R. Davis

Timing in the Laboratory: Hardware and Software Solutions

Abstract: Measurement of time is critical in most scientific experiments. Computers can be used to accurately monitor time. This paper explores a series of software and hardware techniques for measuring time. Software techniques utilizing the IBM-PC, AT and compatible computers are discussed and ways of making timing independent of software changes are emphasized. A general, hardware solution utilizing a multi-function counter/timer integrated circuit is discussed.

Keywords: Timing, computer control, clock, IBM-PC, Metrabyte CTM-05[1].

Introduction

The precise control of timing is critical for just about every laboratory experiment. Computers can monitor time with great accuracy, so when a process is under the control of a computer, it is advantageous also to delegate timing requirements to the computer. Researchers who use data acquisition products (for example Labtech Notebook, ASYST or LabWindows) can handle timing as a black box. This may or may not be an important detail. However, for those who write data acquisition programs, timing issues can be a detail requiring a lot of careful thought and investigation. The purpose of this paper is to explore some options for incorporating timing control into experiments and to demonstrate how the use of a hardware clock can often simplify the software requirements. In addition, a series of flow charts will be provided which can be used as a starting point by the scientific/engineering user as a basis for high precision timing using the Metrabyte CTM-05 Counter/Timer Interface Card. The frame of reference for this discussion is the IBM Personal Computer and equivalent computers.

Rickie R. Davis, U.S. Department of Health and Human Services, Public Health Service, Centers for Disease Control, National Institute for Occupational Safety and Health Division of Biomedical and Behavioral Science, Physical Agents Effects Branch, 4676 Columbia Parkway, Cincinnati, Ohio 45226.

Background

For this paper, there are two forms of timing: ***event timing*** and ***control timing***. Event timing is analogous to a stopwatch. Event timing answers the question "What time (relative or absolute) did a certain event happen?". An example might be a "time code generator" where neuron spikes are tagged with a relative time by a high resolution clock for later analysis (Pfeiffer & Molnar, 1976). Control timing turns events on and off for various lengths of time. Control timing answers the question "What time should an event be started?" Triggering an analog-to-digital converter at 1 second intervals is an example of control timing. The hardware and software associated with these two types of timing are usually very different.

[1]CTM-05 is a trademark of Metrabyte, Inc., 440 Myles Standish Blvd., Tauton, MA 02780. Am9513 and Am9513A are trademarks of Advanced Micro Devices, Inc. Figures 2, 3, and 4 are copyright Advanced Micro Devices, Inc. and are used by permission. IBM, PC and AT are trademarks of International Business Machines. BASICA, OS/2 and Microsoft C 5.0 are trademarks of Microsoft Corp. MSCHRT is a product of Ryle Design, P.O. Box 22, Mt. Pleasant, MI 48804. Labtech Notebook is a product of Laboratory Technologies Corp. LabWindows is a product of National Instruments. ASYST is a product of Asyst Software Technologies, Inc. The use of trade names is for reference only and does not imply endorsement by the author, National Institute for Occupational Safety and Health or the U.S. Public Health Service.

Control Timing

Program Loop

The simplest form of control timing is the program loop. Example 1 shows a double loop in the C programming language (Kernigham and Ritchie, 1978). Converting the program fragment to another language is trivial. For every cycle of the outside loop there are 60000 cycles of the inside loop. By using a stopwatch the programmer can produce loops of a large number of iterations so delays of the correct length occur.

```
.
.
.
for(i=0;i<60000;i++)          /* outside loop */
   for(j=0;j<60000;j++)       /* inside loop */
        ;
.
.
.
```

Example 1.

There are three serious problems with this approach. First, this solution is very computer and compiler specific. If the computer, operating system or compiler is changed, the programmer must re-time all of the loops. This reduces portability between systems (in fact, some "optimizing" compilers will eliminate empty loops of this sort). Second, even if the computer, operating system or compiler is not changed, the exact timing will change from one day to the next depending upon on-going computer activity. Thus, background tasks and system activity may affect the timing of the loop from one day to the next. The effects of background activity may become a more serious problem as multi-tasking operating systems such as OS/2 and Unix become more common in the laboratory. A third criticism of this technique is that, basically, the entire resources of the processor are being used to do a relatively simple task. By moving part of the timing to hardware the processor can accomplish other work.

PC System Clock

A second control technique uses the PC system clock. This clock keeps a running time-of-day, once it is set at startup. About 18.2 times per second, a hardware interrupt is generated which causes the processor to increment this memory location. Reading this memory location is relatively simple and most professional compilers have library routines to change the clock tick value into an hours:minutes:seconds format. However, for timing purposes the raw clock tick count is the only part needed. By storing the initial value, adding in the number of ticks to wait (in 55 ms steps) and reading the clock until the proper number of ticks have occurred, you have a cheap, universal solution which has the resolution of approximately 55 milliseconds. By carefully constructing your program you may be able to have the processor do other useful work while waiting. A disadvantage is that you may need an accuracy of better than 55 ms resolution. Also, some programs, including BASICA, may change the clock timing or turn off interrupts to the system clock to increase performance. This may affect your timing. By issuing a specific instruction, the microprocessor can turn off all maskable interrupts. This is useful when the microprocessor is doing very time-critical code since it no longer has to be "distracted" by side trips to the system clock interrupt code every 55 ms. However, if you depend upon the system clock for keeping track of time, turning off interrupts for long periods of time can be disastrous, since "ticks" are no longer being registered. Example 2 is a code fragment using Microsoft C 5.0 to show how you would use the system clock to wait approximately 1 second.

