COMPUTER TECHNOLOGY

Millisecond timing on the Apple Macintosh: Updating Drexel's MilliTimer

ROBERT F. WESTALL, M. NADINE PERKEY, and DOUGLAS L. CHUTE
Drexel University, Philadelphia, Pennsylvania

In 1986, we published in *Behavior Research Methods, Instruments, & Computers* a small library of assembler code for the public domain that permitted accurate millisecond timing for Apple's Macintosh computer. That version of MilliTimer used the Versatile Interface Adapter chip's Timer 1 (VIA T1), which is part of the standard Macintosh hardware configuration. The VIA T1, however, is also used by the Sound Manager, and thus the use of sound and accurate millisecond timing were incompatible. We offer for the public domain a new version labeled MilliTimer 2.2, which permits psychologists and other researchers to use the full capabilities of the Macintosh. MilliTimer 2.2 is compatible with popular teaching, research, and clinical tools such as HyperCard and MacLaboratory. MilliTimer 2.2 employs the Macintosh Time Manager, which is a part of the Macintosh operating system and is supported by Apple for the 512KE, Plus, SE, and Macintosh II machines. MilliTimer 2.2 is hardware-independent and automatically self-corrects for other operating system activities that also use the Time Manager. The self-correcting function is limited, however, and can result in a nonaccumulating error averaging about -3 msec. To preserve compatibility with earlier machines that do not contain the Time Manager, a simple mode parameter allows MilliTimer 2.2 to use the older, more accurate timing method.

The Apple Macintosh has enjoyed a rapid expansion in teaching, research, and clinical use. Its high resolution graphics, ease of operation, simple interapplication data transfer, and consistent user interface have made possible a number of programs for human performance measurement and control of laboratory apparatus. For example, in MacLaboratory for Psychology, the 1988 EDUCOM/NCRIPTAL award-winning package, the Macintosh is used as an editor to create a number of research-grade experimental protocols for teaching (Chute, 1988a; Chute & Daniel, 1988). In a research application, such as Reaction Time (Chute, 1988b), millisecond timing is used in the control of stimulus presentation and response data collection. In clinical use, the laboratory control application, Controller (Chute, Gaeman, & Ziegler, 1987), can employ its millisecond accuracy to monitor household circuitry for the safety of brain-injured or other cognitively impaired patients (Chute, Conn, DiPasquale, & Hoag, 1988). Readily authored tools such as HyperCard are empowering teachers, researchers, and clinicians with a programming capability. Since millisecond timing is a requisite for many programs and applications, we felt it appropriate to update the Drexel MilliTimer previously

published in this journal (Westall, Perkey, & Chute, 1986). An example of a Pascal implementation for MilliTimer 2.2 is presented in Figure 1 and, along with the code in the Appendix, is intended for the public domain.¹ Basically, MilliTimer 2.2 offers an alternative timing routine when programming needs require sound production and the use of the Macintosh Sound Manager, which was incompatible with the previous version of Drexel's MilliTimer. This new version permits the programmer to have millisecond accuracy along with the full range of Macintosh functions.

MilliTimer was originally designed to use the capabilities of the Versatile Interface Adapter chip's Timer 1 (VIA T1), a free-running interrupt-generating timer, clocked with an effective frequency of 783.36 kHz. This provided the best and most accurate means of millisecond timing on the Macintosh. However, because sound could not be used while the timer was running, and to eliminate hardware-dependence that might cause compatibility problems with future versions of the Macintosh, a new timer was developed. It uses the Time Manager, which provides interruptible task service of the millisecond order. In addition, this method does not require any hardware dependency within the MilliTimer code, making it portable to any machine (512KE, Plus, SE, Macintosh II, etc.) that contains the Time Manager. The Time Manager itself uses VIA T2, which is also used by the disk driver, the operating system, and any other process that wants to queue a task of its own. This means that the overhead of the interrupt manager, combined with the overhead to reload VIA T2 (this timer does not operate in free-run mode) and the over-

MilliTimer development has been supported in part by grants from Drexel University, Apple Computer, Inc., the Solon Foundation, and the Advanced Technology Center of SouthEastern Pennsylvania to Douglas Chute. The first two authors are with the Software Development Group, and the third author is with the Neuropsychology Program, all at Drexel University. Send reprint requests to Douglas L. Chute, Professor of Neuropsychology, Drexel University, Philadelphia, PA 19104.

head of all the tasks that might preempt our task, provides the potential for a drop in accuracy, which nonetheless can be controlled reasonably well. In order to achieve compatibility with earlier Macintosh machines (pre-512KE), which do not contain the Time Manager, a simple mode parameter allows MilliTimer 2.2 to use the original timing method (VIA T1). Care must be taken, however, to completely avoid using the Sound Manager while the MilliTimer is running with the older mode selected.

