to be called each time their contents are read and/or written. The length of the memory area is respectively 1, 2, 4 or 8 bytes depending on whether `ckGetMemPortByte()`, `ckGetMemPortWord()`, `ckGetMemPortDword()` or `ckGetMemPortQword()` is invoked.

The access mode triggering the **handler** invocation is defined by the parameter **mode**, which may take the following values:

`CK_MEMPORT_WRITE` causes the handler to be called upon write access.

`CK_MEMPORT_READ` causes the handler to be called upon read/write access.

The handler is always executed on behalf of the simulation thread which has performed the monitored access. The current access mode, the chunk memory address and the opaque cookie are respectively passed as the first and last parameter to the handler. The access mode passed to the handler is forced to `CK_MEMPORT_READ` by the current implementation.

RETURN VALUES

The address of a memory chunk with special access protection is returned to the caller.

EXAMPLE

```
u_long *memMap;

#define MEMMAP_SET_BIT0() (*memMap |= 1)
#define MEMMAP_SET_BIT1() (*memMap |= 2)
#define MEMMAP_SET_BIT2() (*memMap |= 4)

void portWriteHandler(int mode, caddr_t addr, void *cookie)
{
    const char *name = (const char *)cookie; /* i.e. "psr" */
    ckWarning("memory port %s accessed -- start addr = %p", name, addr);
}

{
```



```
/* Get a 32bits memory port */
memMap = (u_long *)
ckGetMemPortDword(CK_MEMPORT_WRITE,portWriteHandler,"psr");
...
/* Set bit #2 from memory map */
MEMMAP_SET_BIT2();
/* ...portWriteHandler() should be running now... */
}
```

ckOpenTerminal - open a new virtual console

SYNOPSIS

```
#include <ck/terminal.h>

int ckOpenTerminal(const char *title, ckhandle_t *handlep);
```

DESCRIPTION

This service starts a new virtual terminal for the current node. This terminal shall be used to send and/or receive characters through the appropriate services. A CarbonKernel virtual terminal is an external GUI application connected to the simulator through a TCP/IP channel. The maximum numbers of active terminals is virtually unlimited.

The **title** parameter is passed to the window manager as the new window's title. If NULL is passed, a default title string will be picked.

The handle of the new terminal is returned at **handlep**.

CarbonKernel starts an initial virtual console for each emerging node, unless it has been told not to do so (i.e. -Xh startup flag). Whenever this console is closed for the current node, the next virtual terminal created by this service will be considered as this node's new console.

RETURN VALUES

Zero is returned on success, and **handlep** points to the new terminal handle. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The ckOpenTerminal() service fails if:

ENXIO The simulation kernel has failed to spawn and/or connect to the virtual terminal application.

SEE ALSO

ckCloseTerminal(), ckPutString(), ckGetChar(), ckPutChar(), ckWaitChar().

ckCloseTerminal - close a virtual terminal

SYNOPSIS

```
#include <ck/terminal.h>

int ckCloseTerminal(ckhandle_t handle);
```

DESCRIPTION

This service closes the connection with a virtual terminal, making it exit gracefully. **handle** refers to a terminal instance previously opened by the `ckOpenTerminal()` service, or to the current node's virtual console if `CK_NODE_CONSOLE` is passed.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The `ckCloseTerminal()` service fails if:

ESRCH **handle** does not refer to an active terminal.

SEE ALSO

`ckOpenTerminal()`, `ckPutString()`, `ckGetChar()`, `ckPutChar()`, `ckWaitChar()`.

ckGetChar, ckWaitChar - read from a virtual

SYNOPSIS

```
#include <ck/terminal.h>

int ckGetChar(ckhandle_t handle, ckticks_t timeout);
int ckWaitChar(ckhandle_t handle, int c, ckticks_t timeout);
```

DESCRIPTION

ckGetChar() blocks the calling RTOS thread until a character is available from the virtual terminal identified by **handle**. ckWaitChar() waits until the specific character **c** is received. The call is directed to the current node's virtual console if **handle** equals to CK_NODE_CONSOLE.

A watchdog can be set using the **timeout** parameter to unblock the calling RTOS thread if the character is not received within the allotted amount of time. Its value is a number of ticks to wait before the request is aborted. The tick value is defined by the running node. If **timeout** is zero, the calling RTOS thread waits indefinitely.

RETURN VALUES

The received ASCII keycode is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

These services fail if:

| | |
|-----------|------------------------------------------------------------|
| ESRCH | handle does not refer to an active terminal. |
| EPIPE | the connection was lost with the terminal during the wait. |
| ETIMEDOUT | the timeout has elapsed. |

SEE ALSO

ckOpenTerminal(), ckCloseTerminal(), ckPutString(), ckPutChar().

ckPutChar, ckPutString, ckPutFormat - write to a virtual terminal

SYNOPSIS

```
#include <ck/terminal.h>

int ckPutChar(ckhandle_t handle, int c);
int ckPutString(ckhandle_t handle, const char *s, int n);
int ckPutFormat(ckhandle_t handle, const char *format, ...);
```

DESCRIPTION

ckPutChar() writes the character **c** to the virtual terminal identified by **handle**. ckPutString() emits **n** characters from the string starting at **s**. If **n** is negative, a null (unwritten) character is expected to end the string. ckPutFormat() writes a formatted representation of its arguments according to the **format** string which should be conformant with the printf(3) specifications.

The call is directed to the current node's virtual console if **handle** equals to CK_NODE_CONSOLE. If this console is closed, the output is simply discarded unless the --Xc flag has been passed to the simulator, in which case the output goes to the standard output stream.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The services fail if:

ESRCH **handle** does not refer to an active terminal.

SEE ALSO

ckOpenTerminal(), ckCloseTerminal(), ckPutChar(), ckWaitChar().

ckGetNid - get current node's identifier

SYNOPSIS

```
#include <ck/context.h>
cknid_t ckGetNid(void);
```

DESCRIPTION

This service returns the global identifier of the running node. A -fixed- distinct identifier is assigned by the simulation kernel to each emerging node at startup. The 0-based numerical value of this identifier depends on the initialization order of the nodes. The first node declared in the initialization list for the architecture gets id #0, the second gets id #1 and so on. Cloned nodes take consecutive identifiers after the original instance.

RETURN VALUES

The current node's identifier is returned to the caller.

ERRORS

None.

ckGetStdStream - get original STDIO streams

SYNOPSIS

```
#include <ck/context.h>

FILE *ckGetStdStream(int fildes);
```

DESCRIPTION

This service returns a copy of the original STDIO streams passed to the simulator by the host environment as they were set before they are redirected to the virtual console(s) during the initialization process. **fildes** should be used to specify which stream is requested among 0 (stdin), 1 (stdout) or 2 (stderr).

CK_STDIN, CK_STDOUT and CK_STDERR are macros respectively defined as ckGetStdStream(0), ckGetStdStream(1), et ckGetStdStream(2).

RETURN VALUES

A pointer to a copy of the original STDIO stream is returned on success. Otherwise, NULL is returned and *errno* is set to indicate the error.

ERRORS

This service fails if:

EINVAL **fildes** is not in the range [0-2] (inclusive).

EXAMPLE

```
{
    FILE *out = ckGetStdStream(1);
    fprintf(out, "Writing to the shell's output stream...\n");
    /* Same as */
    fprintf(CK_STDOUT, "Writing to the shell's output stream...\n");
}
```

ckGetErrnoAddr - **get per-thread errno variable**

SYNOPSIS

```
#include <ck/context.h>
int *ckGetErrnoAddr();
```

DESCRIPTION

This service returns the address of the `errno` variable for the current simulation thread.

