TOOLBOX-BASED ROUTINES FOR MACINTOSH TIMING AND DISPLAY

R.A. Rensink

Technical Report 89-11
June 1989

TOOLBOX-BASED ROUTINES FOR MACINTOSH TIMING AND DISPLAY [1]

R.A. Rensink [2]

Technical Report 89-11
June 1989

Department of Computer Science
The University of British Columbia
Vancouver, BC, Canada V6T 1W5

Abstract

Pascal routines are described for performing and testing various timing and display operations on Macintosh computers. Millisecond timing of internal operations is described, as is a method to time inputs more accurately than tick timing. Techniques are also presented for placing arbitrary bit-image displays on the screen within one screen refresh. All routines are based on Toolbox procedures applicable to the entire range of Macintosh computers.

---

[1] Submitted for publication

[2] This work has been supported in part by the UBC Center for Integrated Computer Systems Research. The author would like to thank James T. Enns of UBC Psychology for his encouragement and help, without which this paper would not have been written. Eric Ochs provided useful feedback on earlier drafts of this paper, and Jack Moxness supplied code that helped clarify the functioning of slots. Many thanks also to Bob Woodham of UBC Computer Science for his support.

Toolbox-Based Routines for Macintosh Timing and Display

Macintosh computers allow a wide variety of fast and flexible graphics operations to be carried out at relatively low cost. In addition, there exist easy-to-use graphics editors such as FullPaint and PixelPaint that allow complex images to be produced in very little time. Consequently, the Macintosh has been used extensively for applications involving sophisticated visual displays.

However, there is a problem. The Toolbox instructions for high-level control of the Mac do not allow many real-time operations to be easily programmed. Furthermore, much of the documentation on the relevant Toolbox procedures is obscure in key places, and in some cases, important points are completely missed. The programming of real-time operations such as millisecond timing and fast display has therefore relied on a combination of special assembly-language routines (e.g., Westall, Perkey, and Chute, 1986; Lane and Ashby, 1987) and special high-level languages such as Rascal (Reed College, 1985). These approaches work, but suffer from being special-purpose – they are generally designed for only one processor or machine architecture, and are not always portable to other machines. Furthermore, a casual user must spend considerable time and energy to learn to use them correctly.

This article takes a different approach to the problem, one more in spirit with the philosophy of the Toolbox. It describes how timing and display operations can be carried out and tested using available Toolbox routines. Pascal routines are given that show how the Time Manager can be used to carry out millisecond timing of internal operations, as well as timing of input events with an accuracy apparently limited only by the system hardware. Procedures are also presented for placing displays on the screen within one refresh cycle; for Mac IIs, these can be extended to allow an indefinite number of images to be sequentially displayed with each screen refresh. These techniques can be used as the basis of timing and display routines that will operate on all current models of the Macintosh computer, and presumably, all future models as well.

Not all timing and display applications can be foreseen in advance. Consequently, this article is somewhat tutorial in nature. The code for each routine is presented along with the description of its operation, so that the reader can develop a feeling of how to use Toolbox

routines for particular timing and display operations. To keep things simple, there is no error checking and only rudimentary input/output. All routines described in this paper have been extensively tested, and found to operate in a consistent fashion on all Macintosh models currently supported by Apple, viz., the Mac Plus, SE, Mac II, and Mac IIx. Many are applicable to the 128K and 512K models as well, and all have been found to operate correctly on the enhanced version of the 512K. To test these routines on particular machines, they need only be incorporated into the program `TD_Tests`, shown in Fig. 1. This program forms a minimal stand-alone "shell" that allows these routines to be run and tested on all machines and systems.

```
program TD_Tests;
  uses   { Units depend on development system -- these are for Lightspeed[TM] Pascal }
    Memtypes, Quickdraw, OSIntf, ToolIntf, PackIntf, VideoIntf;
  const
    MaxLong = 2147483647;
    TickFactor = 16.625822; { ticks to msec }
    NoSlot = 0; { value used when no slot active }
  type
    TMTaskPtr = ^TMTask;
    VBLTaskPtr = ^VBLTask;
  var
    CurrEvent :  Eventrecord;
    ErrCode :  integer; { code returned for i/o errors }

    DelayStat :  boolean; { global flag for delay }
    TimerTask, TriggerTask, DelayTask :  TMTask; { timing tasks }
    TimerPtr, TriggerPtr, DelayPtr :  TMTaskPtr; { ptrs to timing tasks }

    MainPort, ImagePort :  grafPtr; { graphics ports }
    MainBuffer, ImageBuffer, BackBuffer :  ptr; { ptrs to bit image buffers }
    MonitorX, MonitorY : integer; { dimensions of screen }
    CardWidth; { width of screen buffer (in bytes) }
    CurrMacType :  integer; { machine model }
    SmallScreen :  boolean; { video system }
    CurrSlot :  integer; { slot identifier (large-screen Macs only) }
    SlotTask :  SlotIntQElem; { slot-interrupt task (large-screen Macs only) }
    SlotPtr :  SQElemPtr; { slot-task ptr (large-screen Macs only) }
    VideoTask :  VBLTask; { screen-counter task (large-screen Macs only) }
    VideoPtr :  VBLTaskPtr; { screen-counter ptr (large-screen Macs only) }
    videoCount :  longint; { global counter for screen refreshes }

  { ...  setup and shutdown routines defined here ...  }
  { ...  timing and display routines defined here ...  }

begin { TD_Tests }

  { ...  general initialization procedures placed here ...  }

  setUpTimers; { set up Time Manager tasks }
  setUpBuffers; { set up buffers containing images }
  setUpVCounter; { set up counter for screen refreshes }

  triggerTest; { test trigger-based timing }
  anchorTest; { test anchor-based hybrid timing }
  delayTest; { test millisecond delay }
  inputTest; { test timing of keyboard input }
  screenTest; { test fast display of image onto screen }

  shutDownVCounter;
  shutDownBuffers;
  shutDownTimers;
end; { TD_Tests }
```

