This chapter describes the Vertical Retrace Manager, the part of the Operating System that schedules and executes recurrent tasks during vertical retrace interrupts. You can use the Vertical Retrace Manager to execute simple, repetitive tasks and avoid having to execute those tasks repeatedly in your application's main event loop.

You should read the information in this chapter if you want your application to schedule tasks for execution during a vertical retrace interrupt. For example, you can use the Vertical Retrace Manager to cycle among a series of cursors while some lengthy operation is happening, thus presenting the illusion of a spinning cursor.

In general, you should use the Vertical Retrace Manager only when you need to synchronize actions with the redrawing of the screen or when the tasks don't need to be executed at very precise intervals. As explained later in this chapter, certain conditions can cause the Operating System to turn off vertical blanking interrupts for a period of time. When this happens, the tasks in the vertical retrace task queue are not executed as scheduled. As a result, you should not use the Vertical Retrace Manager to handle tasks that must be executed consistently or with precise timing. For precise, uninterrupted task execution, you should use the Time Manager. See the chapter "Time Manager" in this book for details.

To use this chapter, you need to be familiar with interrupt-time processing and with the general limitations on such processing. The chapter "Introduction to Processes and Tasks" in this book describes these issues in detail. As emphasized in that chapter, you should in general avoid executing tasks at interrupt time. If you must install a VBL task, the code should be as short as possible. In addition, the code and any data it accesses should be locked into physical memory if virtual memory is in operation.

To use this chapter, you might also need to be familiar with techniques for accessing information in your application's A5 world at interrupt time. The chapter "Introduction to Memory Management" in *Inside Macintosh: Memory* describes the A5 world and the routines you can use to manipulate the A5 register. This chapter provides complete code samples that illustrate how to access your application's A5 world in a VBL task. As a result, you might be able to use the Vertical Retrace Manager to accomplish simple, repetitive tasks without reading that chapter.

This chapter describes how the Vertical Retrace Manager works and then shows how you can use the Vertical Retrace Manager to

- install a simple task to be executed during vertical retrace interrupts
- access information about a task record installed in the vertical retrace queue from within that task
- access your application's global variables in a vertical retrace task
- spin the cursor to indicate that the user must wait while the computer completes some lengthy processing
- install a vertical retrace task in the system heap so that it continues to be executed even when your application is switched out

About the Vertical Retrace Manager

The video circuitry in a Macintosh computer, whether built-in or external, refreshes the screen at regular intervals. For built-in monitors, the screen is refreshed approximately 60 times per second; for external monitors, the screen is refreshed at intervals determined by the associated video hardware. To refresh the screen, the monitor's electron beam draws one pixel at a time, starting at the upper-left corner of the screen and moving quickly to the lower-right corner. When the electron beam returns from the lower-right corner of the screen to the upper-left corner, the video circuitry generates a **vertical retrace interrupt** or **vertical blanking (VBL) interrupt.**

The Vertical Retrace Manager is the part of the Operating System that schedules and executes tasks—known as **VBL tasks**—during a vertical retrace interrupt. The Operating System itself uses the Vertical Retrace Manager to perform some important housekeeping operations, such as moving the cursor in response to mouse movements and checking whether the current application's stack has expanded into its heap. Within the limitations described in this chapter, you can use the Vertical Retrace Manager to install your own recurrent tasks. For example, you can use the Vertical Retrace Manager to spin the cursor to indicate that the user must wait while some processing initiated by your application completes.

In general, the Vertical Retrace Manager is useful for small, repetitive tasks that do not allocate or release memory and that you do not want to execute in your main event loop. Whenever possible, it is best to manage periodic tasks within your main event loop. For example, you can call the TextEdit routine `TEIdle` once each time through the loop, thus causing the insertion point in a block of text to blink. However, if you want some task to execute repetitively at a time when you do not want to reenter your main event loop (perhaps because you don't want your application to be switched out during some lengthy operation), it might be possible to use the Vertical Retrace Manager to execute the task.

The principal limitation on VBL tasks (aside from the limitations on any interrupt-time processing) is that they cannot execute more frequently than once per VBL interrupt. The exact amount of time between successive VBL interrupts depends on the refresh frequency of the screen, which varies. On Macintosh computers that have a built-in screen (such as the Macintosh Plus or Macintosh Classic), the vertical retrace frequency is approximately 60.15 Hz, resulting in a period of approximately 16.63 milliseconds. If you need a task to be executed more often than that, you should use the Time Manager, which has a much finer resolution (up to 250 microseconds for drift-free task execution).

Unlike the Time Manager, the Vertical Retrace Manager is not an absolute timing mechanism. Its operations are always relative to the VBL interrupt, which may be disabled (for instance, during disk access). As a result, you should use the Time Manager in cases where absolute time delays are important. Use the Vertical Retrace Manager, however, in cases where the scheduled actions need simply to be synchronized with other VBL tasks, such as moving the cursor or refreshing the screen.

VBL Tasks Installed by the Operating System

The Operating System uses the Vertical Retrace Manager to accomplish a number of repetitive tasks at uniform intervals. These are some of the VBL tasks installed by the Operating System, grouped by the intervals at which they execute:

- Every interrupt
  - Update the value of the global variable `Ticks`, which a program may access through the routine `TickCount`.
  - Call the "stack sniffer" to see if the current application's stack and heap have collided. If so, the task calls the System Error Handler.
  - Update the position of the cursor.
- Every 30 interrupts
  - Check whether the user has inserted a disk or mounted a volume. If so, the task posts a disk-inserted event.
- Every 32 interrupts
  - Check whether a keyboard has been reattached after having been detached. If so, the task resets the keyboard.

Some VBL routines may execute only on certain computers or only in certain versions of system software. For example, on early Macintosh computers, a VBL task checks every other interrupt to determine whether the state of the mouse button has changed from its previous state and then remained unchanged for at least four interrupts. If so, that task posts a mouse-down or mouse-up event, as appropriate. In Macintosh computers equipped with Apple Desktop Bus mouse devices, the Operating System uses a different mechanism for posting mouse-down and mouse-up events.

Note
VBL tasks installed by the Operating System are not maintained in the same queue used for application-defined VBL tasks. ◆

Types of VBL Tasks

There are two general types of VBL tasks. A **slot-based VBL task** is linked to an external video monitor. Because different monitors can have different refresh rates and hence might execute VBL tasks at different times, the Vertical Retrace Manager maintains a separate task queue for each video device attached to the computer. When a VBL interrupt occurs for a particular device, the Vertical Retrace Manager executes any tasks in the queue for the slot holding that monitor's video card. You can install a slot-based VBL task by calling the `SlotVInstall` function.

