Up In Alarms

Draft 2

Michael S. Engber
Apple Computer - Newton ToolBox Group

This article was (will be) published in the May 1994 issue of PIE Developers magazine. For information about PIE Developers, contact Creative Digital Systems at CDS.SEM@APPLELINK.APPLE.COM or 415.621.4252.

Introduction

This articles discuses how third party applications can set alarms on the Newton like the built-in Calendar application. The library for using alarms is included in the NTK 1.0 final platform file. Therefore, you need the NTK 1.0 final (or later) platform file to actually test out the alarms. The alarm library is different than other libraries in the NTK platform in that it permanently installs alarm code on your user's Newton. Make sure you understand this issue thoroughly before distributing code using alarms. I assume the reader has some experience using NTK to write Newton applications and is familiar with how to use library code provided in the NTK platform files.

What Alarms Can Do

The alarm library allows you to set an alarm to go off at some time in the future. If the Newton is asleep at that time the alarm will wake up the Newton. You can specify the alarm time to the nearest minute. The alarm action can be as simple as putting up a Notify alert (^ la the built-in Calendar) or it can execute a callback function to do whatever you like.

Trying Out Alarms From the Inspector

First, recall that if the NTK platform provide a function named Foo that the constant you use to refer to the function is call kFooFunc. The five constants for the alarm API are:

```
kAddAlarmFunc
kRemoveAlarmFunc
kGetAlarmFunc
kGetAppAlarmKeysFunc
kRemoveAppAlarmsFunc
```

The first thing to try is setting an alarm:

```
call kAddAlarmFunc with ("foo:PIEDTS",Time()+1,[kNotifyAlert,"foo","bar"],nil,nil);
#440D559  "foo:PIEDTS"
```

This sets an alarm to go off one minute from now. Within a minute you should see the following message in Figure 1 on the Newton screen. Remember that Time() returns the current time in minutes. If you execute the above code at 2:42:45, then Time() will return 2:42 and the alarm will be set for 2:43, 15 seconds later.

Figure 1 - Simple Alarm Notify Message

The first argument to `AddAlarm` is a string used to uniquely identify the alarm for future reference – for example, if you want to cancel it. Use your signature as a suffix to generate unique alarm keys just as you use it to generate unique package names. The second argument is the time in minutes (since January 1, 1904) that the alarm will go off. The third argument is an array of arguments to pass to Notify when the alarm goes off. The last two arguments are a call back function and arguments to be executed when the alarm goes off. This example doesn't take advantage of call back functions.

Next we can try out `GetAlarm` and `RemoveAlarm`:

```
call kAddAlarmFunc with ("foo:PIEDTS",Time()+2,[kNotifyAlert,"foo","bar"],nil,nil);
#440FA69  "foo:PIEDTS"

call kGetAlarmFunc with ("foo:PIEDTS");
#44116D9  {key: "foo:PIEDTS",
           Time: 46813104,
           notifyArgs: [3, "foo", "bar"],
           callBackFn: NIL,
           callBackParams: NIL,
           _uniqueID: 2}

call kRemoveAlarmFunc with ("foo:PIEDTS");
#44116D9  {key: "foo:PIEDTS",
           Time: 46813104,
           notifyArgs: [3, "foo", "bar"],
           callBackFn: NIL,
           callBackParams: NIL,
           _uniqueID: 2}

call kGetAlarmFunc with ("foo:PIEDTS");
#2        NIL
```

This time we set the alarm for two minutes to give us enough time to execute the rest of the code. Next we use `GetAlarm` to retrieve the alarm. The frame it returns basically contains all the information we originally passed to `AddAlarm`. You should be careful not to modify this frame or its contents. The more observant among you will notice it is a soup entry. Alarms are kept in a soup. The implications of this will be discussed later in the "Warnings and Gotchas" section of this article.