For higher resolution requirements, subroutine libraries are available which can be called from C and other higher level languages which utilize clever techniques to emulate virtual timers in software. The IBM-PC contains 3 internal counter/timers : one is utilized for generating the hardware-interrupt for the previously mentioned time of day counter, one controls the speaker and one generates the "refresh" cycle absolutely required by the dynamic memory used in most computers. Abrash (1989) discussed and demonstrated techniques for timing segments of software from 10 microseconds to 54 milliseconds by reprogramming the internal counter/timer used to generate the time of day interrupts. Other authors have discussed utilizing the dynamic memory "refresh" counter. The refresh counter should never be modified, only read. A number of commercial subroutine packages are available which allow this sort of timing to the one microsecond range. It is good economy to utilize them rather than writing your own (e.g. "Inside!" by Paradigm Systems; "MSCHRT" by Ryle Designs for Microsoft C).

External Clock

A third technique actually involves two solutions, one for control timing and the other for event timing. By using a digital input/output board, a handful of gates and a crystal -controlled clock, an external clock can be built (see Figure 1). This particular external clock allows a resolution of 1 millisecond. The computer software required to service this clock must be carefully written to maintain maximum accuracy, especially on computers using the old 4.77 mHz standard (early PC's). This clock can be used both to time events and to control the timing of experimental processes. For advanced programmers, the "clock out" connection could be attached to a hardware interrupt control line and an interrupt *service routine* written to service a clock generated interrupt. An interrupt-based technique can also be used with the solution discussed later. This is an advanced solution requiring an understanding of how interrupts are generated and serviced in the PC. Examples of interrupts in other contexts have been published by Hunt (1985), Swafford (1988), Holub (1987) and others for the Personal Computer. The second part of this solution is

```
/* holdit - hold up program for one second */

#define TIMER 0x00046c                    /* low address for system clock */
#define ONE_SECOND 1198180 / 65535        /* ticks per second */

void holdit()
{
    int far *rawcount;                    /* pointer to raw count integer */
    unsigned int pause = ONE_SECOND;      /* initialize pause to 18 ticks */

    unsigned int tick;

    rawcount = TIMER ;                    /* set pointer to address of raw count */
    for( ;pause > 0; pause--){
        tick = *rawcount;                 /* read the current value of timer */

        while( tick == *rawcount)         /* hold until next */
                ;                         /* tick of timer */
        }
    }
```

Example 2.

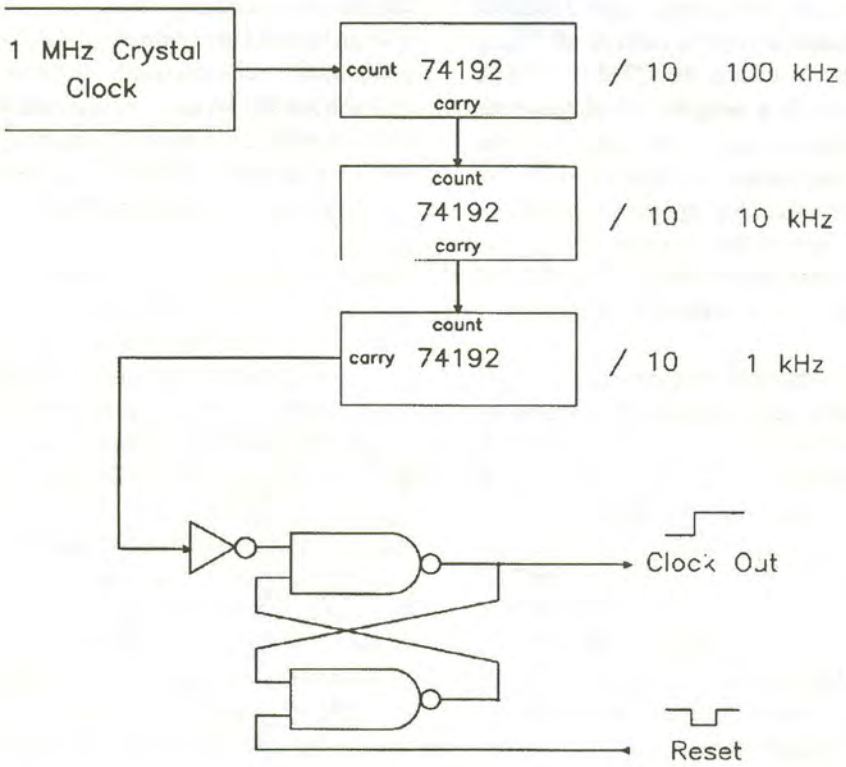


Figure 1: A simple 1 kiloHertz clock for timing. The 1 mHz TTL crystal clock is divided by 74192 decade counters connected in series. The final 7400 is configured as a flip flop so a carry pulse from the final counter "sets" the output to the computer. This prevents the computer from missing the final count. The computer then resets or arms the flip flop for the next counting cycle. The Metrabyte CTM-05 uses the Am9513A chip to produce similar results using programmable counters.

more of a control solution in which a digital output line is used to trigger a 555 "one shot" integrated circuit timer. Using resistors and capacitors, these highly versatile timers can be triggered by a digital output line on the digital i/o card. Using this technique, pulses lasting hours can be generated. However, using a solution based on a 555 integrated circuit leads to lack of flexibility and requires careful calibration. However, in some situations a totally hardware-based solution is the most economical and easily implemented. Lancaster (1974, 1977) has published a series of popular books documenting the abilities of the 555.

Programmable Hardware Clock

The remainder of this article is devoted to a programmable hardware clock-based solution to laboratory timing. The clock of interest is a Metrabyte CTM-05. The CTM-05 is a general solution for timing problems which can be used for event-timing and control-timing. It is based on an Advanced Micro Devices (AMD) Am9513A System Timing Controller integrated circuit. The documentation furnished by Metrabyte is a subset of the more thorough documentation provided in AMD's Technical Manual (AMD, 1984). The Technical Manual is required for anyone interested in doing advanced programming of the CTM-05 card.

