The Phoenix Testbed

Michael Greenberg and David L. Westbrook

COINS Technical Report 90-19

Experimental Knowledge Systems Laboratory
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

The Phoenix project is an experiment in the design of autonomous agents for a complex environment. The Phoenix testbed is the environmental simulation component of Phoenix. The Phoenix testbed is consists of a task model, a map representation, a user interface and a fire simulation. This paper presents the testbed at the conceptual level and at the functional interface level.

This research has been supported by the Air Force of Scientific Research under contract F49620-89-C-0121; ONR University Research Initiative grant N00014-86-K-0764; a grant from Digital Equipment Corporation; DARPA-AFOSR contract F49620-89-C-00113; and DARPA/RADC F30602-85-C0014.

We wish to thank David Hart and Scott Anderson for their skillful proof reading and helpful stylistic contributions.

Contents

Chapter 1

Introduction

The Phoenix testbed[1] contains four components: A task model, a map representation, a user interface and a fire simulation. In this document, each component is described conceptually, what is it and how does it work, and functionally, how is the component used.

The appendices contain a description of the file organization, and installation and maintenance instructions.

In a view, the testbed is an environment for simulating processes that need to be synchronized in *time*. Each process is called a *task*. In addition to tasks, the testbed provides a topological representation of the world. This *map* contains information about vegetation, roads, rivers and buildings. A user interface and a forest fire simulation are also provided.

1.1 Simulation and Time

Phoenix provides for the simulation of the world by allowing the user to define tasks which "run the world." By maintaining a global (between all tasks) notion of time, the system is able to control each task to make sure that the tasks stay synchronized. Thus, each task can implement a part of the world, and Phoenix makes sure all the parts are kept up to date. Tasks communicate via shared data structures and system defined synchronization methods.

A simple example will make this clear. Suppose we want to simulate a world that has forest fires and firefighters. We can define a task which burns the fire (the simulator), and one task for each firefighter. The tasks must be synchronized by some global notion of time. In a five minute period, for instance, the fire can burn only so far and a firefighter can only do a certain amount of problem solving. The tasks interact by modifying a global data structure, in this case the map of the world. The simulator burns things, and the firefighters try to contain the fires.

The *Phoenix Scheduler* is responsible for keeping tasks synchronized. The global clock is called *simulation-time*. Each task must provide a method which specifies how simulation-

[1]The testbed is implemented in Common Lisp on the Explorer II; a high-performance standalone Lisp workstation.

time passes as it (the task) executes. There are three distinct types of time in the system:

Simulation Time This is the global shared clock to which each task is synchronized.

Task Time This is the time that each task thinks it is. Imagine that each task has a watch.

CPU time This refers to cpu time for a single task.

Unless otherwise specified, all time units are measured in *internal time units.* Functions are provided to convert between time units (for example, `minutes->internal-time` and `internal-time->seconds`).

1.2 The Fire-system

While Phoenix is running, one object handles all requests by tasks, namely, the current instance of the `fire-system` flavor. The `fire-system` flavor contains the user interface, task scheduler, sets of tasks and the real world representation. It also provides numerous methods for manipulating those objects. The function `fire-system` returns the current instance of the `fire-system` flavor, which we will call that the current fire system.

Chapter 2

Tasks

Tasks are the organizing component of the Phoenix testbed. A *task* is a process plus a time-keeping mechanism. Each task must specify how *simulation-time* passes as the task's process executes. For example, a task may say "One second of cpu time corresponds to five minutes of simulation time." Three types of tasks exist; they differ by the time-keeping mechanisms they use.

CPU-time tasks. In a cpu-time task, simulation-time is a function of cpu time. When you define such a task, you specify the ratio of simulation time to cpu time. As the task runs, the Phoenix Scheduler keeps track of time appropriately. For example, the firefighting agents "think" via lisp code, and their "thinking speed" is set by some ratio of simulation time to cpu time, so that N cpu seconds of lisp execution is taken to be $C \cdot N$ seconds of simulated real time.

Explicit-time tasks. An explicit-time task is responsible for explicitly telling the scheduler how much time passed while it ran. For example, if there is a task to move an agent in the world, that task computes time as a function of speed and distance.

Periodic tasks. A periodic task runs at a fixed time interval. This interval is the task's *period.*

2.1 Tasks and the Scheduler

The tasks are interleaved on the cpu to simulate parallelism. The Phoenix Scheduler is responsible for allocating each task the appropriate amount of cpu time. The basic control structure of the scheduler is:

1. Select the task that should be executed next.
2. Start the task on the cpu.
3. When the task relinquishes control of the cpu, update its *task-time.* In general, cpu-time tasks relinquish control whenever there is a cpu timer interrupt (on the Explorer, this

means a task gets at most one cpu second before relinquishing control). Explicit-time and periodic tasks explicitly return control to the task scheduler.

4. Repeat.

The scheduler updates each task's task-time by an amount computed based on the task's type. A simple example should illustrate this.

Suppose there are four tasks.

$Task_1$ is a periodic task that runs once every three minutes.

$Task_2$ is an explicit-time task.

$Task_3$ is a cpu-time task that runs at 5 minutes/cpu-second.

$Task_4$ is a cpu-time task that runs at 10 minutes/cpu-second.

When the simulation begins, the task-time for each task is zero. Following is a chronology of what happens:

> $Task_1$ executes at task-time 0 minutes. (This means that from the task's point of view, 0 minutes has elapsed since the simulation began). After it is done, the scheduler updates the task-time for $Task_1$ to be 3 minutes.
>
> $Task_2$ executes at task-time 0 minutes. When it relinquishes control, it reports that it used 7 minutes of simulation-time. The scheduler updates its task-time to be 7 minutes. Note that it is now "out of sync" with $Task_1$ by 4 minutes and $Task_3$ and $Task_4$ by 7 minutes.
>
> $Task_3$ executes at task-time 0 minutes. After using 1 cpu second, it is interrupted and the scheduler updates its task-time to be 5 minutes (1 cpu-second * 5 minutes/cpu-second).
>
> $Task_4$ executes at task-time 0 minutes. After using .8 cpu seconds, it is interrupted and the scheduler updates its task-time to be 8 minutes (.8 cpu-seconds * 10 minutes/cpu-second).

The scheduler maintains a *queue* of tasks sorted by task-time. The task chosen to execute next is the task at the head of the queue. The queue currently looks like this (the number is the task-time in minutes):

> $Task_1$=3, $Task_3$=5, $Task_2$=7, $Task_4$=8
>
> $Task_1$ executes for three minutes. When complete, its time is incremented by three minutes (the task's period).
>
> $Task_3$=5, $Task_1$=6, $Task_2$=7, $Task_4$=8
>
> $Task_3$ executes and gets .1 cpu-seconds. Its task-time is updated to 5.5 minutes.
>
> $Task_3$=5.5, $Task_1$=6, $Task_2$=7, $Task_4$=8

Notice that each task has its own idea about what time it is (that is, their watches disagree). This occurs for two reasons. First, we are simulating parallelism on a serial machine. Second, some tasks represent discrete processes. Given this, the tasks are all out of synchronization by some amount. The maximum "out of sync" time is the maximum over all tasks of *period*, *explicit time interval* and *max cpu quantum* · *simulation time* / *cpu time*. On the Explorer, the max cpu quantum is 1 second.

2.2 Defining and Using Tasks

This section describes how to define each type of task. Examples are given throughout. All tasks start with the `task` flavor, that is, to define a task you must first create a new flavor inheriting from `task`. You must also write a method which will *implement* the task and decide what the task's type is.

2.2.1 Defining CPU-time Tasks

To make a cpu-time task flavor called `generic-cpu-time-task`, do the following:

```
(defflavor generic-cpu-time-task ()
        (task)
    (:default-init-plist
     :initial-method :generic-cpu-time-task-toplevel
     :schedule-type :cpu-time
     :cpu-usec/internal-time  (round 1e6 (minutes->internal-time 5))))
```

This creates a task flavor named `generic-cpu-time-task` of type `:CPU-TIME`. The ratio of cpu-time to simulation-time is expressed in cpu-micro-seconds per internal-time unit. In the example, the ratio is set to 5 simulation-time minutes per one cpu-second. The method that executes when the task is run by the system is specified by `:initial-method`. In this case, the method `:generic-cpu-time-task-toplevel` must be defined. A cpu-time task normally shouldn't return control from the initial method once it is called. If it does, the task is *deactivated* (See section 2.2.4).

Normally, a cpu-time task is interrupted at least once every cpu-second (on the Explorer) and control is returned to the scheduler. If for some reason you want to explicitly return control to the scheduler, call `swap-in-scheduler`. Processing will continue from that point when the task is resumed.

2.2.2 Defining Periodic Tasks

To make a periodic task flavor called `generic-periodic-task`, do the following:

```
(defflavor generic-periodic-task ()
        (task)
  (:default-init-plist
   :initial-method :generic-periodic-task-toplevel
   :schedule-type :periodic
   :period (minutes->internal-time 5)))
```

This is almost identical to creating a cpu-time task. The differences are `:SCHEDULE-TYPE` and `:PERIOD`. When the scheduler decides to execute a periodic task, it executes the initial method. When the method returns, the task-time is incremented by the period, and the task is put back into the queue.

2.2.3 Defining Explicit-time Tasks

To make an explicit-time task flavor called `generic-explicit-task`, do the following:

```
(defflavor generic-explicit-task ()
        (task)
  (:default-init-plist
   :initial-method :generic-explicit-task-toplevel
   :schedule-type :explicit))
```

This is almost identical to creating a cpu-time task. Again, when the scheduler runs an explicit-time task, its initial method is executed. When the this method returns, the task is rescheduled (that is, reinserted into the task queue). Sometime during the execution of the method, the instance variable `restart-time` must be set to the appropriate time to run this task again. If `restart-time` is not reset, an error is signaled. If for some reason the task must restart again at the same time, the initial method must return `:OK-TO-RESTART-AT-SAME-TIME`. At any time during its execution, an explicit-time task can return control to the scheduler. To do so, it should set `restart-time` appropriately, then call the function `swap-in-scheduler`. Processing will continue from the point immediately after the `swap-in-scheduler` the next time the task is run.

2.2.4 The Task Life Cycle

There are several phases in the life of a task: creation, activation, execution, deactivation and termination.

Creation

The first thing to do is make an instance of a task with `make-instance`. When you create a task you should give it a *handle*. A task handle is a name (represented as a symbol) that can be used to index the task. You can also specify an `:after :init` method to perform instantiation-time actions.

```
(make-instance 'generic-cpu-time-task :handle 'task-1 :name "Task 1")
```

Activation

When a task is created, it is not available for execution. To make it available, you must *activate* it.

```
(tsend 'task-1 :activate)
```

Once activated, a task can be run by the scheduler. You can specify an `:after :activate` method. The `:activate` method is run in the calling process, not the process associated with the task. If you want a task to be activated at instantiation time, use the `:activate` initialization option `(make-instance task ... :activate T)`

The function `tsend` is like `send`, except that it takes a task-handle instead of a flavor object.

Execution

All active tasks can be scheduled by the Phoenix Scheduler. Each task runs in its own process. Execution starts at the task's initial method. In a cpu-time task, the method should not return. In periodic and explicit-time tasks, when the method returns, the scheduler may schedule another task. These two types of tasks can assume that they won't be swapped until they either explicitly release control, or the method returns. A cpu-time task may be interrupted any time, at any place in its code.

Deactivation

A task can be stopped by *deactivating* it. Once deactivated, it is no longer eligible for execution by the scheduler. To deactivate a task, send it a `:deactivate` message. A task can deactivate itself:

```
(send self :deactivate)
```

or be deactivated by some other process

```
(tsend task-handle :deactivate)
```

You may specify a `:before :deactivate` method.

Reactivation

After a task has been deactivated, it may be reactivated again. When a task is reactivated, execution *always* begins from the initial method. All internal state information may be lost. Activation and deactivation do not correspond to pausing and resuming. Activation is "reset to initial state and begin execution" and deactivation "stop and clean up." Since a task may be activated and deactivated many times (the UCL command `Reset` deactivates then activates all tasks), the `:after :activate` method should make sure the task's state is properly initialized, taking into account that the task may have been run previously.

Termination

At some point, you will want to kill a task. To do so, send the task a `:kill` message. A task can kill itself or be killed by some other process. Before a task is killed, it is deactivated if it is active. You may write `:after :kill` methods.

2.2.5 Managing Time

The current task-time is available to all tasks, and is returned by the function `exact-time`. In a cpu-time task, time is continuously changing, whereas in the other types, time changes in discrete steps. In cpu-time tasks, the scheduler computes the task-time.

For periodic tasks, task-time is updated by the task's period each time the initial method is executed. A periodic task may change its period by setting the `period` instance variable.

Explicit-time tasks are responsible for updating their own time before the initial method returns. This is done by setting the instance variable `restart-time` to be the next time the task is run. For example, an explicit-time task could say "Next time my initial method is started, start it 5 minutes from now" as follows:

```
(incf restart-time (minutes->internal-time 5))
```

Each time an explicit-time task's initial method is started, the task's task-time is set to its restart-time.

All three types of tasks can say, "Put me to sleep for some amount of time." For explicit-time and periodic tasks:

```
(incf restart-time (minutes->internal-time 5))
(swap-in-scheduler)
;; processing continues here 5 minutes later
```

For cpu-time tasks:

```
(setf cpu-time-adjustment (* cpu-usec/internal-time
                             (minutes->internal-time 5)))
(swap-in-scheduler)
```

2.3 A Real Example

This section contains an annotated example using four tasks that print informational messages. The function `task-format` prints a trace message to the trace pane in the 'process-display' screen configuration. The message automatically includes the task-time and the task's handle. So if the task called "T-1" at 1:00 P.M. on 8/1 executes

```
(task-format "Hello there")
```

the output is

```
[8/1 13:00 T-1: Hello there]
```

The first task will be a periodic task that wakes up once every three minutes. This task will keep track of the number of times it has run in the instance variable `count`. Notice that the `:after :activate` method sets `count` to zero.

```
(defflavor periodic-task (count)
        (task)
  (:default-init-plist
   :initial-method :periodic-toplevel
   :schedule-type :periodic
   :period (minutes->internal-time 3)))

(defmethod (periodic-task :after :activate) ()
    ;; When this task is activated, reset count to 0.
    (setf count 0))

(defmethod (periodic-task :periodic-toplevel) ()
    (incf count)
    ;; task-format prints a trace message
    (task-format "Count is ~d" count))
```

The second task will be an explicit-time task. This task also keeps track of the number of times it has been called with the instance variable `count`. Explicit-time tasks must keep track of their own time. The first time this task is run it will take one minute, the second time two minutes, then three minutes, and so on.

```
(defflavor explicit-task (count)
        (task)
  (:default-init-plist
   :initial-method :explicit-toplevel
   :schedule-type :explicit))

(defmethod (explicit-task :after :activate) ()
    ;; When this task is activated, reset count to 0.
    (setf count 0))

(defmethod (explicit-task :explicit-toplevel) ()
    (incf count)
    (task-format "Count is ~d" count)
    (incf restart-time (* count (minutes->internal-time 1))))
```

The third and fourth tasks will be cpu-time tasks. The difference between them will be their ratio of task time to cpu time. Since a cpu-time task's initial method shouldn't return, a local variable can be used to keep track of iterations. The function `1-second-compute` takes exactly one cpu second to execute (on an Explorer-II).

```
(defflavor cpu-task ()
        (task)
  (:default-init-plist
   :initial-method :cpu-toplevel
   :schedule-type :cpu-time))

(defmethod (cpu-task :cpu-toplevel) ()
  (do ((count 1 (1+ count)))
      (nil)
    (task-format "Count is ~d" count)
    (1-second-compute)))
```

To save time we have already defined these task flavors and methods in the file "*PH:TASKS;DOCUMENTED-TASK-EXAMPLE.LISP*". To run these tasks, load that file and start Phoenix.

