EMS 4.0 pulls together

In search of supernumerary memory for DOS applications, EMS 4.0 unites something old, something borrowed, and something new, says Ted Mirecki.

Until applications arrive that fulfill the promise of OS/2, many users are searching for alternatives that can run widely available DOS applications, yet overcome the limitations of that operating system. Expanded memory is the technology that gives such alternatives a large part of their power. The Lotus/Intel/Microsoft Expanded Memory Specification (EMS) version 4.0 unites features of earlier EMS versions with strengths of the AST Research/Quadram/Ashton-Tate Enhanced Expanded Memory Specification (EEMS), and it adds some traits of its own to create a new and improved EMS. The new version may open all kinds of doors for DOS applications.

Lotus, Intel and Microsoft originally developed EMS to relieve the spreadsheet power user's memory crunch. It was intended to provide 8Mbytes of memory beyond the 640k limit as a repository for data such as spreadsheet cells or a RAM disk. Because most programs of this type already provided their own data space management, expanded-memory management was also left at the application level. Therefore, expanded memory benefited applications only to the extent that they were written specifically to take advantage of it. Such applications and utilities soon proliferated.

However, DOS systems needed additional memory for purposes other than data storage. Programs were growing in size, and users were discovering multitasking in the form of terminate-and-stay-resident (TSR) utilities and task-switching environments that loaded more than one program at a time. Although the original EMS offered the possibility of expanding the memory space by an order of magnitude, the design of its hardware interface with the rest of the system made it inefficient for running multiple programs.

To provide for this further use of expanded memory, AST, Quadram, and Ashton-Tate developed EEMS. For applications and small TSR utilities, EEMS provides the same capabilities, in the same way as EMS. For system-level control programs that create and manage task-switching or multi-tasking environments, it efficiently expands memory available for running programs.

The most notable EEMS success is DESQview from Quarterdeck Office Systems. DESQview and Microsoft Windows run on top of DOS and rely on a hardware solution to expanded memory. Either of these, in conjunction with underlying hardware, can provide two features lacking in DOS: a large address space and multi-tasking.

Quarterdeck's windowing multi-tasking environment is much like Microsoft Windows, but it is text-based and menu-driven instead of graphics- and icon-oriented. It can simultaneously run — not just hold in memory for task switching but actually execute — as many programs as will fit in available expanded memory. By comparison, Windows versions prior to 2.0 used EMS memory for data storage only, so it could multi-task only as many programs as fit in the 640k of conventional memory space at one time.

DESQview demonstrated that expanded memory could go beyond merely providing data storage space. In recognition of this fact, Lotus, Intel, and Microsoft revised their specification extensively to create EMS 4.0. The major changes are increasing the maximum expanded memory space from 8Mbytes to 32Mbytes, incorporating the same hardware interface as EEMS, and adding multi-tasking support in software.

Part of the motivation for this revision was undoubtedly the release of Windows 2.0, which takes advantage of the multi-tasking support provided by EMS 4.0 software running on appropriate hardware. Another reason could have been the design of IBM's 80286 Memory Expanded board for the PS/2 Models 50 and 60, which supports a mapping scheme functionally equivalent to EEMS

(but for reasons other than providing expanded memory).

Something old

Understanding expanded memory requires clear definitions of the terms used in describing the memory architecture of a PC system. The 'system address space' is memory addressable by the CPU. In real mode (the only mode available to 8088 processors), the address space is 1Mbyte. In protected mode, it is 16Mbytes for 80286-based computers and 4Gbytes for 80386-based machines. 'Conventional memory' is the portion of system address space available to DOS. On most PC-compatible systems, this is limited to 640k, but this number is a consequence of hardware design, not an inherent limitation of DOS.

'Extended memory' is memory above 1Mbyte that lies within the system address space of a '286 or '386. It is accessible only after switching into protected mode (for example, under OS/2 or UNIX), whereupon it becomes part of the larger system address space. DOS cannot use this memory, but the BIOS of '286 and '386-based systems has procedures for transferring data between conventional and extended memory by temporarily switching into protected mode. This facility is of use to device drivers (such as VDISK) that implement a RAM disk in extended memory. EMS is not concerned in any way with extended memory.

'Expanded memory' resides outside system address space and therefore cannot be accessed directly in any processor mode. It is applicable in real mode only to provide memory in excess of the conventional memory limit. Expanded memory is divided into 'pages' of 16k and accessed by mapping some number of these pages into unused portions of this address space. Each 16k block of the system address space that can hold an expanded memory page is called a 'page frame'.

The implementation of expanded memory requires both hardware and software components. The hardware must allow changing of the effective address of memory under software control. This capability is built into the '386 processor, but for '286 and 8088-based systems, it must be provided by hardware on the memory board. On such boards, each page frame requires a page register whose contents determine which expanded-memory page appears at the address of that frame.

Software needed to control mapping takes the form of a device driver called the Expanded Memory Manager (EMM).

FUNCTION	DESCRIPTION
1	Get EMM status.
2	Get page-frame base address.
3	Get total number of pages and free-page count
4	Assign an EMS handle ID and allocate pages to it
5	Map a single page into a page frame
6	Close an EMS handle and deallocate all its pages
7	Get the EMM version number
8	Save the EMM status in an internal buffer
9	Restore the EMM status from the internal buffer
10-11	Reserved
12	Get the number of active EMS handles
13	Get the number of pages allocated to a specific handle
14	Get an array of page counts per handle
15	Save, restore, or swap the EMM status to/from an external buffer

Table 1 EMM functions common to all versions: the basic set of EMM functions defined by EMS 3.2 is a proper subset of the functions supported by both EEMS 3.2 and EMS 4.0. Programs using only this set of functions are compatible with all EMS hardware and software

However, programs that use expanded memory communicate with the EMM not through DOS calls or IOCTL functions (the normal means for applications to talk to device drivers) but through interrupt 67h. Typically, interrupt handlers are installed as TSR programs, not device drivers.

The EMS and EEMS implementation of the EMM as a device driver turns out to be an unfortunate choice because it makes expanded memory unusable in the DOS compatibility mode of OS/2. If the EMM were a TSR, it could be installed after switching to OS/2 real mode, and DOS applications running in the compatibility box could fully use expanded memory. But a DOS device driver, even if it is designed for real mode only, cannot be installed under OS/2. Using EMS memory in the DOS box will be possible when and if EMM drivers are rewritten to allow installation under OS/2.

Table 1 lists the basic set of functions common to all implementations of EMS.

Something borrowed

The differences introduced by EEMS (and now adopted by EMS 4.0) have to do with the number and location of page frames in the system address space. Incorporation of these features into EMS acknowledges the role that expanded memory can play in support of multitasking operating environments.

EMS 3.2 specifies four contiguous page frames above conventional memory, creating a 64k window into expanded memory. EEMS allows a page frame in every unused 16k block in the system address space, both above and within conventional memory.

In version 3.2 of each specification, EEMS is a proper superset of EMS, meaning that EEMS performs the same functions in the same way — and adds more. All software written to take advantage of EMS runs flawlessly on EEMS hardware. Little practical difference exists at the applications level, and no widely marketed applications take advantage of features specific to EEMS. These features are meant to be used by systems-level software such as multi-tasking operating environments.

To illustrate the advantages of mapping expanded memory into the conventional address space, consider a system with 256k of memory on the system board, an EGA, and a 2Mbyte expanded memory board with 2Mbyte of memory (see Fig 1). The memory board is configured to backfill conventional memory to 640k. If this board conforms to EMS 3.2, it provides a window of four page frames. By default, these are located at segments C800H, CC00H, D000H, and D400H; the user also can configure the window to begin at CC00H or D000H. In any case, two of the six available 16k blocks remain unused. The 384k used to backfill conventional memory are lost from the expanded memory pool, leaving 1664k (104 pages) of expanded memory available to the EMM.

If the memory expansion board is con-

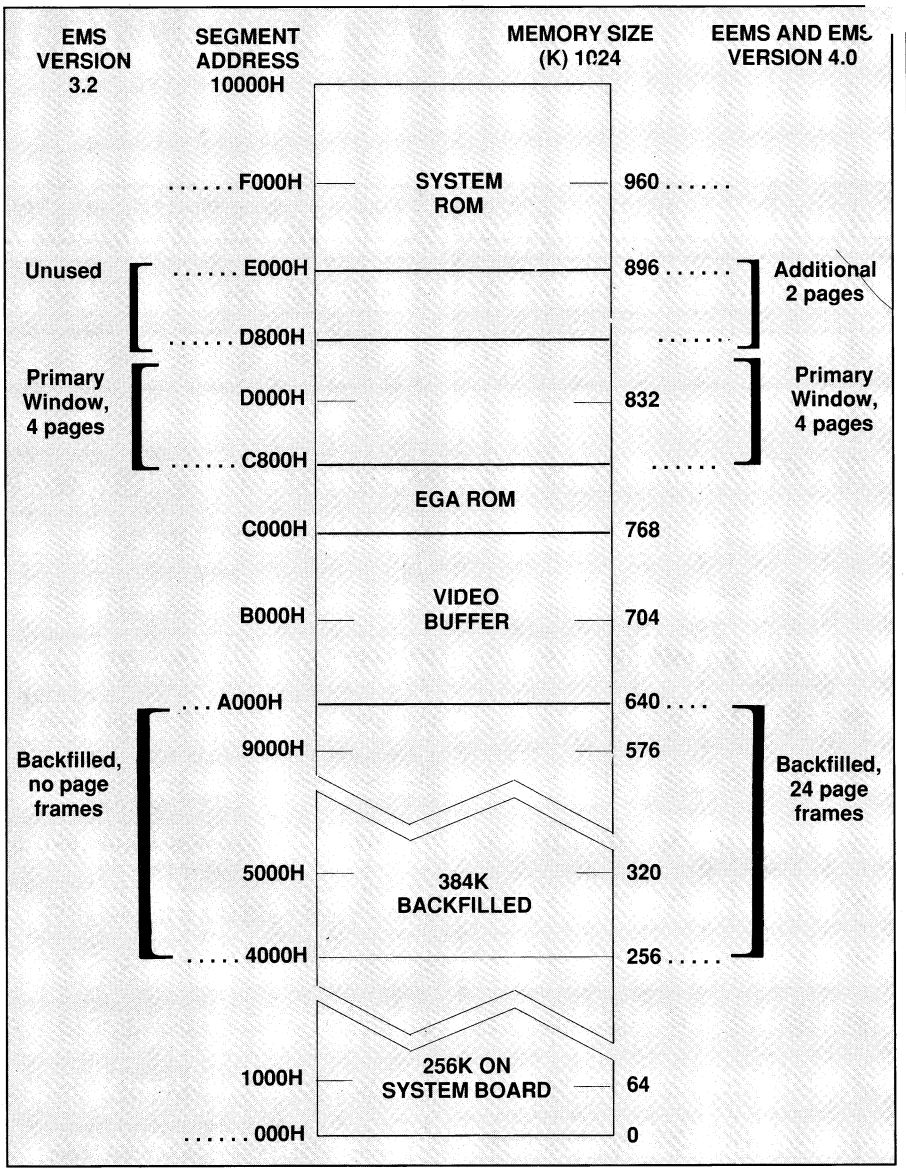


Fig 1 Example memory configuration: previous versions of EMS were limited to providing four page frames above conventional memory. Version 4.0, like previous versions of EEMS, allows a page frame in any 16k block, above or below 640k, not populated by RAM or ROM

structed to the standards of EEMS 3.2 or EMS 4.0, the primary window at segment C800H can extend over six page frames instead of four, allowing an application to manipulate 96k of expanded memory at a time instead of 64k. In addition, the backfilled memory below 640k supports page frames, allowing the mapping of expanded memory pages into conventional memory. Memory used for backfilling remains in the expanded memory pool. While the primary window is meant to be controlled by applications, only an operating system or system-level tasking environment should manage page frames within conventional memory.

If an operating environment running several co-resident applications consumes the entire conventional memory space, and the user requests start-up of yet another application, a program must be unloaded from memory to make room for the new one. In the absence of expanded memory, one of the applications must be written out to disk before a new one is loaded. With EMS memory, the operating environment could copy an application into expanded memory with this sequence:

1. Allocate a sufficient number of expanded memory pages to hold the unloaded application.
2. Map four of those pages into the window at segment C800H.