Figure 1: shell program for timing and display procedures

Millisecond Timing

With the introduction of the Time Manager in the Mac Plus, Toolbox procedures became available to access the millisecond timer on the VIA chip. Previously, this was possible only by special assembly-language programs; for the 128K and 512K Macs, this is still the only way to carry out such timing. These programs are described in great detail elsewhere (see, e.g., Westall, Perkey, and Chute, 1986), and will not not be discussed here. Rather, consideration will be limited to machines with a Time Manager on board, i.e., all Macs currently supported by Apple, as well as the enhanced version of the 512K.

The general characteristics of the Time Manager itself will not be described in great detail here, except where they bear directly on issues of setting up and using millisecond timing. For other information about the Time Manager, see pages IV-299 – IV-301 of Inside Macintosh (1986).

Basics

The operation of the Time Manager is based upon entries of type `TMTask` that are placed in a special Time Manager queue. Each of these entries has a `tmCount` field containing the time (in msec) until the associated task is executed. This number is regularly decremented until it reaches zero, at which point the Time Manager executes the procedure pointed to by the `tmAddr` field of the task. It has been noted by Kieley and Higgins (1988) that `tmCount` is incorrectly specified as being of type `integer`. It is actually a `longint`, and so can be used for durations as long as $2^{31} - 1 = 2147483647$ msec, or roughly 596 hrs.

Procedure `setUpTimers` in Fig. 2 shows how timing tasks can be set up. Here, `TimerTask` and `TriggerTask` are associated with an asynchronous null operation, while `DelayTask` invokes an asynchronous procedure that sets the global variable `DelayStat` to `false`. (Note that asynchronous tasks preserve the value of the A5 registers.) The pointers associated with these tasks are installed in the queue, and the `tmCount` fields are set by the `PrimeTime` procedure. Testing (based on the tests described below) has shown that installing non-zero values of `tmCount` does not always cause activation of Time Manager updating. However, if the task is re-installed after a delay of more than approximately 6 ticks, updating is carried out correctly; for caution, a delay of about 10 ticks is used in `TD_Tests`. The only known condition under which this will not work is when running System version 6.0.3, which, for unknown reasons, apparently does not correctly update the `tmCount` field. For version 6.0.2, as

well as several commonly-used earlier Systems, this method has yielded consistently correct operation over hundreds of trials.

A routine for shutting down the timing tasks (`shutDownTimers`) is also given in Fig. 2. This procedure must be called before exiting the program, or a system crash will occur.

Timing

Timing can be done by accessing the `tmCount` field of a TMTask. In `TD_Tests`, the global variable `TimerTask` is used exclusively for this purpose. Procedure `milliCount` in Fig. 3 shows how this value may be correctly accessed. Since the value of the `tmCount` field is always decremented by the Time Manager, subtraction of `tmCount` from `MaxLong` yields a positive value that constantly increases. Set up in this way, `milliCount` is analogous to `tickCount`.

The procedure `triggerTest` (Fig. 3) is designed to test the consistency of millisecond timing with that based on ticks. Timing begins at the point where `tickCount` reaches the first mark `tVal1`, and stops where it reaches the second mark `tVal1 + duration`. Running `triggerTest` shows that a straightforward access of `tmCount` does not result in precise timing. Apparently (at least on the systems tested) `tmCount` fields are only updated every 5 ticks. Simple access of `tmCount` thus yields lower precision than tick timing! To solve this problem, `milliCount` uses `PrimeTime` to install `TriggerTask` with a zero argument. This acts as a "trigger", forcing the Time Manager to update the `tmCount` fields of all tasks in the queue to the nearest millisecond.

