This chapter describes how your application or device driver can use the Deferred Task Manager to defer the execution of lengthy tasks until interrupts are reenabled. Time-consuming tasks, if executed at interrupt time, can prevent the execution of interrupt tasks having the same or lower priority. The Deferred Task Manager allows you to improve interrupt handling by deferring a task until all other interrupts have been serviced.

Lengthy tasks are often initiated by slot cards. As a result, you probably need to read the information in this chapter only if your application or driver deals with slot-card interrupts. However, you can use the services provided by the Deferred Task Manager whenever you need to install a lengthy interrupt task capable of running with all interrupts enabled. You can, for example, defer the execution of completion routines, Time Manager routines, and VBL tasks.

To use this chapter, you should be familiar with interrupts and interrupt tasks in general. See the chapter "Introduction to Processes and Tasks" in this book for an overview of both interrupt and noninterrupt processing. Because the Deferred Task Manager maintains all deferred tasks in a queue until their execution, you should also be familiar with operating-system queues, as described in the chapter "Queue Utilities" in *Inside Macintosh: Operating System Utilities.*

This chapter begins with a description of interrupt priority levels and explains when you might need to use the Deferred Task Manager. Then it shows how you can use the Deferred Task Manager to defer a task. The chapter concludes with a description of the Deferred Task Manager's data structure and routine.

About the Deferred Task Manager

Every type of interrupt has an **interrupt priority level,** a number that identifies the importance of the interrupt. The microprocessor also maintains several bits in the status register of the CPU that indicate which interrupts are currently to be processed and which are to be ignored. This **processor priority** is always set to the interrupt priority level of the highest-priority interrupt currently executing. For example, if no interrupts are being serviced, the processor priority is 0. If the current application is then interrupted by a vertical retrace interrupt, the interrupt priority is set to 1 during the servicing of the interrupt and restored to 0 upon completion. If, during the servicing of the vertical retrace interrupt, a level-2 interrupt occurs, the processor priority is set to 2 during the servicing of the interrupt and restored to 1 upon completion of any level-2 interrupt tasks.

The microprocessor ordinarily services an interrupt only if its interrupt priority level is higher than the processor priority. Accordingly, when no interrupt routines are executing, the microprocessor can service any new interrupt. If, however, a slot interrupt is executing, the microprocessor ignores other slot interrupts and interrupts of lower priority. As a result, a lower-priority interrupt (for example, a vertical retrace interrupt) might not execute on schedule.

When the microprocessor is servicing one interrupt, it is said to **disable** other interrupts whose priority level is lower than or the same as that of the interrupt being serviced. This feature prevents the interruption of tasks by interrupts of lesser or equal priority. You might, however, initiate an interrupt task that does not need this extra protection. If an interrupt task takes so much time to execute that the disabling of other interrupts during execution becomes significant, you might prefer it to have your interrupt task executed at a time when all other interrupt tasks have been serviced and interrupts are reenabled. The Deferred Task Manager provides a mechanism for this purpose.

Instead of immediately performing the main work of a task, such as a slot-interrupt task, you can **defer** the task, or schedule it for execution when all interrupts have been reenabled. You do this by placing information about the task to be deferred in a **deferred task record,** which you then insert in the **deferred task queue.** The task is then known as a **deferred task.** All system interrupt handlers check the deferred task queue just before returning. If there are tasks in the queue and the microprocessor's status register is about to be reset to 0, the system interrupt handlers reenable interrupts and pass control to the Deferred Task Manager to execute all the deferred tasks.

The Deferred Task Manager checks whether a VBL task is active. If so, the Deferred Task Manager exits, and the deferred tasks remain deferred until the VBL task completes. (The VBL task is interrupt code, and so the Deferred Task Manager is called again when the Vertical Retrace Manager returns control to the primary interrupt handler.) If a VBL task is not active, the Deferred Task Manager checks whether a deferred task is already active. If so, the Deferred Task Manager exits. Otherwise, a deferred task is removed from the queue and executed. When all deferred tasks have been removed from the queue and executed, the Deferred Task Manager returns control to the primary interrupt handler.