Once Phoenix is up, select the 'process-display' screen configuration (user typein is shown after a Phoenix prompt).

```
Phoenix> process-display
```

Now we must create the four flavor instances:

```
(make-instance 'explicit-task :handle 'explicit-task :activate T)
(make-instance 'periodic-task :handle 'periodic-task :activate T)
(make-instance 'cpu-task :handle 'cpu-task-1
     :cpu-usec/internal-time  (round 1e6 (minutes->internal-time 5))
     :activate T)
(make-instance 'cpu-task :handle 'cpu-task-2
     :cpu-usec/internal-time  (round 1e6 (minutes->internal-time 10))
     :activate T)
```

The first cpu-time task runs at 5 minutes/cpu-second. The second runs at 10 minutes/cpu-second. The function `create-task-demo-tasks` (also defined in the *DOCUMENTED-TASK-EXAMPLE* file) creates all the instances, so:

```
Phoenix> (create-task-demo-tasks)
```

The simulation begins on August first at 12 noon. To run the system for 16 minutes, type

```
Phoenix> Run 16
```

When you start the system running, the scheduler will run each task at the appropriate time. You should see the output from the calls to `task-format` in the trace window. The output should start as follows:

```
[8/1 12:00 EXPLICIT-TASK: Count is 1]
[8/1 12:00 PERIODIC-TASK: Count is 1]
[8/1 12:00 CPU-TASK-1: Count is 1]
[8/1 12:00 CPU-TASK-2: Count is 1]
```

If everything works correctly, the explicit task will generate output at 12:00, 12:01, 12:03, 12:06 etc. Each time interval is one minute greater than the previous. The periodic task will count every three minutes. The first cpu-time task should count in five minute intervals, and the second at ten minute intervals. The output should continue with:

```
[8/1 12:01 EXPLICIT-TASK: Count is 2]
[8/1 12:03 EXPLICIT-TASK: Count is 3]
[8/1 12:03 PERIODIC-TASK: Count is 2]
[8/1 12:05 CPU-TASK-1: Count is 2]
[8/1 12:06 EXPLICIT-TASK: Count is 4]
[8/1 12:06 PERIODIC-TASK: Count is 3]
[8/1 12:09 PERIODIC-TASK: Count is 4]
[8/1 12:10 EXPLICIT-TASK: Count is 5]
[8/1 12:10 CPU-TASK-1: Count is 3]
[8/1 12:10 CPU-TASK-2: Count is 2]
[8/1 12:12 PERIODIC-TASK: Count is 5]
[8/1 12:15 EXPLICIT-TASK: Count is 6]
[8/1 12:15 PERIODIC-TASK: Count is 6]
[8/1 12:15 CPU-TASK-1: Count is 4]
```

When this runs, the output may vary slightly because when two tasks should run at the same time, the scheduler picks one arbitrarily.

Suppose you want to change the periodic task to run at 6 minute intervals. To do this you need to change the period and reset the system.

```
Phoenix> (tsend 'periodic-task :set-period (minutes->internal-time 6))
Phoenix> reset
```

If you run it again, the period will change. The `Reset` command sets the clock back to 12:00 and reactivates all the tasks. Note that if the `:after :activate` methods weren't specified, the count would continue from where it left off.

2.4 Input/Output

Since all tasks run as background processes, doing i/o is not straightforward. For all types of tasks, the easiest way to do output is with the functions `task-format` and `debug-format`. These functions are similar to `format`. The difference is that they generate an output message that automatically includes the task handle and task-time. The output is sent to the process-trace pane. The `debug-format` function also sends the output to the Phoenix Lisp Listener pane.

2.4.1 I/O in Periodic and Explicit-time Tasks

In principle, it is OK to perform any i/o function from periodic and explicit-time tasks. It is best not to do i/o to/from the lisp listener or any window that isn't exposed. Use functions like `w:pop-up-prompt-and-read` and `utils:pop-up-msg` for maximum safety.[1] While a task is waiting for input, no other tasks execute.

2.4.2 I/O from CPU-time Tasks

i/o from cpu-time tasks is particularly difficult because the scheduler can't tell that the task is waiting for input. To do i/o, you should only use functions that have been properly configured with `dont-swapout-function`. Currently, it is safe to use the following functions:[2] `w:pop-up-prompt-and-read`, `tv:careful-notify`, and `tv:mouse-confirm`. When i/o is done from a cpu-time task, the cpu time accounting may become slightly inaccurate. i/o should only be done from cpu-time tasks for debugging purposes. This is because any time the i/o takes is charged to the cpu task. This isn't a problem with the other task types.

[1] Currently, `*terminal-io*` (and therefore all the other stream variables) are bound in the task to a deexposed background window. An error will result if output is directed to it.

[2] see "*PH:TASKS;SCHEDULER.LISP*" for further details.

2.5 Debugging

When a task gets an error, the scheduler notices the error and pops up a window that includes the task-handle, the function where the error occurred and a stack backtrace. To deactivate the task, just move the mouse off the window. The execution of all the other tasks can be continued. To enter the debugger, click on the window and an error message will appear on the screen. To select the debugger for the task, type TERM META-S. It is possible to proceed from the debugger, but it doesn't always work as expected. The best thing to do is use the debugger to find and fix the problem, and then `Reset` and start again.

Chapter 3

Task Reference Manual

3.1 Task Object Manipulations

`:activate` [*Method of* `task`]

`:closure` [*Method of* `task`]

`:cpu-time` [*Method of* `task`]

`:cpu-usec/internal-time` [*Method of* `task`]

`:deactivate` [*Method of* `task`]

`:after :deactivate` [*Method of* `task`]

`:edit-parameters` *&optional additional-items* [*Method of* `task`]

See (:method standard-agent :around :edit-parameters) to see how to add items.

`find-task` *handle* [*Function*]

Finds the task associated with the handle. This does no error checking.

`:handle` [*Method of* `task`]

`:after :init` *&rest ignore* [*Method of* `task`]

`:initial-args` [*Method of* `task`]

`:initial-method` [*Method of* `task`]

`:kill` [*Method of* `task`]

Kill a task. The task is deactivated first.

`kill-process` *process* [*Function*]

Really kill a process, no matter what its state is. Unwinds are handled.

`:name` [*Method of* `task`]

`:period` [*Method of* `task`]

`:restart-time` [*Method of* `task`]

`:schedule-type` [*Method of* `task`]

`:state` [*Method of* `task`]

`swap-in-scheduler` [*Function*]

Allow the scheduler to run another task.

`swap-in-scheduler-if-necessary` *task* [*Function*]

Allow the scheduler to run another task unless we are the task that is going to be run.

`task-active-p` *task* [*Function*]

Non-nil if the task is active.

`task-dont-swapout` *task* [*Function*]

`task-wait` *wait-string interval fn &rest args* [*Function*]

Wait until a specific event occurs. 'fn' is tested every 'interval' in the scheduler process. The first argument to 'wait-fn' is always the time at which the function is being called. The function can return T, or the time at which the task should wake up (maybe past or future)

`task-wait-for-interval` *interval* [*Function*]

Wait until a time interval has passed.

`task-wait-until-time` *time* [*Function*]

Wait until a specific time.

`tsend` *task &rest args* [*Function*]

Send a message to a named task.

3.2 Timing

`*cpu-usec/internal-time*` [*Variable*]

`*time-units-per-second*` [*Constant*]

Number of internal time units per second.

`1-second-compute` [*Function*]

This takes one cpu-second on an Explorer-II.

`1/5-second-compute` [*Function*]

This takes 1/5 cpu-second on an Explorer-II.

`base-time` [*Function*]

Base time (in seconds).

`brief-time-stamp` *internal-time &optional (stream nil) (current-time (current-time))* [*Function*]

Prints only those aspects of 'internal-time' which differ from the current time. Never prints seconds.

`cpu-usec->internal-time` *usec* [*Function*]

`cpu-usec/internal-time->minutes/cpu-sec` *time* [*Function*]

`current-time` [*Function*]

Elapsed time (in internal time!).

`estimated-time` [*Function*]

Return a quick approximation of the time (in internal time units).

`exact-time` [*Function*]

Return the time as exactly as it can be determined (in internal time units).

`exact-time-stamp` *internal-time &optional (stream nil) (base-time (base-time))* [*Function*]

Prints 'internal-time' in the format [M]M/[D]D [H]H:MM.SS.

`free-operations` *&body body* [*Macro*]

Evaluate forms, but don't charge CPU time to the process doing the evaluation.

`hours->internal-time` *hours* [*Function*]

`internal-time->hours` *internal-time* [*Function*]

`internal-time->minutes` *internal-time* [*Function*]

`internal-time->seconds` *internal-time* [*Function*]

`internal-time->useconds` *internal-time* [*Function*]

`minutes->exact-internal-time` *minutes* [*Function*]

`minutes->internal-time` *minutes* [*Function*]

`minutes/cpu-sec->cpu-usec/internal-time` *time* [*Function*]

`parse-to-internal-time` *time-string* [*Function*]

Parses 'time-string' into internal-time format.

`real-time` [*Function*]

Current time (in seconds).

`seconds->internal-time` *seconds* [*Function*]

`time-only-stamp` *internal-time &optional (stream nil) (base-time (base-time)) print-seconds* [*Function*]

Prints 'internal-time' in the format [H]H:MM or [H]H:MM.SS if print-seconds is non-nil.

`time-stamp` *internal-time &optional (stream nil) (base-time (base-time)) print-seconds* [*Function*]

Prints 'internal-time' in the format [M]M/[D]D [H]H:MM or [M]M/[D]D [H]H:MM.SS if print-seconds is non-nil.

`useconds->internal-time` *usec* [*Function*]

`useconds->minutes` *usec* [*Function*]

`useconds->seconds` *usec* [*Function*]

3.3 Output Functions

`continuation-format` *format-string &rest args* [*Function*]

Like 'label-format' except that no time or task is printed (but space is left for them). Usefull for continuing 'label-format' messages.

`debug-format` *format-string &rest args* [*Function*]

Prints debugging messages to the Lisp Listener pane in the Phoenix system.

`label-format` *format-string &rest args* [*Function*]

Formats 'format-string' and args on the Pheonix message label pane and the trace pane.

`label-format?` *predicate format-string &rest args* [*Function*]

A conditional version of 'label-format'.

`popup-stop` *&optional format-string &rest args* [*Function*]

Stop the system immediately from within an execution method.

`task-format` *format-string &rest args* [*Function*]

Like 'label-format' except that the task-name and exact time are also printed.

Chapter 4

Phoenix Task Scheduler

The scheduler is responsible for making sure each task gets allocated the appropriate amount of cpu time. Given the notion of the scheduler queue (from chapter 2) and task-time, the basic algorithm is:

1. Current task = pop(queue)
2. Current time = task-time(task)
3. Allow task to be scheduled on the cpu
4. When either: current task is a cpu-time task and a quantum break occurs (on the Explorer this happens every second), or the task isn't a cpu-time task and the task relinquishes control (by changing its restart-time and swapping in the scheduler), disable the task from being scheduled on the cpu.
5. For cpu-time tasks increment task-time(task) by the product of the task's cpu-time-to-real-time ratio and the amount of cpu time used between steps 3 and 4.
6. For non cpu-time tasks, set task-time(task) to the the task's restart-time.
7. Insert the task back into the queue. The queue is sorted by task-time (earliest first).
8. Repeat.

When an error occurs, the scheduler automatically stops to allow debugging (see section 2.5 for details).

4.1 Scheduler Implementation

This section describes the implementation of the Phoenix task scheduler on the Explorer. The scheduler is split into two parts. The first is responsible for selecting which task to execute and enabling that task. This part is implemented as a simple-process and is responsible for

overall control of the task scheduler. The second part runs every time a process associated with a task gets swapped out by the Explorer operating system. This occurs when either a timer interrupt occurs (once a second), or a process explicitly swaps itself out.

4.2 Top Level Scheduler Loop

This section describes the first part of the task scheduler implementation. The scheduler is an instance of the `task-scheduler` flavor, which provides instance variables for the scheduler queue and more. The scheduler is either running (allowing tasks to run) or stopped (not allowing tasks to run). When running, it executes tasks until a specific simulation time. The queue is a priority queue (implemented as a heap). The tasks are ordered by task-time.

The main loop is as follows:

```
;; If stopped, or currently running a task or no tasks available,
;;    do nothing.
If *current-task* is non-NIL Or we're stopped Or the queue is empty
    return

;; Find the next task ready to run.  (Since tasks can have a wait
;; function, it may not be the task at the head of the queue).
Loop
    task = head(queue)
    If task-time(task) > run-until-time then
        stopped = true
        return (from main loop)
    pop(queue)
    If task has no wait function, Or the wait function returns True,
         exit loop
    ;; If the wait function fails, reschedule the task for later
    task-time(task) = task-time(task) + task-wait-interval(task)
    insert(task, queue)
Endloop
;; At this point, the task can run.
update screen (timestamp and scheduler status-window)
wait-function(task) = NIL
*current-task* = task
start-cpu-time(task) = current-cpu-time(task)
enable(task)
```

This pseudo-code selects which task to run, then enables it.

4.3 When Processes Swap Out

The second part of the scheduler implementation disables a task's process and manages task-time accounting. The Explorer operating system has been modified to call the `after-task-execution` function every time a process associated with a task swaps out.

