Many microcomputer systems use a DMA controller between the floppy disk controller and the CPU. This additional hardware may not always be necessary.

Floppy Disk Data Transfer Techniques

Trevor G. Marshall
John A. Attikiouzel

University of Western Australia

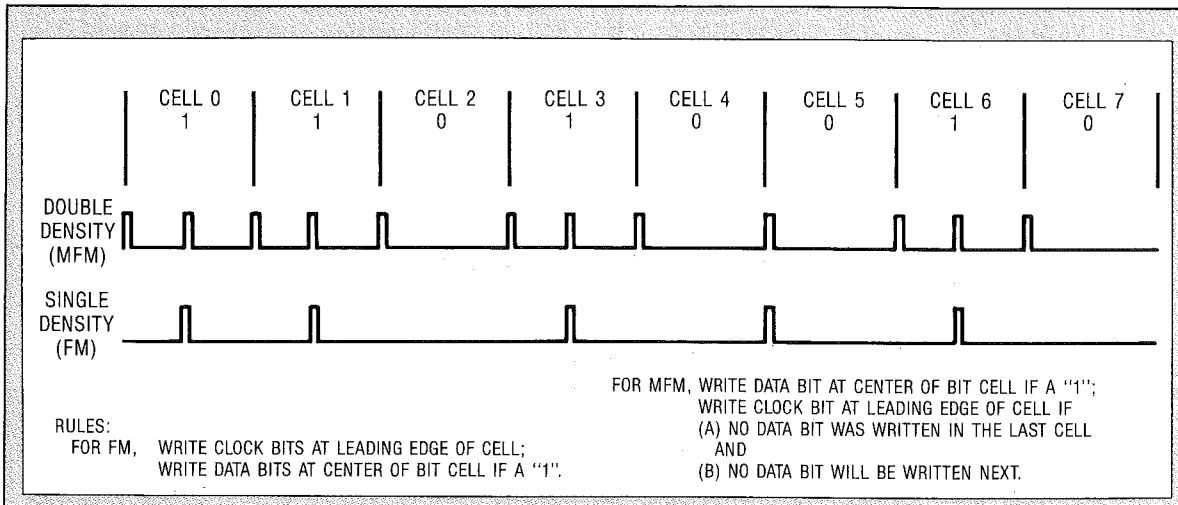
It is often assumed that a direct memory access controller increases the speed of operation of a disk operating system. This is a fallacy for most microcomputer operating systems (such as CP/M) because it presupposes that the CPU can be doing something useful while the disk I/O is being performed. Nevertheless, it is not a trivial task to achieve the required data transfer speeds with a MOS microprocessor. Here, we will examine several data transfer techniques which can be used with the Z80A microprocessor.

The task of a DMA controller, when used with a floppy disk controller (FDC), is to take a burst of data from the FDC and store it into memory at the correct address (the DMA address) requested by the disk operating system. Data are recorded on an 8-inch, double-density floppy disk at a rate of 500K bits per second, so a new byte of data is assembled by the FDC (nominally) every 16 microseconds. The rate at which the bytes are assembled from the serial disk data and presented to the DMA controller depends solely on the speed and synchronization of the disk rotation, and is asynchronous with the operation of the CPU or master system clock.

When a DMA controller is used, it must be able to access the system bus every 16 microseconds to write a new byte of data to the memory. It can do this in three ways. In the byte mode, the DMA controller transfers data one byte at a time, interrupting the processor whenever it requires memory access (every 16 microseconds). This mode is not practical at floppy disk data transfer rates, however, as the CPU needs to spend almost all the DMA interval

0272-1732/83/1200-0017$01.00

servicing and returning from the interrupt. In the continuous mode, the DMA controller takes command of the system bus for the entire DMA interval. However, the CPU is "locked out," unable to do useful computation. The most productive technique is the burst mode—when a byte is ready for transfer, the DMA controller



Representation of the byte "hex D2" in FM and MFM formats.

What is double density?