For Macintosh computers that have only a built-in monitor (such as a Macintosh Plus or Macintosh Classic), there is no need to isolate VBL tasks into separate queues. Instead, the Operating System maintains just one task queue and processes the tasks in that queue when it receives a VBL interrupt. A VBL task that is not linked to an external video device is known as a **system-based VBL task.** You can install a system-based VBL task by calling the `VInstall` function.

To maintain compatibility on modular Macintosh computers for software that uses the `VInstall` function, the Operating System generates a special interrupt at a frequency identical to the retrace rate on compact Macintosh computers. This special interrupt is generated approximately 60.15 times a second and mimics the vertical retrace interrupt on compact models. This ensures that application tasks installed using the `VInstall` function, as well as periodic system tasks such as updating the tick count and checking whether the stack has expanded into the heap, are performed as usual.

To ensure the synchronization of your VBL task with the retracing of the screen, you should check whether the `SlotVInstall` function is available in the current operating environment. If it is, you should use the slot-based routines to install and remove your VBL task. If not, you should use the system-based routines.

However, even if you synchronize your VBL task to the retracing of the screen correctly, tasks may not always execute as scheduled. Some types of system activity, such as disk access, may cause VBL interrupts to be disabled temporarily. (This is why cursor movement sometimes becomes jerky during disk operations.) Also, if a VBL task takes longer to perform than the time it takes to retrace the screen, other interrupt tasks may miss one or more vertical retrace interrupts.

Like all interrupt tasks, VBL tasks cannot do everything that ordinary routines can. The following list summarizes the operations that VBL tasks should not perform. A VBL task that violates one of these rules may cause a system crash:

- A VBL task must not allocate, move, or purge memory, or call any Toolbox routines that might do so.
- A VBL task must preserve all registers other than A0–A3 and D0–D3.
- A VBL task cannot call a routine from another code segment unless it sets up the application's A5 world properly. In addition, that segment must already be loaded in memory.
- A VBL task cannot access your application global variables unless it sets up the application's A5 world properly. This technique is explained in "Accessing Application Global Variables in a VBL Task," beginning on page 4-13.
- A VBL task's code and any data accessed during the execution of the task must be locked into physical memory if virtual memory is in operation.

The VBL Task Record

You install a VBL task by passing the Vertical Retrace Manager the address of a **VBL task record,** which holds information about your VBL task. This information includes the address of the procedure the Vertical Retrace Manager is to execute at interrupt time and the number of interrupts before it should next execute the task. The `VBLTask` data type defines a VBL task record.

```
TYPE VBLTask =
RECORD
   qLink:    QElemPtr;   {next entry in vertical retrace queue}
   qType:    Integer;    {queue type}
   vblAddr:  ProcPtr;    {pointer to task procedure}
   vblCount: Integer;    {interrupts until next execution}
   vblPhase: Integer;    {task phase}
END;
```

Your application needs to fill in only the `qType`, `vblAddr`, `vblCount`, and `vblPhase` fields of the VBL task record. The `qLink` field, which contains a pointer to the next entry in the VBL task's vertical retrace queue, is set by the Vertical Retrace Manager when you install the task by calling `VInstall` or `SlotVInstall`. Your application does not need to set up the `qLink` field.

The Vertical Retrace Manager installs your VBL task record into the appropriate VBL queue. A vertical retrace queue is a standard operating-system queue.

Note
For more information about the structure of operating-system queues, see the chapter "Queue Utilities" in *Inside Macintosh: Operating System Utilities.* ◆

You must set the `qType` field to `ORD(vType)` before you install the task. This specifies that the task's queue is a vertical retrace queue and not some other type of operating-system queue.

The `vblAddr` field holds a pointer to the procedure that the Vertical Retrace Manager is to execute.

When installing a VBL task, you specify, in the `vblCount` field, the number of interrupts before the routine first executes. The Vertical Retrace Manager lowers this number by 1 during each interrupt. If decrementing `vblCount` produces a value of 0, the Vertical Retrace Manager executes the procedure specified in the task record's `vblAddr` field. If you want the procedure to be executed again, that procedure is responsible for resetting the value of the `vblCount` field to the desired value.

If you do not want the Vertical Retrace Manager to execute the task again, your task should leave the value of `vblCount` at 0. Setting the `vblCount` field to 0 is one way of disabling a task. (A more common approach is to remove the task record from its queue by calling `VRemove` or `SlotVRemove`, but this should not be done by the VBL task itself.) Note that if you set `vblCount` to 0 when installing a VBL task, the task will never execute. If you want a task to execute immediately upon installation, set `vblCount` to 1.

The `vblPhase` field specifies the task's phase count, indicating which interrupts are to trigger the execution of the VBL task. You can set two VBL tasks installed at the same time and scheduled for execution after the same number of interrupts out of phase with one another—that is, executed during different interrupts—by specifying different phase counts for each task. Unless you add many tasks to a VBL queue at one time, you can usually set `vblPhase` to 0.

Vertical Retrace Queues

The Vertical Retrace Manager stores application-defined VBL task records in **vertical retrace queues**, which are standard operating-system queues. If multiple tasks in the same vertical retrace queue are scheduled to be executed during the same interrupt, the Vertical Retrace Manager will execute the tasks in the order they were installed in the queue.

Compact Macintosh computers maintain only one vertical retrace queue, because these computers have only one screen. However, computers with multiple screens require multiple vertical retrace queues. Because slot-based task installation and removal routines apply to just one slot, the Vertical Retrace Manager maintains a separate vertical retrace queue for each slot that contains a video card. In addition, to maintain compatibility with the system-based VBL task installation and removal routines, the Vertical Retrace Manager maintains a single, system-based vertical retrace queue for all applications to share.

Ordinarily, you do not need to inspect or manipulate the contents of vertical retrace queues directly. Instead, you can use the Vertical Retrace Manager routines for installing task records in and removing them from vertical retrace queues.

In one case, however, you might need to inspect the header of a vertical retrace queue. If you need to know whether some code is being called in response to a VBL interrupt, you can inspect the `qFlags` field of the queue header. The Vertical Retrace Manager sets bit 6 of the `qFlags` field in the queue header to indicate that a VBL task in the queue is being executed.