The major advantage of using the CTM-05 card is that it has 1 microsecond accuracy with maximum flexibility. It can both time and count responses in the environment up to 7 million/second. It can be programmed to remove most or all timing chores from the computer. In effect, the CTM-05 can be programmed, armed and the clock program exited. Since the CTM-05 can continue to time and count until stopped or reset you may do other, non-related tasks with your PC while the CTM-05 generates control pulses for equipment, counts or times events. The native speed of the computer, the type of compiler and whether interrupts are enabled or disabled is of little consequence since the CTM-05 operates independently from the host computer.

The major disadvantage is the programming complexity of the CTM-05. The assembly language driver provided by Metrabyte with the CTM-05 works in conjunction with interpreted BASIC. The manual indicates that it was written to be simple but flexible. It is fine for doing very general functions with the CTM-05, however, for more specific powerful control of the CTM-05, the experimenter needs to control the hardware more directly. One goal of this article is to provide a series of templates which can be a starting point for exploring the capabilities of the CTM-05 (and the Am9513A). For those interested, the author can provide C programs which demonstrate aspects of CTM-05 programming.

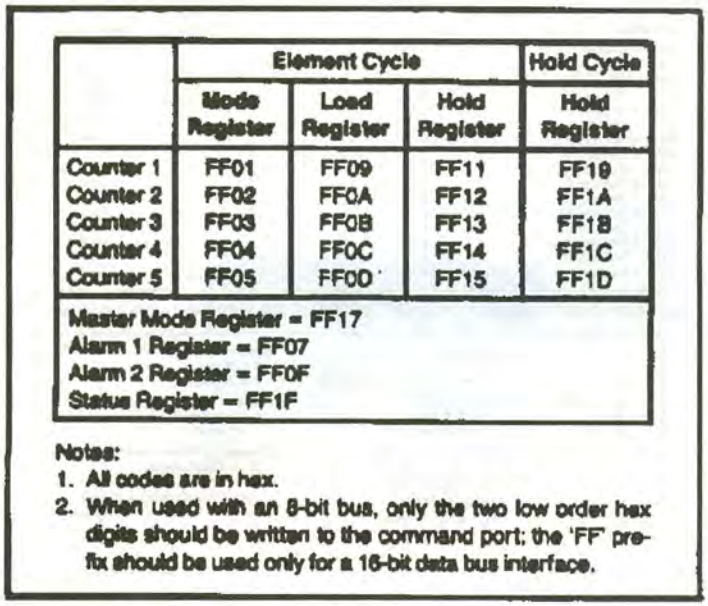
Physical Description of the CTM-05

The Metrabyte CTM-05 card requires one expansion slot in a PC or AT compatible. The card has an onboard 1 mHz crystal for 1 microsecond accuracy. The Am9513A integrated circuit is designed with 5 independent counter/timers in it. Metrabyte has constructed the card with a standard 37 pin "D" plug on the rear bracket which allows interfacing to the 5 counter/timer inputs, gates and outputs.

A pulse on an *input* causes that counter to show an increment or decrement with each transition. With a level voltage applied, a *gate* can either enable counting or disable counting in one or more counter/timers. The *output* produces a pulse (either +5 or ground) or a change in voltage level when the associated counter/timer reaches terminal count (TC). In addition, ground and +5 volts are available on two plug pins. Eight bits of digital *input*, 8 bits of digital *output* plus an *input* for the hardware interrupt line are also linked to external circuits through pins on this plug. Great flexibility can be introduced by wiring between various outputs and inputs using a standard 37 pin "D" socket. However, the inputs and outputs of the counters can be routed internally via programming, and this allows for even greater flexibility. The various options available are programmed by setting and clearing bits in the *Master Mode Register* and each of the counter/timer *mode registers*. The final *output* available on the CTM-05 D plug is the *scaled time base oscillator* output (in the documentation referred to as f_{out}).

The CTM-05 registers require 4 addresses of i/o space. The base address can be modified by changing the on card switches. The highest CTM-05 address is the *digital output address*. The next lower address is the *digital input address*. This article will not be concerned with use of the digital i/o ports since their use is fairly straightforward. The base address on the CTM-05 is the Am9513A *data register* and the next higher address is the Am9513A *command/status register*. The command/status register is the *command register* when written to and is the *status register* when read from. On the PC the command register utilizes 1 byte long operation codes (op codes). The status register returns 1 byte consisting of the current output states of each of the 5 counters (1's or 0's) and a bit called the byte counter. The byte counter (bit 0) is set if the Am9513A is expecting another byte of two bytes of information. Status register bits 1,2,3,4 and 5 map directly to the current outputs of the corresponding counters. This is a handy way to detect the output state of each counter. Bits 6 and 7 should be ignored.

If you are using the CTM-05 in an AT or 80386 based machine you should realize that the computer i/o bus can exceed the speed of the clock on the CTM-05 card. In the driver program provided by Metrabyte they degrade the performance of the AT by using a software looping technique (similar to the one described in example 1) between writes to the CTM-05. If problems are experienced with the CTM-05, you might want to use the second technique described in example 2 to wait one or more ticks of the computer system clock between writes to the card. Of course, there is no reason why you cannot use that time between writes to the CTM-05 to do other useful work, since the CTM-05 will wait for you.



	Element Cycle			Hold Cycle
	Mode Register	Load Register	Hold Register	Hold Register
Counter 1	FF01	FF09	FF11	FF19
Counter 2	FF02	FF0A	FF12	FF1A
Counter 3	FF03	FF0B	FF13	FF1B
Counter 4	FF04	FF0C	FF14	FF1C
Counter 5	FF05	FF0D	FF15	FF1D

Master Mode Register = FF17
Alarm 1 Register = FF07
Alarm 2 Register = FF0F
Status Register = FF1F

Notes:
1. All codes are in hex.
2. When used with an 8-bit bus, only the two low order hex digits should be written to the command port; the 'FF' prefix should be used only for a 16-bit data bus interface.

