New Technical Notes

Macintosh

Developer Support

Deferred Task Traps, Truths, and Tips

Processes

Written by: Jim Luther July 1992

This Technical Note shows how to determine when the Deferred Task Manager is available, points out a compatibility issue with the Macintosh Plus, explains how and when deferred tasks are called, and shows how to access the `dtParm` parameter from deferred tasks written in C and Pascal.

Contrary to a belief started by a Reader's Guide note in Inside Macintosh, the Deferred Task Manager *is* useful for all lengthy interrupt time tasks, even those not initiated by slot cards. Deferring a task is useful anytime interrupt time code (that includes completion routines, Time Manager tasks, etc.) is going to perform a lengthy task that can be accomplished with interrupts enabled (processor priority level 0). This note contains information beyond that included in Inside Macintosh that you may find useful when using deferred tasks.

When you can use the Deferred Task Manager

Applications cannot always depend on having the services of the Deferred Task Manager. In particular, the Deferred Task Manager is not available on the Macintosh Plus and Macintosh SE running Macintosh System 6. There is also no support for the Deferred Task Manager in versions of A/UX prior to version 3.0. Macintosh System 7 adds the Deferred Task Manager to the Macintosh Plus and Macintosh SE.

You should always check to make sure the Deferred Task Manager is available with the following code which checks for the existence of the `_DTInstall` trap.

In Pascal:

```
gHasDeferredTasks := TrapAvailable(_DTInstall);
{see Inside Macintosh Volume VI Chapter 3 for TrapAvailable function }
```

In C:

```
gHasDeferredTasks = TrapAvailable(_DTInstall);
{see Inside Macintosh Volume VI Chapter 3 for TrapAvailable function }
```

Deferred Tasks on the Macintosh Plus

As the compatibility chapter of Inside Macintosh Volume VI advises, "…you should try to avoid even well-known global variables because they may not be available in all environments or in the future." Here's a good example.

Although DTInstall is available on all Macintoshes running System 7, the patch that added the Deferred Task Manager to the Macintosh Plus could not use the low-memory globals DTQueue (the deferred task queue header) and JDTInstall (the jump vector for the DTInstall routine). On the Macintosh Plus, the queue header used for deferred tasks is private and there is no jump vector to the DTInstall routine. Don't use DTQueue or JDTInstall on the Macintosh Plus.

How and when the Deferred Task Manager calls deferred tasks

Since understanding how something works usually answers most questions about it, this note gives a complete explanation of how deferred tasks get called. I'll start with the occurrence of an interrupt.

When an interrupt occurs, the processor stops executing the current program and control is passed to the Macintosh's primary interrupt handler. The primary interrupt handler saves registers A0-A3/D0-D3 and then decides what secondary interrupt handler to dispatch to. If the system has an MMU, was booted 24-bit mode but is presently in 32-bit mode, then the MMU state is saved and the MMU is switched to 24-bit mode. If the system has power management (i.e., is a portable Macintosh) and is in slow speed, the speed is saved and is then switched to full speed. Control is then passed to the secondary interrupt handler which handles the interrupt.

When control returns to the primary interrupt handler from the secondary interrupt handler, the primary interrupt handler checks to see if there are any deferred tasks in the deferred task queue and checks to see if upon exit of this interrupt, the processor priority level will be returned to 0. If both conditions are true, control is passed to the Deferred Task Manager to execute those tasks.

The Deferred Task Manager checks to see if a deferred task is already active; if so, the Deferred Task Manager exits. Otherwise, one by one, each deferred task in the deferred task queue is dequeued, and then called with register A1 containing dtParm from the DeferredTask record and the processor priority level is set to 0 (i.e., all interrupts are enabled). When all deferred tasks have been dequeued and called, control is returned to the primary interrupt handler.

When control returns to primary interrupt handler from the Deferred Task Manager, the primary interrupt handler restores the MMU state and speed if necessary and then restores registers A0-A3/D0-D3. Then the primary interrupt handler returns control to the program executing before the interrupt occurred.