Assembly-Language Note

You can use the global constant `inVBL` to test this bit. ◆

VBL Tasks and Application Execution

Often, a VBL task performs services that are useful only to the application that installed it. For instance, consider the VBL task defined in “Spinning the Cursor” beginning on page 4-16. This task spins the cursor while your application performs some lengthy operation and should be executed only if your application is in the foreground. If the user switches your application into the background while it is occupied with that lengthy operation, you probably want to disable that task for as long as your application is in the background. Otherwise, the cursor will continue to spin, probably confusing the user.

In other cases, a VBL task should continue to be executed even when the application that installed it is no longer in the foreground. For instance, you probably wouldn’t want to disable a VBL task that periodically checks for the arrival of electronic mail just because your application is moved to the background.

The Process Manager automatically disables a system-based VBL task when the application that installed it is swapped out in a major or minor switch, if the address of the VBL task is anywhere in the application’s partition. Then, when that application regains control of the processor, the Process Manager reenables that VBL task. If,

however, the address of a system-based VBL task is in the system partition, the VBL task continues to be executed, regardless of the processing status of the application that launched it.

Note
When your system-based VBL task continues to be executed in this way, the Process Manager does not restore the context of your application before executing the VBL task. In particular, any trap patches installed by your application might not be available to the VBL task. When a VBL task depends on your application context, your task can call the Process Manager function `GetCurrentProcess` to check whether your application is the current process and hence that its context is valid. ◆

The address of a system-based VBL task, not the address of the VBL task record, determines whether the Process Manager disables the task. See "Installing a Persistent VBL Task," beginning on page 4-20, for a technique you can use to prevent the disabling of a task when the application that installed it is switched out.

By contrast, the Process Manager never disables a slot-based VBL task, no matter where the task is located. As a result, if you want to disable a slot-based VBL task when your application is in the background, you must do so yourself, either by removing the task record from the VBL queue or by setting the `vblCount` field of the task record to 0. You can do this in response to a suspend event. Then, when your application receives a resume event, you can reenable the VBL task by reinstalling the task record or by resetting the `vblCount` field of the task record to the appropriate value.

In some cases, you might want to disable a *system*-based VBL task manually, even though the Process Manager also disables it when your application is switched out. This is because the Process Manager reenables system-based VBL tasks when your application receives processing time as a result of a minor switch, when your application is still in the background. If the VBL task should be executed only when your application is in the foreground, you need to disable it when your application receives a suspend event and reenable it when your application receives a resume event. The easiest way to do this is to set and reset the `vblCount` field of the task record, as described in the previous paragraph.

The Process Manager treats VBL tasks slightly differently when your application quits or crashes than when it is switched out. If either the task record for a VBL task or the code of the VBL task is located in your application partition, the Process Manager removes that task record from its VBL queue. (This is true for both slot-based and system-based VBL tasks.) Conversely, if both a VBL task record and the task itself are located in the system partition, the Process Manager doesn't remove the task record from its VBL queue when the application that installed them quits or crashes.

▲ **WARNING**
Failure to remove VBL task records installed in the system partition from their queues can lead to a system crash if the VBL task is located in the system partition but accesses data in your application partition. Because the Process Manager deallocates your application partition when your application quits or crashes, the VBL task may attempt to

access undefined data. The easiest way to avoid this problem is to patch the Process Manager's `ExitToShell` procedure so that it removes all VBL task records installed by your application. ▲

Using the Vertical Retrace Manager

You can use the Vertical Retrace Manager to install VBL task records in and remove VBL task records from system-based or slot-based vertical retrace queues. To install a task record, you must first fill in some of its fields and then call either `VInstall` or `SlotVInstall`. See the next section, "Installing a VBL Task," for information on installing VBL tasks.

If it is to be executed more than once, a VBL task must access the task record and reset the value of the task record's `vblCount` field. The section "Accessing a Task Record at Interrupt Time" on page 4-12 describes this technique. To disable a task temporarily, you can simply set the `vblCount` field of its task record to 0. To remove a VBL task from its VBL queue, call `VRemove` if you installed the task by calling `VInstall` or call `SlotVRemove` if you installed the task by calling `SlotVInstall`.

If your VBL task needs to access your application global variables, you can put the application's A5 value or the global variables themselves into the second field of a record whose first field contains the VBL task itself. The sections "Accessing Application Global Variables in a VBL Task," beginning on page 4-13, and "Spinning the Cursor," beginning on page 4-16, explain these techniques.

Installing a VBL Task

For any particular VBL task, you need to decide whether to install it as a system-based VBL task or as a slot-based VBL task. You need to install a task as a slot-based VBL task only if the execution of the task needs to be synchronized with the retrace rate of a particular external monitor. If the task performs no processing that is likely to affect the appearance of the screen or that depends on the state of an external monitor, it is probably safe to install the task as a system-based VBL task.

▲ **WARNING**
If you do decide that the execution of some VBL task needs to be synchronized with the retrace rate of a monitor, you should first check that the `SlotVInstall` function is available in the operating environment. You can do this by calling the `TrapAvailable` function defined in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*. If you call `SlotVInstall` and it is not available, your application will crash. ▲

If you are uncertain whether to install a task as a system-based or as a slot-based VBL task, you should first install it as a system-based task (by calling `VInstall`). Then test your application on a modular Macintosh computer with an external monitor whose refresh rate is different from the refresh rate on a compact Macintosh computer

(approximately 60.15 Hz). If any screen updating that occurs as a result of processing done by your VBL task has an unacceptable appearance, you probably need to install the task as a slot-based VBL task (by calling `SlotVInstall`). Remember, however, to check whether `SlotVInstall` is available before you call it; if it isn't available, call `VInstall`. You can determine whether `SlotVInstall` is available by calling the `SlotRoutinesAvailable` function defined in Listing 4-1.

Listing 4-1 Checking whether you can use slot-based VBL routines

```
FUNCTION SlotRoutinesAvailable: Boolean;
CONST
   _SlotVInstall = $A06F;
BEGIN
   SlotRoutinesAvailable := TrapAvailable(_SlotVInstall);
END;
```

If the slot-based routines are available and you want to use them, you need to know the slot number of the video device to whose retrace the VBL task is to be synchronized. Listing 4-2 illustrates a way to find the slot number of the main graphics device. To access the device control entry for the main graphics device, you must first find the device's reference number. Then you can cast the device control entry into type `AuxDCEHandle` and access the slot number directly. You can use a similar technique to find the slot number of some other graphics device.

