This document describes a potential problem which one must be careful about when creating a device interface for ACT-R or when using the device interfaces provided for the native windows created in ACL, CCL, or LispWorks. It is technically not only a problem for creating devices, but that is the situation where it is most likely to occur. The basis of the problem is that the ACT-R code is not "thread safe". Only the thread in which the ACT-R model(s) are being run should ever call any of the ACT-R functions or modify data which the ACT-R functions may use (like the items in the visicon).[1] For most situations that is not a problem because the ACT-R running commands operate within a single thread. However, when working with a device one is often connecting the model to some external and/or asynchronous system that is not running within the same Lisp thread as the ACT-R model. In those situations one must take precautions to not execute ACT-R code in any other thread.

For the native window devices provided with ACT-R this problem can occur with the motor actions which the model performs and the event handler methods and button actions which get executed as a result of those motor actions. The problem arises because when the model is interacting with the real windows the motor actions which it performs generate real actions so as to operate the same as if a person were interacting with the window. Those real actions will be handled by the Lisp's event handler mechanisms which may be executing in another thread. That means that the rpm-window-key-event-handler method, rpm-window-click-event-handler method, and any button action functions in those situations should not use any ACT-R commands because they may be run in the Lisp event handler thread instead of the ACT-R model running thread. Another potential problem related to that is that because those event handlers are running asynchronously relative to ACT-R and are designed for interacting with people the model(s) should be run in real-time so as to reduce any potential problems with the timing of the actions as well when the model is running with real windows.

The sections below will describe ways to deal with using those actions safely within a model that is using a potentially unsafe interface. These techniques will also work with the model if it is using a safe interface like the virtual windows. The assumption here is that one always wants to use the interface methods and actions that are built-in instead of just bypassing them with some other mechanism (like !eval! calls in productions). These same mechanisms are also relevant when one is creating a new device interface which will be interacting through other threads (for

[1] While it would be possible to make the ACT-R code thread safe to avoid this issue it is not practical to do so for many reasons.

example a separate thread which is communicating over a network connection to some external simulation). The general mechanism for doing so is to provide some way for the ACT-R running thread to perform the necessary ACT-R commands. There are three basic mechanisms for doing that, and some variations available within those mechanisms.

Before describing the specific mechanisms, there is one general concept which should be addressed. Since these mechanisms will be dealing with communicating between threads in Lisp one must be careful to make sure that this code is itself thread safe. Because the threading mechanisms available within different Lisps provide different capabilities and assumptions about what is and isn't safe one may want to consult the documentation on threads for the Lisp being used to determine the best way to guarantee safety. However, there is a simple (though typically inefficient) means for guaranteeing that safety which is provided with the ACT-R interfacing tools found in the "support/uni-files.lisp" file for those Lisps which can use the ACT-R Environment (that file is loaded automatically for those Lisps). That is through the use of a process lock.

A process lock can be used to ensure that blocks of code protected by the lock cannot be run concurrently in separate threads. There are three functions provided for using locks: uni-make-lock, uni-lock, and uni-unlock. To use them one must create a lock with uni-make-lock providing a name for the lock as a string, and then for the code to be protected start it with a uni-lock on that lock and end it with a uni-unlock of the lock. Here is a very simple example showing the modification of a list being protected:

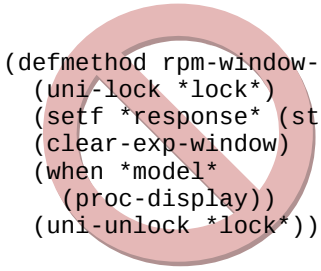
```
(defvar *lock* (uni-make-lock "demo-lock"))
(defvar *data* nil)

(defun add-data (item)
  (uni-lock *lock*)
  (push item *data*)
  (uni-unlock *lock*))

(defun remove-data ()
  (uni-lock *lock*)
  (prog1 (pop *data*)
    (uni-unlock *lock*)))
```

For example purposes we will use this rpm-window-key-event-handler which is unsafe to use when working with native windows that can be found in the demo2 model of the tutorial:

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
```

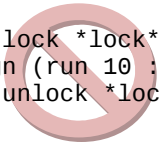
```
  (setf *response* (string key))
  (clear-exp-window)
  (when *model*
    (proc-display)))
```

The problem is that it is calling both clear-exp-window and proc-display when the model is doing the task. Note, when a person does the task it is acceptable to call ACT-R GUI functions like clear-exp-window from an event handler since ACT-R is not running, but when the model is running it is not safe to do so. Just wrapping that code with a lock like this:



```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (uni-lock *lock*)
  (setf *response* (string key))
  (clear-exp-window)
  (when *model*
    (proc-display))
  (uni-unlock *lock*))
```

is not sufficient to fix things because the ACT-R code running the model isn't protected by any locks. Trying to protect that by then wrapping a lock around the call to run the model like this:



```
...
(uni-lock *lock*)
(progn (run 10 :real-time t)
(uni-unlock *lock*)
...
```

would just result in preventing the event handler code from executing until the run had completed.

To properly fix things the actions of that event handler will need to be split so that the ACT-R commands can be executed at the appropriate time by something running in the same thread as ACT-R. For all of the example mechanisms below we will use a variable named *pressed* as a flag to indicate it is time to perform the model actions (we don't want to use the *response* variable as the flag because we need that value after the run completes). We will assume that the *lock* variable is bound to a lock created by uni-make-lock, and we will place all of the model actions into a separate function that will be called when appropriate:

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response* (string key))
  (uni-lock *lock*)
  (setf *pressed* t)
  (uni-unlock *lock*)
  (unless *model* (clear-exp-window)))
```

```
(defun model-pressed-key ()
  (clear-exp-window)
  (proc-display))
```

The setting of the *pressed* variable has been protected with the lock because we will be checking that variable in a different thread and do not want to try and read it while it is being set (we are assuming that setf is not atomic and thus not thread safe). In the thread that is running ACT-R we will want to test that *pressed* variable and call the model-pressed-key function when it is set. Something like this function can be used to do that:

```
(defun check-response ()
  (let (done)
    (uni-lock *lock*)
    (when *pressed*
      (model-pressed-key)
      (setf *pressed* nil)
      (setf done t))
    (uni-unlock *lock*)
    done))
```

The reading and clearing of the *pressed* variable is protected by the lock since that may be changed in a separate thread. The call to model-pressed-key happens to be protected by the lock because of how the *pressed* variable is being used, but that isn't necessary since that code will be called from the thread running ACT-R and doesn't have any connection to code which may run in other threads. This function returns t after it detects a press because that can be used by some of the mechanisms below to stop checking.

Now we will look at ways to call check-response from the thread running ACT-R. Since ACT-R will be running, that means that we must set something so that the ACT-R scheduler will execute check-response, and there are three different ways to add code which will be run by the scheduler.

The first way is to schedule an event to perform the check. However, since we don't know when the event is going to occur, we can't just schedule an event to occur at a particular time, and we will also need to keep checking until it happens so a single event will not work. There are a few ways to handle that.

One is to schedule a periodic event that will just keep firing at a specified interval to check again and again:

```
(schedule-periodic-event .05 'check-response :maintenance t)
```

That call will result in check-response being called every .05 seconds as long as the model runs. In some cases that is a very useful thing to do, for example when connected to an external simulation which is providing information to the model having it update the state for ACT-R at a specific rate can be useful. In this case however we are only expecting a single keypress and don't necessarily need that to be ongoing throughout the model run.

One alternative would be to have the periodic event call a function other than check-response which would be capable of removing the event from the queue once the key is pressed. That would look something like this:

```
(defun check-for-press ()
  (when (check-response)
    (delete-event *event*)))

(setf *event* (schedule-periodic-event .05 'check-for-press :maintenance t))
```

We save the event that is generated and when we detect the occurrence of the keypress we remove the event from the queue. The one downside to that is that it is still going to only perform the check at the specified interval. We could set that for 1ms (the resolution of ACT-R's clock) if we wanted the maximum responsiveness, but that would result in a lot of overhead in executing those events.

An alternative which avoids a lot of overhead would be to schedule the event to only occur after motor module actions since we know that the signal we're looking for is the result of a motor module action. That would be done with something like this:

```
(schedule-event-after-module :motor 'check-response :maintenance t)
```

That call alone however would not be sufficient because it's only going to create one event that will check after the first motor module action, and if the key isn't pressed at that point it won't try again. Thus, we will need to reschedule another event to check again until the keypress is detected. That rechecking event could be a periodic event like above with something like this:

```
(schedule-event-after-module :motor 'start-checking :maintenance t)

(defun start-checking ()
  (unless (check-response)
    (setf *event* (schedule-periodic-event .05 'check-for-press :maintenance t))

(defun check-for-press ()
```

```
  (when (check-response)
    (delete-event *event*)))
```

Or it could be another single event that just keeps rescheduling itself after the next event which occurs:

```
(schedule-event-after-module :motor 'start-checking :maintenance t)

(defun start-checking ()
  (unless (check-response)
    (schedule-event-after-change 'start-checking :maintenance t :delay-value nil))
```

By specifying the :delay-value as nil it guarantees that the event will occur even if there are no other events in the queue. The potential problem with that is that the subsequent checks will only occur after there is some other action performed. Thus, there is no guarantee on the timing of those checks.