Figure 2: List of operation codes (op codes) written to the CTM-05 command register to read back values stored in the various counter/timer registers as well as write values to these registers. Note that in PC applications only a single byte is sent to the command register. If pointer sequencing is disabled "Element Cycle" and "Hold Cycle" are unimportant. Copyright* Advanced Micro Devices, Inc., 1984. Reprinted with permission of copyright owner. All rights reserved.

Functional Description

Upon examining the AMD Technical Manual for the Am9513A one notices there are three kinds of opcodes: those operating on a single *register*, those operating on a *single counter*, and those operating on *multiple counters*. Figure 2 is a reproduction of the figure in the Am9513A manual which shows some of the opcodes for operations on single registers of a single counter/timer. These codes are in hexadecimal. (Single counter and multiple counter op codes will be demonstrated in the programming section). By writing one of these values to the command register you can select (for reading or writing) the mode, load and hold registers for each of the five counter/timers. In addition you may point to the master mode register, status register and the two alarm registers (which we will not use). The master mode register is shown in Figure 3. Note that the 16 bits actually "group" into 9 functional divisions.

The functions controlled within the master mode register are:

1. how the counters count (binary or binary coded decimal),
2. whether the automatic data pointer increment is enabled or disabled,
3. whether the driving frequency (f_{out}) is available on the output pin of the chip or not,
4. the data bus width (8 bits or 16 bits),
5. what the driving oscillator frequency should be divided by in order to drive the counters (1 through 16 are available),
6. the source of f_{out} (possibilities include the 1 mHz chip oscillator frequency including 4 subharmonics or one of 10 pins on the 37 pin D connector),
7. enabling bits for alarm 1
8. enabling bits for alarm 2 ,
9. the time of day mode enable.

The most important bits by far are the f_{out} source bits and the fout divider. The f_{out} source bits allow you to choose 1 of 5 internal frequencies derived from the CTM-05 1 mHz crystal. With the Scaler control bit set to Binary-Coded-Decimal (BCD) the frequencies available are 1 mHz, 100 kHz, 10 kHz, 1 kHz and 100 Hz. (These sources are referred to in the figure as f1, f2, f3, f4 and f5 respectively). With the Scaler control bit set to Binary, the frequencies available, f1 through f5, are 1 mHz, 62.5 kHz, 3.90625 kHz, 244.14 Hz and 15.258 Hz. These options give a wide range of frequencies, however, you may further divide these values to get even lower frequencies using one of these harmonics as input to one or more of the five counter/timers and using that *output* as another counter/timer input. Another possibility is using an external oscillator as a time source.

As with the master mode register, each counter/timer has its own 16 bit *mode* register. Again these 16 bits group into 9 functional divisions. These bit "groupings" control how (and if) external *gating* is applied to the counter/timer, whether counting occurs on a *rising or falling clock edge*, the *source* of the clock pulses (including f1, f2, f3, f4 or f5), the source for *reloading* the counter upon terminal count, whether counting is done *only once or continuously*, whether counting is *binary or BCD*, whether counting is done by *incrementing or decrementing*, and the *dynamic state* of the output of this counter.

In addition to a *mode register*, each counter/timer has a 16 bit *load* and a 16 bit *hold register*. In fact, there is no true access to the working counter/timer 16 bit register. Access is

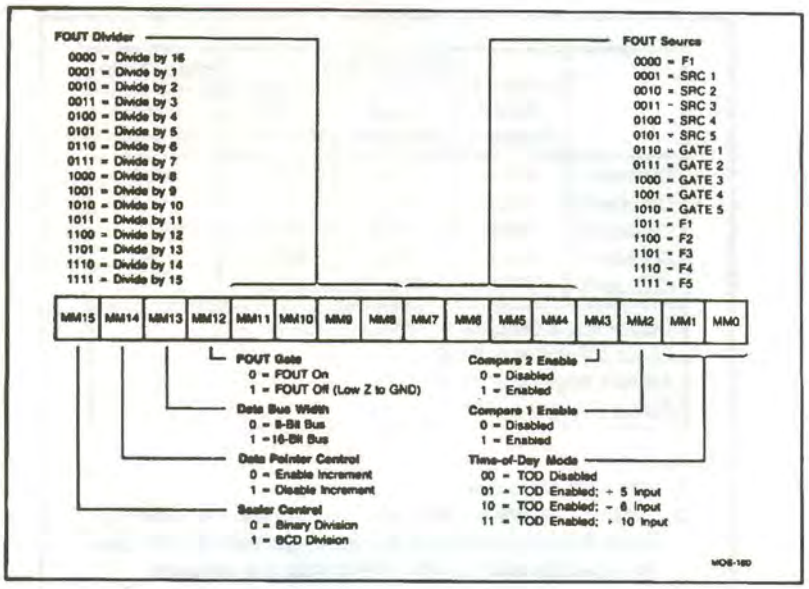


Figure 3. Diagram of the single Master Mode Register. Note that there are 9 groupings within the 16 bits. The alarm registers (compare 1 and compare 2) and the time-of-day modes are not described in this paper.