The CKPI header files define `errno` as a macro whose value is `* ckGetErrnoAddr()`.

RETURN VALUES

A pointer to the `errno` variable for the current simulation thread.

ERRORS

None.

ckGetArgs - get user-defined argument vector

SYNOPSIS

```
#include <ck/context.h>

const char *const *ckGetArgs(int *argcp);
```

DESCRIPTION

This service returns a vector built from the user-defined arguments passed to the simulator through the **-Q** prefix option. The integer at **argcp** will be overwritten by the number of elements stored into the vector on return. The information obtained from this service can be parsed using the standard *getopt(3)* routine.

The **-Q** option allows user-defined arguments to be passed to the application code through the standard option parsing engine implemented by the simulation kernel. The argument to the **-Q** option is identified as a user-defined option, opaque to the simulator. This is the best way to prevent polluting the CarbonKernel's standard option namespace and to avoid raising conflicts with next versions of the simulation kernel.

RETURN VALUES

The address of the user-defined argument vector is returned by value. **argcp** is overwritten by the number of elements stored into the vector.

ERRORS

None.

EXAMPLE

```
#include <stdio.h>
#include <getopt.h>

void parseLocalOptions ()
{
    const char *const *argv;
    int argc, c;

    /* First of all, grab local options passed on the
       command line using the -Q prefix... */

    argv = ckGetArgs(&argc);

    /* Then, parse them using getopt(3)... */

    while ((c = getopt(argc,argv,"d:t")) != EOF)
    {
        switch (c)
        {
            case 'd': /* Write debug to log file */

                debugMode = 1;
                debugFileName = optarg;
                break;
        }
    }
}
```

```
        case 't': /* Toggle test mode */  
            testMode = 1;  
            break;  
            ...  
        }  
    }  
}
```

ckGetIntrLevel - get current interrupt level**SYNOPSIS**

```
#include <ck/context.h>
cklevel_t ckGetIntrLevel()
```

DESCRIPTION

This service is useful for interrupt service routines to obtain the current interrupt level.

RETURN VALUES

The current interrupt level is returned. A value of zero means the current simulation thread is a RTOS thread, a DSR, or a callout (i.e. the caller is running outside any interrupt service routine).

ERRORS

None.

SEE ALSO

ckRaiseIntr(), ckGetContext()

ckGetContext - get current execution context

SYNOPSIS

```
#include <ck/context.h>

ckcontext_t ckGetContext(ckhandle_t *handlep)
```

DESCRIPTION

This service returns the type and identification of the current simulation thread to the caller. If **handlep** is non-null, the CarbonKernel's handle to the thread is written to this address.

RETURN VALUES

The type of the current simulation thread is returned by value. A simulation thread always belong to one of the following types:

CK_INIT_CONTEXT is returned if the caller is running on behalf of a node's warm initialization context, standing for the simulated RTOS' start-up phase. At this point, the scheduling has not started yet.

CK_THREAD_CONTEXT is returned if the caller is running on behalf of a RTOS thread (i.e. A real-time thread as defined by the RTOS model). The CarbonKernel's handle to the thread is written to **handlep**.

CK_ISR_CONTEXT is returned if the caller is running on behalf of an interrupt service routine. The associated interrupt object handle is written to **handlep**. ckGetIntrLevel() can be used to obtain the current interrupt level.

CK_DSR_CONTEXT is returned if the caller is running on behalf of a deferred service routine. The associated interrupt object handle is written to **handlep**.

CK_CALLOUT_CONTEXT is returned if the caller is running on behalf of a callout procedure. The information written to **handlep** remains private to the simulation kernel.

CK_ASR_CONTEXT is returned if the caller is running on behalf of an asynchronous service routine. The information written to **handlep** remains private to the simulation kernel.

It should be noted that some RTOS models may have no support for DSRs, callout procedures or asynchronous service routines.

ERRORS

None.

SEE ALSO

ckRaiseIntr(), ckGetIntrLevel()

EXAMPLE

```
{
    ckhandle_t handle;

    switch (ckGetContext(&handle))
```

```

{
case CK_INIT_CONTEXT:

    printf("Initialization - scheduling not started yet.\n");
    break;

case CK_THREAD_CONTEXT:

    printf("On behalf of thread %s.\n",
           ckGetObjectName(handle));
    break;

case CK_CALLOUT_CONTEXT:

    printf("In RTOS callout handler.\n");
    break;

case CK_ISR_CONTEXT:

    printf("In ISR %s, level=%d\n",
           ckGetObjectName(handle),
           ckGetIntrLevel());
    break;

case CK_DSR_CONTEXT:

    printf("In DSR %s\n",
           ckGetObjectName(handle));
    break;

case CK_ASR_CONTEXT:

    printf("In asynchronous service routine.\n");
    break;
}
}

```

ckDecodeTimeBounds - parse a time specification string

SYNOPSIS

```
#include <ck/context.h>

int ckDecodeTimeBounds(const char *s, ckclock_t range[2],
    cktimeval_t params[2])
```

DESCRIPTION

This service parses the null-terminated string *s* as a time specification conforming to the event source programming format.

The decoded time bounds are respectively written to **range**[0] and range[1]. Up to two additional parameters can be evaluated and written to **params**[0] and params[1].

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The ckDecodeTimeBounds() service fails if:

EINVAL a syntax error was found while parsing the string.

EXAMPLE

```
{
    ckclock_t range[2];
    cktimeval_t params[2];

    ckDecodeTimeBounds("137.68u-155m/10u",range,params);
    printf("Time frame: %.2f usec => %.2f usec, step by %.2f usec\n",
        range[0],range[1],params[0]);

    ckDecodeTimeBounds("-1s,10 msc - 15 msc",range,params);
    printf("Time frame: 0 => %.2f usec, window [%.2f,%.2f] usecs\n",
        range[1],params[0],params[1]);
}
```

ckGetObjectHandle - get a simulation object's handle

SYNOPSIS

```
#include <ck/context.h>

int ckGetObjectHandle(cknid_t nid, ckobjtype_t type, const
char *name, ckhandle_t *handlep);
```

DESCRIPTION

Simulation objects which can be accessed from the CKPI are uniquely identified by a 32bits handle. This kind of identifier is global to the simulation system and can be used each time an object's handle is required by a simulation service. An object may also have an external string-based identifier, which can be set by configuration (i.e. using the ISE's librarian) when instantiating some simulation objects, or programmatically (i.e. by using the **name** parameter of object creation services from the CKPI).

ckGetObjectHandle() attempts to find a simulation object and returns its handle to the caller. The object is searched into the internal tables of the node whose identifier is specified by **nid**. A value of CK_CURRENT_NID makes the current node's internal tables searched for the object. **name** specifies the external name of the object, as given by the user. **type** tells which kind of simulation object is searched for, among the following choices:

CK_NODE_OBJECT causes this service to search for a node. If **name** is NULL, the handle of the node whose identifier is **nid** is returned. If **name** is a valid character string and **nid** refers to the current node, the handle of the node from the simulation network whose name exactly matches the argument is returned.

CK_THREAD_OBJECT causes this service to search for a real-time thread.

CK_INTR_OBJECT causes this service to search for an interrupt object.

CK_MSGPORT_OBJECT causes this service to search for a message port.

CK_MUTEX_OBJECT causes this service to search for a mutex object obtained from a call to the ckCreateMutex() service.

CK_CV_OBJECT causes this service to search for a condition variable obtained from a call to the ckCreateCv() service.