Testing based on `triggerTest` shows this method of timing to be accurate to within one millisecond over intervals of about a second or so, depending on the particular machine. Beyond this range, errors of several milliseconds can accumulate. However, it is possible to avoid cumulative error via a hybrid timer that uses tick timing for coarse "large-scale" timing together with simple millisecond timing for finer "small-scale" measurement.

The procedure `anchorTest` in Fig. 4 provides an example of how this can be done. Coarse timing is done by taking the difference between "anchor points", which are `tickCount` values near the actual starting and stopping times. The difference between the actual times and the anchors is then measured by simple millisecond timing; the upper limit to these differences must be less than a second or so, but otherwise there are no special constraints. The subroutine `aTest` in Fig. 4 shows how the anchor points may be set. The first (`tStart`) is one more

```
{A+} { asynchronous } procedure DummyProc; { dummy procedure for timers }
begin
end;
{A-}

{A+} { asynchronous } procedure DelayProc; { procedure for delay }
begin
  DelayStat := false; { remove delay condition }
end;
{A-}

procedure setUpTimers;
  var
    tVal :  longint;
begin { setUpTimers }
  TimerPtr := @TimerTask;
  TriggerPtr := @TriggerTask;
  DelayPtr := @DelayTask;
  TimerTask.tmAddr := @DummyProc; { dummy task }
  TriggerTask.tmAddr := @DummyProc; { dummy task }
  DelayTask.tmAddr := @DelayProc; { change DelayStat when activated }

  InsTime(QElemPtr(TimerPtr)); { install timing tasks into queue }
  InsTime(QElemPtr(TriggerPtr));
  InsTime(QElemPtr(DelayPtr));
  PrimeTime(QElemPtr(TimerPtr), MaxLong); { set tmCount fields }
  PrimeTime(QElemPtr(TriggerPtr), 0);
  PrimeTime(QElemPtr(DelayPtr), 0);
  tval := tickCount + 10;
  repeat
  until tickCount > tVal; { wait and reset TimerTask }
  PrimeTime(QElemPtr(TimerPtr), MaxLong);
end; { setUpTimers }

procedure shutDownTimers;
begin { shutDownTimers };
  RmvTime(QElemPtr(TimerPtr));
  RmvTime(QElemPtr(TriggerPtr));
  RmvTime(QElemPtr(DelayPtr));
end { shutDownTimers };
```

Figure 2: procedures for setting up and shutting down timing tasks

```
function milliCount :  longint;
begin
  PrimeTime(QElemPtr(TriggerPtr), 0);
  milliCount := MaxLong - TimerTask.tmCount;
end;


procedure triggerTest;
  procedure tTest (triggerStat :  boolean; duration :  integer);
    var
      tVal1, tVal2, mStart, mStop, mDiff :  longint;
  begin { tTest }
    tVal1 := tickCount + 2;
    tVal2 := tVal1 + duration;
    repeat
    until tickCount > tVal1; { wait for first mark }
    if triggerStat then
      mStart := milliCount      { use trigger before access }
    else
      mStart := MaxLong - TimerTask.tmCount;    { use simple access }

    repeat
    until tickCount > tVal2; { wait for second mark }
    if triggerStat then
      mStop := milliCount      { use trigger before access }
    else
      mStop := MaxLong - TimerTask.tmCount;    { use simple access }

    mDiff := mStop - mStart;
    if triggerStat then
      writeln(duration,' ticks (using trigger) is ',mDiff,' msec')
    else
      writeln(duration,' ticks (without trigger) is ',mDiff,' msec');
  end; { tTest }

begin { triggerTest }
  tTest(false,1); { test 1-tick duration without trigger }
  tTest(true,1); { test 1-tick duration using trigger }
end; { triggerTest }
```

Figure 3: procedure for triggered millisecond timing

than the value of `tickCount` at the time the seconds counter (accessed by `getDateTime`) reaches the first mark `sVal1`; the difference `mStart2 - mStart1` consequently becomes the duration from the start of timing to the anchor point. A similar rationale is used to determine the point at which the seconds counter reaches the second mark `sVal2`. The difference between the two mark points is then calculated by converting the difference between the anchors to milliseconds and adding or subtracting the appropriate small-scale measurements.

Testing with `anchorTest` shows that hybrid timing of internal operations can be carried out with millisecond accuracy over arbitrarily long stretches of time, regardless of the load (internal or external) placed upon the system.

This approach can also be used to time inputs, as shown in procedure `inputTest` in Fig. 5. The event queue is first cleared. The procedure then loops until an event is detected by `OSEventAvail`; the loop is subsequently exited and the anchor point set to the value of `tickCount`. The millisecond timer is then triggered, and the difference between it and the time to the next tick is measured. Subtracting this interval from 17 msec (the time for a tick) yields the time from the anchor point.