Each interrupt task is removed from the deferred task queue before it is executed. For this reason, your interrupt code must reinstall the task record into the queue each time the task is to be deferred. If your task is simple enough that reinstalling the task record into the deferred task queue takes about as much time as doing the real work of the task, then the Deferred Task Manager is not useful for your application. Note that interrupts are disabled during the reinstallation of a task record into the deferred task queue, even though they are reenabled before the reinstalled task is executed.

Although you can use the Deferred Task Manager for all types of interrupt tasks, it is especially convenient for slot-interrupt tasks. Interrupts from NuBus™ slot devices are received and decoded by special hardware on the main logic board. This hardware generates level-2 interrupts. Because of the way the hardware works, the microprocessor must disable lower-priority interrupts until it services the level-2 interrupts (otherwise, a system error occurs). During the execution of slot-interrupt tasks, the microprocessor disables other level-2 interrupts, such as those for sound, as well as all level-1 interrupts. By using the Deferred Task Manager, you can defer the processing of slot interrupts until all of the slots are scanned. Just before returning, the slot-interrupt handler executes any tasks having records in the deferred task queue.

It is important to remember that deferred tasks are executed at the end of a hardware interrupt cycle, before the secondary interrupt handler returns. In addition, the tasks in the deferred task queue are executed only if the status register is being restored to 0 (that

is, all interrupts reenabled). If the status register is not being restored to 0, but only to some higher level, deferred tasks are not executed during that hardware interrupt cycle.

This behavior, if not properly understood, can lead to some puzzling situations. For example, applications can mask the CPU's status register to disable certain interrupts. Suppose that your application installs and activates a Time Manager task, which is triggered by level-2 interrupts. If you don't want the task to be executed during a specific period of time, you can set the status register to 2, thus disabling all level-1 and level-2 interrupts. (In this case, the status register is set to 2, but not in response to a level-2 interrupt.)

Now suppose that a level-4 interrupt occurs, perhaps triggered by the arrival of some LocalTalk data at a serial port. The LocalTalk interrupt handler is executed with the status register set to 4. That handler might install a deferred task and then return. Because the interrupt cycle is nearly complete, the system interrupt handler checks whether the status register is about to be restored to 0. In the situation described, the status register is about to be restored to 2, not to 0. As a result, any pending deferred tasks, including the newly installed LocalTalk deferred task, are ignored. Moreover, if the status register remains masked at 2, any additional deferred tasks installed by the LocalTalk interrupt handler remains queued and are not executed.

Eventually, the application that masked the status register (to disable its Time Manager task) will restore the status register to 0. At the end of the next hardware interrupt cycle, all the pending deferred tasks are finally executed.

As you can see, it's possible for an interrupt routine—in this example, the LocalTalk interrupt handler—to install a deferred task that is not executed until after some future hardware interrupt cycle. Indeed, that future hardware interrupt might well be another LocalTalk interrupt. In other words, it's possible for an interrupt routine to install a deferred task and to be called again, *before* the deferred task has been executed. It's even possible for the interrupt routine to interrupt the deferred task that it installed during some previous interrupt cycle. You need to make sure, for instance, that your interrupt code doesn't modify a data buffer that a deferred task is processing.

Keep these points in mind when you use the Deferred Task Manager to defer tasks:

- The purpose of the Deferred Task Manager is to allow lengthy interrupt tasks to be deferred until all interrupts can be reenabled.
- Deferred tasks are executed with all interrupts enabled (that is, with the status register set to 0).
- Deferred tasks are not executed if some other interrupt code is executing. For example, a deferred task will not interrupt a VBL task.
- A deferred task is not executed if some other deferred task is being executed. A deferred task cannot interrupt another deferred task.
- Deferred tasks can be interrupted.
- Deferred tasks are executed within the hardware interrupt cycle, even though the status register is set to 0 before the tasks are executed. As a result, deferred tasks are subject to all the normal limitations on interrupt-level code. In particular, deferred tasks cannot call any routine that directly or indirectly allocates or moves memory,

and cannot depend on the validity of unlocked handles. See the chapter "Introduction to Processes and Tasks" in this book for a complete description of these limitations.