With that explanation out of the way, here are a few things that may or may not be obvious.

Deferred tasks are interrupt time code

That is, deferred tasks are run as a result of an interrupt. A deferred task must follow many of the same rules as a interrupt handler. Here's a warning to go along with the material in the Deferred Task Manager chapter of Inside Macintosh Volume V.

Warning: Deferred tasks are executed at the interrupt level and must preserve all registers other than `A0-A3` and `D0-D3`. Your deferred task must not make any calls to the Memory Manager, directly or indirectly, and can't depend on handles to unlocked blocks being valid. If it uses application globals, it must also ensure that register `A5` contains the address of the boundary between the application globals and the application parameters; for details, see Setting and Restoring the `A5` Register in the Memory Management chapter of Inside Macintosh Volume VI. Your deferred task should avoid accessing a low-memory variables or calling a trap that would access one because while MultiFinder® or System 7 is running, application low-memory global variables are being swapped in and out.

Deferred tasks can be interrupted

That's the purpose of using deferred tasks - to let interrupt driven processes keep running while your deferred task completes a lengthy task. This means that your deferred task must be interruptible—that is, you should not have to do anything such as timing-critical hardware access that can get screwed up with interrupts on.

If the interrupt handler that installs the deferred task can be called again while that deferred task is executing, then you also need to make sure the way the interrupt handler and the deferred task code work together is reentrant. For example, the interrupt code should not modify a data buffer that the deferred task is processing.

Deferred tasks are not prioritized

Deferred tasks are executed in the order they were added to the deferred task queue no matter what interrupt level the code installing the task may have been running at.

Deferred task latency

There may be significant latency incurred between installing a deferred task with `DTInstall` and when that task is performed. This time is not deterministic and cannot be controlled. Real-time response cannot be guaranteed on Macintosh computers with or without the use of deferred tasks.

About the `DeferredTask` record

When you install a deferred task with `DTInstall`, you are giving your `DeferredTask` record to the system; the system owns the record until the routine designated by the `DeferredTask` record's `dtAddr` field is entered. During the time the system owns a `DeferredTask` record, you must not change any of the fields of that record. Similarly, you must not depend on the contents of the record when it is owned by the system. The system is free to alter the record in any fashion while it has ownership and restores the record to its original state prior to invoking the deferred task. You must not install a

`DeferredTask` record which is already owned by the system. Finally, you must not use the `dtReserved` field for any purpose; it should be set to zero prior to installing a deferred task with `DTInstall`.

Accessing `dtParm` from deferred tasks written in Pascal and C

All it takes is a little assembly language glue to get the value of `dtParm` which is passed to the deferred task in register `A1`. Here are examples in Pascal and C showing how to get the value of `dtParm` from register `A1`.

In Pascal:

```
FUNCTION GetDtParm: LongInt;
INLINE
   $2E89;   { MOVE.L  A1,(SP) }

PROCEDURE DoDeferredTask(dtParm: LongInt);
BEGIN
   { your deferred task code goes here }
END;

PROCEDURE MyDeferredTask;
   VAR
      dtParm: LongInt;
BEGIN
   dtParm := GetDtParm;    { get value of dtParm }
   DoDeferredTask(dtParm); { do deferred task }
END;
```

In C:

```
pascal long GetDtParm(void);
   { 0x2E89 };  /* MOVE.L  A1,(SP) */

void DoDeferredTask(dtParm)
long   dtParm;
{
    /* your deferred task code goes here */
}

void MyDeferredTask(void)
{
    long   dtParm;
    dtParm = GetDtParm();   /* get value of dtParm */
    DoDeferredTask(dtParm); /* do deferred task */
}
```

Notice that the deferred task is broken into two parts: MyDeferredTask which is called by the Deferred Task Manager and DoDeferredTask which contains the deferred task code. This is done to prevent problems caused by some optimizing compilers. See Technical Note #208, Setting and Restoring A5 for details.

Further Reference:

- *Inside Macintosh*, Volume V, The Deferred Task Manager
- Technical Note M.OV.A5 —
  Setting and Restoring A5