3. Copy 64k from conventional memory to the window.
4. Loop back to step 2 until the system has copied sufficient memory to hold the new application.

With EEMS's paged memory below 640k, an application in the upper 384k of conventional memory already resides in expanded memory, so it does not need to be copied out to make room for a new program. Instead, the operating system merely allocates pages for the new application and maps them into page frames in conventional memory. Pages formerly in these frames become inaccessible but retain their contents. This process requires the same number of calls to the EMM (one for each page of the new application) but avoids copying each byte of data through the primary window.

The advantage of EEMS and EMS 4.0 over EMS 3.2 is even greater when the system needs to switch between two previously loaded applications — one in conventional memory, the other in expanded. With previous EMS versions, the system must perform two copies: one to roll out the application being suspended, another to roll in the application being activated. With EEMS and EMS 4.0, it performs both functions merely by mapping the pages holding the incoming application. The pages holding the suspended application are mapped out but not overwritten.

An EEMS or EMS 4.0 board can contain two sets of page registers, with each set containing mapping information for a distinct set of expanded memory pages. By specifying one or the other as the active register set, an entire set of pages can be mapped in with one call to the EMM.

Another advantage of EEMS and EMS 4.0 is that they provide more memory for simultaneously holding applications. With EMS 3.2, the example configuration provides 104 pages of expanded memory for holding inactive applications; active applications are replicated in the conventional memory space. With EEMS and EMS 4.0, 128 pages are available because an active application does not appear twice.

Something new

Besides incorporating EEMS features, EMS 4.0 adds others that provide even more support for multi-tasking. These features are described below in functional groups; EMM functions that implement them are listed in Table 2.

Named handles. The set of expanded memory pages a program obtains by one call to the EMM is identified by a number called the 'EMM handle'. To map a page into the system address space, the program identifies it by specifying the handle number and a logical page number ranging from zero to one less than the number of pages allocated to that handle. Function 20 allows the assignment of an eight-character name of the user's choice to any handle. The EMM keeps a directory of handle names, and function 21 searches this directory and returns the handle number for a given name. This feature can be used for communication between co-operating programs.

For example, a database program could allocate some pages and store data in them. If the program gives a name to the handle, another program that knows the name (say a word processor or spreadsheet) can find out from the EMM the handle number associated with the name. Knowing the handle, the second program can access the data stored in these pages. The handle number cannot be hard-coded into programs because this number can change at every execution, depending on the order in which various co-resident programs are loaded.

Physical page addressing. In previous EMS versions, page frames in the system address space are identified by frame number. For the configuration shown in Fig 1, the four frames in the primary window are numbered from 0 for the frame at segment C800H, to 3 for the one at D400H. In the EEMS implementation, the first four frames are numbered the same way; frames at D800H and DC00H are designated as 4 and 5, respectively, and the ones beginning at 4000H in conventional memory are numbered from 6 upward. When mapping an expanded memory page, a program specifies its location in the system address space by supplying the frame number.

For compatibility, version 4.0 accepts the same frame numbering, but function 17 also allows specifying the location of the frame by its physical segment address. When calling this function to map a page into the lowest frame in conventional memory, a program could pass either 4000H or 6 as the frame argument. A subfunction code specifies the argument type.

Function 25 constructs an array of all page-frame addresses in the system; it is similar, but not identical, to EEMS function 41. Entries in the array returned by function 25 are ordered by physical address and consist of two words: the segment address and the frame number. Function 41 returns an array of bytes containing the high-order six bits of each frame's address; entries are ordered by frame number, with the address for frame *N* in the *Nth* byte of the array. The new method is more convenient for physical addressing of page frames, the old one for addressing them by number.

Raw page sizes. The division of expanded memory into pages of 16k is not

FUNCTION	DESCRIPTION
16	Save/Restore partial page map
17	Map/Unmap multiple pages into page frames
18	Change number of pages allocated to a handle
19	Get/Set volatility attribute of a handle
20	Get/Set handle name
21	Get handle number for handle name
22	Alter page map and jump to a far address
23	Alter page map, call a far address, then restore page map
24	Move/Exchange contents of memory block
25	Get array of page-frame addresses
26	Get expanded memory hardware information
27	Allocate pages in sizes other than 16k blocks
28	Control alternate mapping registers
29	Prepare expanded memory hardware for warm reboot.
30	Disable/Enable system-level EMM functions

Table 2 EMM functions added in EMS 4.0: the new EMS 4.0 functions that support the hardware features defined earlier by EEMS are inoperative on existing EMS boards. All EMS boards, however, can benefit from the functions that add software support for multi-tasking control programs

convenient when the hardware implements a paging scheme with a different page size. For example, the '386 processor performs paging in increments of 4k. In version 4.0, an EMM can be written to allocate expanded memory in units that are convenient to the hardware, provided that the size of such a unit, called a 'raw page', is a submultiple of 16k. Function 27 creates a raw handle and allocates to it a specified number of raw pages. All subsequent page operations for this handle, such as mapping pages or changing the allocation, must be specified in raw pages.

Multiple register sets. EEMS memory boards are constructed with two sets of page-mapping registers, so that a multitasking operating environment can switch rapidly between mapping contexts for two applications. Version 4.0 extends this to allow memory boards to provide any number of register sets. For boards with a single set of registers, function 28 simulates multiple sets by storing inactive mapping contexts in memory provided by the calling program. The effect of the simulation is the same as if the caller saved and restored contexts with function 15, but the advantage is that the context-switching method appropriate to the capabilities of the hardware resides in the EMM; the calling program need not incorporate logic for both switching register sets and saving contexts.

An even more significant extension is the capability to dedicate specific register sets for use by the direct memory access (DMA) controller for high-speed transfers to and from expanded memory. A program can allocate a set of mapping registers for use by DMA, initiate a DMA request to or from the memory mapped by that set, and then switch the context by means of function 15 or 28. The DMA process continues to use the memory mapped by the DMA set of registers, while the CPU can access a different set of pages mapped by a different register set.

Version 4.0 does not require that EMS boards provide hardware for multiple register sets for either CPU or DMA access; it merely specifies that the EMM can take advantage of this capability if available. In contrast, an EEMS memory board must have exactly two sets of mapping registers for the CPU to use; a version 4.0 EMM also can use these registers. No boards currently on the market support any DMA registers or more than two sets of CPU registers. Function 26 provides information about the EMS hardware: the raw page size, the number of alternative register sets (after the first), and the number of DMA register set.

Manipulating register sets should be done only by the operating system or task-switching executive. The system program can use function 30 to disable or enable functions 26 (get EMS hardware information), 28 (manipulate alternate register sets), and 30. Once disabled, these functions return error codes to all callers, even the system.

When function 30 is disabled, how does the system re-enable it? Immediately after installation of the EMM, the system-level functions are enabled by default. The first call to function 30 returns a random 32-bit access key that must be used on all subsequent calls to the function. Thereafter, function 30 can be called, even when disabled, if the key value is given as a parameter in the call. During its installation, the operating environment is presumably the first program to call function 30, so it is the only one to obtain the key.

Control and data transfers. In a multitasking environment, transfers of control

to various programs often involve changes in the mapping context. For example, when a task-switching executive activates a dormant program, it must map that program's expanded memory into the system address space before branching to the program's code. This capability is also useful within an application — for example, when a small TSR kernel in conventional memory maps in and branches to its main code in expanded memory.

Function 22 automates establishing a new mapping context and jumping to a far address. Logically, it is equivalent to a call to function 27 (map multiple pages), followed by a far jump to an absolute address. The target can be in expanded or conventional memory. As is the case for function 27, the page frames in the new mapping context can be specified either as frame numbers or absolute segment addresses.

Function 23 performs the equivalent of a far call. It saves the current mapping context in a caller-supplied memory area, establishes a new context (in the same way as function 27), and transfers control to a far address. When a far return is subsequently executed, the EMM restores the saved context before returning control to the instruction following the call to function 23.

Function 24 provides a means of efficiently moving or exchanging large blocks of data between expanded and conventional memory or between two areas of expanded memory. In the latter case, the two blocks might belong to different handles. Blocks need not be aligned on page or segment boundaries. Overlapping source and destination blocks are handled properly for a move but generate an error for an exchange. This function is useful for moving data between blocks that span several pages, especially when the blocks are larger than any available contiguous window. The calling program need not save and restore the context before calling this function, nor provide a save area for the EMM to use.

Nonvolatile handles. An EMS board can provide hardware features to maintain the contents of expanded memory through a warm boot created by a keyboard reset or other event that can be trapped by software. Function 19 applies the 'nonvolatile' attribute to a handle; only the data in pages belonging to nonvolatile handles survive a reboot.

Function 29 called just prior to reboot saves data on the EMS board; its specific action is not specified in EMS documentation because that depends on hardware implementation. Typically, function 29 writes information into EMS hardware registers that notifies the EMM driver (when it subsequently reinstalls itself) which data are nonvolatile and therefore are not to be erased during initialisation.

Calling function 29 requires the detection of an imminent reboot, and this is not possible in all cases. A replacement interrupt 9 handler can detect a keyboard reset, but two other events can cause a reboot. One is to execute a direct jump to the boot code in ROM, the other is to toggle the processor's reset pin with a hardware reset button (this generates a hardware branch to the boot location). Software cannot detect either event, and the boot code, being in ROM, cannot be hooked by overwriting its entry point. Therefore, a keyboard reset is the only event that can be expected to preserve nonvolatile data.

Efficiency improvements. Function 16, 17, and 18 do not add major functionality to the EMM, but they provide services in a more efficient or convenient manner than is available by other means. Function 16 (save partial mapping context) is similar to 15 (save, restore, or swap EMM status to/from an external buffer), but it saves or restores only a specified subset of page-mapping registers, thereby consuming less conventional memory by storing only the essential portion of any context-switching information.

Function 17 (besides supporting physical frame addressing) can map more than one page at a time. It accepts an array of page numbers and page frames, so pages to be mapped need not be consecutive, nor page frames contiguous. As a result, a single call to function 17 can replace many calls to function 5.

Function 18 allows changing the number of pages allocated to a given handle. In previous versions, the page count per handle is fixed; if a process needs more pages, it has to obtain another handle. Handles are a limited resource (the absolute upper limit is 256 and the EMM typically defaults to a lower limit at installation), and obtaining and releasing them involves some overhead.

EEMS 3.2 FUNCTION, SUBFUNCTION	CLOSEST EMS 4.0 EQUIVALENT	DESCRIPTION
33	None	Get addresses of page frames outside of conventional memory
34	None	Generic accelerator card support
41	25	Get addresses of all page frames in system
42	5	Map a page into any frame
43, 0	16, 0	Save partial page map
43, 1	16, 1	Restore partial page map
43, 2	None	Save and restore partial page map
43, 3	16, 2	Get size of save array
43, 4, 5	28, 1, 2	Switch to another set of map registers
43, 6	18	Deallocate pages mapped at initialisation to frames in conventional memory

Table 3 Support for EEMS functions: version 4.0 of EEMS incorporates most of the enhancements introduced by EEMS version 3.2. Although these services are functionally similar in both versions, they require different calling sequences and produce output in a different format

Still an individual

Although version 4.0 brings to EMS the major features of EEMS and adds others, it is not a proper superset of EEMS because it does not support all EEMS features (see Table 3). The missing ones do not diminish the utility of EMS 4.0; they merely indicate different design philosophies.

One of the primary design goals of EEMS is to maintain compatibility with EMS. Therefore, most extensions are implemented by new EMM function calls, not extensions to EMS-defined functions. The standard set of EMM function calls, shared with EMS (see Table 1), handles page frames only above conventional memory. The difference is that EEMS can handle more than four page frames in this area.

All other enhancements are supported by a set of functions with distinct numbers. EEMS deals with two sets of page frames: one containing only those above conventional memory, the other containing all frames in the address space. Each set has a distinct numbering scheme and separate mapping function.

The advantage of this EEMS approach is greater integrity. The upper page frames are for use by applications, the

lower by the operating system. Each program has its own set of functions, and although an application is not prevented from accessing any frame, mapping lower memory requires the conscious effort of a different function call. By contrast, version 4.0 considers all the page frames as one set — it has no EMM service equivalent to function 33, which returns an array of addresses of only those page frames above conventional memory. Both applications and the system use the same functions (5 and 27) to map upper and lower memory, so an errant program can inadvertently map out a page in conventional memory by supplying a wrong frame number or address.