Accuracy

As can be seen in Figure 1, MilliTimer 2.2 requires mode parameter when the MilliTimer is started, which allows the programmer to select the method to be used internally to perform the timing—VIA-driven or Time Manager-driven. If the VIA mode is selected, the original MilliTimer routine will be used, and all rules concerning the VIA apply; sound may not be used while the timer is running. The VIA-driven timer is tested as accurate to 1 msec per 30 min. If the Time Manager mode is selected, the MilliTimer will attempt to self-correct the error caused by the overhead mentioned above, by using the tick count value. With this self-correcting procedure, the maximum error of the MilliTimer at any given time could be as much as -1 ticks, which corresponds to ap-

proximately -16.7 msec. Indeed this is the extreme worst case, and normal maximum error values usually average only -3 msec at any reading. The error, as a percentage of the total timed duration, will rapidly dissipate as the timed interval increases, and will decrease linearly with time.

Operation

The MilliTimer library consists of three routines, which install, read, and remove the MilliTimer.

The function NewMilliTimer is used to install and start the MilliTimer. The function result will be a long integer reference number that should be used in future calls to ReadMilliTimer and DisposeMilliTimer. The parameter mtStorage should contain either NIL or a pointer to a non-relocatable block of storage at least mtStorageSize bytes big. This storage block is used internally by the MilliTimer, and it cannot be disposed of until after DisposeMilliTimer is called. If NIL is passed for mtStorage, the function will create its own nonrelocatable block of storage in the application heap. The mtMode parameter is an integer specifying the mode of operation for the MilliTimer. It must be either mtUseVIA or mtUseTM. Selecting mtUseVIA will cause the older timing method (VIA T1) to be used. Selecting the VIA eliminates the

```

PROGRAM SimpleTest;

CONST
    mtUseVIA      = 0;
    mtUseTM       = 1;
    mtStorageSize = 142;

VAR
    elapsedTime, timerRefNum : LONGINT;

FUNCTION NewMilliTimer (mtStorage: Ptr; mtMode: INTEGER): LONGINT; EXTERNAL;
FUNCTION ReadMilliTimer (mtRefNum: LONGINT): LONGINT; EXTERNAL;
PROCEDURE DisposeMilliTimer (mtRefNum: LONGINT); EXTERNAL;

BEGIN
    timerRefNum := NewMilliTimer (NIL,mtUseTM);

    IF (timerRefNum = 0) THEN
        WRITELN ('The current machine does not contain the Time Manager') ELSE
    IF (timerRefNum = -1) THEN
        WRITELN ('Out of memory') ELSE
    BEGIN
        elapsedTime := ReadMilliTimer (timerRefNum);

        {*** Perform timed event ***}

        elapsedTime := ReadMilliTimer (timerRefNum) - elapsedTime;

        WRITELN ('The elapsed time was ', elapsedTime, ' milliseconds');
        DisposeMilliTimer (timerRefNum);
    END;
END.

```

Figure 1. Example of a Pascal implementation for MilliTimer 2.2.

possibility of using the Sound Manager and does not ensure compatibility with future machines. Selecting `mtUseTM` will cause the operating system's Time Manager to be used. Whenever possible, `mtUseTM` should be used for the reasons mentioned above. Note, however, that the Time Manager is not available on older 64K ROMs. The function result will normally be a valid reference number, with the exception of two possible error conditions. A result of "0" indicates an invalid or unavailable mode selection. A result of "-1" indicates a memory error and can only occur when NIL is passed for `mtStorage`.

The function `ReadMilliTimer` takes as a parameter the reference number obtained from the call to `NewMilliTimer` and returns the elapsed count (in milliseconds) since the call to `NewMilliTimer` was made. The function will return "-1" if a bad reference number is specified.

The function `DisposeMilliTimer` takes as a parameter the reference number obtained from the call to `NewMilliTimer` and shuts down the `MilliTimer`. If a bad reference number is specified, `DisposeMilliTimer` will do nothing.