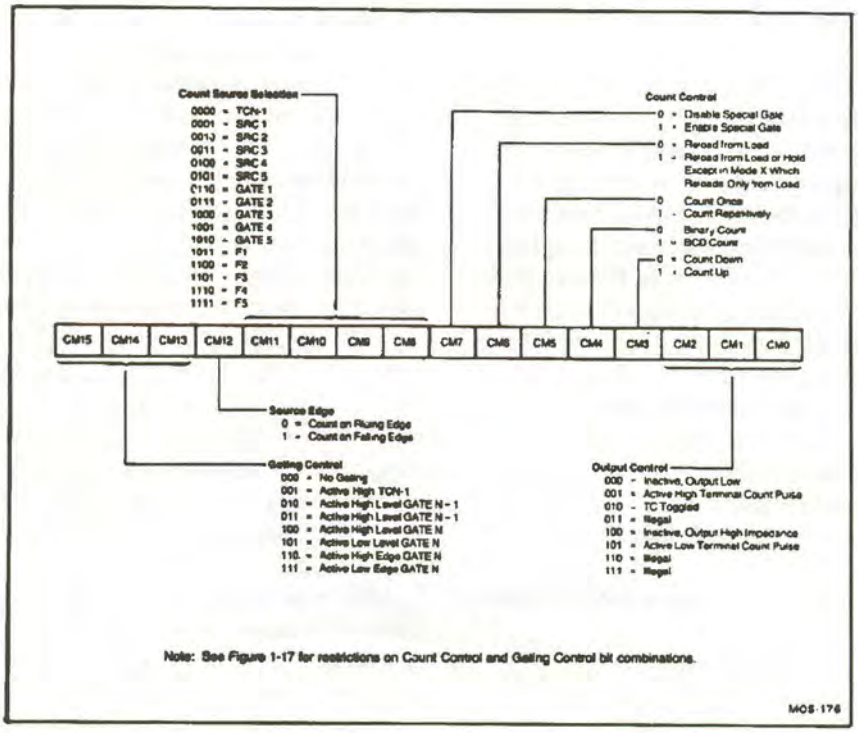


Figure 4: Diagram of an individual counter/timer mode register. There are 5 counter mode registers, one for each counter/timer. Again note that there are 9 groupings within the 16 bits. TCN-1 indicates respond to the terminal count of the next lower numbered timer/counter. GATE N (and N±1) relate to hardwired pins on the chip and the CTM-05, N referring to the gate pin with the same number, minus the next lower and plus the next higher. SRC also refers to hardwired pins. F1 through F5 refer to the 1 mHz on board clock and its subharmonics. By specifying bits 0, 1 and 2 with the counter terminal count the output can pulse high to low, pulse low to high or toggle between high and low.