Although EEMS provides slightly better security than EMS 4.0, all versions of EMS are generally weak in memory protection. This reflects the original purpose of EMS: expanding data storage memory in a single-tasking environment. A program is in no way prevented from accessing expanded memory pages belonging to another program. Access requires only a numeric handle, and any program can obtain a list of active handles. Handle numbers are assigned consecutively from a small range (0 through 255), so a failing program can inadvertently specify a handle belonging to another process. This is especially likely in the case of handle zero, which is owned by the operating system and consists of all pages initially mapped into the backfilled area of conventional memory.

Another feature introduced by EEMS that is not fully incorporated into EMS 4.0 is the interface with caching accelerator boards. Changing the contents of memory by remapping can invalidate the contents of an on-board cache that holds data from the mapped-out location. The caching hardware is unaware of the change because mapping changes the contents of memory by writing to an I/O port, not by direct access to affected locations.

EEMS provides function 34 for communicating with the accelerator board. Support for this function must be provided in both hardware and software by the manufacturer of the accelerator; details are available on request from AST Research. EMS 4.0 does not describe this hardware function, but in the section on EMM implementation guidelines, the documentation states, 'the support of function 34, as defined by AST, is encouraged'.

Just the beginning

Version 4.0 EMM drivers are now available for existing expanded memory boards of both the EMS and EEMS variety. Owners should contact the manufacturers of their boards for information on obtaining updated software.

For this article, four different EMM drivers were tested on five memory boards: Intel's for the Above Board AT, AST's for the RAMpage and RAMpage AT, Quarterdeck's for the IBM 80236 Memory Expansion board for PS/2 Models 50 and 60, and Intel's for the Above Board 2 for Models 50 and 60. The boards for the PC and AT were older models originally supplied with version 3.2 drivers. Version 4.0 drivers for the PC and AT ran only with the boards from their manufacturers. The Intel PS/2 driver refused to install itself without an Above Board 2 but could combine memory from the Above Board 2 with memory from an IBM expansion board. The Quarterdeck PS/2 driver worked with either Intel or IBM memory boards or a combination.

All drivers worked well in the tests. The AST and Quarterdeck software supports functions of both the new version 4.0 and older EEMS version 3.2, giving them full functionality with current versions of both DESQview and Windows. Intel's drivers implement only 4.0, so they do not fully support DESQview (which predates version 4.0 and thus issues only EEMS 3.2 calls). The Above Board 2 (but not Above Board models for the PC and AT) provides equivalent context-switching support for Windows 2.03.

Even with an old board, running a new driver provides many innovations, such as named handles, physical addressing, transfer control, and other improvements in efficiency of managing paged memory. These improvements will lie dormant until programs are written to take advantage of them. However, these software features are not 4.0's primary advantages.

Achieving a spectacular improvement in the performance of DESQview or Windows requires hardware support for mapping conventional memory and multiple register sets; these are not implemented on current EMS boards. Version 4.0 does very little for existing EMS boards because they cannot use the most significant new features. For the most part, the effect of installing a new EMM driver on an old board is that the error response to version 4.0 functions is 'not implemented' instead of the old driver's 'invalid function'.

On the other hand, EEMS boards already have the hardware to support the EMS 4.0 mapping protocol. The only two innovations not already supported by EEMS hardware are more than two sets of general mapping registers and mapping registers that are dedicated to DMA transfers. The EEMS 3.2 and EMS 4.0 drivers for EEMS boards provide the same functionality with one immediate exception: with the older driver, Windows 2.0x cannot take advantage of the multitasking hardware features of the board.

For those who already own expanded memory boards, EMS 4.0 does little — EMS boards cannot use the major new features, and EEMS boards already have them. Still, the creation of a single unified standard that incorporates and adds to the features of the better one is undoubtedly a step in the right direction. This unification is just in time to provide a consistent memory standard for IBM's PS/2 series — the memory-mapping capability of the Micro Channel memory expansion board no doubt played heavily in the decision to create 4.0.

The capabilities offered by the newly unified EMS to task-switching environments breathes new life into DOS-based applications, providing the opportunity to design characteristics not otherwise available in this operating system: large memory spaces for both programs and data, rapid switching between co-resident programs, and concurrent execution.

END

Christensen Protocol

The Christensen file transfer protocol (also called Modem and Xmodem) has been around for quite a few years now and like many things from the past, it's veiled in mystery and misunderstood by many.

The protocol sends data in 128 byte blocks within sequentially numbered packets. Each packet includes a packet header, the data, and a checksum of the packet (excluding the checksum itself which is sent as a separate byte after the 128 byte block). As the receiving computer stores the incoming data in its memory, it performs its own checksum of the packet, then compares it with that in the packet. A successful comparison means that all of the information was sent correctly and the receiving computer acknowledges this by sending an ACK (ASCII for Acknowledge Hex 06) to the sending computer. If the comparison fails, the receiving computer tells the sender to resend the packet by sending a NAK (ASCII for Negative Acknowledge Hex 15).

When a file transfer is initiated there is a possible problem in that the sending computer will wait for the receiving computer and vice versa, which would mean that you get nowhere at all! What happens is that the receiving computer sends a NAK to the sender to initiate the transfer; if that doesn't happen, the sending computer starts transmitting the first block after 10 seconds. This consists of a SOH (ASCII for Start of Heading, Hex 01), the block number, and the two's complement of the block number, then the 128 bytes of data followed by the checksum (132 bytes all up) — see Figure a).

This process continues with the next packet of information. What happens when the receiving computer sends a NAK? Simply, the sender transmits it again (just like when someone says 'Beg your pardon?' when you're talking — see Figure b). What happens if the sending computer doesn't 'see' the ACK/NAK that the receiver sends? Well, after 10 seconds it will resend the packet; the receiving computer knows it already has that packet and just sends an ACK when the packet ends — all fixed!

How does the receiver know what packet it's getting? The block number is incremented by one for each packet sent and the receiver also adds the SOH, Block Number, and the two's complement of the block number and it should get zero (for example 01 [SOH] + 01 + FE is 00) — so there is a double check of the block number, in addition to the check on the whole packet.

Normal completion of a file transfer is when the sending computer transmits an EOT (ASCII for End of Transmission Hex 04), the receiving computer then sends a confirmatory ACK — see Figure c), if the EOT is lost then the receiver will continue to wait for the next block, sending NAKs every 10 seconds; after 10 NAKs, it'll give up and time out.

One thing that is often overlooked is that the receiving computer must be capable of receiving the entire packet at the full speed of the communication link; if it can't, this can cause problems at high speeds. □

```
Receiving Computer
----/NAK/----------------------/ACK/------------------------
     15H                          06H


Sending Computer
-------/SOH/BLK#/BLK2/DATA/CSUM/---/SOH/BLK#/BLK2/DATA/.....
        01H 01H  FEH  .... ddH       01H 02H  FDH  ....
```

Figure a) *Packet Organisation and Block Numbering.*

```
Receiving Computer
-----/ACK/----------------------/NAK/------------------------
      06H                          15H


Sending Computer
CSUM/---/SOH/BLK#/BLK2/DATA/CSUM/---/SOH/BLK#/BLK2/DATA/.....
        01H 03H  FCH  .... ddH       01H 03H  FCH  ....
```

Figure b) *'Beg your pardon?'*

```
Receiving Computer
-----/ACK/---/ACK/ Back to command prompt
      06H     06H


Sending Computer
CSUM/---/EOT/---/ Back to command prompt
        04H
```

Figure c) *End of file sequence.*

NEW SOUTH WALES

NSW Contact:
Prophet TBBS (02) 6285222
FidoNet [712/606]

*** AMENDED ***

ABCOM
Sysop: Ben Sharif
FIDOnet: [712/304]
Phone: (047) 364165
Baud: V21.V22.V22bis.V23
Access: Mem.VA
Computer: IBM XT
DOS: PCDOS
BBSoftware: Fido

ACE (NSW) BBS
Sysop: Larry O'Keefe
Phone: (02) 5292059
Baud: V21
Access: Mem.Reg.LVA
Computer: Atari
DOS: Atari
BBSoftware: Michtron

*** AMENDED ***

Alpha Juno BBS
Sysop: Kevin Withnall
FIDOnet: [620/701]
Phone: (02) 7741543
Baud: V21.V22.V22bis
Access: Mem.VA
Computer: Amstrad PC1512
DOS: MSDOS 3.2

*** AMENDED ***

Amstrad ABBS
Sysop: Riccay Schmahl
FIDOnet: [711/903]
Phone: (02) 9812966
Baud: V21.V22.V22bis.V23
Access: Reg.VA
Computer: Amstrad PC1512
DOS: MSDOS 3.2
BBSoftware: Opus

Apple Users Group BBS
Sysop: Matthew Barnes & Andrew Riley
Phone: (02) 4987084
Baud: V21.V22.V22bis.V23
Access: Mem.VA

*** AMENDED ***

ArcoTel BBS
Sysop: Alex Szx
FIDOnet: [620/601]
Phone: (02) 6833956
Baud: V21.V22.V22bis.V23
Access: Mem.VA
BBSoftware: Opus

*** AMENDED ***

AUGUR TBBS
Sysop: Mark James
Phone: (02) 6614739
Baud: V21.V22.V22bis.V23
Access: Reg.VA
Computer: PC Clone
DOS: PC
BBSoftware: TBBS 2.0m

Ausborne (Osborne) RCPM
Sysop: Daniel Moran
Phone: (02) 4397072
Access: Mem.VA

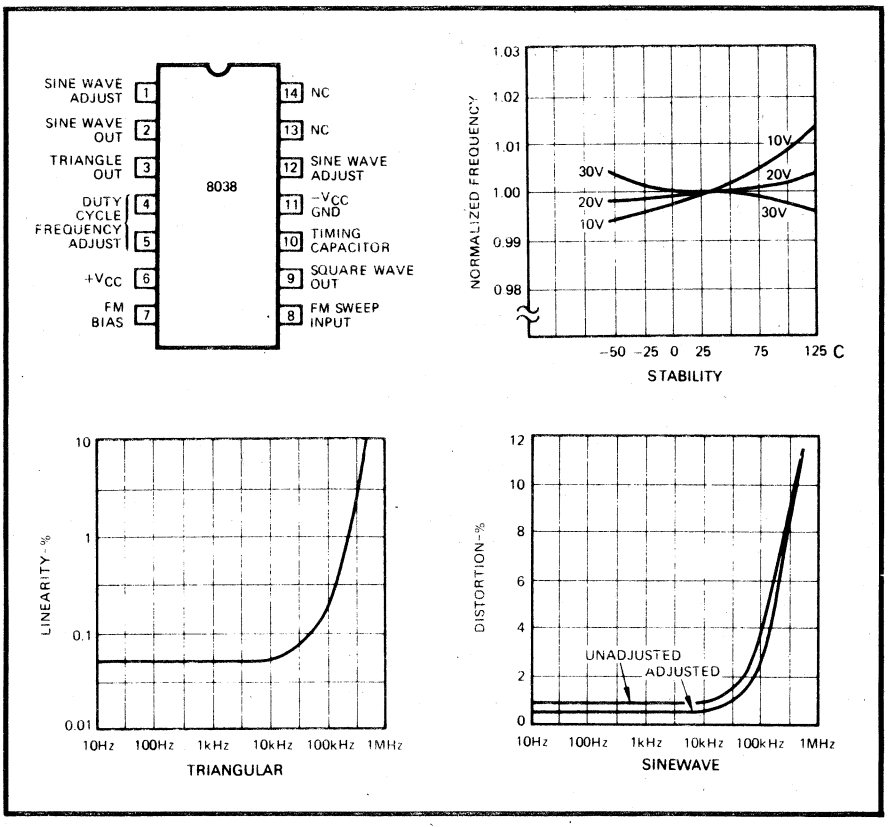
Australian Pick User's BBS
Sysop: Kurt Johannessen
Phone: (02) 6318603
Baud: V21.V22.V22bis.V23
Access: Reg.VA

BeeHive BBS
Sysop: Paul Pinches
Phone: (02) 5205181
Baud: V21.V22
Access: Mem.Reg

ETI data sheet

INTERSIL ICL 8038
Waveform Generator/V.C.O.