Such timing is obviously at least as accurate as that based upon ticks. Furthermore, repeated testing of `inputTest` shows the millisecond difference between anchor and detected response to be fairly evenly distributed between 0 – 16 msec. This is consistent with a very fast and uniform response by `OSEventAvail` to input events; if the system response to inputs has a standard deviation of a millisecond or so, such a distribution would be expected. Lane and Ashby (1987) report that for models earlier than the SE, the time from key press to system interrupt takes a constant 6 msec. For the SE and Mac II, ADB control of keyboard input requires any keyboard event to be signalled within 260 μsec of its occurrence. Provided that the keyboard is the active input device (i.e., the device that has last sent input to the system), and that no service requests are pending from any other device, the keyboard will be continuously polled – without using processor time – until a key is pressed (see pages 11-9 – 11-16 and 11-25 of Macintosh Family Hardware Reference, 1986). The documentation is somewhat vague on this point, but the polling rate will be at least once every 11 msec, and may possibly be much higher than that. Therefore, although proof cannot be given using the routines described here, it is plausible that high accuracy timing of inputs can be achieved by this method. An exact measurement of this accuracy must await experiments based on external timers.

```
procedure anchorTest;
  procedure aTest (anchorStat :  boolean; duration :  integer);
    var
      tStart, tStop, sVal1, sVal2, sCount :  longint;
      mStart1, mStart2, mStop1, mStop2 :  longint;
      mStartDiff, mStopDiff :  longint;
  begin { aTest }
    if duration > 0 then begin
      getDateTime(sCount);
      sVal1 := sCount + 1;
      sVal2 := sVal1 + duration;
      repeat
        getDateTime(sCount);
      until sCount >= sVal1; { wait for first mark point}

      tStart := tickCount + 1; { set 'start' anchor = following tick }
      mStart1 := milliCount;
      repeat
      until tickCount >= tStart;
      mStart2 := milliCount;
      mStartDiff := mStart2 - mStart1;

      repeat
        getDateTime(sCount);
      until sCount >= sVal2; { wait for second mark point}
      tStop := tickCount + 1; { set 'stop' anchor = following tick }
      mStop1 := milliCount;
      repeat
      until tickCount >= tStop; { wait for anchor point }
      mStop2 := milliCount;
      mStopDiff := mStop2 - mStop1;

      if anchorStat then begin
        mDiff := round((tStop - tStart) * TickFactor) + mStartDiff - mStopDiff;
        writeln(duration,' seconds (using anchors) is ',mDiff,' msec');
      end { if }
      else begin
        mDiff := mStop1 - mStart1;
        writeln(duration,' seconds (without anchors) is ',mDiff,' msec');
      end; { else }
    end; { if }
  end; { aTest }

begin { anchorTest }
  aTest(false,10); { test 10-second duration without anchors }
  aTest(true,10);   { test 10-second duration using anchors }
end; { anchorTest }
```

Figure 4: procedure for anchor-based millisecond timing

```
procedure inputTest;
  var
    ix :  integer;

  procedure iTest;
    var
      tStop, mStop1, mStop2 :  longint;
  begin { iTest }
    flushEvents(EveryEvent, 0);
    repeat
    until OSEventAvail(KeyDownMask, CurrEvent); { wait for key press }

    tStop := tickCount; { set 'stop' anchor = current tick }
    mStop1 := milliCount;
    repeat
    until tickCount > tStop; { wait for nearest tick }
    mStop2 := milliCount;

    writeln('tickCount difference is ',tStop - CurrEvent.when,' ticks');
    writeln('difference from anchor is ',17 - (mStop2 - mStop1),' msec');
  end; { iTest }

begin { inputTest }
  flushEvents(EveryEvent, 0);
  repeat
  until OSEventAvail(KeyDownMask, CurrEvent); { make keyboard the active device}

  for ix := 1 to 10 do
    iTest;
end; { inputTest }
```

Figure 5: procedure for testing timing of inputs

```
procedure mDelay (del :  longint);
begin { mDelay }
  if del > 0 then begin
    PrimeTime(QElemPtr(DelayPtr), del);
    DelayStat := true;
    while DelayStat do { DelayStat set by DelayTask }
      ;
  end; { if }
end; { mDelay }

procedure delayTest;
  procedure dTest;
    var
      sVal1, sVal2, sCount :  longint;
  begin { dTest }
    getDateTime(sCount);
    sVal1 := sCount + 1;
    repeat
      getDateTime(sCount);
    until (sCount >= sVal1); { synchronize with seconds timer }
    mDelay(del);
    getDateTime(sVal2);

    writeln(sVal2 - sVal1,'-second change for ',del,' delay');
  end; { dTest }

begin { dTelaytTest }
  dTest(999); { test mDelay for 1-second delay }
  dTest(1000);
  dTest(1001);
end; { delayTest }
```

Figure 6: procedure for millisecond delay

Delays

One other useful real-time capability is the control of delays to millisecond precision. The routine `mDelay` in Fig. 6 shows how this can be done. Delay is carried out via `DelayTask`, which is devoted exclusively to this operation. This task calls the asynchronous procedure `DelayProc`, which sets the global flag `DelayStat` to `false`.