Listing 4-2 Determining the slot number of the main graphics device

```
FUNCTION MainSlotNumber: Integer;
VAR
   mainDeviceRefNum: Integer;    {number of main graphics device}
BEGIN
   mainDeviceRefNum := GetMainDevice^^.gdRefNum;
   MainSlotNumber :=
      AuxDCEHandle(GetDCtlEntry(mainDeviceRefNum))^^.dCtlSlot;
END;
```

Note
For the sake of simplicity, the remainder of this chapter illustrates how to use the Vertical Retrace Manager to handle system-based VBL tasks only. Virtually all of the techniques shown here, however, can be used in connection with slot-based VBL tasks as well. ◆

The `InstallVBL` function defined in Listing 4-3 shows how to fill in a VBL task record and install it in the system-based VBL queue. It assumes that the task record `gMyVBLTask` is a global variable of type `VBLTask` and that you have already defined the procedure `DoVBL`, the actual VBL task. That procedure is subject to all of the usual

limitations on VBL and other interrupt tasks. Also, if DoVBL is to be executed recurrently, it must reset the vblCount field of the task record each time it is executed. The next section, "Accessing a Task Record at Interrupt Time," describes how to do this.

Listing 4-3 Initializing and installing a task record

```
FUNCTION InstallVBL: OSErr;
CONST
   kInterval = 6;                {frequency in interrupts}
BEGIN
   WITH gMyVBLTask DO            {initialize the VBL task}
      BEGIN
         qType := ORD(vType);    {set queue type}
         vblAddr := @DoVBL;      {set address of VBL task}
         vblCount := kInterval;  {set task frequency}
         vblPhase := 0;          {no phase}
      END;

   InstallVBL := VInstall(@gMyVBLTask);
END;
```

Accessing a Task Record at Interrupt Time

A repetitive VBL task must access its task record so that it can reset the vblCount field. As explained in "The VBL Task Record" on page 4-6, the Vertical Retrace Manager decrements the vblCount field during each interrupt and executes the task when that field reaches 0. The task is removed from its queue if the value of the vblCount field is left at 0.

When the Vertical Retrace Manager executes the VBL task, it places the address of the VBL task record into the A0 register. Listing 4-4 defines an inline function that moves this value onto the stack.

Note
You should call the inline function in Listing 4-4 only from a VBL task. It will not work if called from your main program. In addition, the call to this function should be the first line of your VBL task, because other processing might change the value in A0. ◆

Listing 4-4 Finding the address of the task record from within a VBL task

```
FUNCTION GetVBLRec: LongInt;
      INLINE $2E88;              {MOVE.L A0,(SP)}
```

The `GetVBLRec` function defined in Listing 4-4 returns a long integer specifying the address of the task record. Now that you can access the task record, you can easily reset the value of the `vblCount` field. Listing 4-5 provides an example of a generic VBL task that accesses the task record and resets the `vblCount` field.

Listing 4-5 Resetting a VBL task so that it executes again

```
PROCEDURE DoVBL;
CONST
   kInterval = 6;                    {frequency in interrupts}
TYPE
   VBLTaskPtr = ^VBLTask;            {pointer to a VBLTask record}
VAR
   taskPtr: VBLTaskPtr;
BEGIN
   taskPtr := VBLTaskPtr(GetVBLRec);{get address of task record}

   {Put task-specific code here.}

   taskPtr^.vblCount := kInterval;  {reset vblCount}
END;
```

Accessing Application Global Variables in a VBL Task

The Operating System stores the address of the boundary between the current application's global variables and its application parameters in the microprocessor's A5 register. For this reason, most compilers generate references to application global variables as offsets from the address contained in the A5 register. Therefore, if the value in register A5 does not point to the boundary between your application's global variables and its application parameters, your attempts to access your application's global variables will fail.

Ordinarily, applications do not need to keep track of the value in the A5 register. Although all applications share the register, the Process Manager keeps track of the address of your application's A5 world when a major or minor switch causes your application to yield the CPU to other processes, and it restores that value when your application regains access to the CPU. The A5 register is guaranteed to be correct for all code that your application executes directly (that is, for all code that is not executed in response to an interrupt or by a Toolbox or Operating System routine).

Because VBL tasks are interrupt routines, they might be executed when the value in the A5 register does not point to the A5 world of your application. As a result, if you want to access your application's global variables in a VBL task, you need to set the A5 register to its correct value when your VBL task begins executing and restore the previous value upon exit.

Note
For a more complete discussion of the A5 register, see the chapter "Memory Management Utilities" in *Inside Macintosh: Memory.* ◆

The solution to this problem is to find a memory location that both the main program and the VBL task can access. The main program can store the value of register A5 there, and the VBL task can set A5 correctly by reading that value. The functions `SetCurrentA5` and `SetA5` can be used for this purpose. The application can store the value of its A5 register by calling `SetCurrentA5`. Then, at interrupt time, the task can begin by calling `SetA5` to set the register to that value and end by calling `SetA5` again, this time to restore the register to its initial value, the one used by the main program.

The only memory location that a VBL task has access to is the address of the task record, as explained in the previous section, "Accessing a Task Record at Interrupt Time." So, if your application stores the value of A5 directly following the task record in memory, it can locate the value of A5 by first locating the task record. You can do this by defining a new data type (called `VBLRec` in Listing 4-6) whose first field contains the VBL task and whose second field contains a long integer specifying the value of the A5 register.

Listing 4-6 Storing the value of the A5 register directly after the task record in memory

```
TYPE VBLRec =
   RECORD
      myVBLTask:     VBLTask;    {the actual VBL task record}
      vblA5:         LongInt;    {saved value of application's A5}
   END;

   VBLRecPtr = ^VBLRec;
```

Now you can modify the application-defined procedure that installs a VBL task so that it stores the value of register A5 in the `vblA5` field of the `VBLRec`, as illustrated in Listing 4-7.

Listing 4-7 Saving the value of the A5 register when installing a VBL task

```
FUNCTION InstallVBL: OSErr;
CONST
   kInterval = 6;                  {frequency in interrupts}
BEGIN
   WITH gMyVBLRec.myVBLTask DO     {initialize the VBL task}
      BEGIN
         qType := ORD(vType);      {set queue type}
         vblAddr := @DoVBL;        {set address of VBL task}
         vblCount := kInterval;    {set task frequency}
         vblPhase := 0;            {no phase}
```

```
      END;
   myVBLRec.vblA5 := SetCurrentA5;  {get our A5}

   InstallVBL := VInstall(@gMyVBLRec.myVBLTask);
END;
```

You must also modify the VBL task so that it sets and restores the value of register A5 correctly. Listing 4-8 illustrates a simple VBL task that increments the global variable `gCounter` and then resets itself to run again after the specified number of interrupts.