The 8038 has been around for about 5 years — which is a long time in electronics. In fact it has reached the position of becoming an 'Industry Standard' on a par with the 741. An inherently versatile device it has its drawbacks like most chips — but overall has a lot going for it. Intersil even produced a very honest application bulletin (A013) called 'Everything you always wanted to know about the 8038', which explained how to get the best out of this device and admitted its defects — an uncommon event with most manufacturers! Some of the data from A013 has been included in this data sheet, but for more information ask for application bulletins A012, A013, and the latest information sheet. Intersil are distributed in Australia by R & D Electronics Pty. Ltd., 23 Burwood Rd., Burwood, 3125.



Description

The 8038 Waveform Generator is a monolithic integrated circuit, capable of producing sine, square, triangular, sawtooth and pulse waveforms of high accuracy. The frequency (or repetition rate) can be selected externally over a range of less than 1/1000 Hz to more than 1 MHz and is highly stable over a wide temperature and supply voltage range. Frequency modulation and sweeping can be accomplished with an external voltage and the frequency can be programmed digitally through the use of either resistors or capacitors. The Waveform Generator utilizes advanced monolithic technology, such as thin film resistors and Schottky-barrier diodes.

Theory of operation

A block-diagram of the waveform generator is shown in Figure 1. An external capacitor C is charged and discharged by two current sources. Current source #2 is switched on and off by a flip-flop, while current source #1 is on continuously. Assuming that the flip-flop is in a state such that current source #2 is off, then the capacitor is charged with a current I. Thus the voltage across the capacitor rises linearly with time. When this voltage reaches the level of comparator #1 (set at 2/3 of the supply voltage), the flip-flop is triggered, changes states, and releases current source. #2. This current source normally carries a current 2I, thus the capacitor is discharged with a net-current I and the voltage across it drops linearly with time. When it has reached the level of comparator #2 (set at 1/3 of the supply voltage), the flip-flop is triggered into its original state and the cycle starts anew.

Four waveforms are readily obtainable from this basic generator circuit. With the current sources set at I and 2I respectively, the charge and discharge times are equal.

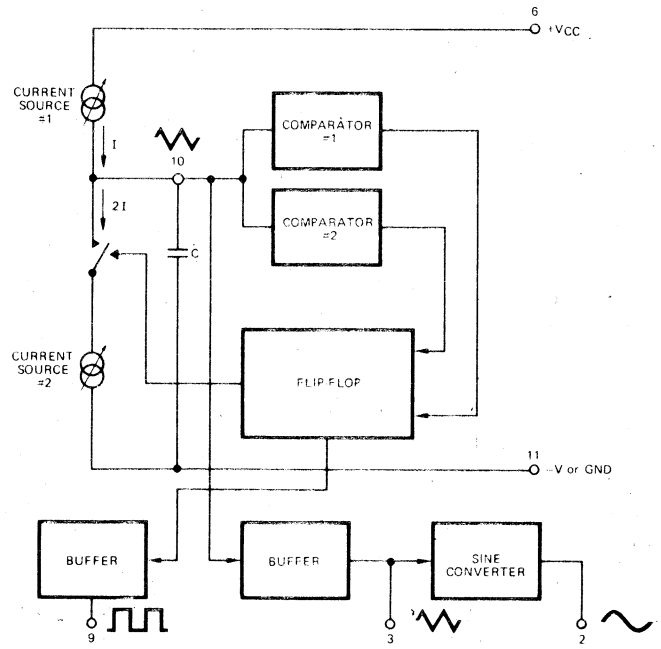


fig 1. BLOCK-DIAGRAM OF WAVEFORM GENERATOR.

Thus a triangle waveform is created across the capacitor and the flip-flop produces a square-wave. Both waveforms are fed to buffer stages and are available at pins 3 and 9.

The levels of the current sources can, however, be selected over a wide range with two external resistors. Therefore, with the two currents set at values different from I and 2I, an asymmetrical sawtooth appears at terminal 3 and pulses with a duty cycle from less than 1% to greater than 99% are available at terminal 9.

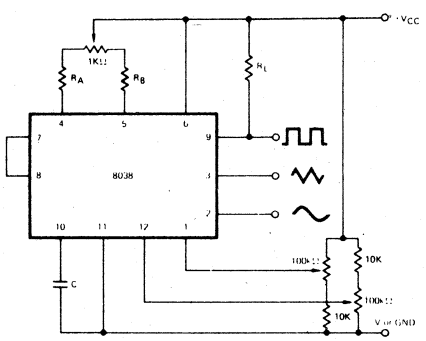
The sine-wave is created by feeding the triangle-wave into a non-linear network (sine-converter). This network provides a decreasing shunt-impedance as the potential of the triangle moves toward the two extremes.

Power Supply

The waveform generator can be operated either from a single power supply (10 to 30 Volts) or a dual power supply (±5 to ±15 Volts). With a single power supply the average levels of the triangle and sine-wave are at exactly one-half of the supply voltage, while the square wave alternates between +V and ground. A split power supply has the advantage that all waveforms move symmetrically about ground.

Also notice that the square wave output is not committed. The load resistor can be connected to a different power supply, as long as the applied voltage remains within the breakdown capability of the waveform generator (30 V). In this way, for example, the square-wave output be made TTL compatible (load resistor connected to +5 Volts) while the waveform generator itself is powered from a much higher voltage.

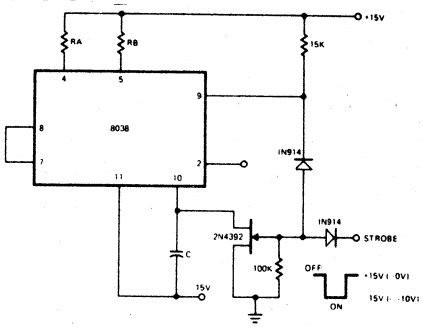
Purity



The symmetry of all waveforms can be adjusted with the external timing resistors. To minimize sine-wave distortion the resistors between pins 11 and 12 are best made variable ones. With this arrangement distortion of less than 1% is achievable. To reduce this even further, two potentiometers can be connected as shown. This configuration allows a reduction of sinewave distortion close to 0.5%.

Both the sine-wave and triangular outputs, are only useful up to about 20kHz if a reasonably pure signal is required. A perusal of the graphs will show why.

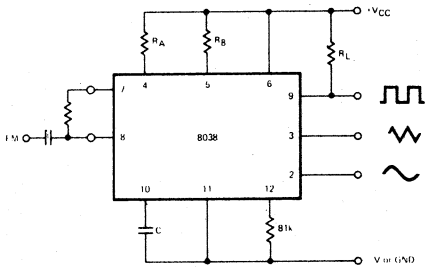
Strobe



With a dual supply voltage (e.g., ±15V) the external capacitor (pin 10) can be shorted to ground so that the sine wave and triangle wave always begin at a zero crossing point. Random switching has a 50/50 chance of starting on a positive or negative slope. A simple AND gate using pin 9 will allow the strobe to act only on one slope or the other.

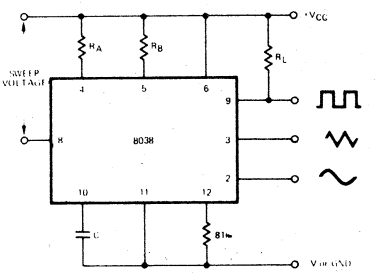
Using only a single supply, the capacitor (pin 10) can be switched either to V+ or ground to force the comparator to set in either the charge or discharge mode. The disadvantage of this technique is that the beginning cycle of the next burst will be 30% longer than the normal cycle.

F.M. and Sweeping



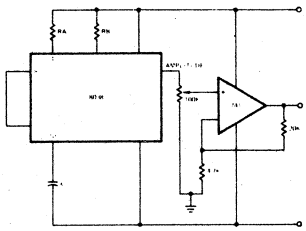
The frequency of the waveform generator is a direct function of the DC voltage at terminal 8 (measured from +VCC). Thus by altering this voltage, frequency modulation is achieved.

For small deviations (i.e. ±10%) the modulating signal can be applied directly to pin 8, merely providing dc decoupling with a capacitor. An external resistor between pins 7 and 8 is not necessary, but it can be used to increase input impedance. Without it (i.e. terminals 7 and 8 connected together), the input impedance is 8k, with it, this impedance increases to (R+8k).



For larger FM deviations or for frequency sweeping, the modulating signal is applied between the positive supply voltage and pin 8. In this way the entire bias for the current sources is created by the modulating signal and a very large (e.g. 1000:1) sweep range is created (f = 0 at V_{sweep} = 0). Care must be taken, however, to regulate the supply voltage; in this configuration the charge current is no longer a function of the supply voltage (yet the trigger thresholds still are) and thus the frequency becomes dependent on the supply voltage. The potential on pin 8 may be swept from V_{CC} to about 2/3 V_{CC}.

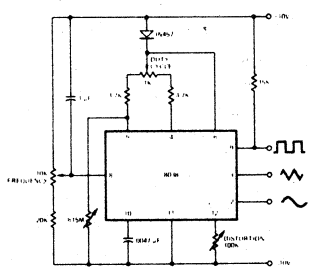
Buffering



The sine wave output has a relatively high output impedance (1K Typ). The circuit provides buffering, gain and amplitude adjustment. A simple op amp follower could also be used.

If the available outputs are all fed through a buffer, extra resistors can be inserted in series with the signal before a switch. Values of 47k (square wave), 15k (triangular) and 10k (sine wave) will ensure equal amplitude signals.

Audio Oscillator



To obtain a 1000:1 Sweep Range on the 8038 the voltage across external resistors RA and RB must decrease to nearly zero. This requires that the highest voltage on control Pin 8 exceed the voltage at the top of RA and RB by a few hundred millivolts.

The Circuit achieves this by using a diode to lower the effective supply voltage on the 8038. The large resistor on pin 5 helps reduce duty cycle variations with sweep. The range of this circuit is 20Hz to 20 kHz, output buffer can be added to make a general purpose bench unit.

Points to Note!

The 8038 runs hot to touch, this is normal, and is due to the resistive nature of the sinewave shaping network.

The optimum supply voltage, for minimum temperature drift is 20V, this can be seen in the stability graph.

ETI data sheet

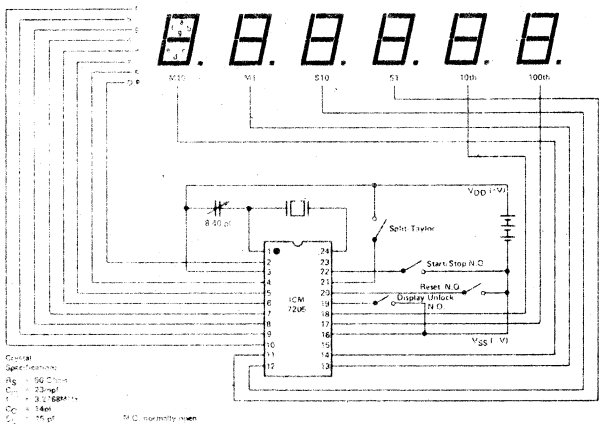
ICM 7205 Stopwatch Chip

The Intersil ICM 7205 is a relatively new device, the main points of interest being: on-chip display drivers, full protection against static – no special handling precautions are required –, and an average current when in operation (including display!) of only 10 mA.

The ICM 7205 is a fully-integrated CMOS six digit stopwatch circuit. The circuit interfaces directly with a six digit/seven segment common cathode LED display. The low battery indicator can be connected directly to the decimal point anode or to a separate LED lamp. The only components required for a complete stopwatch besides the display are: three SPST switches, a 3.2768 MHz crystal, a trimming capacitor, three AA batteries, and an on/off switch. For a two function stopwatch, one additional switch would be required.

The circuit divides the oscillator frequency by 2^{15} to obtain 100 Hz which is fed to the fractional seconds, seconds and minutes counters. An intermediate frequency is used to obtain the 1/6 duty cycle 1.07 kHz multiplex waveforms. The blanking logic provides leading zero blanking for seconds and minutes independently of the clock. The ICM7205 is packaged in a 24 lead plastic DIP.