- Deferred tasks are not prioritized. They are executed in the order they were added to the deferred task queue, no matter what interrupt level the code that installed them was running at.

Using the Deferred Task Manager

You can use the Deferred Task Manager to defer the execution of some code that is to be executed as a result of an interrupt. This section shows how to install a deferred task and how to use a high-level language to access the optional parameter passed to your task in register A1. Because the Deferred Task Manager is not available in all operating environments, you need to check that it is available before using it. The following section shows how to do this.

Checking for the Deferred Task Manager

The Deferred Task Manager was introduced primarily to allow slot handlers to defer lengthy processing initiated by a slot interrupt and, until system software version 7.0, was not available on all computers running the Macintosh Operating System. For example, the Deferred Task Manager is not available on Macintosh Plus or Macintosh SE computers running system software version 6.0. In addition, there is no support for the Deferred Task Manager in versions of A/UX earlier than version 3.0.

As a result, you should always make sure that the Deferred Task Manager is available in the current operating environment before attempting to use it. You can use the function `DeferredTasksAvailable`, defined in Listing 6-1, to do this.

Listing 6-1 Checking for the availability of the Deferred Task Manager

```
FUNCTION DeferredTasksAvailable: Boolean;
CONST
   _DTInstall = $A082;
BEGIN
   DeferredTasksAvailable := TrapAvailable(_DTInstall);
END;
```

The `DeferredTasksAvailable` function simply calls the function `TrapAvailable` to determine whether the trap `_DTInstall` is implemented. See the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities* for a definition of the `TrapAvailable` function.

System software versions 7.0 and later support the Deferred Task Manager on all Macintosh computers, including the Macintosh Plus and Macintosh SE. However, the

system global variables DTQueue (containing the address of the deferred task queue header) and jDTInstall (containing the jump vector for the DTInstall function) are not supported on the Macintosh Plus. You should not use DTQueue or jDTInstall on the Macintosh Plus.

Installing a Deferred Task

The Deferred Task Manager provides a single routine, DTInstall, that you can use to install elements into the deferred task queue. The deferred task queue is a standard operating-system queue whose elements are defined by the DeferredTask data type.

```
TYPE DeferredTask =
RECORD
   qLink:      QElemPtr;   {next queue entry}
   qType:      Integer;    {queue type}
   dtFlags:    Integer;    {reserved}
   dtAddr:     ProcPtr;    {pointer to task}
   dtParm:     LongInt;    {optional parameter passed in A1}
   dtReserved: LongInt;    {reserved; should be 0}
END;
```

Your application or driver needs to fill in only the qType, dtAddr, and dtReserved fields. The dtAddr field specifies the address of the routine whose execution you want to defer. You can also specify a value for the dtParm field, which contains an optional parameter that is loaded into register A1 just before the routine specified by the dtAddr field is executed. The dtFlags and dtReserved fields of the deferred task record are reserved. You should set the dtReserved field to 0.

Listing 6-2 defines a routine, InstallDeferredTask, for installing a task element in the deferred task queue. This element corresponds to the routine MyDeferredTask, which does the real work of your interrupt task. The InstallDeferredTask routine sets up a deferred task record and then installs it in the deferred task queue by calling the DTInstall function. Note that you should call DTInstall only at interrupt time.

Listing 6-2 Installing a task into the deferred task queue

```
PROCEDURE InstallDeferredTask (theTask: DeferredTask);
VAR
   myErr:   OSErr;
BEGIN
   WITH theTask DO
   BEGIN
      qType := ORD(dtQType);     {set the queue type}
      dtAddr := @MyDeferredTask; {set address of deferred task}
      dtParm := 0;               {no parameter needed here}
```

```
      dtReserved := 0;           {clear reserved field}
   END;
   myErr := DTInstall(@theTask);
END;
```

Defining a Deferred Task

You define a deferred task as a procedure taking no parameters and put the address of that procedure in the deferred task element whose address you pass to the `DTInstall` function. When your task is executed, register A1 contains the optional parameter that you put in the `dtParm` field of the task record.