Listing 4-8 Setting up the A5 register and modifying a global variable in a VBL task

```
PROCEDURE DoVBL;
CONST
   kInterval = 6;                   {frequency in interrupts}
VAR
   curA5:   LongInt;                {stored value of A5}
   recPtr:  VBLRecPtr;              {pointer to task record}
BEGIN
   recPtr := VBLRecPtr(GetVBLRec);  {get address of task record}
   curA5 := SetA5(recPtr^.vblA5);   {set our application's A5 }
                                    { and store old A5 in curA5}
   gCounter := gCounter + 1;        {modify a global variable}

   {Reset vblCount so that this procedure executes again.}
   recPtr^.myVBLTask.vblCount := kInterval;

   curA5 := SetA5(curA5);           {restore the old A5 value}
END;
```

Because of the optimizations performed by some compilers, the actual work of the VBL task and the setting and restoring of the A5 register might have to be placed in separate procedures. If necessary, you can define a routine `DoVBL` that loads the proper value of A5, calls another routine called `RunVBL`, and then restores the old value of A5. The `RunVBL` routine does the work of the VBL task and resets the task record's `vblCount` field so that the `DoVBL` routine executes again. Listing 4-9 illustrates a sample definition of the `RunVBL` function that modifies an application global variable.

Listing 4-9 Modifying application global variables in a VBL task

```
PROCEDURE RunVBL (aRecPtr: VBLRecPtr);
CONST
   kInterval = 6;                   {frequency in interrupts}
BEGIN
```

```
   gCounter := gCounter + 1;        {modify global variable}

   {Reset vblCount so that this procedure executes again.}
   aRecPtr^.myVBLTask.vblCount := kInterval;
END;
```

Listing 4-10 shows how to call RunVBL from the VBL task.

Listing 4-10 Setting up and restoring the A5 register in a VBL task

```
PROCEDURE DoVBL;
VAR
   curA5:   LongInt;                {stored value of A5}
   recPtr:  VBLRecPtr;              {pointer to task record}
BEGIN
   recPtr := VBLRecPtr(GetVBLRec);  {get address of task record}
   curA5 := SetA5(recPtr^.vblA5);   {set our application's A5 }
                                    { and store old A5 in curA5}
   RunVBL(recPtr);                  {run the actual VBL task}
   curA5 := SetA5(curA5);           {restore the old A5 value}
END;
```

If this separation of routines is necessary, you must make sure that the two routines (DoVBL and RunVBL) are in the same code segment.

Spinning the Cursor

Some VBL tasks need access only to global variables that they do not share with the main program. For example, you might wish to design a VBL task that animates the beachball or watch cursor to indicate that the user must wait while the computer finishes some lengthy processing. The main application might use the application-defined procedures StartSpinning and StopSpinning to install and remove the VBL task, but the application might not need to know, for example, which beachball or watch cursor the VBL task is displaying at any given time. The VBL task itself would need to know this information, because it must know which cursor to display when it is time to change the cursor.

One way to implement such a VBL task is to use application global variables and set up the A5 register properly, as described in the previous section, "Accessing Application Global Variables in a VBL Task." An alternate method, however, is simply to store the information that the VBL task needs directly after the task record in memory, just as you can store information about the program's A5 value there. Then, because the VBL task has access to all of the information it needs, it does not need to set up and restore the A5 register.

The listings that follow use that strategy to implement cursor spinning. This cursor spinning task implements simple animation of any number of cursor frames stored in contiguous resources in the program's resource fork.

Listing 4-11 provides a type definition for a cursor information record. This record holds the task record and information specific to cursor spinning. Listing 4-11 also defines several constants and a global variable to hold a cursor information record.

Listing 4-11 Defining a cursor information record

```
CONST
   kInterval = 4;                    {frequency in interrupts}
   kNumberOfCursors = 4;             {total number of frames}
   kInitialResID = 128;              {ID of first cursor resource}
TYPE
   CursorsList = ARRAY[1..kNumberOfCursors] OF CursHandle;
   CursorTask =
      RECORD
         myVBLTask:  VBLTask;        {the actual VBLTask}
         myCursors:  CursorsList;    {handles to the cursors}
         myFrame:    Integer;        {cursor frame to display next}
      END;
   CursorTaskPtr = ^CursorTask;
VAR
   gMyCursTask:   CursorTask;        {global cursor info. record}
```

Listing 4-12 shows the VBL task itself. The task changes the cursor and resets the task record's `vblCount` field so that the Vertical Retrace Manager executes the task again.

Listing 4-12 Changing the cursor within a VBL task

```
PROCEDURE ChangeCursor;
TYPE
   BooleanPtr = ^Boolean;                    {to check a low-memory global}
VAR
   recPtr:  CursorTaskPtr;
BEGIN
   recPtr := CursorTaskPtr(GetVBLRec);       {get cursor information}
   {If the cursor is busy, we should not change it.}
   IF NOT BooleanPtr(CrsrBusy)^ THEN
      WITH recPtr^ DO                        {update cursor information}
      BEGIN
         SetCursor(myCursors[myFrame]^^);    {display the next cursor}
         myFrame := myFrame + 1;             {advance to next cursor frame}
```

```
        IF myFrame > kNumberOfCursors THEN
           myFrame := 1;                    {wrap around to first frame}
      END;
   recPtr^.myVBLTask.vblCount := kInterval;  {set task to run again}
END;
```

The `ChangeCursor` procedure retrieves the address of the VBL task record. If the cursor isn't already being changed, then `ChangeCursor` changes the cursor to the next one in sequence and resets the index of the next cursor to display. Finally, `ChangeCursor` sets itself to run again after the appropriate number of interrupts have occurred.

Note
It is permissible to call `SetCursor` at interrupt time, provided that the cursor handle is locked and that some other routine is not currently modifying the cursor. The system global variable `CrsrBusy` has the value `TRUE` if the cursor is busy; in that case, you should not call `SetCursor`. Listing 4-12 illustrates the proper way to change the cursor at interrupt time. ◆

Listing 4-13 defines the procedure `StartSpinning`, which you can call before beginning some lengthy operation. Because VBL tasks cannot depend on the validity of unlocked handles, the `StartSpinning` procedure must lock the cursor handles in memory before `SetCursor` is called in the `ChangeCursor` procedure.