Stopwatch Circuit



Switch Characteristics

The ICM 7205 is designed for use with SPST switches throughout. On the display unlock and reset the characteristics of the switches are unimportant, since the circuit responds to a logic level for any length of time, however short. Switch bounce on these inputs does not need to be specified. The Start/Stop input, however, responds to an edge, and it requires a switch with less than 15 ms of switch bounce. The bounce protection circuitry has been specifically designed to let the circuit respond to the first edge of the signal, so as to preserve the full accuracy of the system.

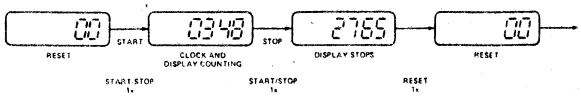
Low Battery Indicator

The on-chip low battery indicator is intended for use with a small LED lamp or with the decimal points on a standard LED display. The output is the drain of a P-channel transistor of approximately half the size of one of the segment drivers. The LBI circuitry is designed always to provide a voltage difference between the LBI trigger voltage and the minimum operating voltage, i.e., the lower the LBI trigger voltage the lower the minimum operating voltage. In this way a stopwatch using three AA batteries will provide at least 15 minutes of accurate timekeeping after the LBI comes on.

Functional Operation

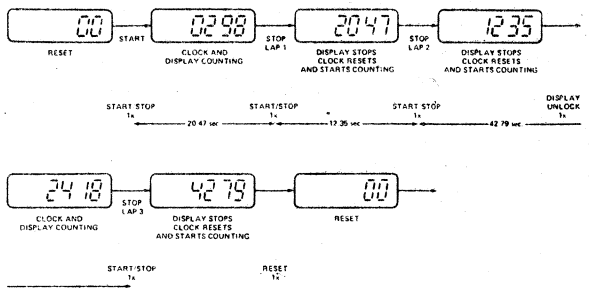
Turning on the stopwatch will bring up the reset state where the fractional seconds are on displaying 00 and the other digits are blanked. This display always indicates that the stopwatch is ready to go.

Start/Stop



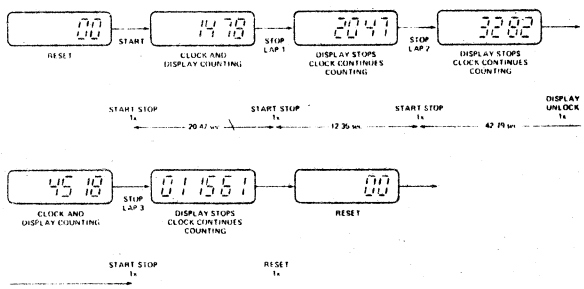
The Start/Stop modes can be used for a single event timing with the Split/Taylor input in either state. The illustration indicates the operations and the results. To time another event the reset switch must be used prior to the start of the event. Seconds will be displayed after one second, minutes after one minute. The range of the stopwatch is 59 minutes 59.99 seconds. If an event exceeds one hour, the number of hours must be remembered by the user. Leading zeros are not blanked after one hour.

Taylor



When the Split/Taylor input is left open circuit or is connected to Vss, the stopwatch can be used in the Taylor or sequential mode. As depicted graphically above, each split time is measured from zero in the Taylor mode, i.e., after stopping the watch, the counters reset to zero momentarily and start counting the next interval. The time displayed is the time elapsed since the last activation of Start/Stop. The display is stationary after the first interval unless the display unlock is used to show the running clock. Reset can be used at any time.

Split



When the Split/Taylor input is connected to VDD the stopwatch is in the Split mode. The Split mode differs from the Taylor in that the lap times are cumulative in the Split mode. The counters do not reset or stop after the first start until reset is activated. Any time displayed is the cumulative time elapsed since the first start after reset. Display unlock can be used to let the display 'catch up' with the clock. Reset can be used at any time.

Points to Note!

Absolute maximum supply voltage is 5V5. Never short outputs to earth or low impedance power supply as this will destroy the device.

ETI data sheet

INTERSIL ICM 7208

FEATURES:

- Useful for:
  a. Unit counter
  b. Frequency counter
  c. Period counter
- Low operating power dissipation < 10mW
- Low quiescent power dissipation < 5mW
- Counts and displays 7 decades
- Wide operating supply voltage range
  $2V \leqslant |V_{DD} - V_{SS}| \leqslant 6V$
- Drives directly 7 decade multiplexed common cathode LED display
- Internal store capability
- Internal inhibit to counter input
- Test speedup point
- All terminals protected against static discharge

DESCRIPTION

The ICM 7208 is a fully integrated seven decade counter-decoder-driver and is manufactured using the Intersil low voltage metal gate C-MOS process. As such it has applications as either a unit, frequency or period counter. For unit counter applications the only additional components are a 7 digit common cathode display, 3 resistors and a capacitor to generate the miltiplex frequency reference, and the control switches.

Specifically the ICM 7208 provides the following on chip functions: a 7 decade counter, multiplexer, 7 segment decoder, digit & segment drivers, plus additional logic for display blanking reset, input inhibit, and display on/off.

The ICM 7208 is intended to operate over a supply voltage of 2 to 6 volts as a medium speed counter or over a more restricted voltage range for high frequency applications.

As frequency counter it is recommended that the ICM 7208 be used in conjunction with the ICM 7207 Oscillator Controller which provides a stable HF oscillator, and output signal gating.

TESTING PROCEDURES

The ICM 7208 is provided with three input terminals: 7,23,27 which may be used to accelerate testing. The least two significant decade counters may be tested by applying an input to the 'COUNTER INPUT' terminal 12. 'TEST POINT' terminal 23 provides an input which bypasses the 2 least significant decade counter. Similarly terminals 7 and 27 permit rapid counter advancing at two points further along the string of decade counters.

COUNTER INPUT DEFINITION

The internal counters of the ICM 7208 index on the negative edge of the input signal at terminal #12.

Format of Signal to be Counted

The noise immunity of the Signal Input Terminal is approximately 1/3 the supply voltage. Consequently, the input signal should be at least 50% of the supply in peak to peak amplitude and preferably equal to the supply. **NOTE: The amplitude of the input signal should not exceed the supply; otherwise, damage**

Fig. 1. Pinout.

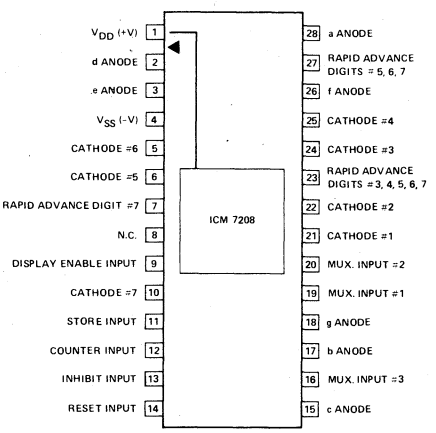


Fig. 2. Absolute maximum ratings.

Power Dissipation (Note 1)	1 watt
Supply voltage $\|V_{DD} - V_{SS}\|$ (Note 2)	6 V
Output digit drive current (Note 3)	150 mA
Output segment drive current	30 mA
Input voltage range (any input terminal)	Not to exceed the supply voltage
Operating temperature range	-20°C to +70°C
Storage temperature range	-55°C to +125°C

*Absolute maximum rating define parameter limits that if exceeded may permanently damage the device.

Fig. 3. Typical operating characteristics.

($V_{DD} - V_{SS} = 5V$, $T_A = 25°C$, TEST CIRCUIT, display off, unless otherwise specified)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
Quiescent Curent	I_{DD1}	All controls plus terminal 20 connected to V_{DD}. No multiplex oscillator		30	100	µA
Quiescent Current	I_{DD2}	All control inputs plus terminal 20 connected to V_{DD} except store which is connected to V_{SS}		70	150	µA
Operating Supply Current	I_{DDS}	All inputs connected to V_{DD}, RC multiplexer osc operating $f_{in} < 25KHz$		210	500	µA
Operating Supply Current		$f_{in} = 2MHz$			700	µA
Supply Voltage Range	V_{DD}	$f_{in} \cdot 2MHz$	3.5		5.5	V
Digit Driver On Resistance	R_D			4	12	ohm
Digit Driver Leakage Current	I_D				500	µA
Segment Driver On Resistance	R_S			40		ohm
Segment Driver Leakage Current	I_S				500	µA
Pullup Resistance of Reset or Store Inputs	Rp		100	400		Kohms
Counter Input Resistance	R_{IN}	Terminal 12 either at V_{DD} or V_{SS} potentials			100	Kohms
Counter Input Hysteresis Voltage	V_{HIN}			25	50	mV

NOTE 1 This value of power dissipation refers to that of the package and will not be obtained under normal operating conditions.
NOTE 2 The supply voltage must be applied before or at the same time as any input voltage. This poses no problems with a single power supply system. If a multiple power supply system is used, it is mandatory that the supply for the ICM 7208 is not switched on after the other supplies otherwise the device may be permanently damaged.
NOTE 3 The output digit drive current must be limited to 150 mA or less under steady state conditions. (Short term transients up to 250 mA will not damage the device.) Therefore, depending upon the LED display and the supply voltage to be used it may be necessary to include additional segment series resistors to limit the digit currents.

may be done to the circuit.

The optimum input signal is a 50% duty cycle square wave equal in amplitude to the supply. However, as long as the rate of change of voltage is not less than approximately $10^{-4}V/\mu sec$ at 50% of the power supply voltage, the input waveshape can be inusoidal, triangular, etc.

Display Considerations

Any common cathode multiplexable LED display may be used. However, if the peak digit currents exceeds 150 mA for any prolonged time, it is recommended that resistors be included in series with the segment outputs (terminals 2, 3, 15, 17, 18, 26, 28) to limit current to 150 mA.

The ICM 7208 is specified with 500 μA of possible digit leakage current. With certain new LED displays that are extremely efficient at low currents, it may be necessary to include resistors between the cathode outputs and the positive supply V_{DD} to bleed off this leakage current.

Display Multiplex Rate

The multiplex frequency reference is divided by eight to generate an 8 bit sequencer. Thus the display multiplex rate is one eighth of the multiplex frequency reference.

The ICM 7208 has approximately 0.5 μs overlap between output drive signals. Therefore, if the multiplex rate is very fast, digit ghosting will occur. The ghosting determines the upper limit for the multiplex frequency reference. At very low multiplex rates flicker becomes visible.

It is recommended that the display miltiplex rate be within the range of 50 Hz to 200 Hz which corresponds to 400 Hz to 1600 Hz for the reference frequency.

CONTROL INPUT DEFINITIONS

INPUT	TMNL	VLTG	FUNCTION
1. Display	9	V_{DD}	Display on
		V_{SS}	Display off
2. Store	11	V_{DD}	Counter Inform. Stored
		V_{SS}	Counter Inform. Transferring
3. Inhibit	13	V_{DD}	Input to Counter Blocked
		V_{SS}	Normal Opertn.
4. Reset	14	V_{DD}	Normal Opertn.
		V_{SS}	Counters Reset

Fig. 4. Typical performance characteristics.