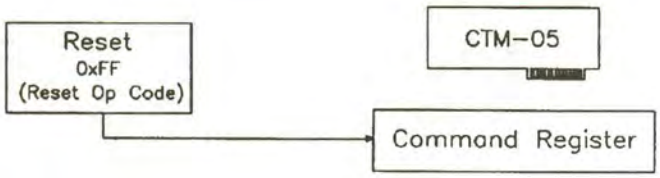


Diagram 1

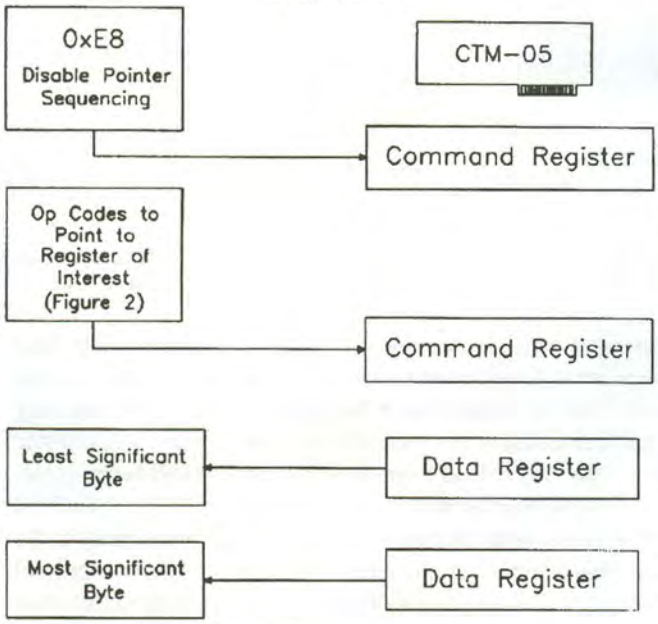


Diagram 2

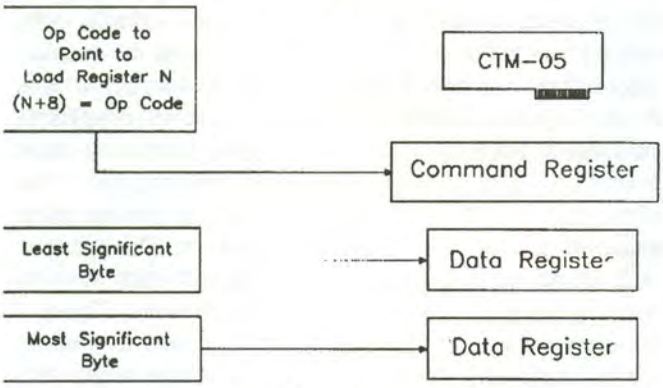


Diagram 3

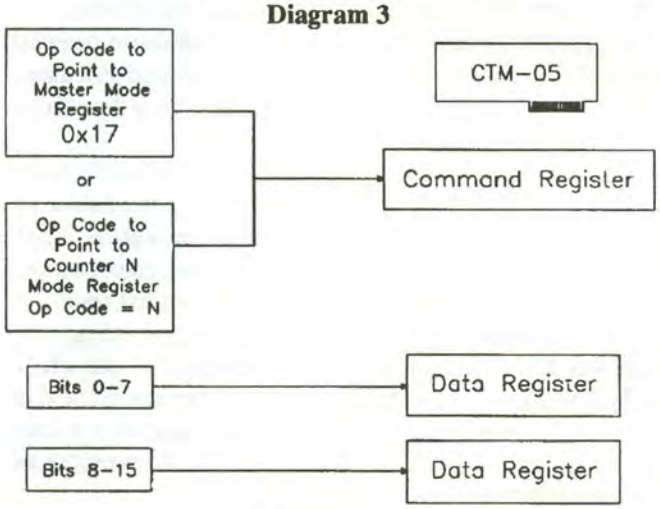


Diagram 4

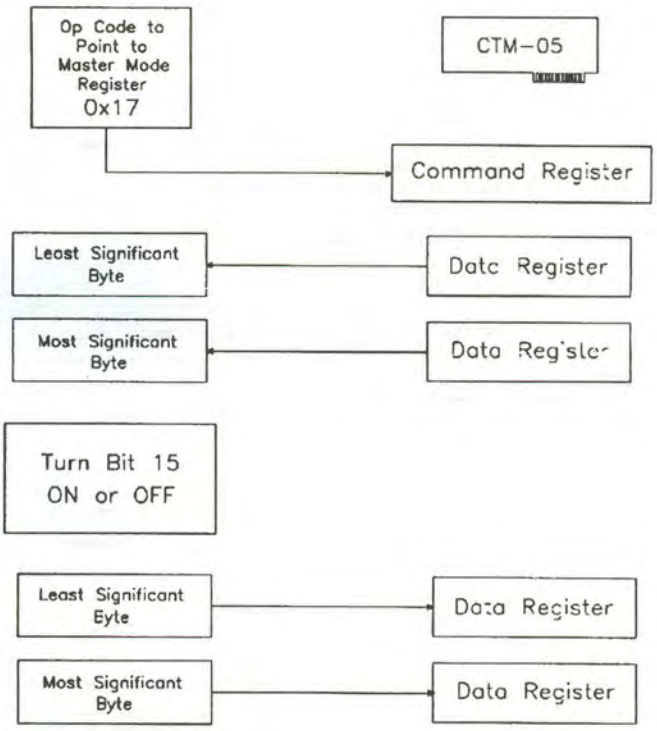


Diagram 5

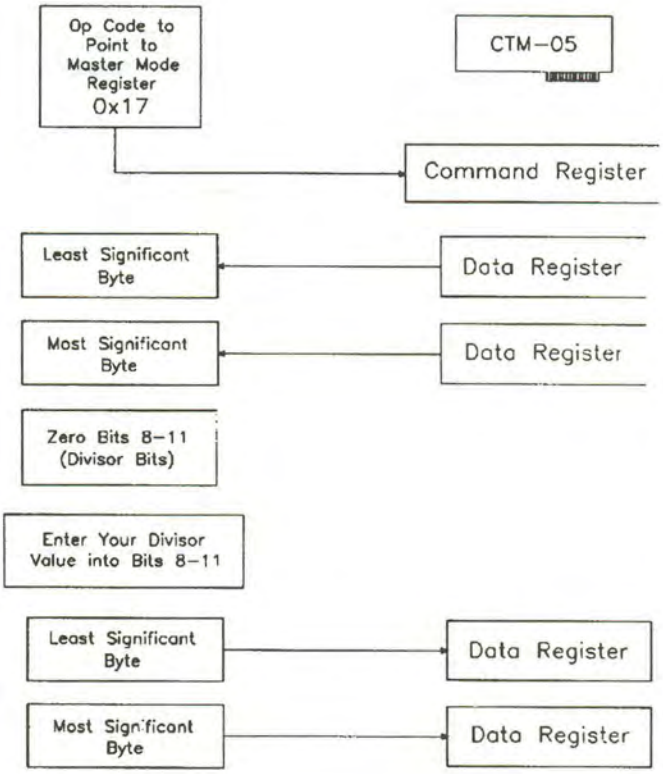


Diagram 6

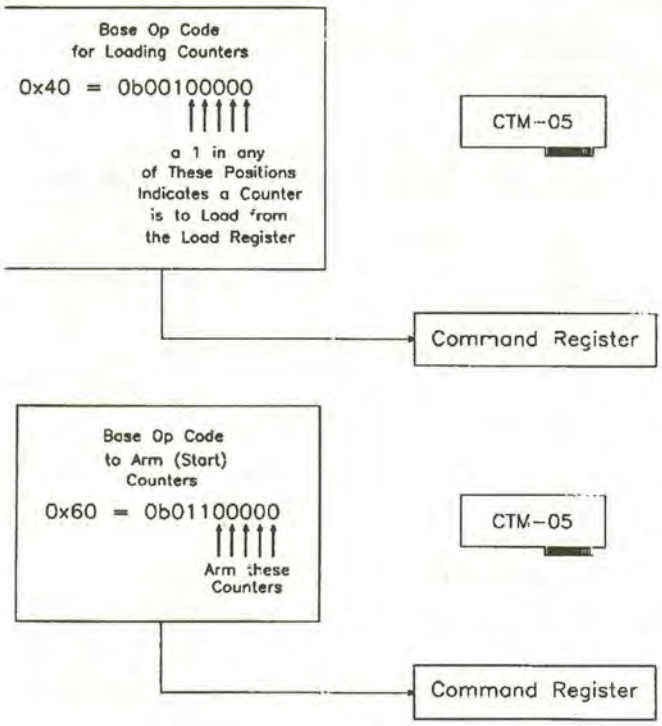


Diagram 7

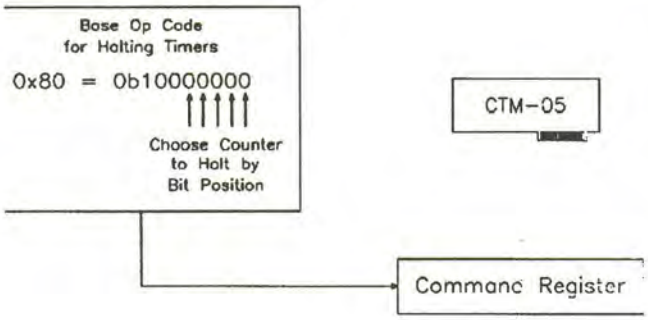


Diagram 8

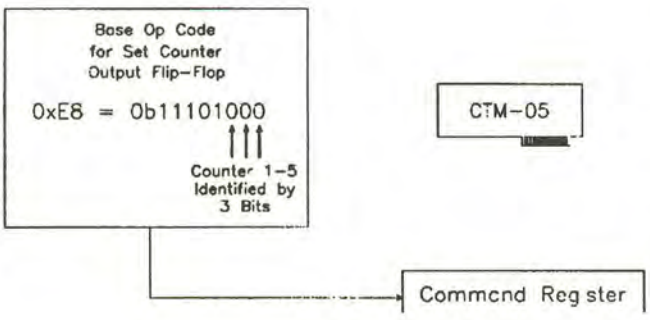


Diagram 9

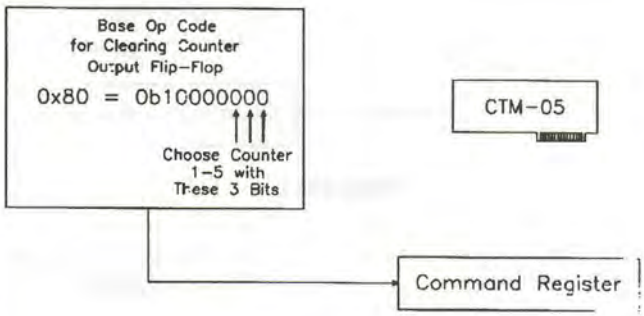


Diagram 10

arranged through the load and the hold registers. The *load register* is always used as a memory cell to contain a *starting value* for the counter/timer beginning count. Upon reaching terminal count (i.e. by overflow or underflow) the counter normally will *reload* from the *load register* and begin again.

The *hold register* has two uses. The default use for the hold register is, upon command, to copy the value currently in the working counter/timer register. By issuing an op code the hold register will access and store the current working count, then the hold register can be read to obtain the count. In this manner the counter/timer is not adversely affected by reading the current count. Another use for the hold register is similar to the load register. For certain modes of operation the counter/timer will count down (or up) for one period from the *value* in the *load register*, then the next period count down (or up) from the *value* in the *hold register*, and repeat, alternating count down (or up) from values in the load and hold registers. This allows the CTM-05, through programming, to produce pulse trains which have an asymmetrical waveform. The output of each counter/timer can be programmed with each terminal count to produce a single pulse, or toggle states. Figure 4 demonstrates using the load and hold registers to produce symmetrical and asymmetrical pulses and how output toggling can be used to produce asymmetrical waveforms.

In addition to the three registers described above counter/timers 1 and 2 each have *alarm registers* associated with them. Refer to the Am9513A manual for information on the utilization of the alarm capabilities.

The terminal count pulse of a counter is passed easily to the next higher numbered counter. This is accomplished by setting the higher numbered counter's *mode register* bits 8 through 11 to 0 (figure 4). As one might predict, counter/timer 1 can be triggered off the counter/timer 5 terminal count. A day contains 86,400 seconds. At the one megaHertz rate one days worth of clock ticks can be represented in 37 bits which is 3 counter/timers. Actually, with the 48 bits represented in the 3 counter/timers you can time almost 9 years to 1 microsecond accuracy. The point is that the CTM-05 provides an almost unlimited capability.