Listing 4-13 Installing the cursor-spinning task into a vertical retrace queue

```
PROCEDURE StartSpinning;
CONST
   kInitialDelay = 120;          {initial delay before starting to spin}
VAR
   myErr:   OSErr;
   count:   Integer;
BEGIN
   {Initialize cursor information.}
   FOR count := 1 TO kNumberOfCursors DO
   BEGIN
      {Load cursor into memory.}
      gMyCursTask.myCursors[count] := GetCursor(kInitialResID + count - 1);
      {Lock cursor so that we can call SetCursor at interrupt time.}
      HLockHi(Handle(gMyCursTask.myCursors[count]));
   END;
   gMyCursTask.myFrame := 1;     {display cursor with kInitialResID first}

   WITH gMyCursTask.myVBLTask DO {initialize the VBL task record}
   BEGIN
```

```
      qType := ORD(vType);       {set queue type}
      vblAddr := @ChangeCursor;  {get address of VBL task}
      vblCount := kInitialDelay; {set task frequency}
      vblPhase := 0;             {no phase}
   END;

   myErr := VInstall(@gMyCursTask.myVBLTask);
END;
```

Notice that the initial delay (specified by the `kInitialDelay` constant in the `vblCount` field) is much larger than the number of interrupts between subsequent cursor changes (specified by the `kInterval` constant). This prevents the cursor from starting to spin until a reasonable time (about 2 seconds) has elapsed.

Listing 4-14 shows how to remove the cursor-spinning task from the vertical retrace queue.

Listing 4-14 Removing the cursor-spinning task from its vertical retrace queue

```
PROCEDURE StopSpinning;
VAR
   myErr:   OSErr;
   count:   Integer;
BEGIN
   {Remove the task record from its queue.}
   myErr := VRemove(@gMyCursTask.myVBLTask);

   {Free memory occupied by the cursors.}
   FOR count := 1 TO kNumberOfCursors DO
      ReleaseResource(Handle(gMyCursTask.myCursors[count]));

   InitCursor;                          {restore the arrow cursor}
END;
```

Depending on the needs of your application, you might want to load the cursors into memory at application-launch time and release them when your application quits. If so, you need to modify the `StartSpinning` and `StopSpinning` procedures accordingly.

Installing a Persistent VBL Task

A **persistent VBL task** continues to be executed even when the Process Manager switches out the application that installed it and that application is no longer in control of the CPU. If you want to install a persistent system-based VBL task, you need to load its VBL task record into the system partition. (Slot-based VBL tasks are always persistent, no matter where you put the task record.) Listing 4-15 illustrates a simple way to load a VBL task record into the system heap.

Listing 4-15 Installing a persistent VBL task

```
FUNCTION InstallPersistentVBL (VAR theVBLRec: VBLTask): OSErr;
TYPE
   ProcPtrPtr = ^ProcPtr;                        {a pointer to a ProcPtr}
CONST
   kJMPInstr = $4EF9;                            {this is an absolute JMP}
   kJMPSize = 6;                                 {size of an absolute JMP}
VAR
   myErr:      OSErr;
   SysHeapPtr: Ptr;
   tempPtr:    Ptr;
BEGIN
   SysHeapPtr := NewPtrSys(kJMPSize);            {get a block in system heap}
   myErr := MemError;
   IF myErr <> noErr THEN                        {make sure we have the block}
      BEGIN
         InstallPersistentVBL := myErr;
         Exit(InstallPersistentVBL);
      END;
   IntegerPtr(SysHeapPtr)^ := kJMPInstr;         {move in the JMP instruction}
   tempPtr := Ptr(ORD(SysHeapPtr)+SizeOf(Integer));
   ProcPtrPtr(tempPtr)^ := theVBLRec.vblAddr;    {move in the JMP address}
   theVBLRec.vblAddr := ProcPtr(SysHeapPtr);     {point record at sys heap}
   InstallPersistentVBL := VInstall(@theVBLRec);{install the VBL task record}
END;
```

The `InstallPersistentVBL` function defined in Listing 4-15 allocates enough bytes in the system heap to hold an integer that encodes an assembly-language `JMP` instruction together with the absolute address to which to jump. It loads into that space the assembly-language instruction and the address of the original VBL task, which is extracted from the VBL task record passed to it as a parameter. Then `InstallPersistentVBL` replaces the address of the original VBL task in that record with the address of the block in the system heap. The net result is that the `vblAddr` field of the VBL task record now contains an address in the system partition, making the VBL task persistent.

Vertical Retrace Manager Reference

This section describes the data structure and routines provided by the Vertical Retrace Manager. The section "Data Structure" shows the Pascal data structure for the VBL task record. The section "Vertical Retrace Manager Routines" describes the routines you can use to install and remove slot-based and system-based VBL tasks; it also describes several utility routines for advanced programmers. The section "Application-Defined Routine" describes VBL tasks.

Data Structure

This section describes the VBL task record, the data structure you use to install VBL tasks in and remove them from vertical retrace queues.

The VBL Task Record

A VBL task record describes a vertical retrace task. It indicates which task record (if any) comes next in the vertical retrace queue, what procedure to use for the task, how many interrupts to wait before the task is executed, and in what phase to execute the task. The `VBLTask` data type defines a VBL task record.

```
TYPE VBLTask =
RECORD
   qLink:      QElemPtr;   {next entry in vertical retrace queue}
   qType:      Integer;    {queue type}
   vblAddr:    ProcPtr;    {pointer to task procedure}
   vblCount:   Integer;    {interrupts until next execution}
   vblPhase:   Integer;    {task phase}
END;
```

Field descriptions

`qLink` A pointer to the next entry in the task's vertical retrace queue.

`qType` The queue type. This field must be set to `ORD(vType)`.

`vblAddr` A pointer to the VBL task.

`vblCount` The number of interrupts between successive calls to the VBL task specified in the `vblAddr` field. If the value of `vblCount` is 0, the task will never be executed. The Vertical Retrace Manager decrements the value of this field after each interrupt. If decrementing `vblCount` produces a value of 0, the Vertical Retrace Manager executes the task. The task must then reset `vblCount`, or its entry will be removed from the queue after it has been executed.

