Technical Note TN1033

Interrupts in Need of (a Good) Time

CONTENTS

Defining the Problem

Techniques for Solving the Problem

Summary

References

Downloadables

One of the classic problems of Macintosh programming is that your code is executing at interrupt time and you wish to execute a Mac OS routine that cannot be called at interrupt time. This Technote presents a number of techniques you can use for communicating between interrupt time code and task level code, along with an analysis of each method. It concludes with a recommendation for the best general-purpose method for solving the problem.

This Note is intended for Macintosh application developers who are writing interrupt time code that needs to execute code at task level. The Note focuses on traditional Macintosh Operating System techniques but it also analyzes the likelihood of future compatibilty for each approach. The Note may also be useful for traditional Mac OS device driver writers, i.e., those who are writing drivers of type `'DRVR'`.

Updated: [Feb 1 1996]

Defining the Problem

Before discussing some of the techniques you can use for communicating between interrupt time code and task level code, it's important to note that the Mac OS supports three execution levels:

- hardware interrupt time
- deferred task time
- task level, also known as system task time

In common parlance, hardware interrupt and deferred task time are collectively referred to as "interrupt time." Application developers usually encounter interrupt time when writing ioCompletion routines, which typically run at deferred task time.

The most important consequence of these different execution levels is that you can only call a limited set of Mac OS routines from interrupt time. For example, you might have an application that issues an asynchronous call to read some data off the network. The call completes and executes your `ioCompletion` at deferred task time. Inside your `ioCompletion` routine you want to call a Mac OS routine that can't be called at interrupt time, such as `NewHandle`. What do you do?

Back to top

Techniques for Solving the Problem

Over time, a number of techniques have been used to solve the problem of calling task level code from interrupt time. Some, but not all, of them are described here, along with an analysis of the pros and cons of each approach.

Solution Framework

All of the solutions that follow, essentially, take the same approach. The program contains two threads of execution, one that runs at interrupt time and the other that runs at task level time. When it executes, the interrupt time code queues a request (using the queue management routines described in *Inside Macintosh: Operating System Utilities*) on to some global queue. The task level code then polls this queue and processes any requests it finds.

The following snippets demonstrate this technique. First, we declare two queues, one that contains a pool of free queue elements (`free_queue`) and the other which contains a list of pending requests (`request_queue`). These are declared as simple extensions to the QElem record declared in OSUtils.h.

```
typedef struct qQElem qQElem;
typedef qQElem *qQElemPtr;
struct qQElem {
    qQElemPtr qLink;
    short qType;
    requestProc request;
    long refcon;
};

static QHdr free_queue;        /* List of queue elements that are
                    currently unused. */

static QHdr request_queue;    /* List of queue elements that
                    hold pending requests. */
```

Populating the free queue is left as an exercise for the reader.

The `QQueuesNewRequest` routine is called at interrupt time to indicate the request procedure to be called at the next available task level time.

```
pascal OSStatus QQueuesNewRequest(requestProc request, long refcon)
    /* Add a request to the "to do" queue.  Request must be a
    native procedure, there is no MixedMode magic in here!*/
{
    OSStatus err;
    qQElemPtr free_element;

    err = noErr;
    /* Get an element from the free queue. */
    free_element = (qQElemPtr) QQueuesGetQueueElement(&free_queue);
    if (free_element == nil) {
        err = noFreeQueueElementsErr;
    }

    /* Now fill out the fields of the element and add it to the
    list of queued requests. */
    if (err == noErr) {
        free_element->request = request;
        free_element->refcon = refcon;
        Enqueue((QElemPtr) free_element, &request_queue);
    }

    return (err);
}
```

The QQueuesProcessRequests routine is called at task level time to execute all of the pending requests.

```
pascal OSStatus QQueuesProcessRequests(void)
    /* Process each of the queued requests.*/
{
    qQElemPtr request;

    do {
        /* Get an element off the "to do" queue. */
        request = (qQElemPtr) QQueuesGetQueueElement(&request_queue);
        if (request != nil) {
            request->request(request->refcon);
                /* Do the request... */
            Enqueue((QElemPtr) request, &free_queue);
                /* ... and put it back on the free
                queue. */
        }
    } while (request != nil);
    return (noErr);
}
```

Oh, and just for the sake of completeness, the QQueuesGetQueueElement routine is a utility routine called by the previous two routines.

```
static QElemPtr QQueuesGetQueueElement(QHdrPtr queue)
    /* An interrupt safe mechanism to remove and return the
    first element of queue. */
{
    OSStatus err;
    QElemPtr first_element;

    /* Pull the first element off the queue, spinning if it
    disappears while we're looking at it.*/
    do {
        first_element = queue->qHead;
        if (first_element != nil) {
            err = Dequeue(first_element, queue);
        }
    } while ((first_element != nil) && (err != noErr));

    /* Return it. */
    return first_element;
}
```

So the original problem can now be restated as "How do I get periodic time in order to process requests?"

Approach #1: Patching SystemTask, Installing a jGNEFilter, et al

One obvious approach is to patch some commonly called trap (for example, `SystemTask`) or low memory global and use it to periodically check for the queued requests. Historically, this is a very common technique, largely inherited from the original Mac OS which could only run one application at a time.