The pseudo-code for this part is as follows: (NOTE: process is the process associated with *current-task*)

```
;; Allow the debugger to run.
If process is in an error state return
;; Allow i/o.
If *current-task* has disabled swapping
      (ie., dont_swapout_task(task) = true)
   return

If *current-task* is not a cpu-time task
   If restart_time(task) has been set or
                         task returned :OK-TO-RESTART-AT-SAME-TIME
       task_time(task) = restart_time(task)
       ;; Allow another task to be scheduled.
       insert(task, queue)
       disable(process)
       *current-task* = NIL
       return
   Else
       return (allow task to continue)

If *current-task* is a cpu-time task
   delta = current_cpu_time(task) - start_cpu_time(task)
   If suspend_cpu_accounting(task)
       Then task_time(task) = task-time(task) +
                              cpu_time_adjustment(task)
       Else task_time(task) = task_time(task) +
                              delta*cpu_ratio(task) +
                              cpu_time_adjustment(task)
   ;; A task can tell the scheduler to modify its cpu usage
   cpu_time_adjustment(task) = 0
   insert(task, queue)
   disable(process)
   *current-task* = NIL
```

If the process was not disabled, the Explorer operating system would continue to execute it. When a task should be discontinued (for now), the process should be disabled and *current-task* should be set to NIL (to allow another task to be executed).

It is difficult to tell when a process is in an error state. To do this, we put advice around the function that enters the error handler.

When a cpu-time task wants to do i/o, it cannot be disabled during the i/o wait. To allow i/o from a cpu-time task, a task is given the ability to say "don't swap me out." But, when doing i/o, the task should not be charged for the time it takes a person to enter the input. To provide for this, a task can suspend cpu time accounting.

4.4 Portability Issues

How difficult would it be to implement this task model on some other hardware? If each task didn't have to run in its own process, it would be easy. Unfortunately, this isn't the case because tasks (especially cpu-time tasks) can be interrupted at any time and need to save current state on the runtime stack. The fine-grained process control needed to deal with non-cpu-time tasks is easy, because those tasks *know* when they need to be swapped out. Cpu-time tasks are more difficult because they need to be disabled based on some external event (timer interrupt or the passage of a specific amount of cpu time). Since these events are operating system dependent, changes may need to be made to the operating system. I can think of one implementation that will work without operating system modifications. The implementation requires several things: the OS works on a round-robin scheme, each process can enable/disable another process, and each process has access to the cpu-accounting of another process. The idea is to have a scheduler task with the following loop:

```
Loop
  enable appropriate task
  repeat
      ;; If this can't be done, try sleep(small-amount-of-time)
      Swap out to allow round-robin scheduling to occur.
      If *current-task* is cpu-time, disable it.
  until *current-task* is disabled
  update accounting
Endloop
```

This partial solution doesn't deal with issues of user interface and shared data.

4.5 Errors and Debugging

Each instance of `fire-system` has its own scheduler instance. Thus, it is possible to have more than one Phoenix running at a time. When the Explorer warm boots, it reinstalls the original version of the Explorer scheduler. *Do not try to run Phoenix after a warm boot!*[1] The "bomb-drop" beep indicates that an error has occurred in the scheduler. To debug this problem, go to the 'process-display' configuration. To ignore it, reset the system with `Reset` command.

4.6 Programming Interface

It is possible to interact with the scheduler programatically. The function `current-scheduler` returns the current scheduler flavor instance. The methods that can be used are described in chapter 5.

[1] Actually, one might have reasonable success using the `install-task-scheduler` function to reinstall the Phoenix scheduler after a warm boot, but it is not for the fainthearted (or those who have not saved all their editor buffers).

Chapter 5

Scheduler Reference Manual

`*current-scheduler*` [*Variable*]

A pointer to the current instantiation of the Phoenix Task Scheduler.

`*current-task*` [*Variable*]

A pointer to the currently executing task.

`*previous-task*` [*Variable*]

A pointer to the last task executed.

`*query-task-errors*` [*Variable*]

If non-NIL, errors are handled interactively. The default is T.

`*scheduler-error-message*` [*Variable*]

Holds the error message when a task error occurs.

`*scheduler-error-where*` [*Variable*]

Holds the location of the error when a task error occurs.

`*scheduler-swapin-count*` [*Variable*]

A counter containing the number of times the Phoenix scheduler runs.

`current-scheduler` [*Function*]

Returns the task scheduler of the current fire system.

`:dequeue-task` *task* [*Method of* `task-scheduler`]

`dont-swapout-function` *fn* [*Macro*]

Adds advice to the function which prevents it (or more correctly, the task running it) from being swapped out while it is being run.

`:edit-parameters` [*Method of* `task-scheduler`]

`:enqueue-task` *task &optional (time (current-time))* [*Method of* `task-scheduler`]

`in-current-task-p` [*Function*]

Return T if currently executing in *current-task*.

`:kill` [*Method of* `task-scheduler`]

Kill the scheduler.

`:macro-step-scheduler` *&optional (n 1)* [*Method of* `task-scheduler`]

Run the system for 'n' macro steps. If 'n' is NIL, run it forever.

`:reset` *&rest ignore* [*Method of* `task-scheduler`]

Set the scheduler to an initial state.

`:run` *n* [*Method of* `task-scheduler`]

Run the system for 'n' minutes. If 'n' is NIL, run it forever.

`:run-until-time` [*Method of* `task-scheduler`]

`:single-step` [*Method of* `task-scheduler`]

Run the system for the smallest time interval possible; in other words, run the system for one internal-time unit or until a task swaps out.

`:stop` *&optional (wait nil)* [*Method of* `task-scheduler`]

Stop the scheduler.

`:trace` [*Method of* `task-scheduler`]

`without-task-swapout` *&body body* [*Macro*]

Execute 'body' without ever swapping this task out for another task. Code that does user i/o or grabs locks should be included here.

Chapter 6

Map

Maps in Phoenix represent topographical features for a land area. The features include ground-cover, elevation, roads, rivers, buildings and fire. The map of Yellowstone is approximately 75 kilometers square. It uses different representations for the various types of information present. The rest of this chapter describes these representations with some illustrative examples.

6.1 Map Basics

6.1.1 Units and Positions

All units of distance are represented in meters. A position on the map is represented as a `point` data structure. A `point` structure contains the x and y coordinates of a point in meters. The position 0,0 is the upper left of a map. Unless otherwise specified, map access functions take positions as a `point`. In general, the values of the x and y coordinates should be integers in the range [0, `*width-in-meters*`] and [0, `*height-in-meters*`] respectively.

To create a point at position x=123 and y=45000:

```
(setf p (point 123 45000)) ==> (123 . 45000)
(point-x p) ==> 123
(point-y p) ==> 45000
```

The x and y components of a point can be accessed and set with the functions `point-x` and `point-y`.[1] The map reference chapter describes many of the point functions and geometry functions that operate on points.

[1] Points are just represented as cons cells, but that doesn't mean you should use `car`, `cdr` and `cons` to manipulate or create them.

6.1.2 Grid Arrays

A `grid-array` is an array-based representation of a map. Each array element corresponds to a square region of the map. The size that an element corresponds to is called the *grid-array-size* and the *resolution* of the grid array is the log of the size to base 2. For instance, the size 256 corresponds to the resolution 8. If the world is 76,000 meters on a side (as is our Yellowstone map), a grid-array representation of Yellowstone at resolution 8 (size=256meters) is a 300x300 array. At resolution 7, the array size is 600x600. `Grid-arrays` are typed arrays. The term *cell* is used to denote an element of a grid-array.

6.2 The Map Representation

A map is represented as an instance of the `firemap` flavor. There are a large number of functions and methods that operate on firemaps. The set of functions is described in chapter 7.

6.2.1 Ground Cover

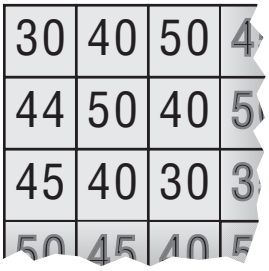
The map representation currently contains 11 types of ground cover. They are agriculture (fields), chapparal, hardwood, lake, marsh, meadow, rocky, softwood, suburban, and urban areas. The eleventh is named boundary, and is used as a sentinel around the borders of the map. Ground cover is represented as a grid array at resolution `*gc-cell-resolution*` (8). So each cell in the ground-cover map corresponds to an area `*gc-cell-size*` (256 meters) on a side. Each element of the grid-array is of type `(mod 16)`. Given a firemap named `firemap`, the functions `cell-ground-cover` and `ground-cover-name` can be used to access ground cover as follows:

```
(cell-ground-cover (point 34000 12000) firemap) ==> 3
(ground-cover-name 3) ==> "Lake"
```

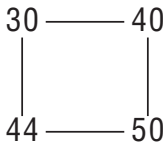
There are eleven constants, each corresponding to the different type of ground cover: `*gc-agriculture*`, `*gc-boundary*`, `*gc-chapparal*`, `*gc-hardwood*`, `*gc-lake*`, `*gc-marsh*`, `*gc-meadow*`, `*gc-rocky*`, `*gc-softwood*`, `*gc-suburban*` and `*gc-urban*`.

6.2.2 Elevation

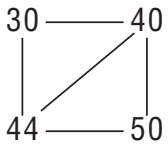
Elevation is represented as a grid array of type `(mod 65536)` at resolution `*elevation-cell-resolution*` (8). Elevations are represented in meters above sea level. Since the elevation data is stored as a grid, it is necessary to do some form of interpolation to prevent large discontinuities. For example, suppose that part of the elevation array looks like



Since each cell represents an area of 256 meters on a side, uninterpolated the elevation at $(253\ .\ 0) = 30$, $(254\ .\ 0) = 30$, $(255\ .\ 0) = 30$ and $(256\ .\ 0) = 40$! Notice the discontinuity at the cell boundary. Phoenix interpolates elevation as follows: Imagine the cell at $(0\ .\ 0)$ from above. The elevation at each corner of the cell is fixed by the elevation data.



Since three points define a plane, a diagonal through the cell defines two planes:



When you ask for the elevation of a position, the elevation is interpolated. This technique guarantees that elevation is continuous over the entire map (If there are supposed to be discontinuities–cliffs–they are lost). The function `cell-elevation` returns the interpolated elevation (rounded to meters).

```
(cell-elevation (point 0 . 0) firemap) ==> 30
(cell-elevation (point 128 . 0) firemap) ==> 35
(cell-elevation (point 256 . 0) firemap) ==> 40
```

`cell-approx-elevation` returns the uninterpolated value directly from the grid array.

```
(cell-approx-elevation (point 0 . 0) firemap) ==> 30
(cell-approx-elevation (point 128 . 0) firemap) ==> 30
(cell-approx-elevation (point 256 . 0) firemap) ==> 40
```

6.2.3 Roads, Rivers and Buildings

Roads, rivers and buildings are represented as *features*. Each feature type has a specific width. Currently there are eleven types of features.

`*f-building*` Buildings. (8 meters wide)

`*f-fireline*` Fireline. (4 meters wide)

`*f-river128*` 128 meter wide river.

`*f-river64*` 64 meter wide river.

`*f-river32*` 32 meter wide river.

`*f-river16*` 16 meter wide river.

`*f-river8*` 8 meter wide river.

`*f-river4*` 4 meter wide river.

`*f-road16*` 16 meter wide road.

`*f-road8*` 8 meter wide road.

`*f-road4*` 4 meter wide road.

Each feature type has a specific width (while not a requirement, feature widths are powers of two for historical reasons). In the map, features are represented as directed line segments. The `feature-edge` structure contains a feature type, start point and end point. Since each feature-edge has a width (the width of the feature type), the shape of a feature edge is shown in Figure 6.1.

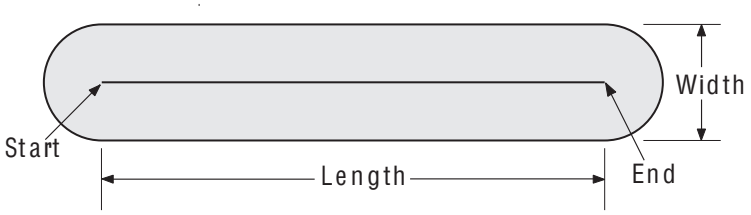


Figure 6.1: A Feature Edge

A point is on a feature-edge iff the distance from the point to the line segment from the start point to the end point is less then 1/2 the width of the feature.

Feature types are split into two types: static and dynamic. Static features never change while the program is running. Dynamic features do change. (Currently, the only dynamic feature is fireline.) There are several reasons for this separation:

- Firemaps are large. By guaranteeing that some features are static, we can reuse the same static data structure objects in many firemaps.
- Space/Time tradeoff. The data structures used to represent static features are large in terms of space, but extremely fast to index. If each copy of the firemap was large (no data sharing), this tradeoff couldn't be made. Dynamic features are indexed more slowly, but in a more space efficient data structure.

- Cleanup. By marking some parts of the map as dynamic, it is easy to reset the map to its initial state—just clear the dynamic parts.

6.2.4 Static Features

There are two indexing methods for static features. The first method answers the question "What feature-edge is at a specific position in the map?" The second answers the question "Given a feature edge, what are the other 'adjacent feature edges?"

To facilitate the position to feature-edge mapping, a grid array at resolution `*feature-cell-resolution*` (8) is used. Stored in each cell is a list of all feature-edges (containing static features) that *touch* the cell in any way. A feature-edge touches a cell if any part of the rectangle defined by the segment and width cover any part of the cell.[2] The question "What feature-edge is at a specific point?" is answered by searching the list of feature-edges in the cell to see if the point is on any edge. This algorithm is quite efficient. Most cells (90 percent or more) contain no features, so no searching is done, and when a cell does contain features, there are usually only two or three. To answer the question "Is a point on a feature?" requires finding the distance from the point to the line segment of the edge. This can also be implemented very efficiently.

Since it is possible for a point to be on more than one feature, a precedence of features is defined. The order of precedence is building, roads (wide to narrow), river (wide to narrow). This way, if a road crosses a river, a point that is on the road over the river is considered to be on the road.

To answer the question "Given an edge, what edges are connected to it" several data structures are used. All edges start and end at a `vertex`. Each `vertex` contains a position and a pointer to all of the edges that start or end at that position. Since edges point to vertices and vice versa, it is possible to follow roads and rivers. For this to work properly anytime *two edges can only meet at a vertex, edges are not allowed to cross.* So when features are added to the map, the system automatically creates vertices at the appropriate places and splits single edges into multiple edges. Figures 6.2 and 6.3 illustrate this.

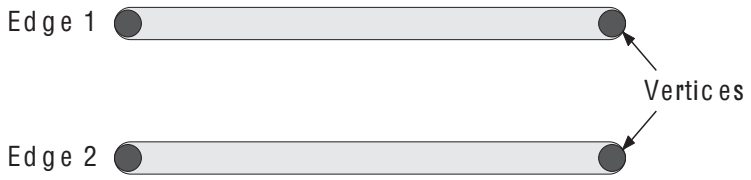


Figure 6.2: Before Adding Edge

Adding a single edge of feature results in two existing edges being split into four edges, and the new edge into three edges. Four new vertices are created.

[2] So the circular positions at the edge of the cell aren't included. This isn't a problem because of the way feature-edges are connected.