If you write your deferred task in a high-level language, such as Pascal, you might need to retrieve the value loaded into register A1. The function `GetA1` defined in Listing 6-3 returns the value of the A1 register.

Listing 6-3 Finding the value of the A1 register

```
FUNCTION GetA1: LongInt;
INLINE
   $2E89;         {MOVE.L A1,(SP)}
```

You can call `GetA1` in your deferred task, as illustrated in Listing 6-4.

Listing 6-4 Defining a deferred task

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

PROCEDURE MyDeferredTask;
VAR
   myParm:     LongInt;
BEGIN
   myParm := GetA1;        {retrieve parameter put in register A1}
   DoDeferredTask(myParm); {run the deferred task}
END;
```

Note that `MyDeferredTask` calls `GetA1` to retrieve the parameter passed in the register A1. Then `MyDeferredTask` calls the application-defined procedure `DoDeferredTask`, passing it that parameter. The `DoDeferredTask` procedure does the real work of the deferred task. (This division into two routines is necessary to prevent problems caused by some optimizing compilers.)

Deferring a Slot-Based VBL Task

As indicated earlier in this chapter, you are most likely to use the Deferred Task Manager when dealing with slot interrupts. All slot interrupts, including slot-based VBL interrupts, disable all other slot interrupts. For this reason, as a slot-interrupt routine (installed using `SIntInstall`) or a slot-based VBL interrupt routine (installed using `SlotVInstall`) runs to completion, interrupts at that level and below are disabled. You can help improve interrupt handling by using the Deferred Task Manager to defer your slot-interrupt processing until interrupts have been reenabled.

Listing 6-5 provides another example of how to use the Deferred Task Manager. The program defined there defers the cursor updating that would normally occur as a slot-based VBL task. The time required to update the cursor can range from about 700 to 900 microseconds for monitors having a screen depth of 1 to 8 bits. Because the cursor updating is done at slot-based VBL time, all other slot interrupts are put off until updating is finished. This might adversely affect interrupt processing by your application. Accordingly, it is useful to defer the cursor updating to noninterrupt time by installing the updating as a deferred task.

The program defined in Listing 6-5 replaces the cursor-updating routine pointed at by the system global variable `jCrsrTask` with a different routine. This new routine installs the original routine as a deferred task.

Listing 6-5 Deferring cursor updating to noninterrupt time

```
*** MyDefTask
TaskBegin
MyDefTask
   DC.L        0                ;qLink   (handled by OS)
   DC.W        0                ;qType   (queue type: dtQType)
   DC.W        0                ;dtFlags (reserved)
   DC.L        0                ;dtAddr  (pointer to routine to be executed)
   DC.L        0                ;dtParm  (optional parameter; not used here)
   DC.L        0                ;dtReserved (should be zero)
SysCrsrTask
   DC.L        0                ;pointer to system jCrsrTask
DefCrsrFlag
   DC.W        0                ;1 if using a deferred task, 0 otherwise
PendingFlag
   DC.W        0                ;1 if a jCrsrTask is pending, 0 otherwise

*** MyjCrsrTask
MyjCrsrTask
   MOVEM.L     A0/A1/D0,-(SP)
   LEA         PendingFlag,A0   ;see if a deferred jCrsrTask task is pending
   TST.W       (A0)
```

```
   BNE.S       bailOut           ;if yes, exit
   MOVE.W      #1,(A0)           ;if no, set the pending flag
   LEA         MyDefTask,A0      ;point to our deferred task element
   LEA         DefjCrsrTask,A1   ;get address of deferred task routine
   MOVE.L      A1,dtAddr(A0)     ;set up pointer to routine
   MOVE.W      #dtQType,dtType(A0)     ;set queue type
   _DTInstall                    ;install the task
   MOVEM.L     (SP)+,A0/A1/D0
   RTS
bailOut
   MOVEM.L     (SP)+,A0/A1/D0
   RTS
DefjCrsrTask
   MOVEM.L     A0,-(SP)
   LEA         SysCrsrTask,A0    ;get system cursor task address
   MOVEA.L     (A0),A0
   JSR         (A0)              ;and call it
   LEA         PendingFlag,A0    ;clear pending call flag
   CLR.W       (A0)
   MOVEM.L     (SP)+,A0
   RTS
TaskEnd