The third line of code removes the alarm. If you entered it in time, the alarm will not go off. `RemoveAlarm` returns an unspecified non-nil value to indicate it successfully removed the alarm. As you can see, it returns the same thing as `GetAlarm`, but this is not a fact you should rely on.

Finally, we use `GetAlarm` again to double check that the alarm was removed. It returns nil indicating no alarm with the specified alarm key was found.

Next we'll use the following code to try out the two other functions from the alarm API.

```
call func()
begin
  local i;
  for i := 1 to 5 do
    begin
      local alarmTime := Time() + i;
      local key := i & "foo:PIEDTS";
      local notifyArgs := [kNotifyAlert,HourMinute(alarmTime),"Alarm#" && i];
      call kAddAlarmFunc with (key,alarmTime,notifyArgs,nil,nil);
    end;
end with ();
#2        NIL

call kGetAppAlarmKeysFunc with (":PIEDTS")
#440B249  ["2foo:PIEDTS", "3foo:PIEDTS", "4foo:PIEDTS", "5foo:PIEDTS"]

call kRemoveAppAlarmsFunc with (":PIEDTS")
#C        3
```

First we use a loop to set up 5 different alarms. Notice that we generate a different alarm key for each alarm. If we didn't do this we'd end up only setting a single alarm. Each time we called `AddAlarm` the new alarm would replace the one we previously set.

Next we use `GetAppAlarmKeys` to see all the alarms we set. We pass it the common suffix all our alarm keys share. You'll notice that only four alarm keys were returned. I wasn't quick enough on the enter key. The first alarm went off before I executed the call to `GetAppAlarmKeys`.

Finally, we use `RemoveAppAlarms` to remove all the remaining alarms we have installed. It takes an alarm key suffix just like `GetAppAlarmKeys` and returns the number of alarms it actually removed. Again, I wasn't quick enough to beat the next alarm we had scheduled. That's why it returned a value of three.

For more examples of how to use alarms, including using callback functions, see the PIE DTS sample code called FalseAlarm.

Alarm API Function Descriptions

This section discusses each of the alarm function in detail.

`AddAlarm`(*alarmKey,time,notifyArgs,callBackFn,callBackParams*)

return value	*alarmKey*
alarmKey	A string used to uniquely identify the alarm. You should use your signature or app symbol as a suffix to guarantee uniqueness among different applications.

For example, `"wakeUpReg:PIEDTS"`. If there is already an alarm with with the same *alarmKey*, it is removed and replaced with the new alarm.

time — Integer encoding the time of the alarm in minutes since midnight, January 1, 1904 (^ la the Time() global function).

notifyArgs — An array of 3 arguments to pass to Notify when the alarm goes off. Pass nil to indicate Notify shouldn't be called.

callBackFn — A closure to be executed when your alarm goes off. Pass nil to indicate no call back function is being used.

callBackParams — An array of parameters to be passed to *callBackFn*. Use nil if no call back functions is being used.

`AddAlarm` is used to register an alarm to execute at a specified time. The alarm will wake up the Newton if necessary. You can specify a system notify message to be displayed. You can take other actions by specifying a closure.

The time an alarm executes is approximate. There may be multiple alarms scheduled for the same time, if one of the alarms does something time consuming it can delay the execution of other alarms. In the case of simultaneously scheduled alarms the execution order is unspecified.

Once an alarm executes it's removed. You only need to call `RemoveAlarm` if you want to abort an alarm. If you want to change an existing alarm, simply call `AddAlarm` with the same alarm key. The new version of the alarm will replace the old one. If you want to schedule an alarm periodically, your *callBackFn* should call `AddAlarm` to schedule the next occurrence.

`RemoveAlarm`(*alarmKey*)

return value — nil if no alarm with *alarmKey* was found. An unspecified non-nil value if the alarm was found and removed.

alarmKey — The unique string passed to `AddAlarm` which identifies the alarm. `RemoveAlarm` unschedules an alarm. If you want your application's alarms to execute only if your application is installed, you'll need to call `RemoveAlarm` in your remove script.