When `mDelay` is called, it places `DelayTask` in the Time Manager queue, and loops until `DelayStat` is set to `false` by `DelayProc`. Testing against the seconds timer (as done in `delayTest`) shows that delays of up to a second or so can be carried out to millisecond accuracy by this simple mechanism. Longer delays can be carried out using an anchor-based mechanism similar to that used for timing.

Fast Display

For machines prior to the Mac II (viz., the 128K, 512K, Mac Plus, and SE), displays can be changed at refresh rates by paging between the main and alternate video buffers. The method for doing this is presented on page III-20 of Inside Macintosh (1985). This method will not be further described here, except to point out that it is limited to two displays, and cannot be applied to Mac II - type machines. This section describes how these limitations can be overcome to allow sequential display of an indefinite number of bit images at refresh rates on all machines with standard display monitors.

It should be emphasized that this section applies only to bit (or binary) images, and not full-color displays. Also, the way that bit images are created and represented will not be described in great detail. For more information about these images, see e.g., pages I-142 – I-145 of Inside Macintosh (1985), or pages 99 – 120 of Macintosh Revealed (Chernicoff, 1985).

Basics

Although there exist specific Toolbox procedures for the machine-independent display of images, these generally require several ticks to transfer an image to the screen. One way of overcoming this problem is to draw the image directly on the screen via Toolbox commands. This will work for extremely simple images, but will still be susceptible to problems of screen synchronization if flicker-free display is required. The more general approach described here is to store images prior to display, either by using graphics commands on an off-screen port or by directly loading preassembled images from some other source. These are then copied directly to the screen buffer for display. If data can be transferred quickly enough, an image of arbitrary complexity can be placed upon the screen within one refresh cycle.

Procedure `setUpBuffers` in Fig. 7 shows how image buffers (together with their associated graphics ports) can be set up. The subroutine `getParams` first sets the values of all global variables pertaining to the buffers, such as the horizontal and vertical dimensions. Memory is allocated for the corresponding bit images, and the associated graphics ports are then set up. The complementary procedure `shutDownBuffers` in Fig. 8 shows how the memory may be released after use.

```
procedure setUpBuffers;
  var
    screenSize :  longint;

  procedure getParams;
    var
      environInfo :  sysEnvRec;
  begin { getParams }
    ErrCode := sysEnvirons(1, environInfo); { get info on current machine }
    CurrMacType := environInfo.machineType;
    SmallScreen := CurrMacType < envMacII;
    with MainPort^ do begin
      MonitorX := portrect.right; { horizontal screen dim }
      MonitorY := portrect.bottom; { vertical screen dim }
      CardWidth := portbits.rowbytes; { width of screen buffer (in bytes) }
    end; { with }
  end; { getParams }

  procedure assignPort (var grport :  grafPtr; grBuffer :  ptr);
  begin { assignPort }
    grport := grafPtr(newptr(sizeof(grafport))); { allocate new port }
    openPort(grport); { initialize port }
    with grPort^.portbits do begin
      baseaddr := grBuffer;
      rowbytes := MonitorX div 8;
      setRect(bounds, 0, 0, MonitorX, MonitorY);
    end; { with - grPort }
    grport^.baseaddr := grBuffer; { assign buffer to port }
  end; { assignPort }

begin { setUpBuffers }
  MainPort := grafPtr(newptr(sizeof(grafport))); { set up main (screen) port }
  openPort(MainPort); { initialize port; screen becomes bit image for port }
  MainBuffer := MainPort^.baseaddr; { address of screen buffer }

  getParams; { read off global variables for screen }
  screenSize := (MonitorX div 8) * longint(MonitorY); { size in bytes }
  ErrCode := compactMem(screenSize); { create Image port & buffer }

  ImageBuffer := newptr(screenSize);
  assignPort(ImagePort, ImageBuffer);

  { ...  image can be created here if required ...  }

  setport(MainPort);
end; { setUpBuffers }
```

Figure 7: procedure for setting up image buffers

```
procedure shutDownBuffers;
begin { shutDownBuffers }
  disposPtr(BackBuffer);
  disposPtr(ImageBuffer);
  closePort(ImagePort);
  disposPtr(ptr(ImagePort));
  closePort(MainPort);
  disposPtr(ptr(MainPort));
end; { shutDownBuffers }
```

Figure 8: procedure for shutting down image buffers

Owing to differences in video circuitry, it is useful to distinguish between two groups of machines: "small-screen" Macs and "large-screen" Macs. The former group encompasses the 128K, 512K, Mac Plus, and SE models. These have in common (1) a fixed (standard) screen 342 pixels high and 512 pixels wide, (2) a screen refresh synchronized with the tick timer, running at 60.15 Hz, and (3) an alternate video buffer that can be displayed by clearing bit 6 in VIA data register A.