`vblPhase` The phase count of the VBL task. In most cases, you can set this field to 0. However, if you install multiple tasks with the same `vblCount` at the same time, you can assign them different `vblPhase` values so that the tasks are not executed during the same interrupt. The value of the `vblPhase` field must be less than the value of the `vblCount` field.

For more information about using the `vblCount` and `vblPhase` fields, see "The VBL Task Record" on page 4-6.

Vertical Retrace Manager Routines

This section describes routines that allow you to install slot-based and system-based task records in vertical retrace queues and to remove task records. This section also describes utility routines that are of interest only to advanced programmers.

Slot-Based Installation and Removal Routines

You can use the functions `SlotVInstall` and `SlotVRemove` to install task records in and remove them from slot-based vertical retrace queues.

SlotVInstall

You can use the `SlotVInstall` function to install a task record in a slot-based vertical retrace queue.

```
FUNCTION SlotVInstall (vblTaskPtr: QElemPtr; theSlot: Integer):
                          OSErr;
```

`vblTaskPtr` A pointer to the task record to add to a queue.

`theSlot` The slot number of the video device to whose vertical retrace queue the task record is added.

DESCRIPTION

The `SlotVInstall` function installs the task record specified by the `vblTaskPtr` parameter in the vertical retrace queue associated with the video device specified by the `theSlot` parameter. The Vertical Retrace Manager executes the task at intervals determined by the task record's `vblCount` and `vblPhase` fields. The task must reset the value of the task record's `vblCount` field if you want the task to be executed again.

The Vertical Retrace Manager continues to execute tasks installed using the `SlotVInstall` function even when the application that installed them is switched out.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SlotVInstall` are

Registers on entry

A0 Pointer to the task record

D0 Slot number of the device associated with the vertical retrace queue

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`vTypErr`	–2	Invalid `qType` value (must be `ORD(vType)`)
`slotNumErr`	–360	Invalid slot number

SlotVRemove

You can use the `SlotVRemove` function to remove a task record from a slot-based vertical retrace queue.

```
FUNCTION SlotVRemove (vblTaskPtr: QElemPtr; theSlot: Integer):
                      OSErr;
```

`vblTaskPtr`
A pointer to the task record to remove from its queue.

`theSlot` The slot number of the video device from whose vertical retrace queue the task record is removed.

DESCRIPTION

The `SlotVRemove` function removes the task record specified by the `vblTaskPtr` parameter from the vertical retrace queue associated with the video device specified by the `theSlot` parameter.

To disable a slot-based VBL task temporarily, you can set the `vblCount` field of the task record to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SlotVRemove` are

Registers on entry

A0 Pointer to the task record

D0 Slot number of the device associated with the vertical retrace queue

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`qErr`	–1	Task record isn't in the queue
`vTypErr`	–2	Invalid `qType` value (must be `ORD(vType)`)
`slotNumErr`	–360	Invalid slot number

System-Based Installation and Removal Routines

You can use the functions `VInstall` and `VRemove` to install task records in and remove them from the system-based vertical retrace queue. These routines exist to provide compatibility with Macintosh computers that have built-in monitors. You can also use these routines when you don't need to synchronize the execution of your VBL task to any monitor.

VInstall

You can use the `VInstall` function to install a task record into the system-based vertical retrace queue.

```
FUNCTION VInstall (vblTaskPtr: QElemPtr): OSErr;
```

`vblTaskPtr`
A pointer to the task record to add to the queue.

DESCRIPTION

The `VInstall` function installs the VBL task record specified by the `vblTaskPtr` parameter in the system-based vertical retrace queue. The Vertical Retrace Manager executes the task at intervals determined by the task record's `vblCount` and `vblPhase` fields. The task must reset the value of the task record's `vblCount` field if you want the task to be executed again.

In current versions of system software, the Vertical Retrace Manager does not continue to execute tasks installed using the `VInstall` function when the application that installed

them is switched out, unless the address in the `vblAddr` field of the task record points in the system partition.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `VInstall` are

Registers on entry

A0 Pointer to the task record

Registers on exit

D0 Result code

RESULT CODES

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

VRemove

You can use the `VRemove` function to remove a task record from the system-based vertical retrace queue.

```
FUNCTION VRemove (vblTaskPtr: QElemPtr): OSErr;
```

`vblTaskPtr`
A pointer to the task record to remove from the queue.

DESCRIPTION

The `VRemove` function removes the task record specified by the `vblTaskPtr` parameter from the system-based vertical retrace queue.

To disable a system-based VBL task temporarily, you can set the `vblCount` field of the task record to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `VRemove` are

Registers on entry

A0 Pointer to the task record

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
qErr	–1	Task record isn't in the queue
vTypErr	–2	Invalid qType value (must be ORD(vType))

Utility Routines

The Vertical Retrace Manager provides several utility routines that allow you to change the slot number of the primary video monitor, execute all tasks in a slot-based vertical retrace queue, and access the head of the system-based vertical retrace queue.

Note
Most applications do not need to use the routines described in this section. ◆

AttachVBL

The AttachVBL function changes the slot number of the primary video monitor.

```
FUNCTION AttachVBL (theSlot: Integer): OSErr;
```

theSlot The new slot number for the primary video monitor.

DESCRIPTION

The AttachVBL function changes the slot number of the primary monitor to the number specified by the theSlot parameter. System software uses this routine to ensure correct cursor updating.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for AttachVBL are

Registers on entry

D0 Slot number

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
slotNumErr	–360	Invalid slot number

DoVBLTask

Slot interrupt handlers for video cards should call the `DoVBLTask` function to handle the execution of VBL tasks.

```
FUNCTION DoVBLTask (theSlot: Integer): OSErr;
```

`theSlot` Slot number corresponding to the vertical retrace queue whose tasks are to be executed.

DESCRIPTION

The `DoVBLTask` function decrements the `vblCount` field of each task in the vertical retrace queue corresponding to the `theSlot` parameter (except for tasks whose `vblCount` field already contains the value 0). The function executes a task if decrementing the `vblCount` field for a task record results in a value of 0.

If `theSlot` designates the slot of the primary video device, the position of the cursor is also updated.