`RemoveAlarm` unschedules an alarm. This function is used to abort a pending alarm.

`GetAlarm`(*alarmKey*)

return value — A frame containing the 5 slots described below or nil if no alarm is found.

key:	the unique string which identifies the alarm
time:	the time at which the alarm will "go off"
notifyArgs:	array of 3 argument for Notify (or nil)
callBackFn:	closure (or nil)
callBackParams:	array of arguments for callBackFn (or nil)

alarmKey The unique string passed to `AddAlarm` which identifies the alarm.

`GetAlarm` returns information on the specified alarm. The frame returned by `GetAlarm` and its contents should not be modified. Any undocumented slots you find in it should be ignored.

`GetAppAlarmKeys`(*alarmKeySuffix*)

return value An array of alarm keys (strings)

alarmKeySuffix A string which specifies the suffix which ends all your alarm keys. For example, `":PIEDTS"` or `":AlarmSample1:PIEDTS"`.

Returns an array of all an application's alarm keys. They will be in execution order, earliest alarm to latest alarm.

`RemoveAppAlarms`(*alarmKeySuffix*)

return value An integer, the number of alarms removed

alarmKeySuffix A string which specifies the suffix which ends all your alarm keys. For example, `":PIEDTS"` or `":AlarmSample1:PIEDTS"`.

Removes all of an application's alarms. This would be a useful routine to call in your `RemoveScript` if your alarms can't meaningfully execute when your application is not installed.

Why is the Alarm Library Different from All Other Libraries?

The NTK platform file provides a variety of libraries which extend the functionality of the Message Pad ROMs. For example, it provides a function to change an existing icon in the extras drawer, something for which no API was provided by the original Newton ROM. Generally, using these libraries simply means including some extra code in your package (somewhat similar linking in a library in to a C program).

The Alarm library is special. In addition to including some extra code in your package, it also installs code on the internal store of your user's Newton. This code uses about 4K of storage and provides alarm functionality even after your package has been removed.

The installation of the alarm functionality happens automatically, the first time any of the routines in the alarm library is called. The installation is permanent. It remains installed until the user does a cold boot.

This is a very serious matter. Your users need to be warned they may lose ~4K of internal storage if they use your application. This storage is not lost every time your application runs or for every different application that uses alarms. It is lost only once, when the alarm functionality is initially installed. Of course, on Newtons that have this functionality already built into ROM, this installation is not necessary and will not take place. The current Message Pad ROMs do not have the alarm functionality built in.

The alarm library is currently the only set of functions in the NTK platform file that does this type of permanent installation. Alarms are a special case. Many applications require the alarm functionality to persist in the Newton even if the application is not installed (e.g. on a card that has been removed). This means the code for the alarm API cannot reside in your application's package as is the case for the other libraries in the platform file.

AlarmPkgUtil

AlarmPkgUtil is a utility for programmers to use in testing their alarm code. It is distributed along with the PIE DTS FalseAlarm sample code. It allows you to easily install and remove the alarm functionality into a Newton - something your users will never have to deal with explicitly.

You will want to use AlarmPkgUtil in order to test your application's robustness. For example, test your application with the alarm functionality already installed, not previously installed, becoming installed while your application is already running, É

The user interface to the AlarmPkgUtil should be self explanatory. There is only a single button that is labeled either "Install Alarm Package" or "Remove Alarm Package" whichever is appropriate.

There is a status box below which tells you the current state of the system with respect to the installation of alarm functionality. If the state is `*error*`, the message box will contain a list of items, each preceded by a `+` or `–` character. This information may be useful to DTS in debugging any problems you encounter.

Warnings and Gotchas

Call Back Functions

Alarms are kept in a soup so they will persist across restarts and when your package is removed. This means the arguments you pass will be stored in a soup. This, in turn, means these arguments will be deeply copied. You should be careful to avoid indirectly referencing large objects.