CK_PANEL_OBJECT causes this service to search for a panel instance created using the ISE's librarian.

RETURN VALUES

A pointer to the first character of the object's name is returned on success. Otherwise, NULL is returned and *errno* is set to indicate the error.

ERRORS

This service fails if:

| | |
|--------|--------------------------------------------|
| EINVAL | nid is not a valid node identifier. |
| EINVAL | type is illegal. |
| ESRCH | the object cannot be found. |

SEE ALSO

ckGetObjectName(), ckGetObjectTag()

EXAMPLE

```
{
    ckhandle_t handle;

    if (!ckGetObjectHandle(CK_CURRENT_NID,
                           CK_INTR_OBJECT,
                           "uartInterruptObject",
                           &handle))
        /* Raise a simulated UART interrupt NOW! */
        ckControlIntr(handle, CK_INTR_RAISE);
}
```


ckGetObjectName - get a simulation object's name

SYNOPSIS

```
#include <ck/context.h>

const char *ckGetObjectName(ckhandle_t handle);
```

DESCRIPTION

Each simulation object whose **handle** is passed from the simulation kernel back to the user carries a name. Such name can be retrieved through this service as a null-terminated character string. The object may belong to any node.

It may occur that NULL is returned for a valid but unnamed object. In such a case, the *errno* variable is also cleared to distinguish this situation from an error condition.

RETURN VALUES

A pointer to the first character of the object's name is returned on success. Otherwise, NULL is returned and *errno* is set to indicate the error.

ERRORS

This service fails if:

ESRCH **handle** does not refer to an existing object.

SEE ALSO

ckGetObjectHandle(), ckGetObjectTag()

ckGetConf - get configuration parameter value

SYNOPSIS

```
#include <ck/context.h>

int ckGetConf(ckcfname_t name, ckcfval_t *u);
```

DESCRIPTION

This service provides a method for an application to determine the current value of a configuration parameter or option. For node-based parameters, the corresponding value indexed on **name** is searched in the configuration of the currently active node.

The returned value is written to the proper field of the union **u**, according the the value's data type.

name should be one of the following:

| | |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| CK_SIMULATOR_NODES | The number of instantiated nodes is written to u->ival . |
| CK_NODE_MAGIC | A magic cookie uniquely identifying the current node's type (usually common to every nodes simulating the same RTOS) is written to u->ival . |
| CK_NODE_CLOCKFREQ | The number of ticks per second as configured for the current node's real-time clock is written to u->fpval . |
| CK_NODE_WARP | The current node's warp factor's value is written to u->fpval . |
| CK_NODE_MAX_ILVL | The maximum interrupt level for the node is written to u->ival . |

RETURN VALUES

Zero is returned on success, and the proper field of the union is updated with the parameter's value. Otherwise, -1 is returned, no field of the union is changed, and *errno* is set to indicate the error.

ERRORS

This service fails if:

EINVAL the **name** argument is invalid.

ckSetObjectTag - tag a simulation object

SYNOPSIS

```
#include <ck/context.h>

int ckSetObjectTag(ckhandle_t handle, ckobjtag_t tag);
```

DESCRIPTION

Each simulation object whose **handle** is passed from the simulation kernel back to the user carries a tag. Such **tag** can be set for further retrieval through this service. Each new object is assigned a null (invalid) tag after creation.

The application can use this tag to group related simulation objects. The designated object may belong to any node.

RETURN VALUES

The new **tag** overrides the previous one in the object's storage, and zero is returned on success. Otherwise, no change is performed, -1 is returned and *errno* is set to indicate the error.

ERRORS

This service fails if:

| | |
|--------|-------------------------------------------------------------|
| ESRCH | handle does not refer to an existing object. |
| EINVAL | tag is null, which stands for the invalid tag value. |

SEE ALSO

ckGetObjectTag(), ckGetObjectName()

EXAMPLE

```
void tagCurrentThread (ckobjtag_t tag)
{
    ckhandle_t handle;

    if (ckGetContext(&handle) != CK_THREAD_CONTEXT)
        oops("not running on behalf of a real-time thread ?!");

    ckSetObjectTag(handle, tag);
}

...
#define DAEMON_THREAD_MAGIC    0xfebbc045
#define JOINABLE_THREAD_MAGIC 0xcbc744b2

{
    tagCurrentThread(DAEMON_THREAD_MAGIC);
}
```

ckGetObjectTag - retrieve a simulation object's tag

SYNOPSIS

```
#include <ck/context.h>
ckobjtag_t ckGetObjectTag(ckhandle_t handle);
```

DESCRIPTION

This service returns the tag currently assigned to the simulation object identified by **handle**. The designated object may belong to any node.

RETURN VALUES

A non-null tag value is returned if the object exists and was previously tagged. Otherwise, zero is returned. If the object does not exist, *errno* is set to indicate the error. If the object exists but with no tag assigned yet, *errno* is cleared.

ERRORS

This service fails if:

ESRCH **handle** does not refer to an existing object.

SEE ALSO

ckSetObjectTag()

ckInitSpinLock, ckGetSpinLock, ckTrySpinLock,
ckRelSpinLock - manage spin locks

SYNOPSIS

```
#include <ck/ksynch.h>

void ckInitSpinLock(rtspinlock_t *spinlock);
void ckGetSpinLock(rtspinlock_t *spinlock);
int ckTrySpinLock(rtspinlock_t *spinlock);
void ckRelSpinLock(rtspinlock_t *spinlock);
```

DESCRIPTION

Spin locks are basic synchronization objects preventing multiple threads from simultaneously executing critical sections of code which access shared data. Spin locks can be used to synchronize threads from different nodes. A thread waiting for a spin lock to become available busy waits” for the current owner to release it. The thread owning a spin lock cannot be preempted by any other thread from any node, until it releases the lock. This object should be used to enforce serialized execution of threads over short critical sections. A spin lock must be initialized before it is used.

Initialization is performed by the `ckInitSpinLock()` routine. A pointer to an abstract data type `ckspinlock_t` must be passed to this routine. This data address will further identify the spin lock to other related routines. Be sure to declare the spin lock in *netshared* memory if you plan to enforce inter-node synchronization with it. You can initialize a spin lock inline, by affecting the special value `CK_SPINLOCK_INITVAL` to it.

Locking is performed by the `ckGetSpinLock()` routine. The calling context must be a synchronous thread (i.e. not an ISR, DSR or callout), otherwise a fatal error is raised. On successful return, the caller owns the lock and may safely enter the critical section. The rescheduling procedure is momentarily locked on the current node until the spin lock is released. `CkTrySpinLock()` is similar to `ckGetSpinLock()`, except that if the spin lock is currently owned by another thread, the call returns immediately with an error, instead of blocking the calling thread.

Unlocking is performed by the `ckRelSpinLock()` routine. The owner of the spin lock calls this routine to release the lock when it exits the critical section. If there are threads waiting for the lock, the access is immediately granted to one of them. Otherwise, the rescheduling procedure is reactivated. The order in which multiple threads pend for a given spin lock is unpredictable.

RETURN VALUES

Most spin lock management routines have no return value, except `ckTrySpinLock()` which returns a zero status if the lock has been successfully granted to the caller, non-zero otherwise.

ERRORS

These routines may cause fatal simulation errors if the calling context is invalid. Unexpected results (usually memory corruption and/or exception) occur if the spin lock address is invalid.