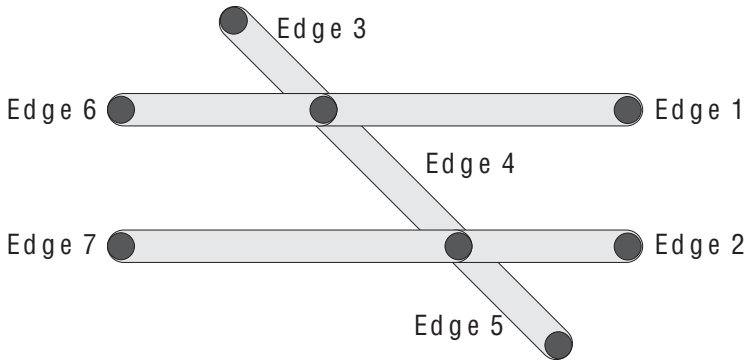


Figure 6.3: After Adding Edge

6.2.5 Dynamic Features

Dynamic features are also represented with `feature-edges` though indexing them is different. Instead of using a grid-array to index features by position, a simple sparse array representation is used. We use a vector representation for the array.To access the list of static-features in a cell at a specific point:[3]

```
(getf (svref vector (round (point-x point) *feature-cell-size*))
      (round (point-y point) *feature-cell-size*))
```

This saves significant space (a 300 element vector instead of a 300x300 array) at a cost of some speed. Since there are usually 5—10 firemaps being used at a time (in the current system), we deemed that this was an appropriate tradeoff. It works reasonably well because (in current applications) the vector is sparse.

We do not represent vertices for dynamic features. Dynamic feature may intersect at non-end points. The cost of this decision is that tracing along dynamic features is more difficult; the advantage is that the cost of creating dynamic features is low.

6.3 Fire

Fire is represented using a grid-array at resolution 7. Thus, the grain size of a fire is 128 meters. The element type of the fire grid array can be either: `bit` , `(mod 16)fixnum` or `T`.

Currently, there are five fire states: no fire, low fire, hot fire, smoldering fire and burned out. These states correspond to different points in a fire's burn-cycle. Each state is a number, corresponding to the constants `*cs-no-fire*` (0), `*cs-low-fire*` (1), `*cs-hot-fire*` (2), `*cs-smoldering-fire*` (3) and `*cs-burned-out-fire*` (4).

Each of the grid-array element-types can be used to represent fire to differing levels of detail.

[3] Actually, instead of `round` we use `(lsh ... *feature-cell-resolution*)`. The use of power of two math saves a significant amount of cpu time.

bit This type represents either the presence or absence of fire. The absence of fire is `*cs-no-fire*`; the presence of fire is `*cs-low-fire*`.

(mod 16) With this type each of the five fire states can be represented.

fixnum This type uses the low order bits (four bits) to represent fire state, and the high order bits to store other information. Currently, the 20 high order bits are used as flags. Functions and variables can be used to mask out the high and low order bits.[4]

T With element type `T`, the data in the grid-array is either a `fixnum` as above, or an instance of a `fire-info` structure. `Fire-info` structures are used by the fire simulation. This structure contains a fixnum (fire-state) as above, and information used by the fire simulation such as the ignition time of cell.

When you create instances of a `firemap`, you can specify what element type the fire grid-array should be, or specify that a fire array shouldn't even be created.

6.4 Creating and Editing Firemaps

To create a firemap, create an instance of the `firemap` flavor. When a firemap is created, the new map shares information about ground cover, elevation and static features from a default firemap (`*default-firemap*`). The default firemap is a map of Yellowstone National Park. Firemaps can be read and written from disk, edited and examined. Care must be taken when editing maps because all maps have common data-structures. Each map gets its own set of dynamic features (initially empty) and fire representation. When instantiating a map, the type (if any) of the fire representation should be specified.

```
;; create a map of Yellowstone with a bit fire representation
(make-instance 'firemap :fire-element-type 'bit)

;; create a map of Yellowstone with no fire representation
(make-instance 'firemap :fire-element-type NIL)

;; create an empty map, and then fill if from a disk file
(setf map (make-instance 'firemap :initialize-from NIL))
(send map :load-map "map-filename")
```

The `firemap` creation options are described in chapter 7.

When a firemap is no longer needed, it is best to reclaim the memory used by the map (they are large) by sending it a `:deallocate` message. In order to return a map to its initial state (no fire, no dynamic features), send it an `:erase-fire` message.

The basic map access functions are:

[4]The use of `fixnum` instead of `(mod 24)` is based upon efficiency concerns. The drawback is that it makes the code non-portable because other Common Lisp implementations might have different length fixnums.

`cell-ground-cover` *point map* returns the ground cover as a number (eg., the value of `*gc-hardwood*`).

`cell-elevation` *point map* returns the interpolated elevation in meters.

`cell-feature` *point map* returns the feature at *point* as a number (eg., the value of `*f-road4*`) or `Nil` if there is no feature.

`cell-fire-state` *point map* returns the fire state as a number. The type of the number depends on the fire element type. If the fire element type is `T` and the cell contains a `fire-info` structure, the fire state from the structure is returned as a fixnum.

Some access functions are implemented as functions, some as methods, and some as both. The choice of technique to use for each is based on efficiency, convenience and style. A complete list of map access routines can be found in Section 7.3.

6.5 The Real World Firemap

There is one firemap which is used to represent the "real world". When tasks look into the world, they should look into the real world firemap. The function `real-world-firemap` returns that map. Tasks can change the "real world." A typical change is the placement of fireline (although this should be done be sending a message to `fire-system` rather than `real-world-firemap`).

6.6 Geometry

Phoenix provides many functions for geometric reasoning over the map. These functions range from simple unit conversions to basic geometry (do two line segments intersect?) to region-growing algorithms. Section 7.8 describes these functions.

Chapter 7

Map Reference Manual

7.1 Firemap Flavor

`firemap-dynamic-edges` *firemap* [*Function*]
`:dynamic-edges` [*Method of* `firemap`]
`firemap-edge-vector` *firemap* [*Function*]
`:edge-vector` [*Method of* `firemap`]
`firemap-elevation` *firemap* [*Function*]
`:elevation` [*Method of* `firemap`]
`firemap-filename` *firemap* [*Function*]
`:filename` [*Method of* `firemap`]
`firemap-fire` *firemap* [*Function*]
`:fire` [*Method of* `firemap`]
`firemap-fire-extents` *firemap* [*Function*]
`:fire-extents` [*Method of* `firemap`]
`firemap-firemap-windows` *firemap* [*Function*]
`:firemap-windows` [*Method of* `firemap`]
`firemap-ground-cover` *firemap* [*Function*]
`:ground-cover` [*Method of* `firemap`]
`firemap-objects-to-display` *firemap* [*Function*]
`:objects-to-display` [*Method of* `firemap`]
`firemap-static-edges` *firemap* [*Function*]
`:static-edges` [*Method of* `firemap`]
`firemap-update-windows` *firemap* [*Function*]
`:update-windows` [*Method of* `firemap`]

`firemap-vertex-vector` *firemap* [*Function*]

`:vertex-vector` [*Method of* `firemap`]

7.2 Map Definitions

7.2.1 Elevation

`*elevation-cell-resolution*` [*Constant*]

`*elevation-cell-size*` [*Constant*]

Size of elevation grid cell side (in meters).

7.2.2 Fire States

`*cs-burned-out-fire*` [*Constant*]

`*cs-low-fire*` [*Constant*]

`*cs-hot-fire*` [*Constant*]

`*cs-mask*` [*Constant*]

`*cs-no-fire*` [*Constant*]

`*cs-smoldering-fire*` [*Constant*]

`*fire-cell-resolution*` [*Constant*]

`*fire-cell-size*` [*Constant*]

Size of fire grid cell side (in meters).

`*fire-names*` [*Constant*]

`*fs-features-present*` [*Constant*]

`*fs-ignitable*` [*Constant*]

`*fs-mask*` [*Constant*]

`*fs-not-ignitable*` [*Constant*]

`*number-of-fire-states*` [*Constant*]

`deffire` *symbol number name &key color-character b&w-character color* [*Macro*]

`fire-name` *fire-state* [*Function*]

Returns a string which describes 'fire-state'.

7.2.3 Fire Info

`allocate-fire-info` [*Function*]

`create-fire-info` [*Function*]

`fire-burn-state` *f* [*Macro*]

Return the burn state from a fire-info fixnum. *cs-no-fire* -> *cs-burned-out-fire*

`fire-flag->number` *fs* [*Function*]

`fire-flags` *f* [*Macro*]

Returns the flag part of a fire-info fixnum.

`fire-info-burn-state` *fire-info* [*Macro*]

`fire-info-change-time` *fire-info* [*Function*]

`fire-info-ignite-time` *fire-info* [*Function*]

`fire-info-point` *fire-info* [*Function*]

`fire-info-state` *fire-info* [*Function*]

`free-fire-info` *arg* [*Function*]

`firep` *f* [*Macro*]

Return T if there is fire in a fire-info fixnum (includes *cs-burned-out-fire*).

`ignitable-p` *f* [*Macro*]

Returns T if there is no fireline in a fire info cell.

`live-fire-p` *f* [*Macro*]

Return T if there is live fire in a fire-info fixnum (does not include *cs-burned-out-fire*).

`not-ignitable-p` *f* [*Macro*]

Returns T if there is fire line in a fire info cell.

`pfi` *&optional (point (spum))* [*Function*]

P rint F ire I nfo at 'point' to *standard-output*. Used for debugging.

7.2.4 Features

`*all-features-flag*` [*Constant*]

`*dynamic-feature-flags*` [*Variable*]

`*f-building*` [*Constant*]

`*f-fireline*` [*Constant*]

`*f-river128*` [*Constant*]

`*f-river16*` [*Constant*]

`*f-river32*` [*Constant*]

`*f-river4*` [*Constant*]

`*f-river64*` [*Constant*]

`*f-river8*` [*Constant*]

`*f-road16*` [*Constant*]

`*f-road4*` [*Constant*]

`*f-road8*` [*Constant*]

`*feature-cell-resolution*` [*Constant*]

`*feature-cell-size*` [*Constant*]

Size of feature grid cell side (in meters).

`*feature-names*` [*Constant*]

Array of feature names.

`*feature-overlay-order*` [*Variable*]

The precedence of features.

`*feature-widths*` [*Constant*]

Array of feature widths (in meters).

`*number-of-features*` [*Variable*]

`*point-feature-flags*` [*Variable*]

`*river-flags*` [*Variable*]

`*road-flags*` [*Variable*]

`*road-or-uncrossable-river-flags*` [*Variable*]

`*static-feature-flags*` [*Variable*]

`*uncrossable-river-flags*` [*Variable*]

`deffeature` *symbol name number width &key roadp riverp uncrossable-river-p b&w-character color-character dynamicp color b&w minimum-display-size pointp* [*Macro*]

`feature-name` *feature* [*Function*]

`feature-width` *feature* [*Function*]

`roadp` *feature* [*Function*]

`riverp` *feature* [*Function*]

7.2.5 Ground Cover

`*gc-agriculture*` [*Constant*]

`*gc-boundary*` [*Constant*]

`*gc-cell-resolution*` [*Constant*]

`*gc-cell-size*` [*Constant*]

Size of ground cover grid cell side (in meters).

`*gc-chapparal*` [*Constant*]

`*gc-hardwood*` [*Constant*]

`*gc-lake*` [*Constant*]

`*gc-marsh*` [*Constant*]

`*gc-meadow*` [*Constant*]

`*gc-rocky*` [*Constant*]

`*gc-softwood*` [*Constant*]

`*gc-suburban*` [*Constant*]

`*gc-urban*` [*Constant*]

`*ground-cover-names*` [*Constant*]

`*ground-cover-types*` [*Constant*]

`*number-of-ground-covers*` [*Variable*]

`burnable-ground-cover-p` *ground-cover* [*Function*]

Returns T if 'ground-cover' is burnable (ie., it has a fuel model).

`defground-cover` *symbol number name &key type color b&w b&w-texture* [*Macro*]

`ground-cover-name` *gc* [*Function*]

`ground-cover-type` *gc* [*Function*]

`vegatation-ground-cover-p` *gc* [*Function*]

7.3 Map Access Functions and Methods

`defmapfn` *slot &key (slot-postfix slot) (resolution (quote *gc-cell-resolution*))* [*Macro*]

Define a set of map slot access functions at a specific resolution. Specifically, this defines a standard "point" reader:
CELL-{SLOT} <point> <map>
an "xy" reader
CELL-{SLOT}-X-Y <x> <y> <map>
and a standard "point" writer function which is also the setf method for the standard reader. If 'slot-postfix' is specified it is used in place of 'slot' to generate the function names.

7.3.1 Elevation

`cell-approx-elevation` *point map* [*Function*]

`cell-elevation` *point map* [*Function*]

Find the real elevation of a point to the nearest meter.

`:highlight-elevation` *el &optional (range 0)* [*Method of* `firemap`]

`:set-cell-elevation` *point new-elevation* [*Method of* `firemap`]

7.3.2 Features

`cell-boundary-crossed-multiply-by-features-p` *point resolution firemap &optional (features *all-features-flag*)* [*Function*]

Return T if the cell boundary is crossed more than once by features.

`cell-dynamic-edges` *point map* [*Function*]

`cell-dynamic-feature` *point map* [*Function*]

`cell-dynamic-feature-edge` *point map* [*Function*]

Return the line feature edge under 'point'.

`cell-dynamic-features-in-area` *point map resolution* [*Function*]

Return the set of dynamic features at 'point' and 'resolution' as a fixnum bit-array.

`cell-feature` *point map* [*Function*]

`:cell-feature` *point* [*Method of* `firemap`]

`cell-feature-edge` *point map* [*Function*]

`cell-feature-flag` *point map* [*Function*]

`cell-features-in-area` *point map resolution* [*Function*]

`cell-static-edges` *point map* [*Function*]

`cell-static-feature` *point map* [*Function*]

Return the feature under 'point'.

`cell-static-feature-edge` *point map* [*Function*]

Return the line feature edge under 'point'.

`cell-static-features-in-area` *point map resolution* [*Function*]

Return the set of static features at 'point' and 'resolution' as a fixnum bit-array.

`:create-static-edge` *from-point to-point type &key (refresh t)* [*Method of* `firemap`]

Create a static edge between the specified points. If the new edge intersects any other edges at a non-vertex, create a new vertex at that intersection point.

`:delete-edge` *edge &key (refresh t)* [*Method of* `firemap`]

Delete an edge.

`:delete-edge-from-array` *edge* [*Method of* `firemap`]

Delete a static edge from a firemap.

`:delete-point-edges` [*Method of* `firemap`]

Delete all edges that start and end at the same vertex.

`:delete-point-feature-near-point` *point &rest ignore* [*Method of* `firemap`]

Delete a feature near 'point'.