Single-density floppy disk format (defined by IBM as IBM 3740 format) calls for a flux transition (corresponding to a clock pulse) to be recorded to define each data cell. It also requires a transition inside the cell if a data 1 is to be recorded, or a lack of a transition if a data 0 is to be recorded. This is usually called the FM (frequency modulation) data format. The accompanying figure shows a typical write pulse pattern generated by a floppy disk controller. The pulses are processed by a flip-flop in the drive to produce the flux transitions on the recording medium.

Double-density format (modified frequency modulation, or MFM, format) specifies the recording of flux transitions according to not only the current data pulses but also the previous data and clock pulses. This ensures that the number of flux transitions will be reduced at a given data rate, and hence the data rate can be increased (doubled). Because each data cell is not uniquely defined by a clock pulse, an analog or digital phase-locked loop is needed to differentiate the data and clock pulses.

To further improve data recovery, "write precompensation" is employed to precompensate for the predicted "bit shift" arising from the limited medium and hardware bandwidth. The data written are shifted slightly "early" or "late" so that the transitions decoded by the read amplifier are more nearly in the center of the data cells. This shifting is most easily applied in the time domain (and in the WD279X FDCs is internally generated), although it is really compensating for medium deficiencies in the frequency domain.

How does a floppy disk controller operate?

The Western Digital WD279X can perform many operations in addition to reading/writing, but only commands which read or write data have critical timing requirements and are thus of concern to us here. These include FORMAT TRACK, READ TRACK, READ ADDRESS, READ SECTOR, and WRITE SECTOR. (Although we examine the specific case of the Western Digital WD279X series of controllers, devices from other vendors have similar interfacing requirements.)

During read operations serial data are assembled from the disk and transferred to the CPU as parallel bytes of data every 16 microseconds (in double-density, 8-inch mode). Two handshake signals are provided. Data request (DRQ) is asserted whenever a byte of data has been assembled from the disk and is available in the data register for transfer to the CPU. Interrupt request (IRQ) is asserted when an error condition has been detected or when the operation is complete. When a read has been completed, both DRQ and IRQ remain inactive until two cyclic redundancy check bytes have been assembled and data integrity has been checked. The floppy disk controller then asserts IRQ to signal the CPU that the status register can be read to verify whether the operation was successful. Write operations are similar, except that the direction of data transfer is reversed.

asserts BUSREQ* to "hold" the CPU in its current state (by requesting bus control), takes control of the system, and writes the data directly to RAM, releasing the CPU for most of the cycle (usually for about 14 of the 16 microseconds).

This protocol is extremely efficient but presupposes that the CPU has some useful operations to perform while waiting for the data from the disk. With CP/M this is not true. (Indeed, we know of no microcomputer DOS which implements true multitasking with its disk I/O.)

Although a DOS can be written so that the CPU can continue to operate usefully during disk I/O, CP/M has been written to ease hardware portability. The DOS kernel is thus quite distinct from the basic I/O system (BIOS), which actually performs the I/O. The DOS requests disk data (a logical sector of 128 bytes), and the BIOS must then place that data into memory before returning control to the kernel. Thus, the DOS kernel loses control of the system whenever a disk I/O (BIOS) request is made. Although user-supplied, interrupt-driven routines to control printers and console I/O can be written to operate during periods of disk latency, the character throughput will rarely require the services of the DMA controller.

The one exception to the above argument occurs when a disk cache buffer (see below) is implemented by the BIOS. The sector requested may indeed be validly in the track buffer while other (currently unnecessary) data are being read from the disk. In order to use the sector from the buffer, the CPU would have to continuously check whether the data currently being read include the requested sector and would then have to begin to process the data for the kernel (transferring the data from the disk cache buffer into the main memory). A time-consuming algorithm would be needed to ensure data integrity. We know of no system currently attempting such a technique.

Replacing the DMA controller

The task of replacing the DMA controller reduces to writing software that

- reads a data byte assembled by the FDC,
- stores that byte in RAM at the DMA address,
- increments the DMA address, and
- waits for the next byte to become available.