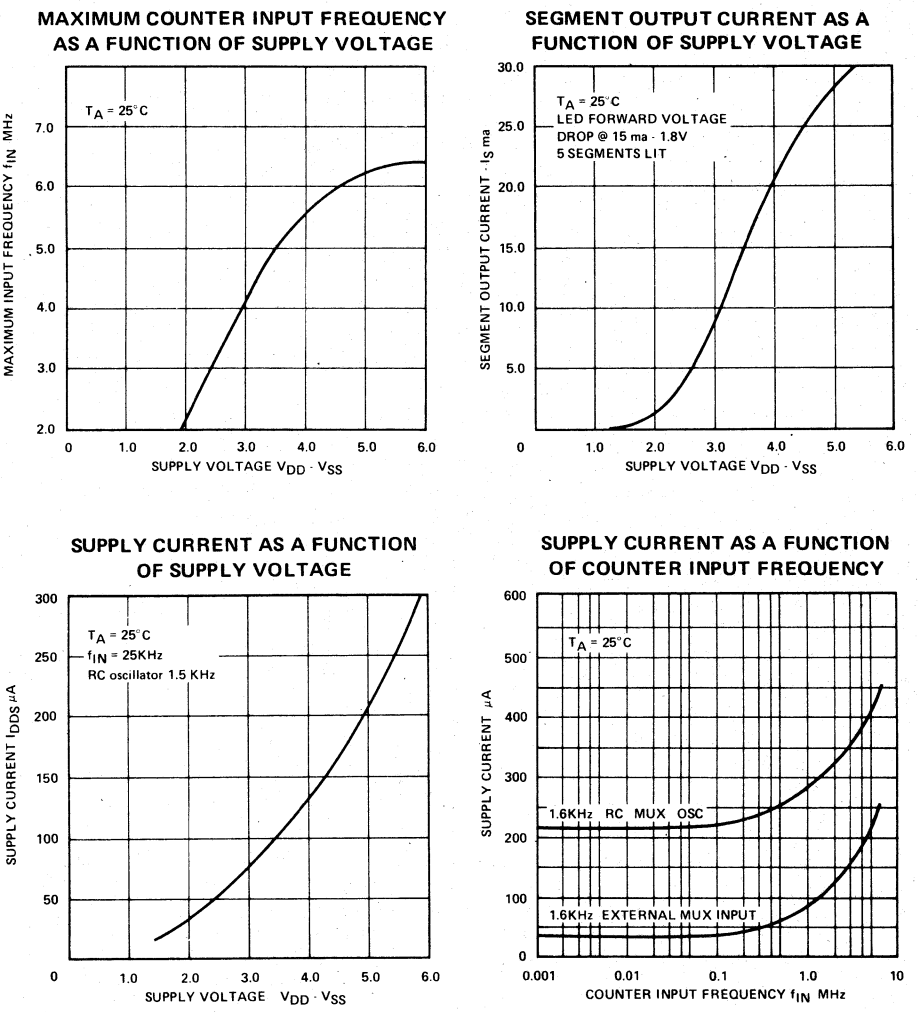
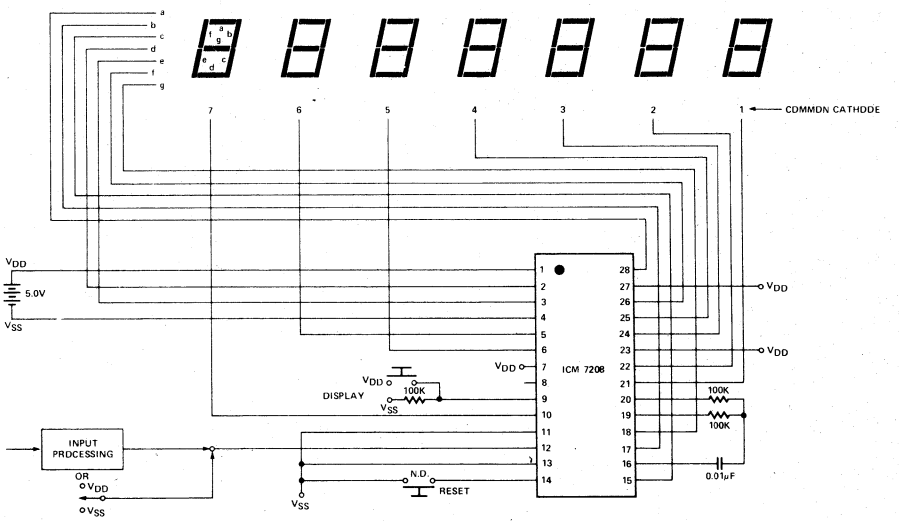


Fig. 5. Unit counter schematic.



Unit Counter

The unit counter updates the display for each negative transition of the input signal. The information on the display will count after reset from 00 to 9,999,999 and then will reset to 0000000 and will begin to count up again. To blank leading zeros actuate reset at the beginning of a count. Leading zero blanking affects two digits at a time.

For battery operated systems the display may be switched off to conserve power.

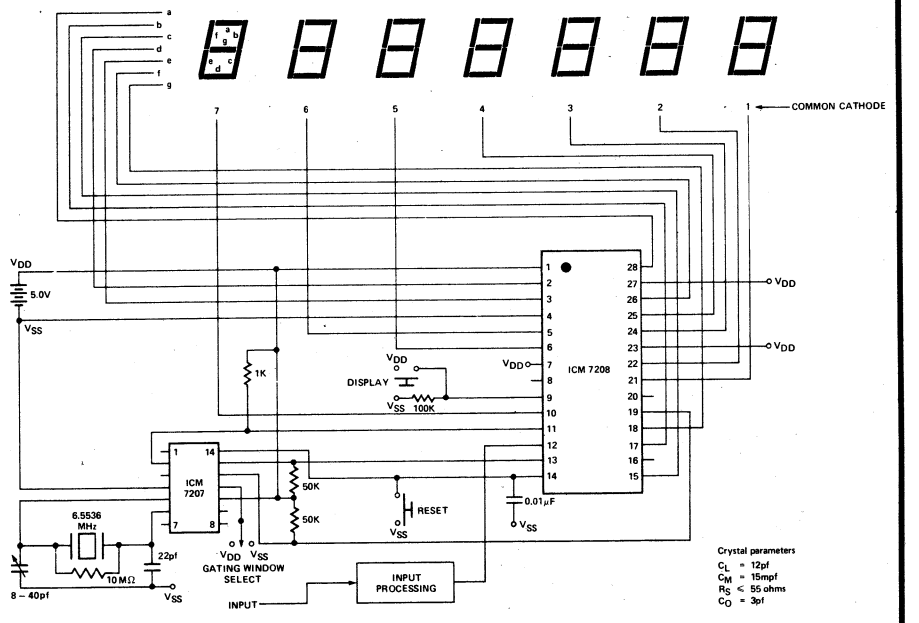
An external generator may be used to provide the multiplex frequency reference (input terminal 20). The signal applied to terminal 19 (terminals 16 and 20 open circuit) should be approximately equal to the supply voltage and for minimum power dissipation should be a square wave.

For stand alone systems two inverters are provided so that a simple but stable RC oscillator may be built using only 2 resistors and a capacitor.

Figure 5 shows the schematic of an extremely simple unit counter that can be used for remote traffic counting, to name one application. The power cell stack should consist of 3 or 4 nickel cadmium rechargeable cells (nominal 3.6 or 4.8 volts). If 4 x 1.5 volt cells are used it is recommended that a diode be placed in series with the stack to guarantee that the supply voltage does not exceed 6 volts.

The input switch is shown to be a single pole double throw switch (SPDT). A single pole single throw switch (SPST) could also be used with a pullup resistor. However, anti-bounce circuitry must be must be included in series with the counter input. In order to avoid all contact bounce problems due to the SPDT switch the ICM 7208 contains an input latch on chip.

Fig. 6. Frequency counter schematic.



Note:
For a 1 sec count window which allows all 7 digits to be used with a resolution of 1 Hz, the ICM7207 can be replaced with the ICM7207A. Circuit details are given on the 7207A data sheet.

Fig. 7. Internal block diagram.

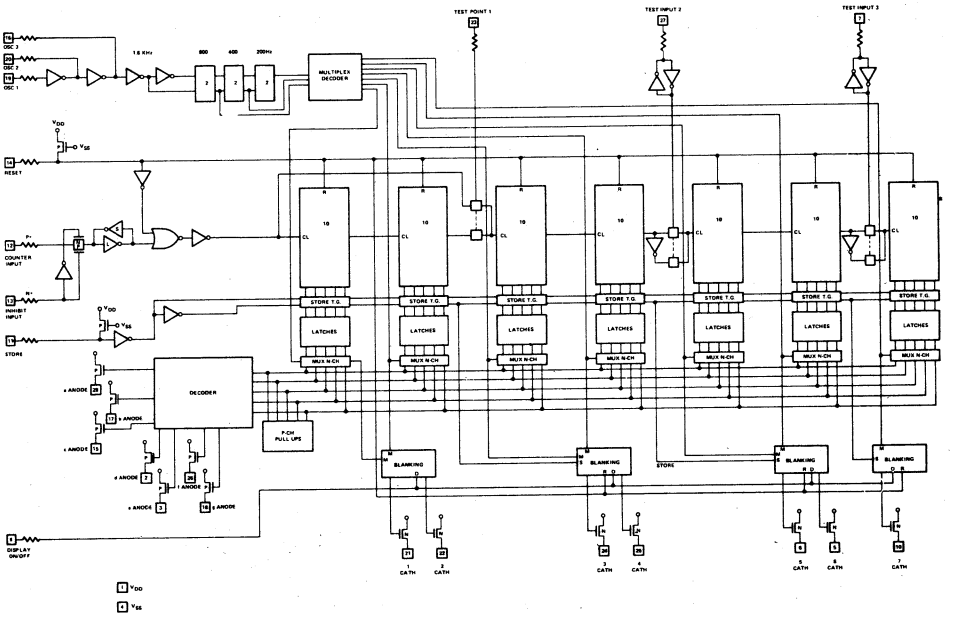
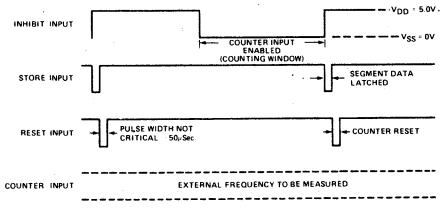


Fig. 8. Frequency counter input waveforms.



USING THE LM3900N

Four amplifiers on a single chip

ONE of the most noticeable trends in modern electronics is for more and more components to be packed into smaller and smaller spaces. One example of this is the fairly new LM3900 device manufactured by the National Semiconductor Company. It contains four separate internally compensated amplifiers in a single 14 pin dual-in-line encapsulation.

All four amplifiers are fabricated on a single silicon chip. Each amplifier contains seven transistors, a diode and a capacitor, whilst other internal components are used in the bias and power supplies.

One might expect that new devices of this type would be quite expensive, but the LM3900 is available at just over $2.00 each — or even less in quantities of ten or more.

CONNECTIONS

The connections of the four separate amplifiers are shown in Fig.1. Each amplifier has a non-inverting input (marked +), an inverting input (marked —) and an output connection.

In addition, there is a single common positive supply connection and a common ground connection (negative supply line) for the whole device.

INTERNAL CIRCUIT

Conventional high gain amplifiers employ a differential input stage to provide inverting and non-inverting inputs, but a rather different approach is employed in the LM3900N. A 'current mirror' is employed in the non-inverting input circuit, the current 'reflected' in this mirror being subtracted from that which enters the inverting input.

This type of amplifier therefore acts as a differential stage by amplifying the difference between two *currents* rather than the difference between two voltages (as in a conventional amplifier).

The type of amplifier used in the LM3900N may be referred to as a 'Norton' amplifier, since Norton is the name of the person who developed a theorem relating the *current* flowing in a circuit to the equivalent current generator and shunt impedance.

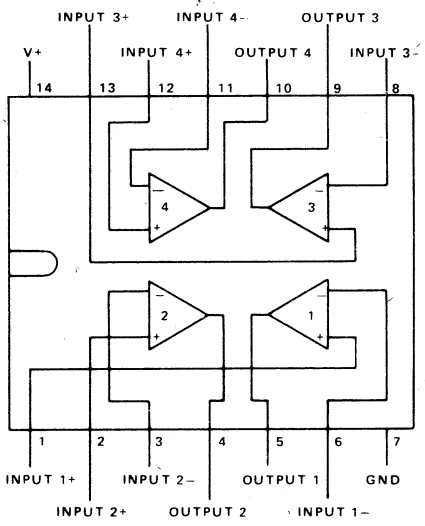


Fig.1. The connections of the LM3900N.

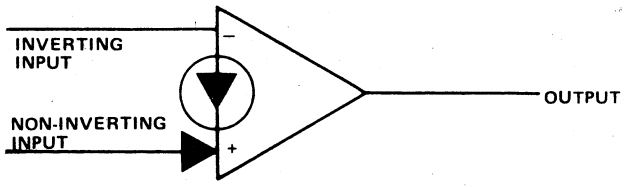


Fig.2. The symbol for one of the Norton amplifiers of the LM3900N.

SYMBOL

The symbol recommended for each of the four Norton amplifier stages in the device is shown in Fig.2. This symbol distinguishes this type of amplifier from the standard operational amplifier symbol and avoids confusion in circuits.

The symbol of Fig. 2 contains an indication that there is a current source between the inverting and non-inverting inputs and implies that the amplifier uses a current mode of operation. In addition, the circuit symbol indicates that current is removed from the inverting input, whilst the arrow on the non-inverting input shows that this functions as a current input.

PERFORMANCE

The LM3900N has the advantage that it can operate from a single supply voltage over the range of four volts to 36 volts. Most conventional operational amplifiers require supplies symmetrical with respect to ground (typically ± 15 V); the LM3900N can be used with such supply lines if desired.