`dynamic-edge-exists-p` *from to firemap* [*Function*]

`:find-edge-nearest-to-point` *point within-distance* [*Method of* `firemap`]

Find the edge closest to 'point' within a certain distance.

`find-point-on-some-feature-in-cell` *cell-point resolution feature-flags map* [*Function*]

Pick a random point on a feature that matches 'feature-flags' in the cell.

`:find-vertex-at-point` *point &key (create nil)* [*Method of* `firemap`]

`:find-vertex-nearest-to-point` *point within-distance* [*Method of* `firemap`]

Find the vertex closest to 'point' within a certain distance.

`fireline-in-cell-p` *point firemap resolution* [*Function*]

`:place-dynamic-feature` *from to type* [*Method of* `firemap`]

Draw a dynamic feature of 'type' from point to point.

`:place-edge-in-array` *edge* [*Method of* `firemap`]

Insert a static edge into the firemap.

`:place-static-edge` *from-point to-point type &key (refresh t)* [*Method of* `firemap`]

Create an edge of 'type' from point to point. Vertices are created at the endpoints if necessary. The new edge should NOT intersect other edges (except at endpoints).

`point-on-feature-of-type-p` *point firemap feature-flags* [*Function*]

Return the feature if 'point' is above a feature of a specified type.

`point-on-lake-p` *point map* [*Function*]

`point-on-road-p` *point map* [*Function*]

`river-in-cell-p` *point firemap resolution* [*Function*]

`road-in-cell-p` *point firemap resolution* [*Function*]

7.3.3 Ground Cover

`cell-ground-cover` *point map* [*Function*]

`:cell-ground-cover` *point* [*Method of* `firemap`]

`:set-cell-ground-cover` *point cover* [*Method of* `firemap`]

7.3.4 Fire

`cell-fire` *point map* [*Function*]

`cell-fire-burn-state` *point map* [*Function*]

Return the burn state at a point in a map.

`:cell-fire-burn-state` *point* [*Method of* `firemap`]

`:cell-fire-display-info` *point* [*Method of* `firemap`]

Return three values; state, change-time, ignite-time.

`cell-fire-flags` *point map* [*Function*]

Return the fire flags for a cell.

`cell-fire-info-structure` *point map* [*Function*]

Create a fire-info structure, or return the existing one for 'point' in 'map'.

`cell-fire-state` *point map* [*Function*]

Return the full state and flags at a point in the map.

`cell-ignite-time` *point* [*Function*]

Return the ignite time of a cell. Return most-positive-fixnum if not known.

`:erase-fire` *&optional confirm (refresh t)* [*Method of* `firemap`]

Reset the map to a nice, clean, no-fire state. This really should be called :reset-all-dynamic-edges.

`:ignite-cell` *&rest args* [*Method of* `firemap`]

`:update-cell-fire` *point new-state &optional old-state* [*Method of* `firemap`]

`:set-cell-fire-burn-state` *point &optional (new-state *cs-low-fire*) (refresh t)* [*Method of* `firemap`]

`:set-cell-fire-state` *point &optional (new-state *cs-low-fire*) (refresh t)* [*Method of* `firemap`]

7.4 Other Firemap Variables and Routines

`*default-firemap*` [*Variable*]

Default-firemap for ground cover and elevation.

`*default-map-file*` [*Variable*]

`*height-in-meters*` [*Constant*]

`*width-in-meters*` [*Constant*]

`all-firemaps` [*Function*]

`:check-for-non-vertex-intersections` [*Method of* `firemap`]

See if there are any interesections that don't meet at a vertex.

`:deallocate` [*Method of* `firemap`]

Return the grid arrays.

`:delete-window` *window* [*Method of* `firemap`]

`:draw-objects` [*Method of* `firemap`]

`:load-map` *&optional file confirm* [*Method of* `firemap`]

`real-world-firemap` [*Function*]

Returns the real-world-firemap of the current fire system.

`:rebuild-vertex-and-edge-vectors` [*Method of* `firemap`]

Given a firemap static-edge-vector, construct a new vertex vector and clean up the edge vector. Vertices and vertex numbers are recomputed from scratch. Everything is recomputed from just the edge-type, start-point and end-point.

`:refresh` [*Method of* `firemap`]

`:save-map` *&optional file confirm* [*Method of* `firemap`]

`:validate-vertex-and-edge-vectors` [*Method of* `firemap`]

Do some consistency checking over features.

7.5 Grids

`grid-array-aref` *array x y resolution* [*Function*]

Use 'x' and 'y' as indices into a grid array. Can be used with 'setf'.

`grid-array-ref` *array point resolution* [*Function*]

Use 'point' as an index into a grid array at the specified resolution. Can be used with 'setf'.

`set-grid-array-aref` *array x y resolution value* [*Function*]

Use 'x' and 'y' as indices to set an element in a grid array.

`set-grid-array-ref` *array point resolution value* [*Function*]

Use 'point' as an index to set an element in grid array at the specified resolution.

`truncate-to-grid` *n* [*Macro*]

7.6 Vertices and Feature Edges

`copy-feature-edge` *object* [*Function*]

`edges-meet-at-vertex-p` *e1 e2* [*Function*]

Return true if the edges share a vertex.

`edge-cell-list` *edge resolution* [*Function*]

Return a list of the cells touched by 'edge'. Each cell is returned only once. The order is random.

`edges-in-area` *edges point resolution* [*Function*]

`feature-edge-bot-point-from` *feature-edge* [*Function*]

`feature-edge-bot-point-to` *feature-edge* [*Function*]

`feature-edge-cen-point-from` *feature-edge* [*Function*]

`feature-edge-cen-point-to` *feature-edge* [*Function*]

`feature-edge-from-point` *feature-edge* [*Function*]

`feature-edge-from-vertex` *feature-edge* [*Function*]

`feature-edge-index` *feature-edge* [*Function*]

`feature-edge-length` *feature-edge* [*Function*]

`feature-edge-p` *object* [*Function*]

`feature-edge-plist` *feature-edge* [*Function*]

`feature-edge-to-point` *feature-edge* [*Function*]

`feature-edge-to-vertex` *feature-edge* [*Function*]

`feature-edge-top-point-from` *feature-edge* [*Function*]

`feature-edge-top-point-to` *feature-edge* [*Function*]

`feature-edge-type` *feature-edge* [*Function*]

`feature-of-type-in-area-p` *point firemap feature-flags resolution* [*Function*]

`find-point-on-edge-in-cell` *edge cell-point resolution* [*Function*]

Given an edge, return some point on that edge within a cell. If the edge doesn't enter the cell, return NIL.

`point-distance-from-edge-squared` *point edge* [*Function*]

Return the square of the distance from the edge centerline to 'point'.

`point-on-edge-p` *point edge* [*Function*]

Return T if point is on 'edge'.

`type-of-connection-between-vertices` *v1 v2* [*Function*]

`vertex-edges` *vertex* [*Function*]

`vertex-index` *vertex* [*Function*]

`vertex-p` *object* [*Function*]

`vertex-point` *vertex* [*Function*]

7.7 Conversion Functions

`*feet-per-km*` [*Constant*]

`*km-per-mile*` [*Constant*]

`*meters-per-chain*` [*Constant*]

`chains/hour->meters/internal-time` *n* [*Function*]

`chains/hour->meters/minute` *n* [*Function*]

`chains/hour->meters/second` *n* [*Function*]

`chains->km` *chains* [*Function*]

`chains->meters` *chains* [*Function*]

`degrees->radians` *degrees* [*Function*]

`km->feet` *km* [*Function*]

`km->miles` *km* [*Function*]

`km/hour->meters/internal-time` *n* [*Function*]

`km/hour->meters/second` *km* [*Function*]

`miles->km` *miles* [*Function*]

7.8 Geometry Functions

`*essentially-zero-threshold*` [*Variable*]

To test if a point is on a line, we find its distance from the line and if it's essentially zero [it might be non-zero due to round-off errors], we deem it to be on the line. This number sets the threshold for being essentially zero.

`angle-between-points` *p1 p2* [*Function*]

Return the angle of the vector from 'p1' to 'p2' (in radians).

`area-of-triangle` *p0 p1 p2* [*Function*]

Return the area of the triangle p0 p1 p2.

`average-point` *new-point &rest points* [*Function*]

'new-point' is set to be the rounded arithmetic mean of 'points'. 'new-point' is not included in the average.

`cell-center-point` *cell-point resolution* [*Function*]

Return the center point of a cell at 'resolution'.

`center-point` *point* [*Function*]

`copy-extent` *object* [*Function*]

`create-extent` *&key upper-left lower-right* [*Function*]

`direction` *vector* [*Function*]

`exact-point-separation` *p1 p2* [*Function*]

Return the distance between two points as a floating-point number. Since points range from [0,0] to [8000,8000], the result is accurate to within +/- 1 meter.

`extend-segment` *p0 p1 extend-amount &optional (extend-to (point))* [*Function*]

Extend the segment from 'p0' to 'p1' by 'extend-amount'. Returns the new segment end [p1] rounded to meters.

`extent-intersection` *ul1 lr1 ul2 lr2* [*Function*]

Given two extents, return their intersection.

`extent-lower-right` *extent* [*Function*]

`extent-p` *object* [*Function*]

`extent-upper-left` *extent* [*Function*]

`half-line-intersects-segment-p` *l0 l1 s0 s1* [*Function*]

Returns T iff the half-infinite line [aka the ray] starting at L0 and going through L1 intersects the segment [S0,S1].

`magnitude` *vector* [*Function*]

`make-extent` *point* [*Function*]

`make-vector` *direction magnitude* [*Function*]

Create a vector. 'Magnitude' and 'direction' should be regular floats.

`nearest-point-on-segment` *pointj line-ptl line-ptk* [*Function*]

Find the point on the segment defined by 'line-ptL' 'line-ptK' nearest to 'pointJ'. The returned point may be 'eq' to a segment end point.

`point` *&optional (x nil x-specified) y (setpoint nil setpoint-specified)* [*Function*]

Make a new point using 'x' and 'y'. If 'setpoint' is specified it is destructively modified and returned.

`point-at-resolution` *point resolution* [*Function*]

Make a copy of 'point'at a specific resolution.

`point-at-resolution*` *point resolution* [*Function*]

`point-at-resolution-with-offsets` *p1 p2 resolution* [*Function*]

Return a point in the same cell as 'p1' with the same offsets as 'p2'.

`point-difference` *p1 p2* [*Function*]

Return p1 - p2.

`point-difference*` *p1 p2* [*Function*]

Return p1 = p1 - p2.

`point-distance-from-extents` *test-point upper-left lower-right &optional (solid? t)* [*Function*]

This returns the distance between a point and a rectangle specified by 'upper-left' and 'lower-right'. The rectangle may be solid; if it is and the point lies inside it, the returned distance is negative and is the distance from the point to the edge of the rectangle.

`point-distance-from-segment-squared` *pointj line-ptl line-ptk* [*Function*]

Copied from Bowyer & Woodark, pg. 47.

`point-distance-from-square` *test-point square-origin size &optional (solid? t)* [*Function*]

This returns the distance between a point and a rectangle specified as an origin and a size. The rectangle may be solid; if it is and the point lies inside it, the returned distance is negative and is the distance from the point to the edge of the rectangle.

`point-distance-squared-from-star-polygon` *point star-point polyline* [*Function*]

Return 0.0 if the point is in the polygon, or the distance squared from the point to the polygon.

`point-extent-sector-code` *point extent* [*Function*]

`point-in-bounds-p` *point* [*Function*]

`point-in-extent-p` *point extent* [*Function*]

Return T if 'point' is within the extent.

`point-in-star-polygon-p` *point star-point polyline* [*Function*]

`point-left-of-line-p` *point p0 p1* [*Function*]

Determine if 'point' is in left half-plane defined by the directed line segment.

`point-on-line-p` *point p0 p1* [*Function*]

Determine if 'point' lies along line defined by segment.

`point-on-segment->parameter` *p0 p1 point &optional error-check?* [*Function*]

If 'point' is on the segment [P0,P1], returns the parametric specification of 'point'. Actually, this works for any point on the half-infinite line [ray] from 'p0' through 'p1' to infinity. If 'point' is on the line but not on the ray, the parameter returned is the negative of the correct answer; that is, the correct answer is negative, but the answer returned is the absolute value of the correct answer.

`point-right-of-line-p` *point p0 p1* [*Function*]

Determine if 'point' is in right half-plane defined by the directed line segment.

`point-rotate-around-point` *theta rotate-point around-point* [*Function*]

`point-rotate-origin` *theta pt* [*Function*]

`point-rotate-origin*` *theta pt* [*Function*]

`point-round` *point* [*Function*]

Rounds 'point'. Returns a new point.

`point-round*` *point* [*Function*]

Rounds 'point'. Destructively modifies 'point'.

`point-sector-code` *point top-left bottom-right* [*Function*]

`point-separation` *p1 p2* [*Function*]

Return the distance between the two points +/- 1 meter. Use exact-point-separation if necessary.

`point-separation-and-sin-cos` *p1 p2* [*Function*]

Find the distance between points. Return the sin and cos of the angle between them.

`point-separation-lessp` *p1 p2 distance* [*Function*]

Return T if distance between 'p1' and 'p2' < distance.

`point-separation-squared` *p1 p2* [*Function*]

Return the square of the distance between the two points. The result is a single-precision floating point number.

`point-sum` *p1 p2* [*Function*]

Return P1 + P2.

`point-sum*` *p1 p2* [*Function*]

Return P1 = P1 + P2.

`point-wrt-line` *point p0 p1* [*Function*]

Return 0 if 'point' is on the line, < 0 if left, > 0 if right.

`point-x` *point* [*Function*]

`point-y` *point* [*Function*]

`point=` *p1 p2* [*Function*]

`pointp` *point* [*Function*]

`points-in-same-cell-p` *p1 p2 resolution* [*Function*]

Return T/NIL if the two points share the same cell at the given resolution.

`polyline->segments` *polyline &key (closedp t)* [*Function*]

Converts 'polyline' into a list of segments [represented as a list of two points].

`polyline-intersects-cell-p` *polyline point resolution* [*Function*]

Returns T iff some segment of 'polyline' goes through the cell of size 'resolution' containing 'point'.

`polyline-length` *polyline &key (closedp t)* [*Function*]

`quick-segment-intersects-cell-p` *p0 p1 cell-point resolution* [*Function*]

True iff the segment [P0,P1] passes through the cell containing 'cell-point' at 'resolution'. Returns the point [rounded] where the segment intersects a diagonal of the cell. [BUG: fails if the segment ends inside the cell without intersecting a diagonal.]

`radians->degrees` *rad* [*Function*]

`rounded-point-p` *point* [*Function*]

`same-side-p` *line-pt1 line-pt2 point1 point2* [*Function*]

Return T if 'point1' and 'point2' are both in the same half plane determined by 'linep1' and 'linep2'.