There are other ways one could schedule events to perform the checking as well, but those are the only ones we will discuss here. Those examples should provide some useful details on what is necessary and issues to consider.

The next mechanism for calling code while ACT-R is running is to use an event hook. Event hooks can be set to call functions before or after every event which the model executes. Thus, a simple solution would be to just call check-response from an event hook function. The event hook functions are passed the current event as a parameter so check-response itself can't be set as the function, but something like this would work:

```
(add-post-event-hook (lambda (x) (declare (ignore x)) (check-response))
```

or, as with the periodic event, one could have it remove itself after the action occurs:

```
(defun check-event (event)
  (declare (ignore event))
  (when (check-response)
    (delete-event-hook *event*)))

(setf *event* (add-post-event-hook 'check-event))
```

The downsides of using an event hook are: there must be some events which occur otherwise the checking will stop, the resolution of the timing will be driven by the times of the events which occur, and the added overhead of performing the check for every event which occurs.

Using event hooks, one thing that might seem like it would provide protection for the handlers operating in other threads would be to use a process lock and lock it in the pre-event hook and unlock it in the post-event hook and then test that in the event handler with something like this:

```
(setf *lock* (uni-make-lock "lock-all"))

(add-pre-event-hook (lambda (x) (declare (ignore x)) (uni-lock *lock*)))

(add-post-event-hook (lambda (x) (declare (ignore x)) (uni-unlock *lock*)))

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (uni-lock *lock*)
  (setf *response* (string key))
  (clear-exp-window)
  (when *model*
    (proc-display))
  (uni-unlock *lock*))
```

That is not recommended however because it has some potentially dangerous issues. One is that there may be other event hooks set that would run before the lock is locked or after it is unlocked and thus could conflict with the key-event-handler code (the ACT-R Environment uses event hooks as do the model tracing and BOLD prediction tools). Another problem that could occur happens if the model stops because of an error in an event or if the user breaks the run. If that were to happen between the lock being locked and unlocked the lock may remain locked and prevent the event handler from ever occurring (some systems require every lock to be paired with a corresponding unlock).

The last mechanism for having code executed in the ACT-R running thread is to adjust the functions responsible for controlling the model's clock when it is run in real time model. Using this mechanism is usually not appropriate for a simple task like the example here, but can be very effective when interfacing the model to an external simulation to both keep things synchronized and provide the model with updates as they occur in the simulation. There are two functions which one can set for controlling the clock: one to provide the time and one to call when the model has to wait for time to pass (for details on how those functions are used and when they are called consult the "Configuring Real Time Operation" section of the reference manual). For this example the clock function seems like the right one to use because we do not know if the model will ever be "waiting", but it will always have to check the clock.

```
(defun new-clock ()
  (check-response)
  (get-internal-real-time))
```

```
(mp-real-time-management :time-function 'new-clock)
```

We define a function which will be used as the clock and have it call check-response. That function returns the result of get-internal-real-time so that it operates the same as the default real time clock. Then all we need to do is set that function as the clock to use with the mp-real-time-management function. That could be made to stop checking after the press has occurred as was done with the other mechanisms because calling mp-real-time-management with no parameters sets it back to the default values:

```
(defun new-clock ()
  (when (check-response) (mp-real-time-management))
  (get-internal-real-time))

(mp-real-time-management :time-function 'new-clock)
```

This only works if the model is being run in real time mode, but that is recommended when interacting with a real interface window or an external simulation.

Which of these mechanisms to choose will depend on exactly what is needed for the task and how one wants to control the interactions. More than one may be necessary for some complex situations, and using these can provide the modeler with a lot more control over what happens as the model runs. For more information on using any of the functions shown here one should consult the reference manual and the sections “Running the system” and “Scheduling Events” in particular.