We will examine two methods of implementing such an algorithm.

Synchronization via the nonmaskable interrupt. The Z80 has a nonmaskable interrupt input. When this line is activated (edge triggered), the current instruction is completed and a restart to location 66 is executed. This is the fastest interrupt response mode on the Z80. In addition to completing the current instruction, the CPU takes five clock cycles (T states) to service the interrupt and another six to perform the stack-write operations. Response time depends on the length of the instruction being executed, but if the interrupt occurs from a HALT state (the example described below), a total of 15 T states (3.75 microseconds at a 4-MHz clock rate) can elapse before the interrupt routine gets control.

What is a disk cache buffer?

When a floppy disk is rotating (at 360 rpm), any particular sector passes the read head only once every 166 milliseconds. This gives rise to "disk latency." A sector requested by the DOS can take up to 166 milliseconds to reach the head before it can be read into RAM. Then, unless the DOS can process that data and request the next sector before it too has passed the head, another 166-millisecond wait may be necessary. To reduce this delay, an "interleave" is used when the disk is read. Instead of trying to read sectors 1, 2, and 3 sequentially, the DOS instead regards physical sectors 1, 7, and 13 as being logical sectors 1, 2, and 3. The operating program then has five sector times (lasting approximately 3 milliseconds) to process the data from the disk and request the next sector before that sector has physically rotated past the head. Thus, waiting occurs for only a partial revolution rather than for the complete 166 milliseconds.

Interleaving is only a partial solution to latency, however, as the time the user program will take to process the data from the sector must be predetermined. Programs written in high-level languages such as Basic generally take much longer than those written in assembly language. Thus, neither can operate at optimum speed and a compromise interleave factor is required. This slows down faster programs and yet may still not be adequate for those written in higher-level languages.

A disk cache buffer is used to read (typically) a whole track of data from the disk whenever even one byte of data is requested from that track. This algorithm assumes that further data from that track will be requested and that such data can be transferred from memory to memory (a fast process) rather than from disk to memory (a process incurring latency). Provided the data have been written in an orderly manner (for instance, by a DOS such as CP/M), considerable improvement in file I/O operations can be achieved (a two- to five-time improvement in access times is typical). The complete track is read sequentially in one revolution, and thus optimized disk I/O programming is required.

A system constructed by the authors, with 39,168 bytes of disk cache and double-density, double-sided, 8-inch floppy disks, benchmarked faster than microcomputers with both 5.25-inch and 14-inch Winchester disks. (The 39K was distributed as one 6528-byte read buffer, one 6528-byte write buffer, and one 6528-byte directory buffer for each of two drives.)

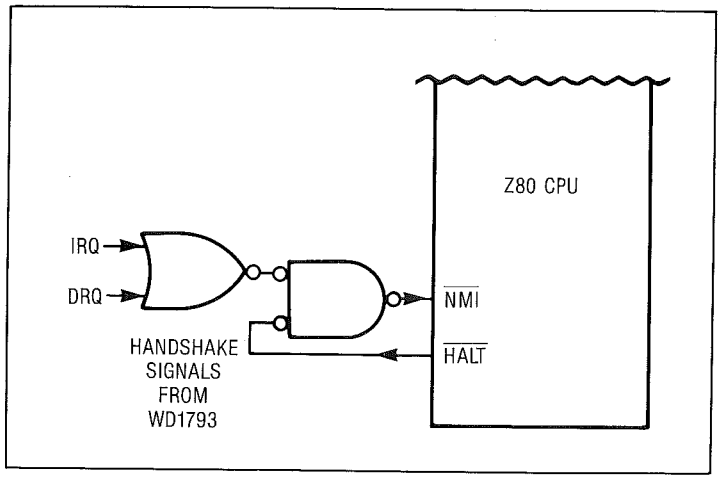


Figure 1. Hardware to implement nonmaskable interrupt synchronization.