"Large-screen" machines, which include the Mac II and Mac IIx, operate quite differently. These machines have (1) video cards containing screen buffers designed for a variety of screen sizes, (2) a screen refresh that depends on the particular monitor used, and (3) no alternate screen buffer. Although many different types of monitor are possible, one of the most commonly used is the "standard" Mac II monitor. This has a screen 480 pixels high and 640 pixels wide, with a refresh rate of 67 Hz.

It should be noted that because of the two different kinds of video circuitry, the procedures described below depend on the kind of machine running the program. However, the global variable `SmallScreen` (determined in procedure `getParams` in Fig. 7) can be used to select the appropriate procedures for the machine being used, as is shown in the simple routine `screenTest` in Fig. 9. In this way, routines can achieve a fair degree of machine-independent operation.

```
procedure screenTest;

  { ...  procedure smScreenTest defined here ...  }
  { ...  procedure lgScreenTest defined here ... }

begin { screenTest }
  if SmallScreen then
    smScreenTest
  else
    lgScreenTest;
end; { screenTest }
```

Figure 9: machine-independent procedure for testing fast transfer of images

Small-Screen Macs

The (main) screen buffer for small-screen Macs has a size of 342 rows × 512 columns; images to be displayed must therefore be stored in bit images of size 21888 bytes. The images themselves are created prior to display, either by carrying out a set of standard graphics commands using the port associated with the image (cf. `setUpBuffers` in Fig. 7), or by directly loading the images in from some other source. The number of images that can be set up in this fashion is limited only by the available RAM.

A routine for displaying images within one screen refresh is given by procedure `moveImage` in Fig. 10. This routine relies on a sequence of `blockMove` commands that transfer data between the buffer and the screen within 18 – 24 msec, depending on the particular machine used. Although data transfer using this method is not quite fast enough to move an image within one screen refresh, careful synchronization with the beginning of a video scan (by the use of `mDelay`) allows the location of the pixels being loaded into RAM to lag behind the scanning beam during the first tick, and to lead it during the second tick. Consequently, the image is displayed on the screen within one refresh cycle. Thus, provided that a pre-display duration of one tick is reserved for each change of display, an indefinite number of images can be transferred to the screen without flicker. It is important to realize that once an image has been moved to the screen, it must remain there during the pre-display interval of the succeeding image. Thus, images must remain on the screen for a minimum of two ticks. Apart from this, however, there are no constraints on the duration of a screen display.

An important note concerning `moveImage` is that a *sequence* of `blockMove` commands must be used, rather than just a single instruction. This is necessary because `blockMove`

```
procedure moveImage( fromPtr, toPtr :  ptr); { small-screen data transfer }
  const
    DelayVal = 4; { delay for start of transfer }
    SmallBlock = 1368; { 1/16 of screen (in bytes) }
    LargeBlock = 5472; { 1/4 of screen (in bytes) }
  var
    ix :  integer;
begin { moveImage }
  mDelay(DelayVal);
  for ix := 1 to 4 do begin { transfer first quarter }
    blockMove(fromPtr, toPtr, SmallBlock);
    toPtr := ptr(longint(toPtr) + SmallBlock);
    fromPtr := ptr(longint(fromPtr) + SmallBlock);
  end; { for }

  for ix := 1 to 2 do begin { transfer next two quarters }
    blockMove(fromPtr, toPtr, LargeBlock);
    toPtr := ptr(longint(toPtr) + LargeBlock);
    fromPtr := ptr(longint(fromPtr) + LargeBlock);
  end; { for }

  for ix := 1 to 4 do begin { transfer last quarter }
    blockMove(fromPtr, toPtr, SmallBlock);
    toPtr := ptr(longint(toPtr) + SmallBlock);
    fromPtr := ptr(longint(fromPtr) + SmallBlock);
  end; { for }
end; { moveImage }


procedure smScreenTest;
  var
    mStart, mStop :  longint;
    tVal, tStart, tStop :  longint;
begin { smScreenTest }
  fromPtr := ImageBuffer; { prepare for transfer to screen }
  toPtr := MainBuffer;
  tVal := tickCount + 2;
  tStart := tVal + 1; { image appears one tick after transfer starts }
  repeat
  until tickCount >= tVal; { synchronize with screen refresh }
  mStart := milliCount;
  moveImage(fromPtr, toPtr);
  tStop := tickCount;
  mStop := milliCount;

  writeln('transfer to screen took ',mStop - mStart,' msec');
  writeln(' and ',tStop - tStart,' refresh cycles');
end; { smScreenTest }
```

Figure 10: procedure for fast image transfer on small-screen Mac

transfers data in the reverse direction to that of the scanning beam, beginning at the end of the buffer and ending at the start. If a single `blockMove` instruction is used to transfer an entire image, this will completely disrupt the synchronization of data transfer with the scanning beam, resulting in a noticeable distortion of briefly-displayed images.