A classic example of accidentally referencing a large object is dynamically creating closure you pass for your callback function. Closures contain references the lexical environment and the current receiver (self) at the time they are created. For example, closures created from a view method will reference the root view through their parent chain. So if you pass such a closure to `AddAlarm` the Newton will attempt to copy the entire root view into the alarm soup. Functions created in the NTK inspector also do not have empty environments. This means if you're hacking in the Inspector and you pass a closure to `AddAlarm`, the Newton will appear to hang.

The best way to avoid this problem is to create your alarm callback closures when your package is built. (i.e. make sure the `func(É)É` expression is evaluated at build-time, not at run time) Functions defined in "Project Data" or evaluate slot should will have empty environments.

Debugging your call back function will be difficult. When it executes, any exceptions raised will be caught and ignored by the alarm mechanism. I suggest debugging your call back functions thoroughly before passing them to `AddAlarm`.

Note, that when your alarm runs your application may not be open. In fact, your application may not even be installed. You need to be sure and handle these cases appropriately. For example, the following code tests for the presence of the application before sending a message to it.

```
if GetRoot().(kAppSymbol) then
 GetRoot().(kAppSymbol):DoSomething()
else
 GetRoot():Notify(...)
```

If your alarms aren't useful when your application isn't installed, you should consider simply removing them, using `RemoveAlarm` or `RemoveAppAlarms`, in your `RemoveScript`. There is no point in wasting space with useless alarms that put up Notify messages like "Sorry, this alarm can't execute since application Fwiblob isn't installed."

Using Alarm Functions From Install and Remove Scripts

Alarm functionality may not be added to a Newton during an `InstallScript` or a `RemoveScript`. This means, if the alarm functionality is not already present in the Newton and you use one of the five alarm functions it may not succeed. This situation can cause the Newton to hang, requiring a reset. Note that a reset will cause your `InstallScript` to run again. So, it may require a cold boot to finally end the cycle.

You can avoid this problem entirely by not using the alarm functions during your install or remove scripts. Alternately, your install or remove scripts can spawn deferred actions to execute alarm related code.

Using AddAlarm and RemoveAlarm as Global Functions

You will find that once the alarm functionality has been installed on a Newton that you can simply call `AddAlarm` and `RemoveAlarm` as if they were part of the ROM. However, this will not be the case for the rest of the functions in the alarm API. Do not take advantage of this in your application. Remember, your users will not initially have the alarm functionality installed in their Newtons.

Installation and Removal of Alarm Functionality

It is important that you **do not delete the alarm soup** (take note, "Souper" users). If you do so, AlarmPkgUtil will not work properly until you cold-boot the Newton. The details about the alarm soup are being purposely left undocumented. You should only interact with it indirectly, through the alarm API functions.

Using AlarmPkgUtil is the only safe way to remove the alarm functionality from a Newton without doing a cold-boot. The AlarmPkgUtil will also let you check if a Newton already has the alarm functionality installed.

Alarms and Sound

You may have noticed the Alarm Sound Effects checkbox in the sound preferences panel. This controls whether or not the system beeps when an alarm goes off. If your alarms make their own sounds you may want to tell your users to turn off this setting in Prefs.

Courteous Usage of Alarms

Applications need to be courteous in their usage of alarms. Limiting your applications to a single alarm might be too restrictive. On the other hand, scheduling a daily wake-up alarm for the next year by scheduling 365 different alarms would chew up a lot of the internal store. Use reasonable judgment. Remember, each alarm you schedule takes up space in the Newton's internal store.

Similarly, your alarm actions should be quick so you don't block other alarms. If you need to do something time consuming or repetitive, use a deferred action or set up an idle script.

Recall that idle scripts stop when the Newton goes to sleep. So using an idle script can only ensure the user notices your alarm next time they turn on the Newton. You can set more alarms to make sure the Newton wakes up periodically and tries to get the user's attention. However, each wake up uses power so you probably want to put a limit on this or you will eventually drain their Newton.