SEE ALSO

mutexes, condition variables

EXAMPLE

```
/* We need inter-node synchronization, so the spin lock
   must be "netshared" */

netshared ckspinlock_t spinlock = CK_SPINLOCK_INITVAL;

void nodeExclusiveInit ()
{
    ckGetSpinLock(&spinlock);

    /* This section can't be traversed by more than one
       simulation thread from a single node at a time... */
    ...
    ckRelSpinLock(&spinlock);
}
```

ckCreateMutex, ckLockMutex, ckTryMutex, ckUnlockMutex, ckDestroyMutex - manage mutexes

SYNOPSIS

```
#include <ck/ksynch.h>

int ckCreateMutex(const char *name, ckhandle_t *handlep);
int ckLockMutex(ckhandle_t handle);
int ckTryMutex(ckhandle_t handle);
int ckUnlockMutex(ckhandle_t handle);
int ckDestroyMutex(ckhandle_t handle);
```

DESCRIPTION

Mutexes are synchronization objects preventing multiple threads from simultaneously executing critical sections of code accessing shared data. Unlike spin locks, mutexes are named objects that are local to the current node. A thread waiting for a mutex to become available is blocked until the current owner releases it. Because this object has ownership, only the thread which acquired a mutex may release it. The simulation kernel implements the *priority inheritance protocol* for CKPI mutexes to prevent thread priority inversion. A mutex must be initialized before it can be used.

Creation is performed by the ckCreateMutex() routine. A symbolic **name** can be given to the new mutex; on success, its handle is written to the memory pointed to by **handlep**. This handle will further identify the mutex to other related routines.

Locking is performed by the ckLockMutex() routine. The calling context must be a synchronous thread (i.e. not an ISR, DSR or callout), otherwise a fatal error is raised. On successful return, the caller owns the mutex and may safely enter the critical section. ckTryMutex () is similar to ckLockMutex(), except that if the mutex is currently owned by another thread, the call immediately returns with an error, instead of blocking the calling thread.

Unlocking is performed by the ckUnlockMutex() routine. The owner of the mutex should call this routine to release the lock when it exits the critical section. If there are threads waiting for the lock, the access is immediately granted to the one having the highest priority.

Deletion is obtained by a call to the ckDestroyMutex() routine. Threads waiting for the mutex when it is destroyed are unblocked and ckLockMutex() returns with an error code. After this call, the mutex no longer exists in the simulation kernel.

RETURN VALUES

Mutex management routines return 0 on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

These routines may cause fatal simulation errors if the calling context is invalid, or an attempt by a thread to re-lock a mutex it currently owns is made.

ckLockMutex(), ckTryMutex(), ckUnlockMutex() and ckDestroyMutex() fail if:

ESRCH the **handle** argument does not identify an existing mutex on the current node.

ckLockMutex() also fails if:

EIDRM the mutex was destroyed by another thread while the caller was pending on it.

ckTryMutex() also fails if:

EBUSY the mutex is currently owned by another thread.

ckUnlockMutex() also fails if:

EINVAL the mutex was not owned by the calling thread.

SEE ALSO

spin locks , condition variables

EXAMPLE

```
void initSched (struct scheduler *sched)
{
    ckCreateMutex("schedMutex",&sched->mutex);
    initSlot(&sched->sharedSlot);
    initSlot(&sched->privateSlot);
}

void synchSched (struct scheduler *sched)
{
    ckLockMutex(&sched->mutex);

    /* This section can't be traversed by more than a
       single thread at a time... */

    synchSlot(&sched->sharedSlot,&sched->privateSlot);

    ckUnlockMutex(&sched->mutex);
}
```


ckCreateCv, ckWaitCv, ckSignalCv, ckBroadcastCv, ckDestroyCv - manage condition variables

SYNOPSIS

```
#include <ck/ksynch.h>

int ckCreateCv(const char *name, ckhandle_t *cvhp);

int ckWaitCv(ckhandle_t cvh, ckhandle_t mutexh, ckticks_t
timeout);

int ckSignalCv(ckhandle_t cvh);

int ckBroadcastCv(ckhandle_t cvh);

int ckDestroyCv(ckhandle_t cvh);
```

DESCRIPTION

Condition variables are thread synchronization objects. Like mutexes, they are named objects that are local to the current node. A condition variable is used in association with a mutex which ensures that a condition can be checked atomically and the thread can wait for such condition without missing either a change or a signal that a change occurred. A condition variable must be initialized before it is used.

Creation is performed by the `ckCreateCv()` routine. A symbolic **name** can be given to the new condition variable; on success, its handle is written to the memory pointed to by **cvhp**. This handle will further identify the condition variable to other related routines.

Waiting for the condition is achieved by the `ckWaitCv()` routine. The calling context must be a synchronous thread (i.e. not an ISR, DSR or callout), otherwise a fatal error is raised. The mutex must be acquired by the caller before `ckWaitCv()` is entered. This mutex is released and the thread is put to sleep atomically, ensuring that no other thread can signal the condition until the caller is blocked. The mutex is re-locked by the routine before it returns to its caller. A watchdog can be set to limit the time the thread is allowed to wait for the condition using the **timeout** parameter. This value is a number of clock ticks elapsed on the current node. A value of `CK_INFINITE` for **timeout** allows the thread to wait indefinitely. A value of `CK_NONBLOCK` makes the call return with an error status immediately.

Signaling the condition variable is performed using the `ckSignalCv()` routine. This routine unblocks one thread. All threads blocked on a condition variable can be unblocked by calling `ckBroadcastCv()` on this object. The mutex must be acquired by the caller before `ckSignalCv()` or `ckBroadcastCv()` are issued.

Deletion is obtained by a call to the `ckDestroyCv()` routine. Threads waiting for the condition variable when it is destroyed are unblocked and `ckWaitCv()` returns with an error code. After this call, the condition variable no longer exists in the simulation kernel.

RETURN VALUES

Condition variable management routines return 0 on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

These routines may cause fatal simulation errors if the calling context is invalid.

`ckWaitCv()`, `ckSignalCv()`, `ckBroadcastCv()` and `ckDestroyCv()` fail if:

ESRCH the **cvh** argument does not identify an existing condition variable on the current node.

`ckWaitCv()` also fails if:

ESRCH the **mutexh** argument does not identify an existing mutex on the current node.

EINVAL the mutex referred to by **mutexh** was not owned by the calling thread.

EIDRM the condition variable or the mutex was destroyed by another thread while the caller was waiting for the condition to be signalled.

ETIMEDOUT the condition was not signalled within the allotted amount of time.

EAGAIN **timeout** was given the `CK_NONBLOCK` value.

SEE ALSO

spin locks , mutexes

EXAMPLE

```
void initSched (struct scheduler *sched)
{
    ckCreateMutex("schedMutex",&sched->mutex);
    ckCreateCv("schedCv",&sched->cv);
    sched->idle = 0;
}

void idleSched (struct scheduler *sched)
{
    ckLockMutex(&sched->mutex);
    sched->idle = 1;
    /* Allow all waiters to receive this signal */
    ckCvBroadcast(&sched->cv);
    ckUnlockMutex(&sched->mutex);
}

void waitSched (struct scheduler *sched, ckticks_t timeout)
{
    int rc = 0;

    ckLockMutex(&sched->mutex);

    while (!sched->idle && !rc)
        /* Wait for somebody to call idleSched() */
        rc = ckCvWait(&sched->cv,&sched->mutex,timeout);

    if (rc < 0)
    {
        if (errno == ETIMEDOUT)
        {
            /* Timed out */
        }
    }
}
```

```
    }  
    else  
        oops("failed waiting for IDLE event ?!");  
    }  
  
    /* Not idle anymore until we call idleSched() */  
    sched->idle = 0;  
  
    ckUnlockMutex(&sched->mutex);  
}
```

ckBindPort, ckReadPort, ckWritePort, ckSelectPort - manage message ports

SYNOPSIS

```
#include <ck/msgport.h>

int ckBindPort(const char *name, int flags, ckhandle_t
*handlep);

int ckSelectPort(ckhandle_t *set, int nsel, ckticks_t timeout,
ckhandle_t *handlep);

ckWritePort(ckhandle_t handle, expr);

ckReadPort(ckhandle_t handle, expr, ckticks_t timeout,
ckclock_t *stime);
```

DESCRIPTION

Message ports are access points to the internal event bus managed by the simulation kernel. The application can read and/or write to message ports (after they are bound) in order to pass events and exchange bulks of unformatted binary data between simulation threads.