The maximum peak to peak output amplitude of an LM3900N amplifier is only 1 V less than the supply voltage employed. The current consumed from the power supply is typically 6.2 mA (maximum 10 mA).

The typical voltage gain of each amplifier is 2800 or nearly 70 dB. The

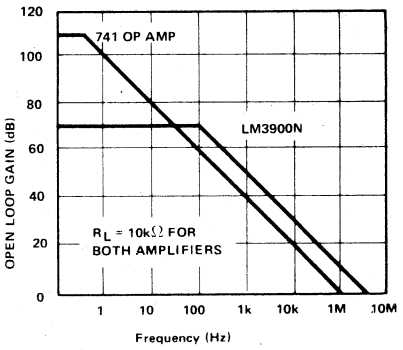


Fig. 3. Comparison of the gain of the LM3900N with that of a 741 amplifier at various frequencies.

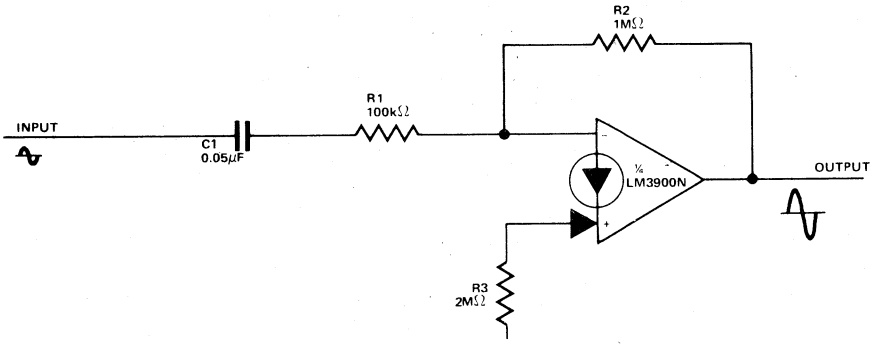


Fig.4. A simple a.c. amplifier circuit.

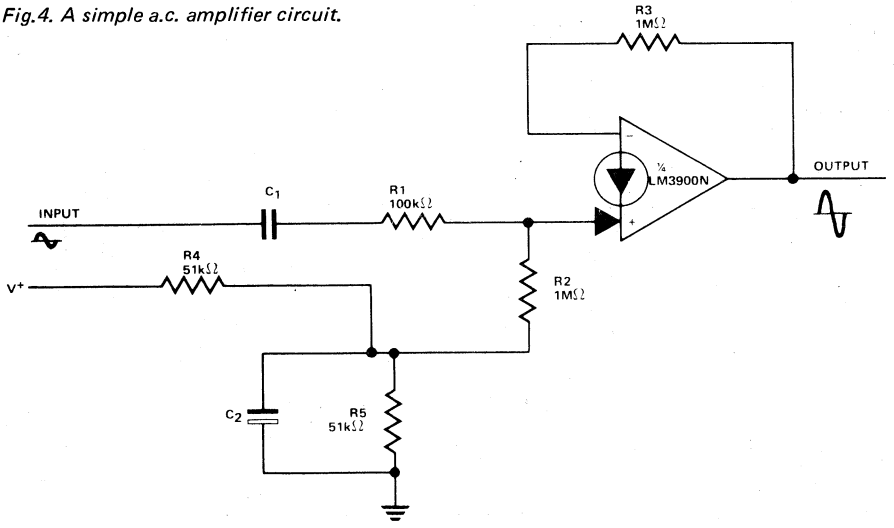


Fig.5. A simple non-inverting a.c. amplifier.

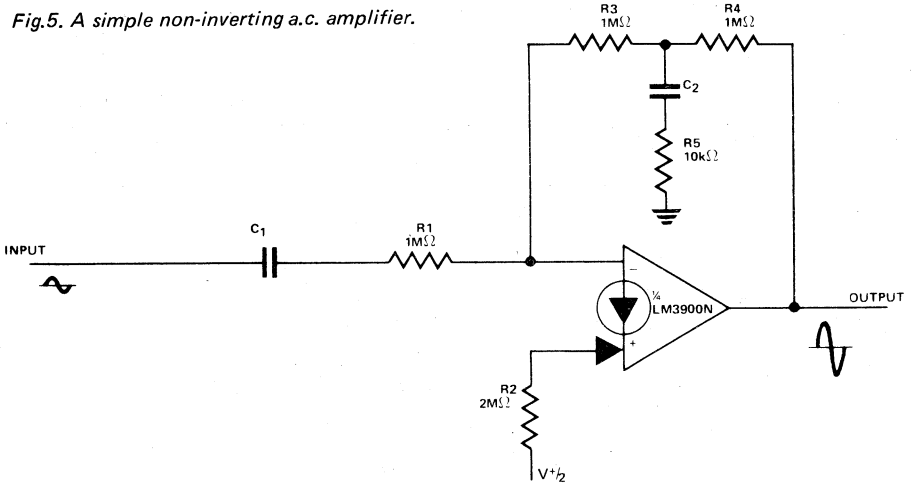


Fig.6. An amplifier which has a high gain and a high input impedance.

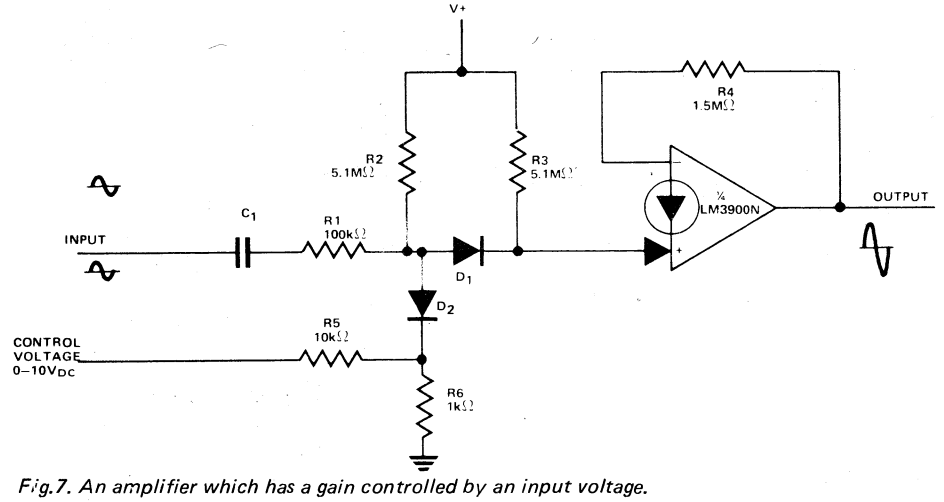


Fig.7. An amplifier which has a gain controlled by an input voltage.

minimum gain of any amplifier is 1200. The variation of this gain with frequency is compared with that of the well known type 741 operational amplifier in Fig. 3. It can be seen that the LM3900N amplifiers provide about 10 dB more gain at all frequencies above 1 kHz.

APPLICATIONS

The Norton amplifiers used in the LM3900N device entail the use of somewhat different circuit design techniques than those used with conventional operational amplifiers.

The inverting input of the LM3900N amplifiers must be supplied with a steady biasing current. The current to the non-inverting input modulates that to the inverting input. The fact that current can pass between the input terminals leads to some unusual applications.

Both inputs of each of the amplifiers in the LM3900N are clamped by diodes so as to keep their potentials almost constant at one diode voltage drop (about 0.5 V) above the ground potential of pin 7. External input voltages must therefore be converted to input currents by placing series resistors in each input circuit.

USE AS AN AC AMPLIFIER

The LM3900N forms a useful ac amplifier, since its output can be baised to any desired steady voltage within the range of the output voltage swing. The ac gain is independent of the biasing level and the single power supply required greatly simplifies circuit design.

A simple ac amplifier circuit is shown in Fig.4. The gain is approximately equal to R_2/R_1 or 10 with the circuit values shown, the mean potential at the output is half the supply voltage. The value of R_3 should be twice that of R_2 since the current' passing through of these resistors is then the same. The positive supply and ground connections are not shown in Fig. 4

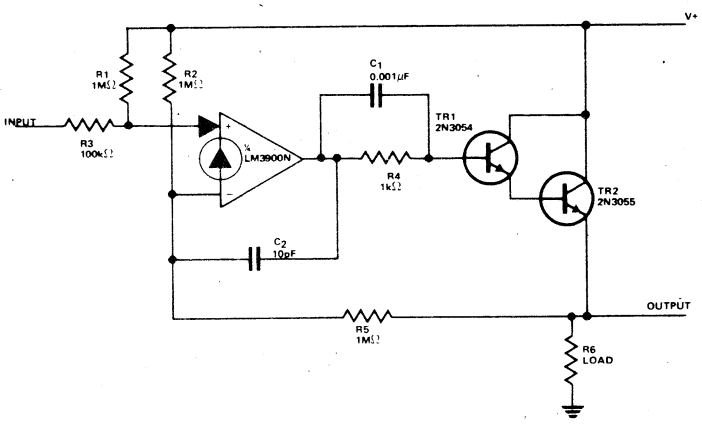


Fig.8. A direct coupled power amplifier.

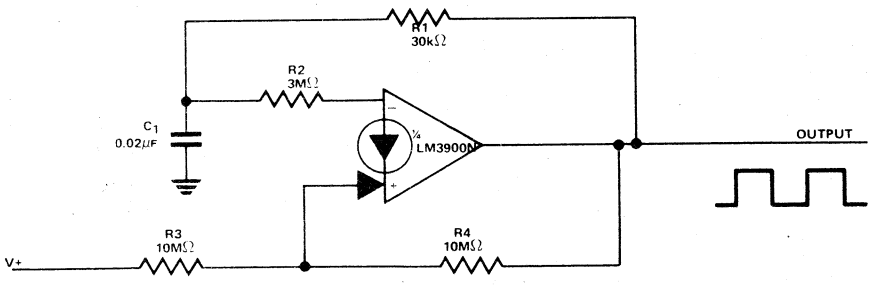


Fig.9. A simple square-wave generator.

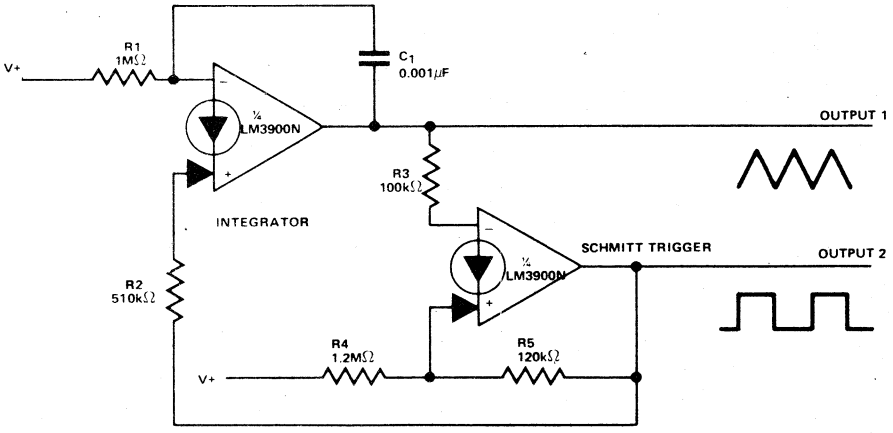


Fig.10. A circuit for generating triangular and square-wave.

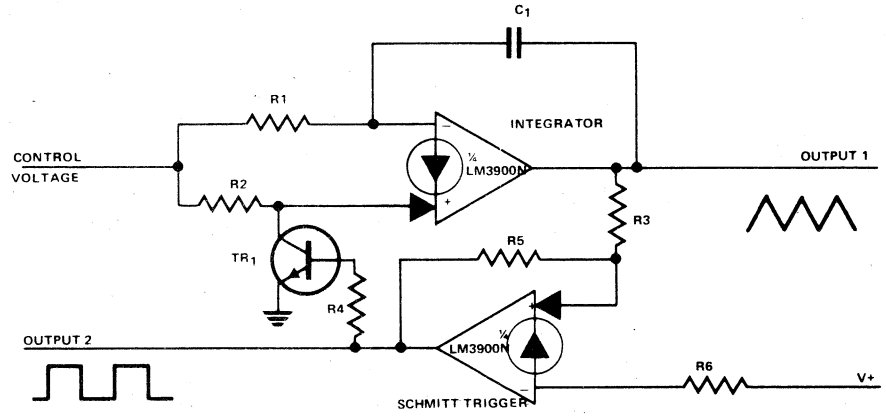


Fig.11. A voltage controlled oscillator which produces triangular and square-waves.

for simplicity, but R_3 should be returned to the same positive supply line as that used to feed pin 14.