`segment&parameter->point` *p0 p1 parameter* [*Function*]

Given a segment and a parameter, return a point on the segment. The point is not rounded.

`segment&parameter->point*` *p0 p1 parameter* [*Function*]

Given a segment and a parameter, return a point on the segment. The new point is not rounded and is returned in 'p0'.

`segment&parameter->point2` *p0 p1 parameter p2* [*Function*]

Given a segment and a parameter, return a point on the segment. The point is rounded and is returned in 'p2'.

`segment-cell-intersection` *p0 p1 cell-point resolution* [*Function*]

Return the first intersection point of the segment [P0,P1] with the cell containing 'cell-point' at 'resolution'. If there is no intersection, returns NIL. The returned point

isn't rounded. Caution: The border of a cell is defined by four segments, the North, South, East and West edges. Points on the North and West edges are IN the cell, while points on the South and East edges are NOT in the cell; they are in the cells to the South and East, because those same segments are the borders of other cells, and each point can only be in one cell. Essentially, the points IN a cell are (x,y), such that

CELL-SIZE = 2^RESOLUTION
FLOOR[x/CELL-SIZE] <= x < FLOOR[x/CELL-SIZE]+CELL-SIZE

and similarly for y. Note the <= versus the <. See 'segment-cell-interior-intersection' for an alternative function.

`segment-cell-intersection-parameter` *p0 p1 cell-point resolution* [*Function*]

Return the parameter corresponding to the first intersection point of the segment with the border of the cell. This point is not guaranteed to be in the cell, since the North and West borders of a cell are in the cell, while the South and East borders are not—they belong to the cells to the South and East, respectively. See 'segment-cell-interior-intersection-parameter' for an alternative function. In more detail: The border of a cell is defined by four segments, the North, South, East and West edges. Points on the North and West edges are IN the cell, while points on the South and East edges are NOT in the cell; they are in the cells to the South and East, because those same segments are the borders of other cells, and each point can only be in one cell. Essentially, the points IN a cell are (x,y), such that

CELL-SIZE = 2^RESOLUTION
FLOOR[x/CELL-SIZE] <= x < FLOOR[x/CELL-SIZE]+CELL-SIZE

and similarly for y. Note the <= versus the <.

`segment-cell-interior-intersection-parameter` *p0 p1 cell-point resolution* [*Function*]

Return the parameter corresponding to the first intersection point of the segment with the points IN the cell. This point IS guaranteed to be in the cell. See 'segment-cell-intersection-parameter' for an alternative function.

`segment-edge-intersection-parameter` *p0 p1 edge* [*Function*]

`segment-feature-border-intersection-in-cell` *p0 p1 cell-point resolution featureflags map* [*Function*]

Return the first intersection point of the segment with the border of an edge of type 'featureflags'. The returned point is not rounded.

`segment-feature-centerline-intersection-in-cell` *p0 p1 cell-point resolution featureflags map* [*Function*]

Return the intersection point of the segment with the centerline of an edge of type 'featureflags'.

`segment-intersection` *pk pl pm pn* [*Function*]

Return the point of intersection or nil. Copied from Bowyer & Woodwark.

`segment-intersection-parameter` *pk pl pm pn* [*Function*]

Return the parameter of intersection on the segment [Pk,Pl]. If the segments don't intersect, return NIL. If the segments are coincident, an endpoint of [Pm,Pn] that is on [Pk,Pl] is returned.

`segment-intersection-parameters` *pk pl pm pn* [*Function*]

Return the parameters of the intersection of the lines determined by segments [Pk,Pl] and [Pm,Pn]. The first value is the parameter on [Pk,Pl], the second on [Pm,Pn]. If the segments intersect, both parameters will be between 0 and 1, inclusive. If the lines do not intersect, NIL is returned. If the lines are coincident, the correct result is calculated.

`segment-intersects-cell-p` *p0 p1 point resolution* [*Function*]

Returns T iff the segment [P0,P1] goes through the cell of size 'resolution' containing 'point'.

`segment-intersects-edge-type-in-cell-p` *start finish cell-point flags map* [*Function*]

Return T iff the segment from start to finish intersects an edge with flags within cell-point.

`segment-side-segments` *p1 p2 width &optional (s10 (point)) (s11 (point)) (s20 (point)) (s21 (point))* [*Function*]

Return the two segments on the boundary of the argument segment (rounded).

`segment-to-implicit-line` *pk pl* [*Function*]

Given a line segment, return the implict form Ax + By + C = 0.

`segment-to-parametric-line` *p0 p1* [*Function*]

Given a segment, return the parameters of the line
x = Xo + Fs
y = Yo + Gs.

`segment-touches-edge-p` *p0 p1 edge* [*Function*]

Return T if the segment from 'p0' to 'p1' comes in contact with 'edge'.

`segments->polylines` *segments* [*Function*]

Converts an unsorted list of 'segments', each represented as a list of two points, into a list of polylines by joining segments.

`segments-intersect-p` *pk pl pm pn* [*Function*]

Return the point of intersection or nil. Copied from Bowyer & Woodwark.

`set-point-x` *point value* [*Function*]

`set-point-y` *point value* [*Function*]

`vector-end-point` *start-point angle radius &optional (end-point (point 0 0))* [*Function*]

Given a vector specified by 'start-point', 'angle' and 'radius', calculate and return an endpoint.

`xy-segment-to-implicit-line` *xk yk xl yl* [*Function*]

Given a line segment, return the implict form Ax + By + C = 0.

`xy-segments-intersect-p` *xk yk xl yl xm ym xn yn* [*Function*]

Return the point of intersection or nil. Copied from Bowyer & Woodwark.

7.9 Iteration Constructs

`*circle-neighborhood-radius-index*` [*Variable*]

`*circle-neighborhoods*` [*Variable*]

`*fire-neighborhood*` [*Variable*]

`*square-neighborhood*` [*Variable*]

`do-feature-edges` *(edge cell-point map &key feature-flags (edge-type both)) &body body* [*Macro*]

Iterate over all edges in a cell matching 'feature-flags'. 'edge-type' is either :both (default), :static or :dynamic. Body is repeated for both types of edges if necessary. Dynamic features are done first. A (return) will get you out of the current edge type loop, not the do-feature-edges loop!

`do-point-set` *(point seed-point &key (resolution (quote *gc-cell-resolution*)) (x (gensym)) (y (gensym)) visited-point-bitarray) (&rest finish-forms) &body body* [*Macro*]

Iterate over a bunch of points. Body adds points to the set of points. 'do-point-set' guarantees that each point is visited only once. Use the macro (visit-xy x y) or (visit-point point) to add a point to the set to visit. Body is executed until the set of points is empty. 'point' can be modified by body, but its value is changed each time through the loop (so use 'copy-point' to keep the point around). Visited points are remembered at the specifed resolution. This also defines the macro (do-point-neighbors (connectedness) &body body). 'point' is set to each neighbor (4 or 8 connected) in turn and body is executed. The expansion is of the form (setf point n1) body (setf point n2) body ...

`do-polyline` *(p0 p1 polyline &key (closedp t) (return nil)) &body body* [*Macro*]

Iterate over each segment in 'polyline'. If body doesn't do an explicit return,'return' is returned.

`do-vertex-neighbors` *(neighbor connection vertex &key (edge (gensym))) &body body* [*Macro*]

Iterate over the neighbors of 'vertex'. 'Vertex' and 'neighbor' are structures.

`dofiremap` *(point &key upper-left lower-right (resolution *gc-cell-resolution*)) &body body* [*Macro*]

Iterate over a rectangular area of a map inclusive of upper-left and lower-right points. If the bounds aren't specified, use the full extent of the map.

`doneighbors` *(neighbor point &key (neighborhood *square-neighborhood*) neighborlist min-radius max-radius distance angle path (resolution *gc-cell-resolution*) (grid-size nil) (check-bounds nil) (index nil)) &body body* [*Macro*]

Iterate over neighbor points in neighborhood array. If 'neighborlist' is not specified, iterate over all neighbors. Note: 'neighbor' is destructively modified.

`map-fire-region` *function fire map* [*Function*]

Iterate over every point contained in the region of fire.

`map-fire-to-boundary` *function fire map &key (resolution *fire-cell-resolution*) (stop-at-natural-boundaries t) (boundary-in-region-p nil) (natural-boundaries-in-region-p nil) (visited-point-bitarray nil)* [*Function*]

Iterate over every point contained in the region of fire, up to the fire boundary. Stop spreading at natural boundaries.

`map-pixels-on-line` *function p1 p2 resolution &rest other-args* [*Function*]

Map over all pixels between two points. The traversal is from p1 to p2. The point argument to the function is destructively modified. If the function returns two values, and the first value is :return, then the mapping is terminated and the second value is returned; otherwise, returns NIL.

`some-pixel-on-line` *predicate start finish resolution* [*Macro*]

Execute 'predicate' for each point on 'line'. If 'predicate' returns non-nil, return that value.

`visit-neighbors` *connectivity* [*Macro*]

Visit all neighbors 4 or 8 connected from the current point. Can only be used within the lexical scope of 'do-point-set'.

Chapter 8

Interface

This chapter describes how to *interface* with Phoenix. There are three broad interfacing activities: Defining new commands, displaying things on the firemap windows and adding new types of windows on the `desktop`.

When Phoenix is started, an instance of the `fire-system` flavor is created. The entire user interface is based on the Explorer Universal Command Loop (UCL), therefore the `fire-system` flavor is built up from UCL flavors and mixins. Almost all interface commands are messages to the current fire system.

8.1 Adding New Commands

To define a new UCL command decide which command table to add the command to or, if necessary, create a new command table. The convention used is that each directory of code may contain two files: *COMMAND-METHODS.LISP* and *MAKE-COMMANDS.LISP*. The command-method file contains all of the methods used to implement the commands and the make-command file contains all of the calls to define the commands and build the command tables. A number of new UCL command tables are defined by Phoenix. All command methods must be methods of the `fire-system` flavor. Look at some of the existing files to see how command tables and commands are defined.

8.2 I/O and Firemaps

A library of functions can be used to perform graphical operations on windows that display firemaps (eg., `highlight-line`). In addition, the mouse can be used to select positions and icons from a map. These functions are described in chapter 9.

8.3 Icons

An `icon` is an graphical object that can be displayed and moved around on a firemap. All firemaps contain sets of icons that are automatically displayed. Icons can be created, deleted and moved on a firemap. All agents in Phoenix are displayed by icons (bulldozers, watchtowers, etc.) Icons and the routines for manipulating them are described in more detail in chapter 9 .

8.4 Defining New Desktop Windows

To build a new window suitable for use on the desktop, add the `utils::desktop-mixin` flavor to the component flavors of the window. Write a "make" function to make a new instance of your window and initialize it – the function `select-or-create-window-on-desktop` should be useful for this. The `define-desktop-window` macro will associate the window type with the make function and will also add the window type to the list of window types that can be created by left-clicking on the desktop. The following code was written for the system to take an existing flavor, `firemap-window`, and build a new flavor that can be used on the desktop.

```
(pushnew 'desktop-firemap-window *standalone-flavors*)

(defflavor desktop-firemap-window
           ()
           (utils::desktop-mixin
            firemap-window)

  (:default-init-plist
    :font-map *fire-system-font-map*
    :foreground-color *fire-system-fg-color*
    :background-color *fire-system-bg-color*
    :border-color *fire-system-border-color*
    :scroll-bar-mode :maximum))

(defmethod (desktop-firemap-window :after :init) (ignore)
  (declare (ignore ignore))
  (send self :set-firemap (real-world-firemap)))

(define-desktop-window "Firemap" make-desktop-firemap-window)

(defun make-desktop-firemap-window ()
  "Create a firemap window on the desktop."
  (let* ((window (select-or-create-window-on-desktop
                   'desktop-firemap-window)))
    (send window :set-label
        `(:string ,(format nil "Real World Firemap") :centered))
    (send window :expose)
    window))
```

Notice that this did some of initializations in `make-desktop-firemap-window` and used an `:after :init` method to do the rest. This is purely a matter of style.

Chapter 9

Interface Reference Manual

9.1 Bitmap Caching

`*bitmap-pathname-defaults*` [*Variable*]

Place where bitmaps are written and can always be read.

`*use-cached-maps*` [*Variable*]

Determines whether the firemap bitmap caching is enabled.

`create-bitmaps` *&optional (*fire-system* (fire-system))* [*Function*]

`load-bitmaps` *&key (background t)* [*Function*]

Load the bitmaps for firemap bitmap caching.

`save-bitmaps` [*Function*]

Save the bitmaps for firemap bitmap caching. See *bitmap-pathname-defaults*.

9.2 Colors

`*b&w-desktop-color*` [*Variable*]

`*banded-color-map*` [*Variable*]

A pointer to the phoenix color map.

`*color->highlight-b&w*` [*Variable*]

`*color->highlight-color*` [*Variable*]

`*color-desktop-color*` [*Variable*]

`*highlight-b&w-mappings*` [*Variable*]

`*highlight-colors*` [*Variable*]

`*highlight-cyan*` [*Variable*]

`*highlight-orange*` [*Variable*]

`*highlight-purple*` [*Variable*]

`*highlight-red*` [*Variable*]

`*highlight-white*` [*Variable*]

`*highlight-yellow*` [*Variable*]

`*use-color-p*` [*Variable*]

Set to NIL to force B&W.

`save-color-map` *&optional (filename ph:fonts;banded-color-map)* [*Function*]

Use this function to write the color map to disk.

`using-band` *(band) &body body* [*Macro*]

Use a band just in a single window. This alters the plane-mask of the window so that only the specified color planes will be changed by drawing routines.

9.3 Firemap Window

`*default-elevation-gradient*` [*Variable*]

`*default-map-editor-state*` [*Constant*]

`*distance-for-sensitivity*` [*Constant*]

The pixel distance at which the mouse is considered to be on top of an object.

`*initial-resolution*` [*Constant*]

The resolution at which a firemap-window initially displays the firemap.

`*leave-ghost-objects*` [*Variable*]

A weird drawing variable that results in objects leaving a trail of where they have been.

`do-exposed-map-windows` *(mapwindow &optional window map) &body body* [*Macro*]

Execute 'body' with 'mapwindow' bound to each of the exposed firemap windows in succession.

`domapwindows` *(window &optional point (can-be-deexposed nil)) &body body* [*Macro*]

Iterate over each of the windows viewing this map. If 'point' is specified, it must be visible. If 'can-be-deexposed' is nil, then 'window' must be exposed.

`:edit-parameters` [*Method of* `firemap-window`]

`:find-object-near-point` *point* [*Method of* `firemap-window`]

Returns the object nearest to point.

`:firemap` [*Method of* `firemap-window`]

`make-desktop-firemap-window` [*Function*]

Create a firemap window on the desktop.