The hardware used to implement the handshake protocol with a WD279X† floppy disk controller is shown in Figure 1. IRQ (interrupt request) and DRQ (data request) are ORed, and the result is ANDed and the HALT status output from the Z80. Thus, whenever either IRQ or DRQ is asserted by the FDC and the CPU is HALTed, an NMI* is asserted to the CPU.

Consider the Z80 assembly code shown in Figure 2. Assume that the FDC has been commanded to begin the read operation and that the HALT instruction has been executed. When the NMI* occurs (due to DRQ being asserted), a restart to the RET instruction is effected. Execution then recommences at the INI instruction. INI reads a byte of data from the I/O port addressed indirectly by the contents of register C (where the FDC data register is located), stores the byte at the memory address pointed to by HL, increments the contents of the HL pointer double register, and decrements register B, setting the zero flag when B = 0. The JP NZ instruction therefore either branches back to the HALT (and waits to receive more data) or allows the program to flow onto the routines checking the FDC error status preparatory to returning control to the DOS kernel. If B were initially loaded with 10, then 10 bytes of data would be accepted from the FDC before the read loop was exited. As disk sectors usually have lengths of 128, 256, 512, or 1024 bytes per sector, B is usually loaded so that a complete sector of data bytes is read before the read loop is exited. As B is only a single register (max count = 256), a more complex read loop is usually required. The read loop shown in Figure 3 is satisfactory for sector lengths to 1024 bytes.

Although the first three INIs decrement B, it is immediately incremented to its previous value by the INC instructions. Consequently only one decrement remains for each pass of the loop, although four bytes have been read. Thus, 256 × 4 = 1024 bytes can be processed by this loop.

The worst-case timing delay occurs from the INI at 800D to the INI at 8001, a total of 51 T states (12.75 microseconds at 4 MHz). This must be less than the 16-microsecond nominal FDC data rate. The FDC service time (byte-to-byte), however, depends not only on disk speed variation but also on the worst-case bit shifts and FDC overhead figures. Western Digital states a worst-case service time of 11.5 microseconds for write operations and 13.5 microseconds for read operations. The longer loop satisfies only the write criterion when the total loop time is considered. If it is desired, the loop can be extended further to improve the service time. (The service time is asymptotic to 11.25 microseconds.) In practice this has never been found necessary—indeed, for several years we have operated a system with a 14-microsecond service time without any lost-data errors which could be attributed to insufficient FDC service time.

If wait states are needed to accommodate slow RAM (see top opposite), the M1 wait-state generator causes six extra T states to be executed, giving a worst-case loop delay of 14.25 microseconds. This is not fast enough to meet the worst-case byte timing specifications, but it has proven adequate in practice.

If an error occurs while the sector data are only partially read, IRQ is asserted and DRQ is held inactive. This causes the remaining portion of the DMA memory to be filled with whatever data are present on the data output of the FDC. (The FDC usually retains the last byte assembled or, if a seek error has occurred, the number of the sector being sought.) Although this is undesirable, it does not cause any real problems in practice.

It is also possible to use the Z80 in Interrupt Mode 1 if the NMI* is needed for other system functions (power down, for instance).

It is usual for the disk I/O routines to save the value of any byte of data present at address 66H upon entry, replace it with the RET instruction during execution, and restore the original value when the routines are done.

Synchronization via the wait control line. The Z80 has a WAIT* input. When WAIT* is asserted, the CPU inserts additional T states into the instruction currently being executed until WAIT* goes inactive. Use of the WAIT* input is the fastest and most accurate way to synchronize CPU operation with asynchronous peripherals. The S-100 bus structure implements WAIT* via XRDY* and RDY* signals.[1] Thus, it is common to find this synchronization technique used in S-100 systems.