*** Entry
TaskSize       EQU   TaskEnd-TaskBegin

Entry
   MOVE.L      #TaskSize,D0      ;put TaskSize into D0
   _NewPtr     SYS,CLEAR         ;make a block in the system heap
   BNE.S       Quit              ;no room in system heap, so quit
   MOVE.L      0,A2              ;got a good pointer; keep a copy
   MOVE.L      A0,A1             ;set up registers for BlockMove
   LEA         MyDefTask,A0
   MOVE.W      #TaskSize,D0
   _BlockMove                    ;copy the task etc. into system heap
   LEA         dtQElSize(A2),A0  ;move original task pointer into our
   MOVE.L      jCrsrTask,(A0)    ; pointer holder
   LEA         dtQElSize+4(A2),A0;replace jCrsrTask pointer with a pointer
   MOVE.L      A0,jCrsrTask      ; to our jCrsrTask

Quit
   RTS                           ;exit the program
END
```

This code allocates a block of memory in the system heap. The allocated block is large enough to hold a deferred task element, a pointer to the original cursor-updating routine, and the replacement routine. The replacement routine simply retrieves the relevant information (namely, the deferred task element and the saved address of the original cursor-updating routine) stored in that block of memory and calls `_DTInstall` to install a deferred task. The address of the replacement routine is placed into the low-memory global variable `jCrsrTask`, whose original contents are stored in the system heap.

Once the program defined in Listing 6-5 is run, the cursor-updating routine is subsequently performed with interrupts enabled, thereby allowing other interrupts. Because the cursor-updating routine is run with interrupts enabled, you may see a slight flickering of the cursor when using this technique.

Deferred Task Manager Reference

This section summarizes the structure of the deferred task record and describes the `DTInstall` function, which you can use to install a deferred task record into the deferred task queue. It also describes the application-defined deferred task.

Data Structure

The deferred task queue is a standard operating-system queue. The `DeferredTask` data type defines an element in the deferred task queue.

```
TYPE DeferredTask =
RECORD
   qLink:      QElemPtr;   {next queue entry}
   qType:      Integer;    {queue type}
   dtFlags:    Integer;    {reserved}
   dtAddr:     ProcPtr;    {pointer to task}
   dtParm:     LongInt;    {optional parameter passed in A1}
   dtReserved: LongInt;    {reserved; should be 0}
END;
```

Field descriptions

`qLink` A pointer to the next entry in the deferred task queue, or `NIL` if there are no more entries in the queue. You do not need to set this field; the Deferred Task Manager does it for you.

`qType` The queue type. You must set this field to `ORD(dtQType)`.

`dtFlags` Reserved.

`dtAddr` A pointer to the task to be executed. Set this field to the address of the routine that you want to execute after interrupts have been enabled.

`dtParm`	An optional parameter that is loaded into register A1 just before the routine specified by the `dtAddr` field is executed.
`dtReserved`	Reserved. You should set this field to 0.

Deferred Task Manager Routine

The Deferred Task Manager provides a single routine for installing task records into the deferred task queue, the `DTInstall` function.

DTInstall

After defining the fields of a deferred task record, you can call the `DTInstall` function to install the record into the deferred task queue.