Message ports are named using an arbitrary long C string identifier. A logical broadcast communication path is established between all ports bound to the same port identifier. When data is written to a port, it can be received by listeners of any other ports sharing the same identifier than the sender's.

Services allowing to read/write from/to message ports need additional information which is automatically generated by the application instrumenter (e.g. **ckcc** for C/C++). This means that *portions of code accessing message ports must be processed by the CarbonKernel instrumenter*.

Because this service is instrumenter-aided, you do neither need to pass the address nor the size of the data to send or receive: you just *pass a variable expression whose address must be computable*; this means that you cannot emit literal constants on a message port, but rather a variable which has been initialized with the constant's value. Any C/C++ data type can be passed, including aggregate types, such as structures, unions or arrays. The instrumenter will determine the source data type and its size, and collect the necessary information to give to the simulation kernel when reading/writing the port.

Binding is performed by the ckBindPort() routine. The **name** argument uniquely identifies the broadcast communication path. If **flags** equals to CK_MSGPORT_SHARED, the communication path will span over node boundaries, allowing data to be exchanged between threads from different nodes. Otherwise, 0 should be passed. On success, the port handle is written to the memory pointed to by **handlep**. This handle will further identify the bound message port to other related routines.

Writing data is achieved by inserting ckWritePort() statements in the application code. ckWritePort() is a macro-definition which expands appropriately to emit additional information concerning the data to be sent when processed by the instrumenter. This macro has 2 parameters : the port's handle to write to, and the variable **expression** which should be

emitted through the port. The submitted expression is always sent *by value*, not by address.

Reading data is achieved by inserting `ckReadPort()` statements in the application code. `ckReadPort()` is a macro-definition which expands appropriately to emit additional information concerning the data to be received when processed by the instrumenter. This macro has four parameters : the port's handle to listen to, the variable **expression** which should receive the data, a **timeout** parameter, and a pointer to a timestamp variable. The **timeout** parameter can be set to limit the time the thread is allowed to wait for input data. Its value is a number of clock ticks elapsed on the current node. A value of `CK_INFINITE` for **timeout** allows the thread to wait indefinitely. A value of `CK_NONBLOCK` makes the statement return with an error status if no data is immediately available. The **stime** parameter is a pointer to a variable which will receive the exact time the message was sent. **stime** can be `NULL` if this information is of no interest to the caller. The calling context must be a synchronous thread (i.e. not an ISR, DSR or callout) unless **timeout** equals to `CK_NONBLOCK`, otherwise a fatal error is raised.

Multiplexing input from multiple message ports can be done by calling the `ckSelectPort()` routine. This routine blocks the calling thread until an incoming message arrives on a message port which is a member of the input set. **set** is the start address of an array of **nsel** message port handles which should be monitored for input. A **timeout** parameter can be configured to prevent the calling thread from blocking indefinitely. Its value is a number of clock ticks elapsed on the current node. A value of `CK_INFINITE` for **timeout** allows the thread to wait indefinitely. A value of `CK_NONBLOCK` makes the call return with an error status if no data is immediately available on any port from the input set. On success, the handle of the port having available input is copied to the memory pointed to by **handlep**. Like `ckReadPort()`, the calling context must be a synchronous thread.

RETURN VALUES

Almost all message port management routines return zero on success, except `ckWritePort()` which returns the number of identified recipients of the message it sent. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

These routines may cause fatal simulation errors if the calling context is invalid.

`ckSelectPort()`, `ckWritePort()` and `ckReadPort()` fail if:

ESRCH the **handle** argument (or one of the handles from the input set for `ckSelectPort()`) does not identify an existing message port on the current node. Message ports bound using the `CK_MSGPORT_SHARED` flag are known from all nodes.

EINVAL the **expression** is `NULL`.

`ckReadPort()` and `ckSelectPort()` also fail if:

ETIMEDOUT no data was available for input within the allotted amount of time.

SEE ALSO

dataports, magnets

EXAMPLE

```
dataport int counter;

void logUpdates (FILE *logfp)
{
    ckclock_t updateTime;
    ckhandle_t handle;
    int _counter;

    /* A dataport is a message port receiving variable updates
       (the preceeding '/' means "dataport counter variable") */
    ckBindPort("/counter",0,&handle);

    for (;;)
    {
        /* Each time "counter" is updated, we will unblock from
           ckReadPort() with "_counter" holding the new value */
        ckReadPort(handle,_counter,CK_INFINITE,&updateTime);
        fprintf(logfp,"at %.3f: counter value = %d\n",updateTime,_counter);
    }
}
```

ckSetHostSpeed - set host simulation speed

SYNOPSIS

```
#include <ck/scontrol.h>

int ckSetHostSpeed(unsigned speed)
```

DESCRIPTION

This service sets the host simulation speed, the same way the speed selector from the ISE does. Lower values slow down the simulation process.

A value of 0 is equivalent to calling the `ckSuspendSimulation()` service which suspends the simulation, unless the application is not bound to the ISE.

The maximum value of 10 tells the simulator to run at full speed. This is the initial default value.

COMMENTS

The host-related simulation speed should not be confused with the *Target Warp* factor which controls the perceived velocity of a given simulated node.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL **speed** is not in the range [0-10] inclusive.

SEE ALSO

`ckSuspendSimulation()`

ckSetTargetWarp - set current node velocity

SYNOPSIS

```
#include <ck/scontrol.h>

int ckSetTargetWarp(double warp)
```

DESCRIPTION

The *Target Warp* factor is a floating-point value ranging from 0 to 10, whose setting affects the perceived processing speed of a simulated node, independently of the other nodes'. The higher the *Target Warp* factor, the shorter the time quantum charged per source statement executed on behalf on the running node. The time quantum is computed in microseconds from the *Target Warp* factor by the following formula: **1 / exp(factor)**. For instance, a node configured with a *Warp* factor of 3.7 will be charged for $1 / \exp(3.7) = 0.02472$, that is to say 24.72 (simulated) nanoseconds for each source statement.

In other words, raising a node's *Warp* value causes its virtual processor to go faster. Conversely, lowering this value slows down the current node's processing speed.

COMMENTS

The *Target Warp* factor should not be confused with the host simulation speed which controls the actual simulation process speed from the host system's standpoint.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL **warp** is not in the range [0-10] inclusive.

ckSpawn - spawn an external command

SYNOPSIS

```
#include <ck/scontrol.h>

int ckSpawn(const char *cmd, const char *argv[])
```

DESCRIPTION

This service executes the external command **cmd** with arguments **argv**. **Argv** is an argument vector which must be null-terminated. **CkSpawn()** does not wait for the command to exit before returning to the caller.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

| | |
|---------------|-----------------------------------------------------------------------------------|
| EACCES may | cmd cannot be started. The command path or the argument vector be invalid. |
|---------------|-----------------------------------------------------------------------------------|

ckSetHandler - set simulation event handler

SYNOPSIS

```
#include <ck/scontrol.h>
```

```
ckhandler_t    ckSetHandler(ckevtype_t    event,    ckhandler_t
handler)
```

DESCRIPTION

This service allows the application code to get informed of significant events occurring in the simulation system. The **handler** routine will be called each time an **event** of the designated kind occurs. Passing a null **handler** removes any current handling of the specified event.