Large-Screen Macs

The large variety of monitor configurations possible for large-screen machines introduces a few more complications into video display. To begin with, large-screen machines use "slots" to handle monitor operations, each slot being assigned to a particular monitor. Furthermore, monitors may be driven by different video cards, each with its own particular characteristics. These matters are discussed on pages V-426 – V-428 and V-566 – V-569 of Inside Macintosh (1988), and will not be described in detail here. In the interests of simplicity, attention will be restricted to single-slot systems. It is assumed that the slot number (which must be between 9 and 14) is known.

The basic ideas behind fast display on large-screen Macs are very similar to those used for small-screen systems. Since `tickCount` cannot be used to synchronize data transfer with screen scans, it is necessary to use a vertical-retrace task for this purpose. In a fashion analogous to that used in the Time Manager queue, the `vblCount` field of a vertical-retrace task is decremented at the beginning of each screen refresh cycle; when this value reaches zero, the associated procedure is executed (see pages V-566 – V-569 of Inside Macintosh (1988) for further details). Procedure `setUpVCounter` in Fig. 11 shows how the vertical-retrace task `VideoTask` is set up. The `VideoProc` procedure associated with this task increments the global counter `videoCount` each time it is called. It also resets the `vblCount` field so that `VideoTask` is called during each vertical blanking interval.

Once it has been set up, `VideoTask` is installed into the vertical-retrace queue associated with the slot. The slot-interrupt task `SlotTask` is then placed into the slot-interrupt queue. This task, set up just after `VideoTask`, is called at each slot interrupt, using `doVBLTask` to carry out the tasks in the queue associated with the slot. Since an interrupt is generated at the beginning of every vertical blanking interval, this ensures that VideoTask increments the value of `videoCount`, so that it functions analogously to `tickCount`.

Exactly as in the case of the Time Manager queue, care must be taken to shut down the vertical-retrace tasks before exiting the program. Procedure `shutDownVCounter` in Fig. 11

shows how this can be done.

The screen buffer for a large-screen Mac is located in the RAM of the video card being used for display. The width of the screen buffer (in bytes) can be read off the `rowbytes` field of the screen bitmap, as shown in `getParams` in Fig. 7. When a monitor of width w is used, the displayed image is held in the w leftmost columns of the buffer, the other columns remaining unused. Buffers for bit images, of course, need only be large enough to hold the images to be displayed.

Procedure `lgScreenTest` in Fig. 12 shows how a bit image can be transferred to a "standard" monitor of size 480×640 pixels. Data transfer is based on the use of typecasting, which allows rows of the image to be declared as arrays. By assigning these arrays to the corresponding locations in the screen buffer, only the image bits are replaced, rather than the entire contents of the buffer. In this way, large-screen machines can generally move an image to the screen within one refresh cycle. For example, using procedure `lgScreenTest` on a standard Mac II, it has been found that an image can be transferred to the screen in 14 msec, which is less than the 15 msec used for a screen refresh. Thus, an indefinite number of bit images can be sequentially displayed on this screen at refresh rates.

Although this technique is rather specialized, it can be straightforwardly adapted for use on other machines and other screen sizes. Furthermore, if the screen size is so large that an image cannot be transferred sufficiently quickly, the synchronization technique for small-screen Macs can almost always be used to ensure single-refresh display of images. And finally, if a general display system indifferent to monitor size is required, it is only necessary to create image buffers the size of the screen buffer, and to transfer their contents with a variant of the `moveImage` routine given in Fig 10.