`:mouse-select-window-point` *&optional (mouse-doc select a screen point.) &key (mouse-char mouse-glyph-hollow-circle-pointer)* [*Method of* `firemap-window`]

Select a point from the map using the mouse. Return the point selected and the button clicked. Only single clicks are accepted. Mouse-M is bound to Abort (that is, it returns NIL as the point selected).

`move-an-object` *object &optional old-position new-character* [*Function*]

`:move-object` *object old-position new-character* [*Method of* `firemap-window`]

Erase 'object' at 'old-position' draw it at its new position using 'new-character'.

`move-rectangle-within-window` *window x y width height* [*Function*]

`:object-menu` *x y* [*Method of* `firemap-window`]

Called when right click near object

`:origin-x` [*Method of* `firemap-window`]

`:origin-y` [*Method of* `firemap-window`]

`:point-on-screen-p` *point* [*Method of* `firemap-window`]

`:point-screen-x` *point* [*Method of* `firemap-window`]

`:point-screen-y` *point* [*Method of* `firemap-window`]

`:redraw-cell` *point* [*Method of* `firemap-window`]

Redraw a single cell.

`:refresh-highlights` [*Method of* `firemap-window`]

`:refresh-objects` [*Method of* `firemap-window`]

`:set-firemap` *map &optional (refresh t)* [*Method of* `firemap-window`]

`spum` [*Function*]

S elect P oint U nder M ouse - selects a point with the mouse and returns it.

`with-method-clipping` *&body body* [*Macro*]

Should be used if drawing methods are used.

9.4 Fire System

`*blip-alist*` [*Variable*]

The alist of mouse character blips that the fire-system window handles.

`*command-command-table*` [*Variable*]

`*command-tables*` [*Variable*]

A list of all command table names.

`*debug-screen*` [*Variable*]

`*initial-screen-configuration*` [*Variable*]

The initial fire-system window configuration.

`*phoenix-command-menu*` [*Variable*]

`*query-about-selecting-phoenix*` [*Variable*]

If true, ask for confirmation before selecting a current Phoenix system.

`*task-command-table*` [*Variable*]

`*task-inspector-menu*` [*Variable*]

`*task-menu*` [*Variable*]

`:activate-all-tasks` [*Method of* `fire-system`]

`:activate-task` *&optional task* [*Method of* `fire-system`]

`:active-agents` [*Method of* `fire-system`]

`:all-firemaps` [*Method of* `fire-system`]

`:all-inferiors` [*Method of* `fire-system`]

`:all-tasks` [*Method of* `fire-system`]

`:base-time` [*Method of* `fire-system`]

`:clear-highlights` [*Method of* `fire-system`]

`:clear-trace-window` [*Method of* `fire-system`]

`:deactivate-task` *&optional task* [*Method of* `fire-system`]

`define-desktop-window` *name function* [*Macro*]

Add this window type to the menu of windows that can be created on the desktop.

`:delete-task` *task* [*Method of* `fire-system`]

`:describe-task` *&optional task* [*Method of* `fire-system`]

`:edit-environment` [*Method of* `fire-system`]

`:edit-fire` [*Method of* `fire-system`]

`:edit-task` *&optional task* [*Method of* `fire-system`]

`:elapsed-time` [*Method of* `fire-system`]

`:erase-fire` *&optional (confirm t) (refresh t)* [*Method of* `fire-system`]

`:exposed-firemap-windows` [*Method of* `fire-system`]

`find-phoenix-system` [*Function*]

Called when SYSTEM S is hit.

`fire-system-all-tasks` *fire-system* [*Function*]

`fire-system-base-time` *fire-system* [*Function*]

`fire-system-elapsed-time` *fire-system* [*Function*]

`fire-system-real-world-firemap` *fire-system* [*Function*]

`fire-system-scheduler` *fire-system* [*Function*]

`:get-environment` [*Method of* `fire-system`]

`:get-environment-parameter` *param &optional default* [*Method of* `fire-system`]

`:inspect-task` *&optional task* [*Method of* `fire-system`]

`:macro-step-scheduler` *&optional (n 1)* [*Method of* `fire-system`]

Macro step the system n 'times'.

`:macro-step-scheduler-and-wait` *&optional (n 1)* [*Method of* `fire-system`]

`:meter-task` *&optional task* [*Method of* `fire-system`]

Turn on task metering.

`:real-world-firemap` [*Method of* `fire-system`]

`:refresh-all-windows` [*Method of* `fire-system`]

`:refresh-objects` [*Method of* `fire-system`]

`:reinitialize` *&optional (query t) (create-agents t)* [*Method of* `fire-system`]

`:reset-and-activate-all-tasks` *&optional (query t)* [*Method of* `fire-system`]

Reset all the tasks and the scheduler. Erase the fire.

`:run` *&optional time-to-run* [*Method of* `fire-system`]

Run the system. Optionally, run it for 'time-to-run' minutes.

`:scheduler` [*Method of* `fire-system`]

`:select-configuration` [*Method of* `fire-system`]

`:set-all-tasks` *.newvalue.* [*Method of* `fire-system`]

`:set-base-time` *.newvalue.* [*Method of* `fire-system`]

`:set-environment` *plist* [*Method of* `fire-system`]

`:set-environment-parameter` *param value* [*Method of* `fire-system`]

`:set-real-world-firemap` *.newvalue.* [*Method of* `fire-system`]

`:single-macro-step-scheduler` [*Method of* `fire-system`]

Macro step the system one time.

`:single-step` [*Method of* `fire-system`]

Run the system for the smallest time interval possible.

`:start` [*Method of* `fire-system`]

Start the system.

`:start-fire` *&optional radius point &optional point-y* [*Method of* `fire-system`]

`:stop` [*Method of* `fire-system`]

Stop the system.

`:task-menu-items` *&optional (task-type t) initial-task-list* [*Method of* `fire-system`]

`:toggle-firemap` *&optional (nextp t)* [*Method of* `fire-system`]

Make the left firemap pane in a 'two-view' configuration display a new firemap. The firemap chosen is the next one in the firemap list. If nextp is NIL, use the previous map on the list.

`:view-firemap` *&optional task* [*Method of* `fire-system`]

9.5 Highlighting

`*remember-highlights*` [*Variable*]

If non-nil then highlights are stored so that the won't be lost during a refresh.

`highlight-cell` *point &key resolution color type alu map window* [*Function*]

Highlight a cell. 'type' is :filled or :outline. All visible maps are highlighted unless 'map' or 'window' is specified.

`highlight-edge` *edge &key full width color type alu units arrowp map window* [*Function*]

Highlight an edge. 'width' is width of line (in units :pixels (default) or :meters). 'type' is :solid or :dashed. All visible maps are highlighted unless 'map' or 'window' is specified.

`highlight-extent` *p1 p2 &key width color type alu map window* [*Function*]

Highlight a rectangular region. 'p1' can be either a point or an extent structure. 'type' is :filled, :outline, or :dashed. All visible maps are highlighted unless 'map' or 'window' is specified.

`highlight-fire` *fire &rest args &key (spokes nil) (color *highlight-cyan*)* [*Function*]
(projections nil) &allow-other-keys

`highlight-fire-cell` *point &key color type alu map window* [*Function*]

Highlight a cell at fire resolution. 'type' is either :filled, :outline. All visible maps are highlighted unless 'map' or 'window' is specified.

`highlight-line` *from-point to-point &key width color type alu units arrowp* [*Function*]
arrowheadsize fill-arrow-p band clip map window

Highlight a line between two points. 'width' is width of line in 'units' (:pixels (default) or :meters)). 'type' is :solid or :dashed. If 'arrowp' is T, draw an arrow head at 'to-point'. All visible maps are highlighted unless 'map' or 'window' is specified.

`highlight-point` *point &key radius color type alu units map window* [*Function*]

Highlight a point (draw a circle around it). 'units' is the units of 'radius' (:pixels (default) or :meters.) 'type' is either :outline or :filled (default). All visible maps are highlighted unless 'map' or 'window' is specified.

`highlight-polyline` *point-list &key width color type alu units arrowp closedp* [*Function*]
center map window

Highlight a chain of line segments defined by the list of points. 'width' is width of line (in units :pixels (default) or :meters). 'type' is :solid or :dashed. If 'arrowp' is T, draw an arrow head at the end point. if 'closedp' is T (default), the points form a closed polygon. All visible maps are highlighted unless 'map' or 'window' is specified.

`highlight-vector` *point angle radius &key width color type alu units arrow map* [*Function*]
window

Highlight a vector. 'width' is width of line (in units :pixels (default) or :meters). 'type' is :solid or :dashed. If 'arrowp' is T, draw an arrow head at the end point. 'length' is the length of the vector in :length-units (:meters (default) or :pixels) All visible maps are highlighted unless 'map' or 'window' is specified.

`outline-cell` *point &rest args* [*Function*]

`outline-fire-cell` *point &rest args* [*Function*]

`random-highlight` [*Function*]

Picks a random color from '*highlight-colors*'.

`with-highlight` *(highlight-form) &body body* [*Macro*]

Executes body with highlight specified in 'highlight-form' turned on then turns it off.

9.6 Icons

`:add-firemap` *map* [*Method of* `icon`]

`:after :init` *&rest ignore* [*Method of* `icon`]

`:after-draw-character-function` [*Method of* `icon`]

`icon-after-draw-character-function` *icon* [*Function*]

`:set-after-draw-character-function` *.newvalue.* [*Method of* `icon`]

`:b&w-character` [*Method of* `icon`]

`icon-b&w-character` *icon* [*Function*]

`:set-b&w-character` *.newvalue.* [*Method of* `icon`]

`:bg-color` [*Method of* `icon`]

`icon-bg-color` *icon* [*Function*]

`:set-bg-color` *.newvalue.* [*Method of* `icon`]

`:color-character` [*Method of* `icon`]

`icon-color-character` *icon* [*Function*]

`:set-color-character` *.newvalue.* [*Method of* `icon`]

`:current-character` [*Method of* `icon`]

`icon-current-character` *icon* [*Function*]

`:set-current-character` *.newvalue.* [*Method of* `icon`]

`:fg-color` [*Method of* `icon`]

`icon-fg-color` *icon* [*Function*]

`:set-fg-color` *.newvalue.* [*Method of* `icon`]

`:firemaps` [*Method of* `icon`]

`icon-firemaps` *icon* [*Function*]

`:set-firemaps` *.newvalue.* [*Method of* `icon`]

`:before :kill` *&rest ignore* [*Method of* `icon`]

`:minimum-display-size` [*Method of* `icon`]

`icon-minimum-display-size` *icon* [*Function*]

`:set-minimum-display-size` *.newvalue.* [*Method of* `icon`]

`:name` [*Method of* `icon`]

`icon-name` *icon* [*Function*]

`:set-name` *.newvalue.* [*Method of* `icon`]

`:object-size` [*Method of* `icon`]

`icon-object-size` *icon* [*Function*]

`:set-object-size` *.newvalue.* [*Method of* `icon`]

`:orientations` [*Method of* `icon`]

`icon-orientations` *icon* [*Function*]

`:set-orientations` *.newvalue.* [*Method of* `icon`]

`:position` [*Method of* `icon`]

`icon-position` *icon* [*Function*]

`:set-position` *.newvalue.* [*Method of* `icon`]

`:wrapper :set-position` *&body body* [*Method of* `icon`]

`:print-self` *stream &rest ignore* [*Method of* `icon`]

`:remove-firemap` *map* [*Method of* `icon`]

`:set-color` *color* [*Method of* `icon`]

9.7 Startup Window

`display-phoenix-icon-window` *&optional label &key (font *icon-window-label-font*) fg (bg (aref *phoenix-icon* 0 0))* [*Function*]

Display the Phoenix startup icon.

`set-phoenix-icon-background-color` *color* [*Function*]

Changes the background color of the Phoenix icon and the window it is being displayed in if one exists.

Chapter 10

Fire Simulation

The fire simulation is controlled by a periodic-task which is an instance of the `fire-simulation` flavor. By default, the fire-simulation updates the fire every five minutes. The computation of rate of spread is based on ground-cover, elevation gradient, wind speed, wind direction, humidity and temperature. The simulator implements most parts of the fire model in [4], though it doesn't include season, cloud cover, time of day and slope orientation (ie. south side of a mountain) as provided in the model. Information about burn times and spotting was empirically derived.

The basic implementation idea is as follows: whenever a cell catches on fire, the ignite time of all its neighbors is computed. If a neighbor already has an ignite time, and the ignite time from the newly ignited cell is earlier, the earlier time is used. The simulator maintains a large queue of cells to be ignited, sorted by ignite time. Since fire changes state, (low->hot->smoldering->burned-out), burning cells have state change times. Burning cells are also kept in the queue.

```
While the next event in the queue occurs before the end of the
      current five minute cycle
  If the event is an ignition event
      ignite the cell
      compute its change state time
      enqueue the change event
      for all neighbors of the cell
        if neighbor is ignitable
         it = time it takes for fire to spread from cell
              to neighbor + ignite_time(cell)
         if ignite_time(neighbor) = nil
              or it < ignite_time(neighbor)
            ignite_time(neighbor) = it
            enqueue(neighbor)
   If event is a change state event
       set cell fire state
       if cell is still burning
           compute new change state time
           enqueue(cell)
```

The "neighbor" set of a cell consists of all adjacent cells (8 connected) plus all cells a knight's move away; thus a cell has 16 neighbors. The reason for the including the knight's tour is to smooth out the shape of a fire. With only 8 neighbors, fires tend to be oddly shaped unless the wind comes from one of the eight compass points. Even with 16 neighbors the fire looks odd when the wind comes from certain directions. This can be fixed, at a cost in cpu time, by increasing the number of directions in which a fire could spread.

Before deciding to compute the ignition of a neighbor, the simulation checks to see if the neighbor is ignitable. A neighbor is ignitable if:

- The neighbor contains burnable ground cover (ie. no lake) AND
- Either there are no obstructions between the cell and neighbor (roads & rivers) or the obstruction can be jumped (tested probabilistically)

When the simulator changes a fire state it modifies the `real-world-firemap`. The simulator stores simulation information in the real-world map in `fire-info` data structures.

Chapter 11

Fire Simulation Reference Manual

11.1 Fuel Model

`copy-fuel-model` *object* [*Function*]

`fuel-model` [*Structure*]

`fuel-model-moisture->base-ros` *fuel-model* [*Function*]

`fuel-model-non-array-wind-factor` *fuel-model* [*Function*]

`fuel-model-p` *object* [*Function*]

`fuel-model-slope-percent->slope-factor` *fuel-model* [*Function*]

`fuel-model-type` *fuel-model* [*Function*]

`fuel-model-wind->wind-factor` *fuel-model* [*Function*]

11.2 Parameters

`*default-fire-increment*` [*Variable*]

Default period for the fire simulation task.