`MilliTimer 2.2` is very flexible in many respects. Although the `MilliTimer` code listed in the Appendix is not segmented, the programmer may place any of the routines in any segment and can even unload the code while the `MilliTimer` is running. However, any time needed to load in the segment during calls to `ReadMilliTimer` will inflate the millisecond count, so it is good practice to leave the `MilliTimer` routines loaded during timing intervals or place them in the main segment. `MilliTimer 2.2` may be used in applications, desk accessories, drivers, XCMDs, and so forth, and it works well in development environments and under Multifinder. Caution should be used, however, when keeping the `MilliTimer` running during `MultiFinder` switches and CPU intensive operations, for the `MilliTimer` is interrupt-driven and may exhibit performance degradation. When the `MilliTimer` is not being

used, it should be shut down and reinstalled when needed. It is possible to install more than one `MilliTimer`, with successive calls to `NewMilliTimer`. Each concurrently running `MilliTimer`, however, installs its own interrupt task of the millisecond order. When more than one `MilliTimer` is running at a given time, latency of normal system functions will be evident. A properly constructed application should never need more than one `MilliTimer` in conjunction with one or more counter variables. It is important to note that although `NewMilliTimer` starts the internal millisecond counter counting from zero, a millisecond or more may elapse before control returns from `NewMilliTimer` back to the calling code. For this reason, timed events should be taken as the difference between two successive calls to `ReadMilliTimer`.

REFERENCES

- CHUTE, D. L., GAEMAN, D., & ZIEGLER, T. (1987). MacLaboratory Controller: A switch and A/D interface between Apple's Macintosh and peripheral apparatus. *Behavior Research Methods, Instruments, & Computers*, **19**, 205-209.
- CHUTE, D. L., & DANIEL, R. S. (1988). *MacLaboratory for psychology laboratory manual*. New York: McGraw-Hill.
- CHUTE, D. L. (1988a). *MacLaboratory for psychology courseware documentation*. New York: McGraw-Hill.
- CHUTE, D. L. (1988b). Reaction time. In *MacLaboratory for psychology: Part 2*. Santa Barbara, CA: Kinko's Academic Courseware Exchange.
- CHUTE, D. L., CONN, G., DiPASQUALE, M. C., & HOAG, M. (1988). Prosthesis Ware: A new class of software supporting the activities of daily living. *Neuropsychology*, **2**, 41-57.
- WESTALL, R. F., PERKEY, M. N., & CHUTE, D. L. (1986). Accurate millisecond timing on Apple's Macintosh using Drexel's `MilliTimer`. *Behavior Research Methods, Instruments, & Computers*, **18**, 307-311.

NOTE

1. The `MilliTimer` source listing, provided in the Appendix, was written for the Macintosh Programmer's Workshop Assembler v3.0.

APPENDIX

```

;
;
;   FILENAME
;       MilliTimer.a (written for MPW Assembler v3.0)
;
;
;   VERSION
;       2.2
;
;
;   AUTHOR
;       Robert Westall, Software Development Group, Drexel University
;
;
;   DESCRIPTION
;       Contains assembler source which provides an interrupt-driven timer with millisecond accuracy.
;
;
;   INTERFACE (Pascal)
;   CONST
;       mtUseVIA = 0;
;       mtUseTM = 1;
;       mtStorageSize = 142;
;

```

APPENDIX (Continued)

```
;
;
;   FUNCTION NewMilliTimer (mtStorage: Ptr; mtMode: INTEGER): LONGINT;
;   FUNCTION ReadMilliTimer (mtRefNum: LONGINT): LONGINT;
;   PROCEDURE DisposeMilliTimer (mtRefNum: LONGINT);
;
;
```

; the following are generic equates which make the source code more easily readable.

```
FALSE      EQU      0
TRUE       EQU      1
BYTE       EQU      1
WORD       EQU      2
LONG       EQU      4
```

; load in the files which tell the assembler where to find the values for the several constants
; and traps used in the MilliTimer code. the two equates directly below are flags which are
; needed so that the assembler can access some of the private equates in the hardware file.

```
HWNNonPortable EQU TRUE ;flags used to pull in "private" equates
onMac          EQU TRUE ;from hardware equate files

INCLUDE 'Traps.a' ;include OS and toolbox traps
INCLUDE 'SysEqu.a' ;include OS and toolbox equates
INCLUDE 'TimeEqu.a' ;include time manager equates
INCLUDE 'HardwareEqu.a' ;include hardware equates (VIA)
```

; tell the assembler that modules outside this unit need to access these routines by name.

```
EXPORT NewMilliTimer
EXPORT ReadMilliTimer
EXPORT DisposeMilliTimer
```