The **handler** is passed a pointer to a *ckeventinfo_t* struct. This aggregate contains the event code, and a union which contents gives additional information on the current event. This struct has the following layout:

```
typedef struct _ckeventinfo {
    ckevtype_t event;

    union {
        struct {
            ckhandle_t target;
            ckhandle_t originator;
        } tasking;
    } u;
} ckeventinfo_t;
```

The **event** parameter can take one of the following values:

CK_TCREATE_EVENT is called whenever a new thread is created. The *tasking* part of the union is updated, with *tasking.target* being the handle of the new thread, and *tasking.originator* being the handle of its creator.

CK_TSWITCH_EVENT is called whenever a thread switch occurs. The *tasking* part of the union is updated, with *tasking.target* being the handle of the incoming thread, and *tasking.originator* being the handle of the outgoing one.

CK_TDELETE_EVENT is called whenever a new thread is created. The *tasking* part of the union is updated, with *tasking.target* being the handle of the destroyed thread, and *tasking.originator* being the handle of its destructor.

Whenever the originator is the idle thread, a null handle is passed.

RETURN VALUES

The previous handler set for the specified event is returned on success, which may be NULL if no previous handler was installed. In the latter case, *errno* is also cleared. Otherwise, NULL is always returned and *errno* is set to indicate the error.

ERRORS

EINVAL **event** is illegal.

EXAMPLE

```
void threadSwitchHook (void *cookie)
{
    ckeventinfo_t *evinfo = (ckeventinfo_t *)cookie;

    printf("THREAD SWITCH: IN %s, OUT %s\n",
           ckGetObjectNames(evinfo->u.tasking.target),
           ckGetObjectNames(evinfo->u.tasking.originator));
}
...
{
    ckhandler_t oldHandler;
    /* Attach a handler on the "thread switch" internal event */
    oldHandler = ckSetHandler(CK_TSWITCH_EVENT,threadSwitchHook);
}
```

ckControlPanel - apply command to a panel object

SYNOPSIS

```
#include <ck/panel.h>

int ckControlPanel(ckhandle_t handle, ckpanelop_t cmd);
```

DESCRIPTION

This service applies a command to a panel object identified by **handle**, which may belong to any node. The appropriate handle should be retrieved by the `ckGetObjectHandle()` service. Some requests may be transparently routed to the ISE for the action to take place. The command is specified by **cmd** which may take one of the following values:

`CK_PANEL_OPEN` causes the designated panel display window to pop up. If the ISE is not active, or the panel is already displayed, this command has no effect.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The `ckControlPanel()` service fails if:

`ESRCH` **handle** does not refer to a valid panel object.

`EINVAL` **cmd** is not a valid command word.

SEE ALSO

`ckGetObjectHandle()`

EXAMPLE

```
{
    ckhandle_t handle;

    if (!ckGetObjectHandle(CK_CURRENT_NID,
                           CK_PANEL_OBJECT,
                           "lcdPanel",
                           &handle))
        /* Make the ISE display the LCD panel */
        ckControlPanel(handle, CK_PANEL_OPEN);
}
```

ckSetTestExpr – attach a test expression to a dataport

SYNOPSIS

```
#include <ck/dataport.h>

int ckSetTestExpr(caddr_t datap, const char *expr, ckhandler_t
handler, void *cookie);
```

DESCRIPTION

This service ensures a **handler** is called whenever an assertion is true after a *dataport* variable's value has changed. The handler is passed the opaque **cookie** as its single argument. **datap** is the pointer to the monitored *dataport* variable, which will cause the test **expression** to be evaluated each time its value is changed by the application.

The handler always executes on behalf of the context changing the *dataport* variable's value.

A single test expression can be attached to a *dataport* variable at any time. Calling ckSetTestExpr() more than once for a single dataport variable only retains the last test expression. The *dataport* variable address is not restricted to the current node context.

Syntax:

The syntax of the test expression supports simple arithmetic and relational operators between constant and/or variable terms. Bracketed expressions affecting the evaluation order are supported. The following tokens may appear in expressions:

Signed integer values and high-precision (double) floating-point literal values

Boolean constants *true* and *false*

Arithmetic operators *, /, + and -

Relational operators <, <=, >, >=, ==, !=

Logical operators &&, ||

Special variable #T standing for the current simulation clock value (i.e. same result as calling ckGetTime()).

Special variable #D standing for the *dataport* variable itself. If the variable is an aggregate (i.e. a struct or union), standard C references using the dot operator can be used to access its members. However, array subscripts are not supported.

Evaluation:

The expression is evaluated from left to right, with no special operator precedence. Sub-expressions should be bracketed in order to get a different evaluation order.

Values from the special variables are fetched dynamically at the time the containing expression is evaluated.

The result of the evaluation is converted (if needed) to a boolean value. A truth outcome causes the **handler** to be fired immediately. Otherwise, the execution continues.

A simple way of obtaining unconditional handler execution upon each variable update is to set **expr** to constant *true*.

RETURN VALUES

Zero is returned on success. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The `ckSetTestExpr()` service fails if:

ESRCH **datap** is not the address of a *dataport* variable.

EINVAL an error occurred while parsing **expr**.

EXAMPLE

```
dataport int collisions;

void handleUnstableNetwork (void *cookie)
{
    ckFatal("TOO MANY COLLISIONS (> 1500): %d\n", collisions);
}
...
{
    /* Install a test expression on the collision number */
    ckSetTestExpr(&collisions, "#D > 1500", handleUnstableNetwork, NULL);
}
```

2 The Magnet Interface

This chapter describes the interface between the CarbonKernel panels and Tcl scripts implementing magnets. How to create a panel and add magnets to a panel is beyond the scope of this discussion. Please refer to the CarbonKernel user manual for this purpose.

2.1 What's a magnet ?

A magnet is a graphical object which can be connected to the simulation via a *dataport*, a message port, or a simulated device driver, to display a simulation variable's content. Thus, the current value and/or state of the variable can be monitored graphically, and may be changed interactively.

A magnet is logically bound to a message port to send and/or receive data to/from the application. Because *dataports* are specialized message ports exporting the contents of programming variables, they are first choice candidates for exposure in magnets. But almost any basic message ports can be connected to a graphical magnet, provided that both sides (i.e. the application and the magnet code in Tcl) agree upon the format of the exchanged data.

Data are exchanged as Tcl lists between the simulation kernel and the magnet code written in Tcl. The simulation kernel is responsible for converting the contents of a binary object (e.g. a programming variable) to a Tcl list representation for the purpose of sending the value to the magnet hosted by the ISE, and conversely, from a Tcl list to a binary form when the magnet requests the simulation kernel to change the object's value.

2.2 Using Tcl/Tk for magnets

A magnet is implemented using the Tcl/Tk scripting language, and can also use the Tix extension. A magnet can be a simple text field, or a complex graphic object, using multiple Tk/Tcl objects and/or bitmaps, such as a LCD display or a keypad.

Tcl (Tool Command Language) is a popular open source scripting language. Tcl and the Tk (Tool Kit) GUI extension enable developers to quickly and easily create applications that are both powerful and easy to extend.