Programming the CTM-05

Rather than present code in one language or another, I have opted to show the logic of programming the CTM-05 using pseudo-block diagrams. The following are skeletons of a series of programs that have been developed for our needs over the past year of working with the Metrabyte CTM-05 card. For the most part, they are relatively simple, but can serve as a starting point for experimentation by those scientists and engineers who have more complex requirements. These diagrams along with the programs in the Am9513A reference manual provide a reasonable orientation to this hardware in a PC environment. It is a very useful exercise to attach an oscilloscope to the outputs of the counters of interest to actually observe changes occurring in the counter/timers. The oscilloscope should be *triggerable* and have at least a dc to 15 or 20 mHz bandwidth.

Reset

When the computer is first turned on or reset, the Am9513A is reset. In addition, the chip can be reset in software by issuing a single op code to the command register. Resetting the chip loads zeros in all of the hold, load and master mode registers and 0B00 hexadecimal in the counter mode registers. Diagram 1 is very simple in that all it really does is send the reset op code to the command register. Since this program is useful for bringing the CTM-05 back under control, it should be the first program written. A "reset" can also be the first command issued in a multicommand series to ensure that the CTM-05 is in a known state.

Reading a register

Diagram 2 looks more complex but is actually similar to resetting. Note in Figure 1 that the op codes for pointing to a register are in ascending order. Certain values are not used (e.g. 0x06) and should not be sent to the command register. Otherwise, reading the series of registers is a matter of looping through 0x01 to 0x1F. In diagram 2, automatic pointer sequencing is first turned off. Automatic pointer sequencing is a feature built into the Am9513A in which every reading of the data register automatically causes the internal register pointer to read the next register in the sequence. I have not found the automatic pointer sequencing feature of the Am9513A to be helpful. My personal preference is to issue the proper op code to point to the register of interest. Turning off pointer-sequencing is done with a single op code. Then an op code request is written to the command register and 2 bytes are read back from the data register. A program which displays all of the registers of the CTM-05 is extremely useful for debugging purposes as well as understanding how registers are read.

Writing to the counter mode, hold and load registers

The next logical sequence of programs allows writing values to the individual counter mode, hold or load registers.