Figure 4 illustrates one technique for implementing the hardware-based handshake protocol. The simplest read loop that can be used with this technique is shown in Figure 5. The WAIT* line is asserted every time the FDC data port is addressed by the INI instruction, and is reset by a DRQ or IRQ from the FDC indicating that it has data ready for transfer. (However, the data may be invalid if IRQ has been asserted.) Thus, when the INI instruction is executed and the CPU tries to read the data byte from the FDC, WAIT* is asserted until the data are ready; when the data are ready, the byte is placed on the

†The WD179X series of FDCs has been superseded by the WD279X devices, which have an on-chip data separator and write precompensation. These additional facilities work well. We have found the WD279X devices to be slightly superior in data recovery. They should be preferred for new designs.

Why are wait states often needed with a Z80?

Z80 memory timing exhibits peculiarities reflecting the state of technology at the time of its design. Memory was much slower then than now, and I/O devices were slower still.

When the Z80 does an instruction fetch (an M1 cycle), it holds the address valid for 2 T states (clock cycles). With a 4-MHz clock, the RAM must have an access time of less than 500 nanoseconds (even less when delays due to buffers are taken into account). If the memory cannot operate reliably at this speed, wait states (extra T states) are necessary and may be added by hardware. When a normal memory read or write is performed, three T states are provided; hence, the wait states are usually only provided on the faster M1 (instruction fetch) machine cycles. The penalty paid for the use of wait states is a slower execution speed.

bus and the CPU recommences execution. The loop is terminated when B = 0, as before. The FDC service time of this loop is only 6.5 microseconds.

A more complex (but slower) read loop, capable of examining IRQ and DRQ separately, is shown in Figure 6. In this loop, IRQ has been made available to the CPU via a control port, which may be polled. If an error condition occurs during the read, IRQ will be asserted; when the RR is executed, carry will be set, causing a jump to the error routine.

The use of both relative and absolute jumps here reveals an important but seldom-appreciated quirk of the Z80's behavior. The execution time for a JP instruction is 10 T states whether the jump is taken or not. JR, however,

```
ADDRESS    MNEMONIC    COMMENTS

0066       RET         ;The NMI restart vector address
                       ;Control merely returns to remain routine

;
;The following initialization is performed prior to reaching the
; read loop routine (at 8000H in this example)
;
           LD C,FDC.DATA.PORT   ;Point C to the FDC data port address
           LD HL,DMA.ADDR       ;Point HL to the base DMA address
           LD B,LOOP.COUNT      ;B determines the number of read loops
;
;The read loop follows
;
8000  LP1:  HALT          ;A hypothetical base address for I/O routine
8001        INI           ;A Z80 block I/O instruction
8003        JP NZ,LP1     ;The looping control instruction
8006                      ;Various error status testing routines follow
```

Figure 2. Z80 assembly code for synchronization via the nonmaskable interrupt.

```
; (Register initialization and NMI vector setup as before)
;
8000  LP1:  HALT        ;4 + 11 + 10 T states before INI is executed
8001        INI         ;16 T states
8003        INC B       ;4 T states
;
8004        HALT
8005        INI
8007        INC B
;
8008        HALT
8009        INI
800B        INC B
;
800C        HALT
800D        INI         ;16 T states
800F        JP NZ,LP1   ;10 T states
```

Figure 3. Read loop.

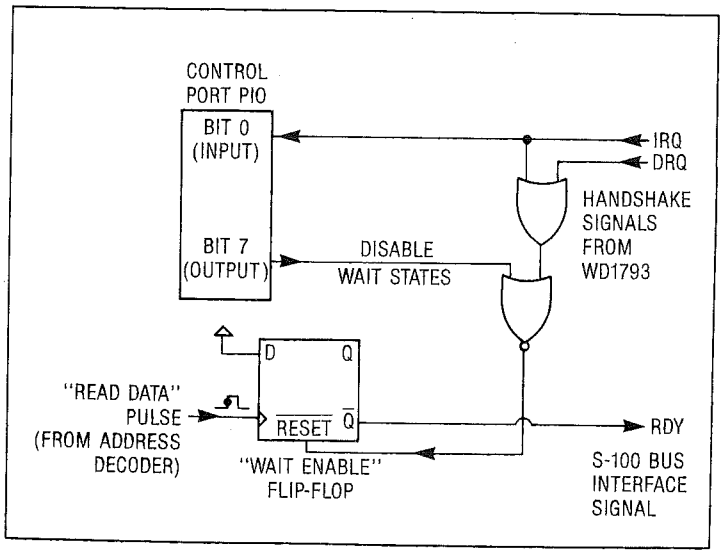


Figure 4. Handshake hardware for wait-state synchronization.