```
FUNCTION DTInstall (dtTaskPtr: QElemPtr): OSErr;
```

`dtTaskPtr` A pointer to a queue element to add to the deferred task queue.

DESCRIPTION

The `DTInstall` function adds the specified task record to the deferred task queue. Your application should fill in all fields of the task record except `qLink` and `qFlags`.

Ordinarily, you call `DTInstall` only at interrupt time. The `DTInstall` function does not actually execute the routine specified in the `dtAddr` field of the task record. Each system interrupt handler executes routines stored in the deferred task queue after reenabling interrupts. After a routine in the queue is executed, it is removed from the deferred task queue.

If the `qType` field of the task record is not set to `ORD(dtQType)`, `DTInstall` returns `vTypErr` and does not add the record to the queue. Otherwise, `DTInstall` returns `noErr`.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `DTInstall` are

Registers on entry

A0 Pointer to new queue entry

Registers on exit

D0 Result code

To reduce overhead at interrupt time, instead of executing the `DTInstall` trap, you can load the jump vector `jDTInstall` into an address register other than A0 and execute a `JSR` instruction using that register.

RESULT CODES

`noErr`	0	No error
`vTypErr`	–2	Invalid `qType` value (must be `ORD(dtQType)`)

Application-Defined Routine

The Deferred Task Manager allows your interrupt routines to install an application-defined routine whose execution is deferred until after all interrupts are reenabled.

Deferred Tasks

You pass the address of an application-defined deferred task in the `dtAddr` field of a deferred task record.

MyDeferredTask

A deferred task has the following syntax:

```
PROCEDURE MyDeferredTask;
```

DESCRIPTION

The `dtAddr` field of a deferred task record contains the address of a procedure that is executed at the end of a hardware interrupt cycle when all interrupts are reenabled.

SPECIAL CONSIDERATIONS

Because the deferred task is executed during a hardware interrupt cycle, it should not allocate, move, or purge memory (either directly or indirectly) and should not depend on the validity of handles to unlocked blocks.

If a deferred task uses application global variables, it must ensure that register A5 contains the address of the boundary between the application global variables and application parameters. For details, see the discussion of setting up and restoring the A5 register in the chapter "Memory Management Utilities" in *Inside Macintosh: Memory*.

A deferred task should avoid accessing system global variables or calling a trap that would access one.

ASSEMBLY-LANGUAGE INFORMATION

When the deferred task is called, register A1 contains the value of the `dtParm` field in the deferred task record passed to `DTInstall`.

A deferred task must preserve all registers other than A0–A3 and D0–D3.

Summary of the Deferred Task Manager

Pascal Summary

Data Type

```
TYPE DeferredTask =
   RECORD
      qLink:      QElemPtr;         {next queue entry}
      qType:      Integer;          {queue type}
      dtFlags:    Integer;          {reserved}
      dtAddr:     ProcPtr;          {pointer to task}
      dtParm:     LongInt;          {optional parameter passed in A1}
      dtReserved: LongInt;          {reserved; should be 0}
   END;
```

Deferred Task Manager Routine

```
FUNCTION DTInstall          (dtTaskPtr: QElemPtr): OSErr;
```

Application-Defined Routine

```
PROCEDURE MyDeferredTask;
```

C Summary

Data Type

```
struct DeferredTask {
      QElemPtr    qLink;            /*next queue entry*/
      short       qType;            /*queue type*/
      short       dtFlags;          /*reserved*/
      ProcPtr     dtAddr;           /*pointer to task*/
      long        dtParm;           /*optional parameter passed in A1*/
      long        dtReserved;       /*reserved; should be 0*/
};
```

Deferred Task Manager Routine

```
pascal OSErr DTInstall      (QElemPtr dtTaskPtr);
```

Application-Defined Routine

```
pascal void MyDeferredTask  (void);
```

Assembly-Language Summary

Deferred Task Manager Queue Element

0	`qLink`	long	pointer to next queue entry
4	`qType`	word	queue type
6	`dtFlags`	word	reserved
8	`dtAddr`	long	pointer to task
12	`dtParm`	long	optional parameter to be passed in A1
16	`dtReserved`	long	reserved; should be 0

Global Variables

`DTQueue`	10 bytes	Deferred task queue header.
`jDTInstall`	long	Jump vector for `DTInstall` function.

Result Codes

`noErr`	0	No error
`vTypErr`	–2	Invalid `qType` value (must be `ORD(dtQType)`)