Combining Timing and Display

The procedures described above can be straightforwardly combined and/or modified to perform various interesting tasks. For example, it is possible to combine routines from `anchorTest` and `screenTest` into a simple routine that measures the time from image presentation to key press. The only complication is that the small-screen version must ensure that the start anchor is one tick *greater* than the actual value of `tickCount` at the beginning of data transfer, since the image does not begin to appear on the screen until the transfer is halfway completed.

```
{A+} { asynchronous } procedure VideoProc; { increment video counter }
begin
  videoCount := videoCount + 1; { increment global counter }
  VideoTask.vblCount := 1; { reset .vblCount field }
end;
{A-}

{A+} { asynchronous } procedure SlotProc;
begin
  Errcode := doVBLTask(CurrSlot); { carry out vertical-retrace task }
end;
{A-}

procedure setUpVCounter;
begin { setUpVCounter }
  CurrSlot := NoSlot; { dummy value }
  if not SmallScreen then begin { set up active slot }
    videoCount := 0; { initialize global value }
    VideoPtr := @VideoTask;
    with VideoPtr^ do begin { set up vertical-retrace task }
      qType := Ord(vType);
      vblAddr := @VideoProc;
      vblCount := 1;
      vblPhase := 0;
    end; { with }

    SlotPtr := @SlotTask;
    with SlotPtr^ do begin { set up slot-interrupt task}
      sqType := 6; { slot queue type }
      sqPrio := 255; { highest priority }
      sqAddr := @SlotProc;
      sqParm := 0;
    end; { with }

    readln(CurrSlot); { must have value between 9 and 14 }
    ErrCode := slotVInstall(QElemPtr(VideoPtr), CurrSlot);
    ErrCode := SIntInstall(SlotPtr, CurrSlot);
  end; { if }
end; { setUpVCounter }

procedure shutDownVCounter;
begin
  if CurrSlot <> NoSlot then begin
    ErrCode := SIntRemove(SlotPtr, CurrSlot);
    ErrCode := slotVRemove(QElemPtr(VideoPtr), CurrSlot);
  end;
end;
```

Figure 11: procedures for setting up and shutting down video task

```
procedure lgScreenTest (duration :  integer); { for monitor 480 high and 640 wide }
  type
    rowType = array[1..20] of longint; { 20 longints = 640 pixels }
    rowPtr = ^rowType;
  var
    row, monitorWidth :  integer;
    vVal, vStart, vStop :  longint;
    mStart, mStop :  longint;
    toBase, fromBase :  rowPtr;
begin { lgScreenTest }

  fromBase := rowPtr(ImageBuffer); { prepare for transfer to screen }
  toBase := rowPtr(MainBuffer);
  monitorWidth := MonitorX div 8; { horizontal size of buffer in bytes }
  vStart := videoCount + 1;
  repeat
  until (videoCount >= vStart); { synchronize with screen refresh }
  mStart := milliCount;
  for row := 1 to MonitorY do begin
    toBase^ := fromBase^;
    toBase := rowPtr(longint(toBase) + CardWidth); { video width = CardWidth * 8 }
    fromBase := rowPtr(longint(fromBase) + monitorWidth);
  end; { for }
  vStop := videoCount;
  mStop := milliCount;

  writeln('transfer to screen took ',mStop - mStart,' msec');
  writeln(' and ',vStop - vStart,' screen scan(s)');
end; { lgScreenTest }
```

Figure 12: procedure for fast image transfer on large-screen Mac

Such a procedure suffers from one minor restriction, viz., if a key press occurs while data is being moved, the response will not be noticed until after the data transfer is ended. This might be a problem for processes such as animated display, where data transfer takes place during a significant fraction of the time. However, transfer can be carried out in blocks that each require less than a millisecond to move. Precise timing can then be carried out by checking the status of `OSEventAvail` after each of these moves.

Finally, it should also be pointed out that the display routines can be readily modified to allow the rows of an image to be transferred to the screen at any rate and in any order. Thus, for example, if interlacing of the displayed rows is done carefully, an image can be transferred over an extended period of time, leading to a smooth transition from the image previously on the screen.

References

Apple Computer. (1985). *Inside Macintosh, Volumes I–III.* Menlo Park, CA: Addison Wesley.

Apple Computer. (1986). *Inside Macintosh, Volume IV.* Menlo Park, CA: Addison Wesley.

Apple Computer. (1988). *Inside Macintosh, Volume V.* Menlo Park, CA: Addison Wesley.

Apple Computer. (1988). *Macintosh Family Hardware Reference.* Menlo Park, CA: Addison Wesley.

Chernicoff, S. (1985). *Macintosh Revealed, Volume 1 : Unlocking the Toolbox.* Hasbrouck Heights, NJ: Hayden.

Kieley, J.M., and Higgins, T.S. (1988). *Precision Timing Options for the Apple Macintosh Family of Computers*, presented at the Eighteenth Annual Meeting of the Society for Computers in Psychology.

Lane, D.M., and Ashby, B. (1987). PsychLib: A library of machine language routines for controlling psychology experiments on the Apple Macintosh computer, *Behavior Research Methods, Instruments, & Computers*, **19**, 246–248

Reed College. (1985). *Rascal user manual: Macintosh language for real time I/O oriented development.* Portland, OR: Metaresearch.

Westall, R., Perkey, M.N., and Chute, D.L. (1986). Accurate millisecond timing on Apple's Macintosh using Drexel's MilliTimer, *Behavior Research Methods, Instruments, & Computers*, **18**, 307–311