takes only 7 T states if the jump is not taken, or 12 if it is. Thus, in the above loop JR at 8003 is more efficient than JP, as the error jump is seldom taken. At 8007, however, the reverse is true and JP is more efficient. With this loop, 48 T states (12 microseconds at 4 MHz) elapse between synchronization intervals. With wait states, this increases to 13.5 microseconds.

For sectors longer than 256 bytes, read loops similar to those used with NMI* synchronization can be constructed.

Satisfactory operation can also be obtained if the wait states are triggered by a read from the status port.

Normally the wait-state circuitry is only enabled immediately before the read/write is to take place, via setting of the wait enable flip-flop. This prevents wait states from being generated during other FDC operations, when they are not required.

Choosing between the two techniques. Dynamic RAM can be refreshed by means of the Z80's inherent refresh capability. If such RAM is present in the computer system, it will not be possible to use the wait-state synchronization technique without violating the two-millisecond refresh interval of the RAM. The Z80 executes one refresh cycle with each instruction. (An exception is the HALT instruction, which causes a succession of NOPs to be executed, thus maintaining refresh.) If an instruction is "stretched" by the addition of wait states, refresh of the memory will not be performed. Worst-case waiting periods of up to one revolution (166 milliseconds) will occur under normal conditions; the waiting periods will be even longer if error conditions are present.

NMI* synchronization is more common in smaller (typically single-board) computers which use the Z80 refresh capability to minimize chip count. Wait-state synchronization is more common in larger (bus-oriented) systems which generally cannot take advantage of processor-dependent features such as an NMI* input.

One can construct write routines by simply substituting OUTI instructions for the INI instructions in the read routines. However, any hardware implementation must generate an early WAIT* signal, as the Z80 samples its WAIT* line before asserting the WR* strobe. The WRITE instruction cycle can be anticipated by the presence of an active (low) IORQ* signal and the absence of active (low) RD* or MI* status signals.

```
;(No NMI vector is required, as interrupts are not used)
;The following initialization occurs before the read loop
; is entered
;
            LD HL,DMA.ADDR   ;Same initialization values as before
            LD B,LOOP.COUNT
            LD C,FDC.DATA.PORT
            DI               ;Interrupts must be inhibited
;
LP1:        INI              ;The read loop
            JP NZ,LP1
```

Figure 5. Simplest read loop that can be used with the hardware-based handshake protocol.

```
;(Initialization same as before)
;
8000  LP1:  IN A,(CONTROL.PORT)  ;in which bit 0 has the DRQ status
8002        RR    A    ;Quicker than AND 01
8003        JR C,ERROR.ROUTINE
8005        INI
8007        JP NZ,LP1
```

Figure 6. Read loop capable of examining IRQ and DRQ separately.

Implementation examples

Intelligent disk controller. We constructed an intelligent disk controller which performs the same functions as the disk controller used in Intel MDS-800 microcomputers.† Our controller uses only a Z80, a parallel port (PIO), an EPROM, and a handful of gates to fully emulate the instruction set of the bit-slice-based Intel controller, thereby effecting a large reduction in chip count, operational complexity, and power consumption. Because we designed the controller for use in an S-100-based microcomputer, we used the wait-state synchronization technique. We did not use RAM for a stack but instead devised a novel technique employing ROM vectors for the interrupt and return stack operations.