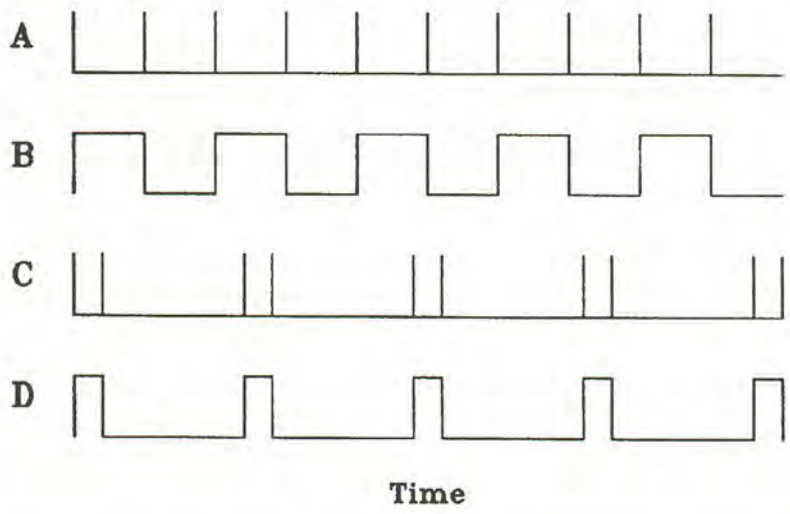


Figure 5: **A.** Symmetrical pulse stream. Bits in counter mode register are programmed to cause counter to reload from the load register and output flip flop is disabled.
B. Symmetrical waveform. Bits in counter mode register are programmed to cause counter to reload from load register and output flip flop is enabled.
C. Asymmetrical pulse stream. Bits in counter mode register programmed to cause counter to reload from load and hold registers. Output flip flop is disabled.
D. Asymmetrical waveform. Bits in counter mode register are programmed to cause counter to reload from load and hold. Output flip flop is enabled.

Diagram 3 shows how values can be entered into a counter's load register. This is a simple matter of writing the correct op code to the command register and then writing the two byte values to the data register. Note that the individual counter mode register op codes are the same as the counter number, the load register op codes are the counter number + 8 and the hold register op codes are the counter number + 16 or counter number + 24. These sorts of short cuts are not highlighted in the Technical Reference Manual but can significantly reduce the code size. By loading a register with this routine then using a register reading program, you can get a good idea of what is happening inside the CTM-05.

Diagram 4 is more complex in that when writing to the master or individual mode registers, the bit patterns have importance. This importance varies depending on the registers, master mode or individual counter/timer mode. Diagram 3 shows how to write directly to the master or counter mode register, but in a real program you should first read the mode register value. By first reading the mode register, then changing the bit patterns of the functional group, then rewriting the values, you will not change functions outside of your interest. There are no shortcuts for manipulating the mode registers and most of the code specific to the mode registers is relatively ugly.

Manipulating master mode functions

Diagram 5 demonstrates the above point by manipulating the bit in the master mode register which shifts the f_{out} scaler from BCD to binary and vice versa. Diagram 6 changes the bits which manipulate the divisor for the f_{out} clock. The implementation of these changes can be observed using an oscilloscope attached to the f_{out} pin on the 37 pin D connector of the CTM-05. In both cases the original mode register is read into memory and the necessary bits are set or cleared to change the values which are written back to the master mode register.

Any use of the CTM-05 consists of three steps: setting the mode registers, setting the load (and possibly hold) registers and finally arming the counter to begin counting. Diagram 7 shows how the op code to load then arm one or more counters is issued so counting can begin. Note that these op codes are single op codes which may affect 1 or more counters at one time.

Initializing counter output states

Each of the five counter timers has an output flip/flop which can be used to produce level outputs. These flip/flops can be initialized to 0 or 1 (In addition, the output can be programmed so that terminal count results in a pulse rather than a change in level). Diagram 9 demonstrates how manipulating a counter mode register bit "sets" the output flip flop of the requested counter to +5 volts (logic 1) and diagram 10 manipulates the counter mode register bit which "clears" the output flip flop of the requested counter to ground (logic 0). Note that these two op codes operate on a single *counter*.

Stop. Diagram 11 demonstrates how to disarm and save the requested counters into their hold registers. Once the counters are stopped the program can examine the hold registers for current counts. The stop op code can operate on one or more counters at one time.

Conclusion

The Metrabyte CTM-05 counter/timer card is a very useful solution to many timing problems. With experience, many sophisticated timing regimens can be developed in which the processor need take only a secondary role. With the proper software, the CTM-05 is invaluable for generating control timing pulses to trigger experimental equipment and for timing experimental processes. In addition, event timing can easily be accomplished by using an event to cause the count to be stored in the counter's hold register. With the CTM-05's flexibility some thought has to go into planning the software, but that planning can pay off with simpler software and more flexible hardware in the long run.

Acknowledgement

I would like to thank Dwight Werren for his help in generating the example diagrams. I'd like to also thank Glenda White and her staff for their help.

References

Abrash, M. (1989) Measuring Performance. Programmer's Journal, 7(4):20-30.

Advanced Micro Devices (1984), Am9513A/Am9513 System Timing Controller Technical Manual. 901 Thompson Place, P.O. Box 453, Sunnyvale, CA 98086, (800)538-8450.

Holub, A. (1987) The Ultimate Metronome: Writing Interrupt Service Routines in C. Dr. Dobb's Journal of Software Tools. 12(9): 106-121 & 82-96.

Hunt, W.J.(1985) The C Toolbox, Addison-Wesley Publishing Co.:Reading, MA.

Kernighan, B.W. and Ritchie, D.M. (1978), The C Programming Language. Prentice-Hall, Inc.: Englewood Cliffs, NJ.

Lancaster, D. (1974), The TTL Cookbook. Howard S. Sams:Indianapolis, IN.

Lancaster, D. (1977), The CMOS Cookbook. Howard S. Sams:Indianapolis, IN.

Pfeiffer, R.R. and Molnar, C.E. (1976) "Computer Processing of Auditory Electrophysiology data", pp 280-305, in Smith, C.A. and Vernon, J.A. (eds), Handbook of Auditory and Vestibular Research Methods, Thomas:Springfield, IL.

Swafford, B.E. (1988) "PC-based Communications Using Interrupts", Micro-Systems Journal, 4:50-54.