Slot interrupt handlers for video cards need to call this function to execute any tasks in the queue for that slot. You can also call this function if you need to simulate vertical retrace interrupts.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `DoVBLTask` are

Registers on entry

D0 Slot number

Registers on exit

D0 Result code

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

RESULT CODES

`noErr`	0	No error
`slotNumErr`	–360	Invalid slot number

GetVBLQHdr

You can obtain the header of the system-based vertical retrace queue by calling the `GetVBLQHdr` function.

```
FUNCTION GetVBLQHdr: QHdrPtr;
```

DESCRIPTION

The `GetVBLQHdr` function returns a pointer to the header of the system-based vertical retrace queue. In general, you need to call this function only if you want to manipulate the contents of the system-based vertical retrace queue directly or if you want to read the information stored in the queue header.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `VBLQueue` contains the header of the system-based vertical retrace queue.

The global variable `ScrnVBLPtr` contains a pointer to the header of the vertical retrace queue associated with the slot for the primary monitor.

Application-Defined Routine

The Vertical Retrace Manager allows your software to install an application-defined routine that is executed during vertical retrace interrupts.

VBL Tasks

You pass the address of an application-defined VBL task in the `vblAddr` field of the VBL task record.

MyVBLTask

A VBL task has the following syntax:

```
PROCEDURE MyVBLTask;
```

DESCRIPTION

The `vblAddr` field of a VBL task record contains the address of a VBL task that is executed after the number of interrupts specified in the `vblCount` field of the task record. The task can be set to execute at any frequency (up to once per vertical retrace interrupt). If the task uses application global variables or calls routines in another code

segment, it must ensure that register A5 contains the address of the boundary between the application global variables and the application parameters. In addition, if your task calls routines in another code segment, that segment must already be loaded in memory.

Because of the optimizations performed by some compilers, the actual work of the VBL task and the setting and restoring of the A5 register might have to be placed in separate procedures. See Listing 4-9 and Listing 4-10 for an example of how you can do this.

Your VBL tasks shouldn't call `VRemove` or `SlotVRemove` to remove its entry from the queue. Instead, either your application should call one of those functions at noninterrupt time or your task should simply not reset the `vblCount` of the task record.

SPECIAL CONSIDERATIONS

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

The code of the VBL task and any data accessed during its execution must be locked into physical memory if virtual memory is in operation.

Unless directed to do otherwise, some compilers insert code into your compiled application to facilitate debugging operations. This additional code can, however, cause trouble for VBL tasks and other interrupt processing. You might need to disable the generation of debugging code by enclosing the interrupt code between the appropriate compiler directives. Here's an example:

```
{$PUSH}
{$D-}
{Don't generate debugging code for this procedure.}
PROCEDURE DoVBL;
BEGIN
   ...
END;
{$POP}
```

Consult the documentation for your development system to see whether this is necessary and, if it is, how to do it.

ASSEMBLY-LANGUAGE INFORMATION

When the VBL task is called, register A0 contains a pointer to the VBL task record associated with that procedure.

A VBL task must preserve all registers other than A0–A3 and D0–D3. It must exit with an `RTS` instruction.

SEE ALSO

See the section “Accessing Application Global Variables in a VBL Task” beginning on page 4-13 for instructions on how to access your application’s global variables in a VBL task.

Summary of the Vertical Retrace Manager

Pascal Summary

Data Type

```
TYPE VBLTask   =              {VBL queue element}
   RECORD
      qLink:      QElemPtr;   {next entry in vertical retrace queue}
      qType:      Integer;    {queue type}
      vblAddr:    ProcPtr;    {pointer to task procedure}
      vblCount:   Integer;    {interrupts until next execution}
      vblPhase:   Integer;    {task phase}
   END;
```

Vertical Retrace Manager Routines

Slot-Based Installation and Removal Routines

```
FUNCTION SlotVInstall       (vblTaskPtr: QElemPtr; theSlot: Integer): OSErr;
FUNCTION SlotVRemove        (vblTaskPtr: QElemPtr; theSlot: Integer): OSErr;
```

System-Based Installation and Removal Routines

```
FUNCTION VInstall           (vblTaskPtr: QElemPtr): OSErr;
FUNCTION VRemove            (vblTaskPtr: QElemPtr): OSErr;
```

Utility Routines

```
FUNCTION AttachVBL          (theSlot: Integer): OSErr;
FUNCTION DoVBLTask          (theSlot: Integer): OSErr;
FUNCTION GetVBLQHdr         : QHdrPtr;
```

Application-Defined Routine

```
PROCEDURE MyVBLTask;
```

C Summary

Data Types

```
typedef pascal void (*VBLProcPtr)(void);

typedef struct {              /*VBL queue element*/
      QElemPtr    qLink;      /*next entry in vertical retrace queue*/
      short       qType;      /*queue type*/
      VBLProcPtr  vblAddr;    /*pointer to task procedure*/
      short       vblCount;   /*interrupts until next execution*/
      short       vblPhase;   /*task phase*/
} VBLTask;
```

Vertical Retrace Manager Routines

Slot-Based Installation and Removal Routines

```
pascal OSErr SlotVInstall   (QElemPtr vblTaskPtr, short theSlot);
pascal OSErr SlotVRemove    (QElemPtr vblTaskPtr, short theSlot);
```

System-Based Installation and Removal Routines

```
pascal OSErr VInstall       (QElemPtr vblTaskPtr);
pascal OSErr VRemove        (QElemPtr vblTaskPtr);
```

Utility Routines

```
pascal OSErr AttachVBL      (short theSlot);
pascal OSErr DoVBLTask      (short theSlot);
#define GetVBLQHdr()        ((QHdrPtr) 0x0160)
```

Application-Defined Routine

```
pascal void MyVBLTask       (void);
```

Assembly-Language Summary

Constants

```
vType           EQU     1       ;VBL queue element type
inVBL           EQU     6       ;bit index for VBL active flag
```

Data Structures

VBL Queue Element

0	`vblink`	long	next entry in vertical retrace queue
4	`vblType`	word	queue type
6	`vblAddr`	long	address of task procedure
10	`vblCount`	word	interrupts until next execution
12	`vblPhase`	word	phase count

Global Variables

`CrsrBusy`	byte	Set to `TRUE` if the cursor is being changed.
`jDoVBLTask`	long	Jump vector for `DoVBLTask` routine.
`ScrnVBLPtr`	long	Pointer to the primary monitor's vertical retrace queue's header.
`VBLQueue`	10 bytes	Header of the vertical retrace queue.

Result Codes

`noErr`	0	No error
`qErr`	–1	Task entry isn't in the queue
`vTypErr`	–2	Invalid `qType` value (must be `ORD(vType)`)
`slotNumErr`	–360	Invalid slot number