Our controller can read a complete track sequentially without intervention by the host CPU and can operate in a manner analogous to an intelligent DMA controller. When the host CPU requests an operation via the I/O disk parameter block, the controller's Z80 acquires the bus, operates on the request, and then returns control to the host (main) CPU.

A disk cache buffer implementation. To test disk cache buffering,†† we devised a BIOS for a low-cost, single-board computer.††† We used the NMI synchronization technique. The cache technique was a modification of that used by Van Valzah.[2] By replacing 16K dynamic RAMs with 64K devices, we made an extra memory bank of 48K available. We designed a double-density disk controller (using the WD1793) to plug into the WD1771 (single-density) controller socket. One 48K bank of memory was assigned to hold the cache buffers. For each of the two drives, storage was allocated as follows:

- 6528 bytes for holding the disk directory track,
- 6528 bytes dynamically assigned to the most recently read track, and
- 6528 bytes dynamically assigned to the most recently written track.

This left approximately 9K bytes for other tasks. We experienced no difficulty in reading a track sequentially—hence, the use of a DMA controller would not have increased disk throughput.

The operational speed of the cache buffer system with disk-intensive programs was never less than three times that of a standard nonbuffered system. In a benchmark test against a system with a 14-inch Winchester disk and no cache buffering, the cache technique provided slightly faster execution. To be fair, however, we must point out that for the 14-inch disk CP/M had to search 511 directory entries for each file entry/close/extend, while the cache system had to search only 256. Indeed, the most obvious reason for the excellent performance of the cache system was the directory buffering. CP/M frequently needs to get more information about the extent of a file from the directory. Normally this requires a seek back to the directory tracks. However, if these tracks have been kept in RAM, no physical head movement is necessary.

No operational advantage can be gained from adding a DMA controller to a microcomputer using a 4-MHz Z80A and running a CP/M-based DOS. A Z80-based system that employs software DMA and cache disk buffering can perform as well as the most sophisticated 8-bit system currently available. In addition, the techniques described here can be advantageously used with other microprocessors—such as the Z8000—which possess block I/O instructions. ■

†Copies of the controller software are available in Volume 40 of the *SIG/M Users' Group Software Library,* available from SIG/M, PO Box 97, Iselin, NJ 08830, or from the authors at the address given at the end of this article. Hardware diagrams and a detailed description are also available from the authors.

††Copies of the software are available in Volume 141 of the *SIG/M Users' Group Software Library* (see preceding note for address).

†††The "Ferguson Big Board" from Digital Research Computers, Garland, Texas.

References

1. K. A. Elmquist, H. Fullmer, D. B. Gustavson, and G. Morrow, "Standard Specification for S-100 Bus Interface Devices," *Computer,* Vol. 12, No. 7, July 1979, pp. 28-52.
2. R. Van Valzah, "FAST.ASM, FASTNSTL.DOC, FMAN.PRN," Vol. 38, CP/M Users Group. (Available from the Group at 1651 Third Ave., New York, NY 10028.)

Trevor G. Marshall is an independent consultant working in California. He graduated from the University of Adelaide in 1973 and, after a year as a tutor at the University of Technology, Papua New Guinea, joined the Department of Electrical Engineering of the West Australian Institute of Technology. In 1978 he received his ME from the University of Adelaide. He then left the Institute to concentrate on consulting activities and to commence doctoral studies at the University of Western Australia. His interests range from microcomputer applications to RF design.

Marshall's address is 3423 Hill Canyon Ave., Thousand Oaks, CA 91360.

John (Yianni) Attikiouzel is a senior lecturer at the University of Western Australia. After receiving his PhD in 1972 from the University of Newcastle-upon-Tyne, he worked there until 1975 as a research officer. From 1975 to 1976 he was a lecturer at Portsmouth Polytechnic. He has published over 50 technical papers on aspects of analog and digital filtering and medical electronics. His current interests include network analysis and synthesis, approximation theory, and microprocessor applications.

Attikiouzel's address is Dept. of Electrical and Electronic Engineering, University of Western Australia, Nedlands, Western Australia 6009.