Tix (Tk Interface eXtension) is an extensive set of mega-widgets, designed to speed up development of Tk-based applications.

2.3 Hosting a magnet

A magnet is hosted in a display window managed by the ISE called a “panel” (see the CarbonKernel user manual). A panel is basically a Tk canvas, and acts as a magnet manager: it draws the magnet's decoration and is also in charge of the dragging and resizing operations. It also keeps track of connections between magnets and data sources, and notifies the appropriate magnet of incoming messages.

To implement a magnet, a script must define a set of mandatory Tcl procedures. Those procedures are described later in this document, and are mainly used to identify and draw the given magnet.

2.4 Naming convention

The prefix used to compose the canonical Tcl procedure names must be the same as the containing script file name, i.e. a magnet called “flashyMagnet” must export canonical procedures prefixed with “flashyMagnet:”, defined in a file called “flashyMagnet.tcl”. We strongly suggest making the prefix end with the word “Magnet” to reduce the odds of naming conflicts with other unrelated procedures.

Magnet:describe

SYNOPSIS

```
proc PREFIX:describe{}
```

DESCRIPTION

This Tcl procedure is called by the panel manager to get the initial information describing the magnet. This is the first procedure called for a given magnet after the panel manager has detected its presence through the entry of its implementation script file, in the standard directory reserved to the ISE's magnets.

This procedure is mandatory.

RETURN VALUES

describe should return a three-elements Tcl list containing the icon name, the tooltip, and a Tcl list of flags, such as { **icon tooltip { flags } }**.

An **icon**, in any format Tk/Tix can handle, should be present in the CarbonKernel icon repository. The icon size should be 20x20 pixels. This icon will appear in the panel's toolbar. Clicking on it will start the creation procedure for this magnet.

The **tooltip** string will be displayed each time the mouse enters the magnet's icon area.

The currently supported **flags** are:

- ▣ *noresize* indicates a fixed size for the magnet, and no resize handle will be drawn by the panel manager. This is generally used with magnets made of bitmaps (which cannot be resized that way).
- ▣ *customizable* indicates that the magnet provides a customization popup menu. The panel manager checks for this flag to determine whether a “Customize” item should appear in the popup menu bound to the right mouse button. The **customizeFill** procedure is called whenever the user selects this menu entry.

The **flags** list may be empty, in which case the magnet is by default resizable and not customizable.

SEE ALSO

customizeFill, customizeApply

EXAMPLE

```
proc buttonMagnet:describe {} {
    return { button.gif "A Button Magnet" {noresize customizable} }
}
```

Magnet:new

SYNOPSIS

```
proc PREFIX:new {tag triggercmd private}
```

DESCRIPTION

This Tcl procedure is called by the panel manager to create a new graphical instance of the magnet. This may be a really new magnet the user has just created, or a previously saved magnet which gets displayed after its hosting panel is re-opened.

The **tag** parameter specifies a unique tag which should be used by the magnet code to mark its components inside the canvas (using for example the “-tag \$tag” option to the “create” command of any Tcl canvas object). This tag should identify all parts of a magnet.

A magnet can send events to the simulation object it is bound to. The message to send should be appended to the contents of the **triggercmd** argument before evaluating the resulting command. This parameter may be null if running in editing mode.

The set of private information the magnet previously returned by its **getprivate** procedure is passed back into the **private** parameter. At first creation time, this parameter contains an empty list.

This procedure is mandatory.

RETURN VALUES

None.

SEE ALSO

getprivate

EXAMPLE

```
proc buttonMagnet:new {tag triggercmd private} {
    # Use this array to store per-magnet info.
    global buttonMagnetInfo

    # We used to save the image name in our private area
    set imagefile [lindex $private 0]

    if {$imagefile == {}} {
        # First creation, use a default value
        set imagefile bdisplay.gif
    }

    # Update the per-magnet info. indexed on $tag
    set buttonMagnetInfo($tag,triggercmd) $triggercmd
    set buttonMagnetInfo($tag,imagefile) $imagefile
    set buttonMagnetInfo($tag,image) [tix getimage $imagefile]
}
```

Magnet:draw

SYNOPSIS

```
proc PREFIX:draw {canvas tag rect}
```

DESCRIPTION

This Tcl procedure is called by the panel manager whenever it needs the magnet to redraw itself. **canvas** is the Tk window name of the hosting canvas the magnet is supposed to draw on. The panel manager draws the magnet's decoration (title bar, name, borders) before entering this procedure. At exit, it draws the resize handle if the magnet is resizable.

A unique **tag** referring to the magnet is passed to the procedure, and should be used by the magnet code to mark its components inside the canvas using the *-tag* option to the *create* command of the Tcl canvas object. This will be used by the panel manager for operating on the magnet as a whole.

Finally, the client area of the magnet inside the canvas is passed in **rect**, which is a Tcl list containing four elements: leftmost-x coordinate, topmost-y coordinate, height, and width (all expressed in pixels). The magnet can draw anywhere inside this rectangular area. If the magnet cannot draw itself in the given client area (such as magnets using larger or smaller bitmaps), it can ignore the **rect** parameter, and return its actual size. The returned value should be a Tcl list specifying the actual height and width of the magnet. In case the magnet needs a specific client size, the panel manager redraws the decoration, so that it fits around the magnet.

This procedure is mandatory.

RETURN VALUES

If the magnet can scale itself inside the given client area, it should return an empty Tcl list. Otherwise, a Tcl list containing its actual height and width should be returned.

EXAMPLE

```
proc buttonMagnet:draw {canvas tag rect} {
    global buttonMagnetInfo

    # fetch the magnet's bitmap width and height
    # we saved in the per-magnet information array.
    set imagewidth $buttonMagnetInfo($tag,imagewidth)
    set imageheight $buttonMagnetInfo($tag,imageheight)

    $canvas create image 0 0 \
        -image $buttonMagnetInfo($tag,image) \
        -anchor nw \
        -tags $tag

    # force a specific size
    return [list $imageheight $imagewidth]
}
```

Magnet:getprivate

SYNOPSIS

```
proc PREFIX:getprivate{canvas tag}
```

DESCRIPTION

This Tcl procedure is called by the panel manager to get the magnet's private information before its status is written to disk. This information will be saved along with the other magnet's properties, and passed back to the magnet when the *new* procedure is called.

Private information can be used to hold the magnet's local settings, such as color names, bitmap filename, and so on.

The Tk window name of the hosting **canvas** and the magnet's unique **tag** is passed to the procedure.

This procedure is mandatory.

RETURN VALUES

A Tcl list of private information. The content of this list will be saved to disk, and passed back to the *new* procedure when called to reinstate the same magnet.

SEE ALSO

new

EXAMPLE

```
proc buttonMagnet:getprivate {canvas tag} {

    global buttonMagnetInfo

    return [list \
        $buttonMagnetInfo($tag,imagefile) \
        $buttonMagnetInfo($tag,switchmode)]
}
```

Magnet:update

SYNOPSIS

```
proc PREFIX:update {canvas tag rect value {flags {}}}
```

DESCRIPTION

This Tcl procedure is called by the panel manager whenever an update message has been received on the associated message port. The magnet should update its display according to the new value. **canvas** is the Tk window name of the hosting canvas.

The magnet's unique **tag** is passed along with its client area in the canvas. The client area defined by **rect** is a four-element Tcl list which respectively contains the leftmost-x corrdinate, the topmost-y coordinate, the height, and the width of the magnet (all expressed in pixels). The magnet code can draw anywhere inside this rectangular area.