; the following are the local equates used in the MilliTimer routines.

```
mtUseVIA      EQU $0 ;mode setting when using VIA
mtUseTM       EQU $1 ;mode setting when using Time Manager
mtMSPerTick   EQU $11 ;milliseconds per tick (rounded up)
mtCorrectNeed EQU $3 ;ticks needed before fractional correction
mtSignature    EQU $4D543232 ;MilliTimer signature "MT22"

mtSigBytes    EQU 0
mtMode        EQU mtSigBytes+LONG
mtNeedDispose EQU mtMode+BYTE
mtMilliSecs   EQU mtNeedDispose+BYTE
mtSubMilliSecs EQU mtMilliSecs+LONG
mtLastTick    EQU mtSubMilliSecs+LONG
mtFractCount  EQU mtLastTick+LONG
mtTask        EQU mtFractCount+WORD
mtEndVars     EQU mtTask+tmQSize
mtVarsSize    EQU mtEndVars
```

```
;
;   INTERRUPT MilliTask;
;
;
;   PURPOSE
;       task called internally at interrupt time by the VIA or Time Manager.
;
```


APPENDIX (Continued)

```

;      FUNCTION    - returns a refnum to be passed in future calls to "ReadMilliTimer"
;                  and "DisposeMilliTimer". a result of "0" indicates that the requested
;                  mode is either invalid or unavailable on the current machine. a result
;                  of "-1" indicates that the function was unable to allocate memory for
;                  storage - this could only occur when "NIL" is passed in "mtStorage".
;
;

```

mtStorageSize EQU	EndTask-MilliTask
-------------------	-------------------

NewMilliTimer	PROC	
LINK	A6,#0	;setup local frame
MOVE.L	#mtStorageSize,D0	;size of task code and variables
MOVE.L	10(A6),-(SP)	;get storage pointer parameter
BNE.S	@1	;no need to allocate memory
MOVE.L	MinusOne,14(A6)	;assume memory error will occur
MOVE.L	D0,-(SP)	;save task code size on stack
_NewPtr		;get a block of memory
MOVE.L	(SP)+,D0	;restore task code size from stack
MOVE.L	A0,(SP)	;test returned pointer for NIL
BNE.S	@1	;valid pointer, short branch
MOVE.L	(SP)+,D0	;memory error, cleanup stack
BRA	@0	;branch to exit
@1: LEA	MilliTask,A0	;prepare to copy task code block
MOVEA.L	(SP),A1	;into static storage area
_BlockMove		;copy code
MOVEA.L	(SP)+,A1	;restore pointer to static block
ADDQ.L	#2,A1	;reg now points to MilliTimer vars
TST.L	10(A6)	;test storage pointer parameter
SEQ	mtNeedDispose(A1)	;if "NIL", set internal flag
CLR.L	14(A6)	;assume bad mode requested
MOVE.B	9(A6),mtMode(A1)	;store mode internally
ANDL.W	#\$FFFE,8(A6)	;test mode parameter range
BNE	@0	;invalid mode, branch to exit
MOVE.L	VIA,D0	;get VIA base address
OR.B	mtMode(A1),D0	;combine it with selected mode
TST.L	D0	;can't both be zero
BEQ.S	@0	;mode unavailable, branch to exit
TST.B	mtMode(A1)	;test mode again
BEQ.S	@3	;VIA selected, branch to install
TST.W	ROM85	;TM selected, ROMs OK?
BMI.S	@0	;mode unavailable, branch to exit
MOVE.L	A1,D0	;move storage pointer to data reg
MOVE.L	D0,-(SP)	;save it on stack
SUBQ.L	#2,D0	;fix it to point to actual task
MOVE.L	D0,mtTask+tmAddr(A1)	;initialize the task record
LEA	mtTask(A1),A0	;get task element pointer
_InsTime		;install task into time manager
BSR.S	@4	;sync timer with vbl
MOVEA.L	(SP),A1	;restore pointer to vars in reg
MOVE.L	Ticks,mtLastTick(A1)	;save current tick count
MOVEQ	#1,D0	;prime value of 1 millisecond
LEA	mtTask(A1),A0	;get task element pointer
_PrimeTime		;prime the time manager task
BRA.S	@2	;installation almost done, branch
@3: MOVEA.L	VIA,A0	;load base address of VIA into reg
MOVE.B	##01000000,vIER(A0)	;disable T1 interrupts (just in case)
BSR.S	@4	;sync timer with vbl