The drawback to this approach is that it involves either patching or changing low memory globals, both of which are considered bad. Still, if you have already written an extension that installs a `SystemTask` patch or a `jGNEFilter`, this technique might be useful.

> **Note:**
> If you're installing a `jGNEFilter` you should check out Pete Gontier's jGNE Helper [soon to be] on the developer CD series.

Approach #2: Installing a Device Driver in the Device Manager's Unit Table

Another commonly used technique is to install a device driver in the Device Manager's unit table and set the `dNeedTime` bit in the `DCtlEntry`. The Mac OS will then periodically call the device with the accRun control code.

This technique has a number of drawbacks. First, it requires you to install a driver into the unit table, something that is tricky and may involve walking on low memory globals another compatibility liability.

The second drawback is a bit more obscure. If another device driver (or desk accessory) brings up a modal dialog in its `accRun` handler, your device driver won't get time, even though the other driver is calling `SystemTask`. This is because the system explicitly guards against dispatching `accRun` events reentrantly. This is in direct contradiction to the statements in the old Technical Note DV 19 - "Drivers & DAs in Need of (a Good) Time."

Incidentally, the other main point of Technote DV 19 -- that traditional Mac OS device drivers should be careful about which heap they're allocating their storage in -- is still very relevant.

The third drawback is that device drivers of type `'DRVR'` as we know them under System 7 are rapidly becoming a thing of the past. The new guidelines for writing native device drivers on PCI machines have explicitly outlawed the practice of making Toolbox calls from a device driver.

In summary, this approach is only appropriate if you're working on a traditional Mac OS device driver of type `'DRVR'`, not `'ndrv'`.

Approach #3: Posting a Notification Request

This technique is hinted at in *Inside Macintosh: Devices* and further described in Macintosh Technical Q &A NW 13. The gist of the idea is to use NMInstall to post a notification request with a response procedure but no sound, icon, string or mark. The Notification Manager polls its internal queue of notification requests at task level and calls your response procedure.

The technique works well. Contrary to popular belief, the Notification Manager does not serialize all requests and so your response procedure will be called even if there is another notification dialog up.

One caveat is that your notification response procedure is called in the context of some other application. You should tread lightly! Be careful about allocating too much memory and don't make assumptions about the current resource chain.

About the only problem with this approach is that it works against the spirit of the Notification Manager, which is meant as a simple method for notifying users about asynchronous tasks, not as a Poor Droid's Scheduling System. Before using this technique, make sure you read Technical Note TB 39 "Toolbox Karma."

> **Note:**
> This is an excellent opportunity to reiterate the advice that you should not leave a notification dialog posted indefinitely. If the user doesn't respond to your notification within a minute or so, you should use `NMRemove` to cancel it and repost it at some later time. A notification dialog on the screen will prevent the foreground process from executing, which will seriously annoy a user who has left a ray tracer running overnight only to find that "The file server Womble has unexpectedly shut down" and that the ray tracer has unexpectedly not finished.

Approach #4: Application Processing Requests

This technique uses an application to process the queued requests in its main event loop. Your interrupt routine puts the request onto a queue in the application's globals and wakes up the application using WakeUpProcess. When the application runs, it looks at the global queue and processes any requests on it.

If you don't have a suitable application handy, you can just create a background-only application (BOA) dedicated to this function. You can even put an INIT resource in BOA, as described in Technical Note PS 2 - "Background-Only Applications."

This technique is the best general-purpose method for solving the problem. It involves no trap patches and it doesn't touch low memory.

The only drawback to this approach is the memory requirements (approximately 50K) if you need a dedicated BOA to process requests. However, if you already have an application running, this technique is definitely the way to go.

Approach #5: Open Transport

Open Transport provides a good technique for scheduling task level time from interrupt time, namely `OTScheduleSystemTask`. OpenTransport will run on most modern Macintoshes and provides this service as an adjunct to its networking facilities.

However, using OpenTransport restricts the systems your code will operate on. The decision as to whether to use this technique, and hence make your software dependent on OpenTransport, is for you to make based on both technical and marketing considerations.

Approach #6: Software Interrupts

Another approach that will be available in the future is the software interrupt routines, currently provided as part of the PCI driver services library. See Technote 1001, "On Power Macintosh Interrupt Management," for details.

Back to top

Summary

There are a variety of techniques you can use to process requests received at interrupt time. The technique you choose depends on a number of factors outlined in this Technote. An application processing the requests on behalf of your interrupt code is the best general-purpose method for solving the problem because it represents the least compatibility liability.

Back to top

References

Inside Macintosh: Operating System Utilities

Inside Macintosh: Devices

Inside Macintosh: Processes

Designing PCI Cards and Drivers for Power Macintosh Computers

Technical Note PS 2 - Background-Only Applications

"Be Our Guest: Background-Only Applications in System 7" in *develop* Issue 9.

Technical Note DV 19 - Drivers & DAs in Need of (a Good) Time

Technical Note TB 39 - Toolbox Karma

Macintosh Technical Q &A NW 13

PCI Device Driver article in *develop* (May 1995)

Back to top

Downloadables

 Acrobat version of this Note (K). Download

Back to top