A - possibly empty - set of **flags** describing the internal object's state is passed.

The currently supported flags are:

- ▣ **ro** indicates that the magnet cannot write back to the object. Otherwise, the source data emitted by the simulator is assumed to be overwritable, and thus can be overwritten as a result of the proper trigger command.

This procedure is optional.

RETURN VALUES

None.

EXAMPLE

```
proc buttonMagnet:update {canvas tag rect value {flags {}}} {

    global buttonMagnetInfo

    # save the last received value in our private info.
    set buttonMagnetInfo($tag,value) $value

    # update the display to exhibit the received value.
    buttonMagnet:drawImage $canvas $tag $rect
}
```

Magnet:delete

SYNOPSIS

```
proc PREFIX:delete{canvas tag}
```

DESCRIPTION

This Tcl procedure is called whenever the magnet should remove itself from the canvas, usually as a consequence of a user request for deletion. **canvas** is the Tk window name of the hosting canvas. **tag** is the magnet's unique tag.

The magnet's local information is of no use anymore after this procedure has returned, and should be discarded from the Tcl module's variables when applicable.

This procedure is optional. If it is not defined, the panel manager acts as if an empty *delete* procedure was defined and returned *true*.

RETURN VALUES

This procedure should return *“true”* if the operation was successful, *“false”* otherwise.

EXAMPLE

```
proc buttonMagnet:delete {canvas tag} {

    global buttonMagnetInfo

    # destroy the bitmap we put on the canvas
    $canvas delete magnetimage
    # destroy some sub-object we have created
    destroy $canvas.subc$tag

    return true
}
```

Magnet:popdown

SYNOPSIS

```
proc PREFIX:popdown{canvas tag menu mode name}
```

DESCRIPTION

This Tcl procedure is called whenever a popdown menu (bound to the right mouse button) is about to be displayed by the panel manager for this magnet. The magnet is given the ability to add some commands to the next-to-be-displayed menu.

canvas is the Tk window name of the hosting canvas. **tag** is the magnet's unique tag.

The Tk **menu** object about to be displayed already contains the general entries set by the panel manager.

A **mode** parameter tells which state the manager is currently in, whether *edit* if we are currently off-line, of *run* if the magnets are bound to a running simulator.

name is the user-defined name of the magnet.

This procedure is optional.

CAUTION: This procedure is not to be confused with the graphical customization of the magnet. A set of procedures is dedicated to the customization, while the intent of this one is to provide a way to add very specific user-defined menu actions.

RETURN VALUES

None.

SEE ALSO

customizeFill, customizeApply

Magnet:customizeFill

SYNOPSIS

```
proc PREFIX:customizeFill {tag dlgf}
```

DESCRIPTION

This Tcl procedure is called by the panel manager whenever the user requires the magnet customization from the popdown menu (bound to the right mouse button) over this magnet. This procedure should build the dialog box that will permit its customization.

The magnet's unique tag and the hosting Tk frame object are passed to the procedure. The frame should be the parent object the dialog items are put on.

The panel manager automatically adds three buttons to the dialog box after this procedure has returned:

- ▣ **Ok** applies all modifications to the magnet, and closes the dialog box;
- ▣ **Apply** applies all modifications to the magnet, but leaves the dialog box open;
- ▣ **Cancel** closes the dialog box without applying any modification.

This procedure is mandatory if the *customizable* flag was returned by the **describe** procedure, and ignored otherwise.

RETURN VALUES

None.

SEE ALSO

describe, popdown, customizeApply

EXAMPLE

```
proc buttonMagnet:customizeFill {tag lbf} {

    global buttonMagnetInfo tkbridge_prefixdir

    tixComboBox $lbf.images -label "Image:" \
        -variable buttonMagnetInfo($tag,tmpImagefile) \
        -options {
            label.width 7
            label.anchor e
        }
    foreach file \
        [glob -nocomplain -- $tkbridge_prefixdir/share/ck/images/bt*] {
        set fname string range $file \
            [expr 1 + [string last "/" $file]] \
            [expr [string last "." $file] - 1]]
        if {$fname != {}} {
            $lbf.images insert end $fname
        }
    }
    set buttonMagnetInfo($tag,tmpImagefile) \
        $buttonMagnetInfo($tag,imagefile)
```



```
    pack $lbf.images -expand yes -fill both -padx 8 -pady 3 -side top  
}
```

Magnet:customizeApply

SYNOPSIS

```
proc PREFIX:customizeApply{tag dlgf}
```

DESCRIPTION

This Tcl procedure is called by the panel manager whenever the user selects the **Ok** or **Apply** buttons from the customization dialog box. The magnet should modify itself according to the new customized values.

The magnet's unique tag and the hosting Tk frame object are passed to the procedure.

This procedure is mandatory if the *customizable* flag was retruned by the **describe** procedure, and ignored otherwise.

RETURN VALUES

None.

SEE ALSO

describe, popdown, customizeFill

EXAMPLE

```
proc buttonMagnet:customizeApply {tag lbf} {

    global buttonMagnetInfo

    set buttonMagnetInfo($tag,imagefile) \
        $buttonMagnetInfo($tag,tmpImagefile)
    set buttonMagnetInfo($tag,image) \
        [tix getimage $buttonMagnetInfo($tag,imagefile)]
}
```

Index**C**

ckBindPort 4, 59, 61
 ckBroadcastCv 56p.
 ckCloseTerminal 33pp
 ckControlIntr 25p., 29p., 47
 ckControlPanel 67
 ckCreateCv 46, 56p.
 ckCreateIntr 15, 22pp, 29
 ckCreateMutex 46, 54p., 57
 ckDecodeTimeBounds 45
 ckDelay 18p.
 ckDestroyCv 56p.
 ckDestroyIntr 23pp
 ckDestroyMutex 54
 ckFatal 11p., 69
 ckFinishSimulation 10
 ckGetArgs 40
 ckGetChar 33pp
 ckGetConf 23, 49
 ckGetContext 25, 42p., 50
 ckGetErrnoAddr 39
 ckGetIntrLevel 25, 42pp
 ckGetMemPortByte 3, 31
 ckGetMemPortDword 3, 31p.
 ckGetMemPortQword 3, 31
 ckGetMemPortWord 3, 31
 ckGetNid 37
 ckGetObjectHandle 6p., 22, 26, 46pp, 67
 ckGetObjectName 7, 44, 47p., 50, 66
 ckGetObjectTag 47p., 50p.
 ckGetSpinLock 52p.
 ckGetStdStream 38
 ckGetTicks 16p.
 ckGetTime 16p., 68
 ckInitSpinLock 52
 ckLockMutex 54p., 57
 ckLockTime 6, 14
 ckMaskIntr 25, 30
 ckOpenTerminal 33pp
 ckPendSynch 5, 18p.
 ckPoll 20p.
 ckProgramIntr 15, 23, 27pp
 ckPutChar 33pp
 ckPutFormat 36
 ckPutString 33pp
 ckRaiseIntr 25p., 29p., 42p.
 ckReadPort 4, 59pp
 ckRelSpinLock 52p.
 ckResumeSimulation 8p.
 ckSelect 20p., 59p.
 ckSelectPort 59p.
 ckSetHandler 65p.
 ckSetHostSpeed 62
 ckSetObjectTag 50p.
 ckSetTargetWarp 63
 ckSetTestExpr 68p.
 ckSignalCv 56p.
 ckSpawn 64
 ckSuspendSimulation 8p., 62
 ckTick 15, 17
 ckTrace 13
 ckTryMutex 54p.