`*default-spotting-scale-factor*` [*Variable*]

Amount to modify the base probabilities of fire spotting. 0 = no spotting. This is a multiplier for the values in *jump-probabilities*.

`*features-that-stop-fire*` [*Constant*]

A set of all the features that can stop a fire from spreading.

`*gc-transition-times*` [*Variable*]

An array containing the amount of time to change from state to state (nothing -> low -> hot -> smouldering -> burnedout) indexed by ground cover.

`*humidity-temperature->fuel-moisture*` [*Variable*]

A two dimensional array referenced by humidity and temperature. Each element contains a value that is used as an index into the moisture->base-ros array in a fuel-model.

`*initial-environment*` [*Variable*]

The default environmental conditions in Phoenix.

`*jump-probabilities*` [*Variable*]

An array [indexed by feature types] of probabilities that a fire-cell containing a feature of the specified type will catch fire.

`*wind-randomization-factor*` [*Variable*]

Random deviation of effective wind speed.

11.3 Simulator Functions and Methods

`burnable-cell-p` *cell-point map* [*Function*]

T iff the cell has burnable ground cover [that is, it isn't water].

`calculate-ros` *from-gc angle slope-percent &optional (verbose nil) (randomize t)* [*Function*]

Given a cell on fire, calculate the rate of spread in the direction of to-cell. Return a value in meters/minute.

`cell-stops-fire-p` *p map &optional (resolution *gc-cell-resolution*)* [*Function*]

Returns T if cell is not currently on fire and contains ground cover that can't burn or a feature that the fire normally will not cross (normally meaning except due to spotting).

`chop-up-segment-at-nonburnable-cells` *p0 p1 map* [*Function*]

Returns a new list of segments on the segment [P0,P1] which don't cross any non-burnable cells [lakes], but with endpoints that are in the lakes.

`current-wind-direction` [*Function*]

`current-wind-speed` [*Function*]

`fire-simulation` [*Function*]

Returns the current instance of the fire simulator.

`:highlight-queued-for-ignition` [*Method of* `fire-simulation`]

Highlight the cells that are on the simulators event queue and are scheduled to ignite.

`:ignite-cell` *point &rest ignore* [*Method of* `fire-simulation`]

Interactively set cell on fire.

`randomize-wind-magnitude` *mag* [*Function*]

Modify wind magnitude +/- *wind-randomization-factor*. This is used to simulate local wind speed variances.

`rate-of-spread` *gc direction grade* [*Function*]

Calculate the rate of spread from a cell with ground cover 'gc' along the vector 'direction' with a gradient of 'grade.' 'direction' should be an angle, measured in radians. 'grade' should be the percent gradient; that is, zero is flat and +100% is a 45 degree angle up while -100% is a 45 degree angle down. Value returned is in meters/minute.

`re-calc-ignite-time` *point &optional (combination-method minimum)* [*Function*]

Recalculate the ignite time for the specified point. 'combination-method' is either :set or :minimum.

`:recalculate-all-ignite-times` [*Method of* `fire-simulation`]

Called to recalculate all fire cell ignite times when there are new environmental conditions.

`:recalculate-ignite-time` *point* [*Method of* `fire-simulation`]

Recalculate the ignite time for the specified point. This should be called anytime something happens that may effect the ignite time (ie., fire-line, retardent, wind change, etc.).

`ros` *p1 p2 map* [*Function*]

Return the rate of spread of the fire from p1 to p2 in meters/second. Return NIL if fire can't spread over the ground cover.

`:set-update-maps` *.newvalue.* [*Method of* `fire-simulation`]

`simulate-forward` *&optional (steps 1) (refresh t)* [*Function*]

Run the fire simulator for 'steps' steps of its default period. If 'refresh' is non-nil the display is updated.

`time-to-ignite` *gc distance angle delta-elevation* [*Function*]

Return the time ignite-cell should catch on fire (in seconds)

`wind-magnitude-fn` *magnitude angle* [*Function*]

Given a wind magnitude and an angle of deflection (ie., the difference from the wind angle), returns the apparent wind magnitude at that angle.

Chapter 12

Miscellaneous Utilities Reference Manual

12.1 Numbers

`fpi` [*Constant*]

pi as a single float.

`div2` *i &optional (power 1)* [*Macro*]

Divide positive fixnum 'i' by 2 or a power of 2.

`exp2` *n* [*Macro*]

2^n

`log2` *n* [*Macro*]

Log of 'n' to base 2.

`max-using-zero-if-nil` *a b* [*Function*]

`mod2` *n power* [*Macro*]

Find 'n' mod a power of 2.

`phoenix-float` *n* [*Macro*]

A faster version of (float n 1.0f0).

`square` *x* [*Function*]

(* x x)

`times2` *i &optional power* [*Macro*]

Multiply 'i' by a power of 2.

`trunc-coord` *coord* [*Macro*]

Equivalent to (values (round coord *gc-cell-size*)), only faster.

`trunc-x` *point* [*Macro*]

Equivalent to (values (round (point-x point) *gc-cell-size*)), only faster.

`trunc-y` *point* [*Macro*]

Equivalent to (values (round (point-x point) *gc-cell-size*)), only faster.

`trunc2` *n power* [*Macro*]

Truncate 'n' to a power of 2.

`truncate-to-factor` *n factor* [*Macro*]

Equivalent to (* factor (truncate n factor)), only faster.

12.2 Sequences

`assocf` *alist item &optional default* [*Function*]

The same as (rest (assoc item alist :test #'eq)), except that it takes 'default' to return if the item is not found.

`break-string` *string length* [*Function*]

Break a string at spaces and hyphens across several lines of length.

`deletef` *item list &rest delete-args* [*Macro*]

Same as (setf list (apply #'delete item list delete-args)).

`make-initialized-array` *&rest inits* [*Function*]

Creates an array initialized from using 'inits' which is a list of the form
((<index-1> <init-form-1>)
(<index-2> <init-form-2>)
...
(<index-n> <init-form-n>))
Each element indexed will be filled with the result of evaluating the corresponding initialization form. The array returned will be just large enough to hold the element with the maximum index.

`nmerge-list` *list1 list2 compare-fn &key (key (function identity))* [*Function*]

Destructive merge of 'list2' into 'list1'. Both lists are may be changed. This works best if the lists are NOT cdr-coded.

`ordered-insert` *elt list &optional (test-fn (function <)) key* [*Function*]

Insert 'elt' into an ordered set.

`ordered-insertf` *item list &rest args* [*Macro*]

Same as (setf list (ordered-insert item list args).

`removef` *item list &rest delete-args* [*Macro*]

Same as (setf list (apply #'remove item list delete-args)).

`some*` *(element list) &body body* [*Macro*]

An iterative version of the function 'some'. Bind each element of list in turn. If the body returns non-nil, return the result. This is faster than the regular function, especially when lexical variables are used in the predicate.

12.3 Sets and Flags

`flag->object` *flag* [*Macro*]

Extracts the object from 'flag'.

`object-set` *&rest objects* [*Macro*]

Forms a set out of a bunch of objects. Same as (set-add (object->flag o1) (object->flag o2) ...).

`object->flag` *object* [*Macro*]

Converts 'object' into a flag.

`set-add` *set &rest flags* [*Macro*]

Adds 'flags' to 'set'.

`set-addf` *set &rest flags* [*Macro*]

Same as (setf set (apply #'set-add set flags)).

`set-clear` *set &rest flags* [*Macro*]

Removes 'flags' from 'set'.

`set-clearf` *set &rest flags* [*Macro*]

Same as (setf set (apply #'set-clear set flags)).

`set-test` *set flag* [*Macro*]

T if 'set' contains 'flag'.

12.4 Symbols and Functions

`call-stack` *&optional (process current-process)* [*Function*]

Return a list of the functions on the call stack.

`name-of` *object* [*Function*]

Tries to return a reasonable handle for object.

`nice-call-stack` [*Function*]

Return a call stack starting at the right spot.

`object-in-resource-p` *resource-name object* [*Function*]

Determine if object is in the free pool of resource 'resource-name'.

`ph-apropos` *string &key predicate boundp fboundp* [*Function*]

Looks for 'string' in the Phoenix package only.

`ph-who-calls` *symbol-or-symbols* [*Function*]

Print who calls 'symbol-or-symbols' in the Phoenix package.

`with-phoenix-package` *&body body* [*Macro*]

Execute the body with *package* bound to the Phoenix package.

12.5 Variables

`*fire-system*` [*Variable*]

Bound within all subprocesses to the current fire-system.

`*firemap-area*` [*Variable*]

GC area where firemap data is stored.

`*phoenix-package*` [*Variable*]

The "Phoenix" package.

Appendix A

File Organization

The Phoenix testbed sources are distributed among various directories on the logical host "*PH:*".

"*COLOR;*" Color enhancements.

"*DOC;*" Documentation.

"*FIREMAP;*" Firemaps and firemap window definitions.

"*FONTS;*" Fonts and color tables.

"*FS;*" Interface and top level definitions.

"*IN;*" Instrumentation utilities.

"*MAC;*" microExplorer specific files.

"*MAPS;*" Compiled map files.

"*PATCHES;*" Patch files and directories.

"*PH;*" The top level Phoenix directory.

"*SIM;*" Fire simulator code.

"*TASKS;*" Tasks and task scheduler code.

Appendix B

System Maintenance

B.1 Loading Phoenix

To incrementally load Phoenix into an existing band:

```
(load-phoenix)
```

To load Phoenix onto a 'bare' machine without making a band:

```
(Load "christa:eksl;eksl-init")
(si:find-system-named :phoenix)
(make-phoenix)
```

B.2 Compiling Phoenix

Before making a new load band (see below), the Phoenix system files should be compiled as needed. There are three different 'preparation' methods.

- Normal method. This won't necessarily work if there are changes to macros, constants or inline functions.[1]

```
In existing phoenix band
(compile-phoenix-for-load-band)
```

[1] The current compilation scheme used by `compile-phoenix-for-load-band` and the `:COMPILE` keyword to `make-system` only compiles those files whose sources are newer than the binaries. Explicit dependencies between code modules are not represented. As a consequence, if the definition of a macro, constant or inline function changes, code which uses that definition might not be recompiled to reflect those changes. See the files "*PH:PH;PHOENIX-SYSTEM.LISP*" and "*PH:PH;PHOENIX-LOAD-UTILS.LISP*" and the Explorer LISP Reference manual for more information.

- Long method: This will catch more compile-time bugs then the normal method and correctly compile all changes to macros, constants and inline functions.

```
In existing phoenix band
(make-phoenix :compile)
Boot non-phoenix band
(load "christa:eksl;eksl-init")
(si:find-system-named :phoenix)
(compile-phoenix-for-load-band)
(make-phoenix-specific :recompile)
```

- Very long method. This will catch all possible compile time bugs. The resulting band should be the same as the long method.

```
Boot into non-phoenix-band
(load "christa:eksl;eksl-init")
(si:find-system-named :phoenix)
(make-phoenix-utils :compile)
(make-phoenix-specific :recompile)
Reboot
(compile-phoenix-for-load-band)
(make-phoenix-specific :recompile)
```

B.3 Making Phoenix Load Bands

See the Explorer I/O Reference manual for more information on `disksave` and load bands.

- Quick method

```
Boot into non-phoenix band
(load "christa:eksl.utils;make-load-band")
(eksl:make-load-band) and select phoenix on the menu
Do disk save
```

- Alternate method

```
Boot into non-phoenix band
(Load "christa:eksl;eksl-init")
(si:find-system-named :phoenix)
(load-phoenix-for-load-band)
Do disk save
```

The Phoenix testbed provides a customized functional interface to the Explorer system maintenance utilities. The following list contains all the functions that were utilized in the previous examples. See the Explorer LISP Reference manual for more details.

`compile-phoenix-for-load-band` *&optional force-increment-patch-version* [*Function*]

Compiles and loads Phoenix, rebuilds flavors and clears patches.

`find-component-systems-with-files` *system* [*Function*]

Return the list of component-systems for a system. The components are returned in an order suitable for compilation.

`make-phoenix` *&rest make-system-args* [*Function*]

Run make-system for all Phoenix subsystems.

`make-phoenix-specific` *&rest make-system-args* [*Function*]

Run make-system for all Phoenix specific systems.

`make-phoenix-utils` *&rest make-system-args* [*Function*]

Run make-system for the Phoenix External Utils system.

`phoenix-patch-level` [*Function*]

Returns the number of patches that have been made to the current Phoenix system.

B.4 Initialization

The Phoenix testbed uses the Explorer initializations software to maintain a list of routines that need to be executed before the Phoenix system can be used. See the Explorer LISP Reference manual for more information about initializations.

`*do-phoenix-init*` [*Variable*]

Determines whether the initializations on '*phoenix-initialization-list*' are run when the system is loaded.

`*phoenix-initialization-list*` [*Variable*]

This initialization list is run during the initialization of the Phoenix system.

`add-phoenix-initialization` *name form* [*Function*]

Add a form to be run at Phoenix initialization time.

Glossary

band A bit pattern used to mask the color lookup table during graphics drawing operations. Also known as a *color plane mask*.

cell An element in a grid array.

command typein The process by which a user uses the keyboard and/or mouse to select a predefined interfacing action to execute.

cpu time The representation of time that an Explorer process maintains.

desktop An overlapping window interface to Phoenix.

Explorer Texas Instrument's standalone Lisp workstation.

Explorer II A high-performance Explorer based upon a TI's VLSI Lisp microprocessor.

feature edge A representation of a topographical surface feature which has a start point, an end point and a width.

firemap A comprehensive data structure which represents topographical information such as ground cover, roads, rivers, buildings, fire and firelines.

firemap pane A tiled window component of the Phoenix window interface which displays the firemap.

grid array A two dimensional matrix which holds topographical information.

internal time The representation of time which is used by the internal data structures of the testbed.

neighborhood A descriptive specification using relative positions of the cells near a cell in a grid array.

pointset A data structure consisting of a set of points. Often used in region-growing algorithms.

polyline A data structure consisting set of lines. Used to define curves.

process An Explorer process. A set of sequential operations in shared virtual address space with a program counter, stack of function calls and special-variable bindings.

simple process A process that does not save its state between calls.

simulation time The representation of time which is presented to the user by the Phoenix interface.

task The basic computational organizational unit in the testbed. Built upon Explorer processes.

task time The representation of internal time that a task maintains.

Bibliography

[1] Adrian Bowyer and John Woodwark. *A programmer's geometry.* Butterworths, 1983.

[2] Paul R. Cohen, Michael Greenberg, David Hart, and Adele Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, Fall 1989. also Technical Report 89-61, COINS Dept, University of Massachusetts.

[3] David M. Hart and Vipul Gupta. The phoenix user manual. Technical report, COINS Dept., University of Massachusetts, Amherst, MA, 1990.

[4] National Wildfire Coordinating Group, Boise, Idaho. *NWCG Fireline Handbook*, November 1985.

Index