APPENDIX (Continued)

```

MOVE.L    Ticks,mtLastTick(A1)           ;save current tick count
MOVE.L    A1,D0                           ;move storage pointer to data reg
MOVE.L    D0,-(SP)                        ;save it on stack
SUBQ.L    #2,D0                           ;fix it to point to actual task
LEA       lv1DT,A1                        ;get address of level 1 dispatch table
MOVE.L    D0,24(A1)                       ;store task pointer into T1 int vector
ORL.B     #%01000000,vACR(A0)             ;select T1 free-run mode
ANDL.B    #%01111111,vACR(A0)            ;disable pulse triggering on T1 interrupts
MOVE.W    #NTicks,D0                     ;store counter value into reg
MOVE.B    D0,vT1C(A0)                    ;load low order byte into T1 counter/latch
ASR.W     #8,D0                           ;shift counter value down a byte
MOVE.B    D0,vT1CH(A0)                   ;load high order byte into T1 counter
MOVE.B    #%11000000,vIER(A0)            ;enable T1 interrupts
@2: MOVEA.L (SP)+,A1                      ;restore pointer to vars in reg
MOVE.L    #mtSignature,mtSigBytes(A1)     ;validate MilliTimer with signature
MOVE.L    A1,14(A6)                      ;vars pointer is refnum result
@0: UNLK    A6                            ;undo local frame
MOVE.L    (SP),6(SP)                     ;bump return address up on stack
ADDQ.L    #6,SP                           ;remove parameters
RTS                                              ;return to caller
@4: MOVE.L    Ticks,D0                    ;put current tick count into reg
CMP.L     Ticks,D0                        ;has tick increased yet?
BEQ.S     @4                              ;loop until sync with vbl is achieved
RTS                                              ;return to caller
ENDPROC

;
;
FUNCTION ReadMilliTimer (mtRefNum: LONGINT): LONGINT;
;
;   PURPOSE
;       returns the elapsed time (in milliseconds) since the call to "NewMilliTimer".
;       a return value of "-1" signifies that "mtRefNum" is invalid.
;
;   PARAMETERS
;       mtRefNum    -   refnum which was returned from the call to "NewMilliTimer".
;
;
ReadMilliTimer  PROC
    MOVEM.L (SP)+,A0/A1                    ;pop return address and refnum
    MOVE.L   A0,-(SP)                      ;put return address back
    MOVE.L   MinusOne,4(SP)                ;assume that refnum is bad
    MOVE.L   A1,D0                          ;put refnum into a data reg
    ASR.B    #1,D0                          ;test for odd address
    BCS.S    @0                             ;bad refnum, branch to exit
    CMPL.L   #mtSignature,mtSigBytes(A1)   ;check for valid signature
    BNE.S    @0                             ;invalid refnum, branch to exit
    MOVE.L   mtMilliSecs(A1),4(SP)         ;set the function result
@0: RTS                                          ;return to caller
ENDPROC

;
;
PROCEDURE DisposeMilliTimer (mtRefNum: LONGINT);
;
;   PURPOSE
;       shuts down the MilliTimer having the specified refnum and de-allocates
;       all private storage used (if any).
;

```

APPENDIX (Continued)

```

;      PARAMETERS
;      mtRefNum      -  refnum which was returned from the call to "NewMilliTimer".
;

DisposeMilliTimer PROC
    MOVEM.L  (SP)+,A0/A1          ;pop return address and refnum
    MOVE.L   A0,-(SP)             ;put return address back
    MOVE.L   A1,D0                ;put refnum into a data reg
    ASR.B    #1,D0                ;test for odd address
    BCS.S    @0                   ;bad refnum, branch to exit
    CMPL.L   #mtSignature,mtSigBytes(A1) ;check for valid signature
    BNE.S    @0                   ;invalid refnum, branch to exit
    MOVE.L   A1,-(SP)             ;save refnum on stack
    TST.B    mtMode(A1)           ;test current mode
    BEQ.S    @1                   ;using the VIA, short branch
    LEA      mtTask(A1),A0        ;using the time manager
    _RmvTime                               ;dequeue task from time manager
    BRA.S    @2                   ;short branch over VIA code
@1: MOVEA.L  VIA,A0               ;load base address of VIA into reg
    MOVE.B   #%01000000,vIER(A0) ;disable T1 interrupts
@2: MOVEA.L  (SP)+,A0             ;restore refnum from stack
    CLR.L    mtSigBytes(A0)       ;invalidate the MilliTimer vars
    TST.B    mtNeedDispose(A0)    ;did we allocate storage?
    BEQ.S    @0                   ;no, branch to exit
    SUBQ.L   #2,A0                ;convert refnum to storage pointer
    _DisposPtr                       ;de-allocate storage
@0: RTS
    ENDPROC
END

```

(Manuscript received May 8, 1989;
accepted for publication June 20, 1989.)