The circuit of Fig. 4 provides a phase inverted output. Any ripple on the power supply line will appear on the output at half amplitude.

NON-INVERTING AC AMPLIFIER

The circuit of Fig. 5 shows an amplifier which provides an output in phase with the input. The gain is equal to $R_3/(R_1 + r_d)$ where r_d is the small signal impedance of the input diode. The value of r_d is equal to 0.026 divided by the current passing through R_2 to the non-inverting input.

The capacitor values should be chosen so that the impedance of these components is considerably less than the circuit impedance at the points concerned.

HIGH IMPEDANCE AND HIGH GAIN

The circuits of Figs. 4 and 5 have an input resistance, R_1 or 100 k ohm. If this resistor is increased to provide a higher input impedance, the gain of the circuit will fall. However, the circuit of Fig. 6 has been designed so that it provides both a high input impedance and a high gain using a simple amplifier. With the component values shown, the input impedance is one megohm and the gain 100.

The voltage applied to R_2 is made equal to the output voltage (which is half the supply voltage). The value of R_2 is equal to the sum of R_3 and R_4; these resistors set the dc bias. If desired, R_2 may be made four megohms and its lower end connected to the V+ supply.

Resistors R_4 and R_5 form a potential divider so that only 1/100 of the alternating output voltage is developed across the $C_2 - R_5$ circuit. This fraction of the output voltage is fed back to the inverting input via R_3. As R_3 and R_1 are equal, the gain is R_4R_5. As R_5 is decreased, the gain approaches the open loop gain of the amplifier.

VOLTAGE CONTROLLED GAIN

An amplifier with a gain which can be controlled by the value of a steady applied voltage is shown in Fig. 7.

A current flows from the positive supply through R_3 to provide a bias which prevents the output of the amplifier from being driven to saturation as the control voltage is varied. When D_2 is non-conducting, the currents passing through both R_2 and R_3 enter the non-inverting input and the gain is of maximum. This occurs when the control voltage approaches 10 V.

The gain is a minimum when the control voltage is zero. In this case D_2 is conducting and only the current passing through R_3 enters the non-inverting input of the amplifier.

DIRECT COUPLED POWER AMPLIFIER

In the circuit of Fig. 8, the output from an LM3900N amplifier is fed to a Darlington pair of power transistors. This circuit can deliver over three amps into a suitable load when the transistors are correctly mounted on heat sinks.

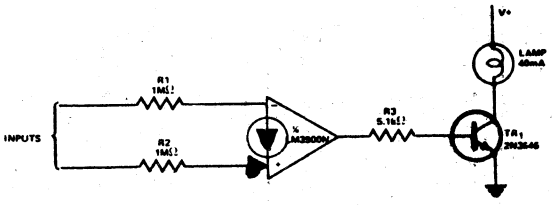


Fig. 12. A voltage comparator with an indicator lamp.

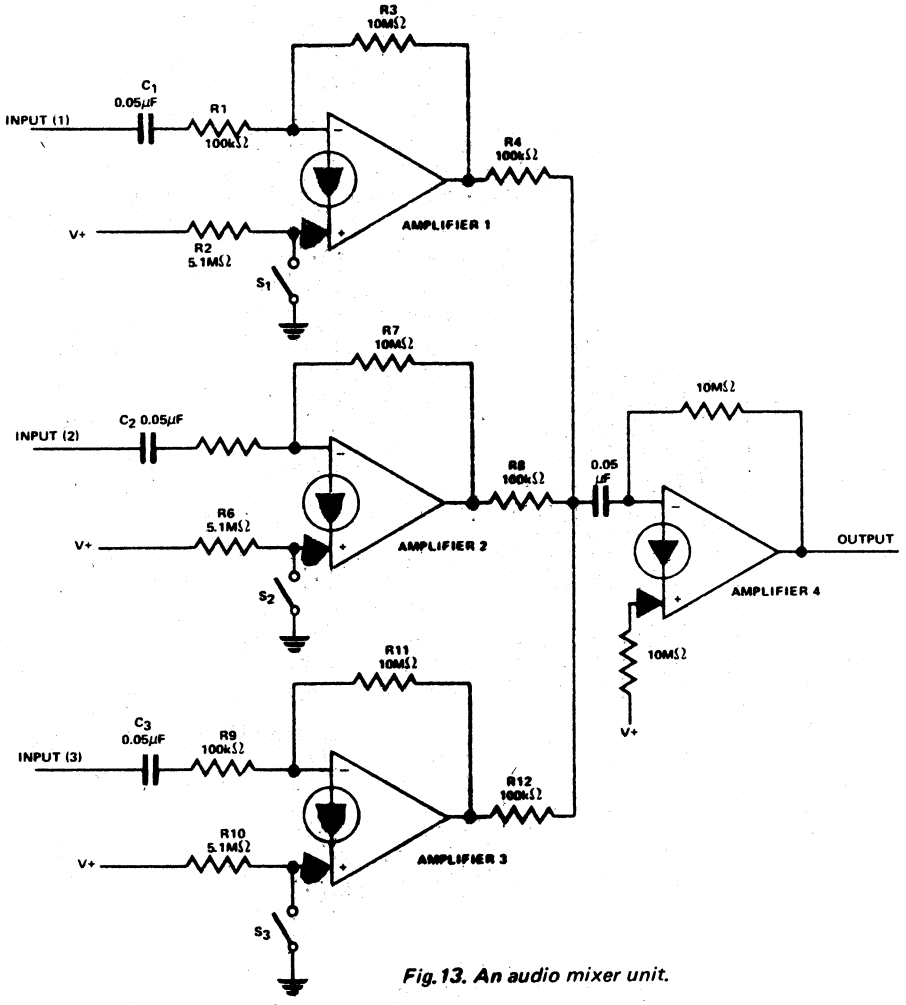


Fig. 13. An audio mixer unit.

SQUAREWAVE GENERATOR

The multiple amplifiers in the LM3900N device are very suitable for use in waveform generators at frequencies of up to about 10 kHz. Voltage controlled oscillators (the frequency of which is dependent on an input voltage) can also be designed using the device.

A simple square wave generator is shown in Fig. 9. The capacitor C_1 alternately charges and discharges between voltage limits which are set by R_2 R_3 and R_4. The circuit is basically of the Schmitt trigger type, the voltages at which triggering occurs being approximately V+/3 and 2V+/3.

TRIANGULAR WAVEFORM GENERATOR

A triangular waveform generator can be made by using one amplifier of a LM3900N device as an integrator and another amplifier as a Schmitt trigger circuit. A suitable circuit is shown in Fig. 10; it has the unusual advantage that only the one power supply is required.

When the output voltage from the Schmitt trigger circuit is low, the current flowing through R_2 is integrated by C_1 to produce the negative slope of the triangular wave at output 1. When the output 2 voltage from the Schmitt trigger is high, current flows through R_2 to produce the rising part of the waveform at output 1.

The output waveform will have good symmetry if $R_1 = 2R_2$. The output frequency is given by the equation:

$$f = \frac{V^+ - V_{BE}}{2R_1C_1V}$$

where $R_1 - 2R_2$, V_{BE} is the steady voltage at the inverting input (0.5 V) and V is the difference between the tripping points of the Schmitt trigger.

VOLTAGE CONTROLLED OSCILLATOR

A simple voltage controlled oscillator circuit which produces both triangular and square wave outputs is shown in Fig. 11. As in Fig.10, one amplifier is employed as an integrator.

When the output of the Schmitt trigger is high, the clamp transistor TR_1 is conducting and the input current passing through R_2 is shunted to ground. The current passing through R_1 causes a falling ramp to be formed.

When the Schmitt circuit changes state, its output switches TR_1 to the non-conducting state. The current flowing through R_2 can be made twice that flowing through R_1 ($R_2 = R_1/2$) so that the rising part of the ramp has a similar slope to the negative part.

The greater the value of the control voltage in Fig.11, the greater the frequency of oscillation. However, the voltage must exceed the constant input voltage (V_{BE}) or the circuit will fail to oscillate.

VOLTAGE COMPARATOR

The circuit of Fig. 12 shows how an LM3900N amplifier may be employed to compare two input voltages and to indicate the result by means of a small lamp. If the input voltage connected to the non-inverting input is appreciably more positive than the other input, the output of the amplifier will provide a positive voltage which renders TR_2 conducting. The lamp will then be illuminated.

One of the inputs may be a reference voltage so that one can then compare a single input voltage against this constant reference.

AUDIO MIXER

The amplifiers of a LM3900N device can be conveniently used to make a mixer unit for audio purposes; the unit enables three separate audio signals to be mixed together to produce a composite output. The circuit shown in Fig. 13 provides this facility using only a single LM3900N device and also enables any one channel to be selected be switches. The currents passing through the resistors R_4 R_8 and R_{12} are summed in the input circuit of the fourth amplifier.

If S_1 is open, amplifier 1 will be driven to saturation by the current passing through R_2. It will therefore be inactive.

CONCLUSION

This short article has attempted to show a few of the numerous applications of this economical integrated circuit. Many more applications (such as phase locked loops, temperature sensing circuits, differentiators, tachometers, staircase generators, active filters, etc) are given in a report AN-72 produced by National Semiconductor.

LANGUAGES

TEACH YOURSELF ASSEMBLER

Paul Overaa discusses the arithmetic operations of addition, subtraction, multiplication and division on the 6502, Z80 and 8080 processors.

This is part six of APC's Teach Yourself Assembler series. It's unique in using Basic as its point of reference, and avoiding the 'drop you in it' approach often used on this subject. Three processors, the Z80, 6502 and 8080 are covered in detail, but enough information is provided to enable users of other processors to follow the course. Copies of earlier articles in the series, which started in March 1984, may be obtained from our Back Issues dept.

The basic arithmetic instructions available on the 8080, Z80 and 6502 processors are for addition and subtraction. The 6502 operates on 8-bit operands only, but both the 8080 and Z80 have certain instructions that enable 16-bit operands to be dealt with.

Addition Z80

On the Z80, addition instructions take the form ADD A, operand. The specified operand is added to the value present in the accumulator, and in symbolic form we can write A ← A + operand. Various forms of addressing are possible, as follows:

ADD A,8: adds the immediate value 8 to the accumulator — that is, is performing the function A ← A + 8.

ADD A,B: adds the contents of the B register to the accumulator, thus performing the function A ← A + B.

ADD A,(HL): adds to the accumulator the contents of the byte whose address is specified by HL — that is, A ← A + (HL).

ADD A,(IX+d): in the indexed addressing form, the address of the byte to be added is found by adding a specified displacement to the address held in index register IX. The symbolic representation is A ← A + (IX+d).

Instructions for 16-bit operations use HL, IX or IY as destination registers. Typical examples are as follows:

ADD HL,DE; adds the contents of the DE pair to the contents of HL, thus performing HL ← HL+DE.

ADD IX,BC: in a similar fashion, this adds the contents of BC to the index register IX.

On the Z80, the instruction 'add with carry' (ADC) will include, in the 'addition', the carry flag value: ADC A,B will perform the function A ← A + B + Carry. The usefulness of this instruction can be seen from the example in Fig 1. We add two 'two byte numbers' — 255 and 257 — by adding the two low bytes first and then adding the two high bytes.

The addition of the low bytes causes a 'carry' to occur: the ADC instruction takes it into account when the high bytes are added. As a general rule, multibyte addition is performed by using a normal addition instruction for the first (least significant) bytes, and using the 'add with carry' instructions for succeeding bytes. The program in Fig 2 adds the contents of two 'two byte numbers' held in locations labelled FIRST$NUMBER and SECOND$-NUMBER.

Because of the existence of double register addition instructions, it's possible to write a much simpler 16-bit addition program on the Z80. DE and HL can be loaded directly with the numbers to add, and an ADD HL,DE instruction used to perform the 16-bit addition with one addition instruction (Fig 3).

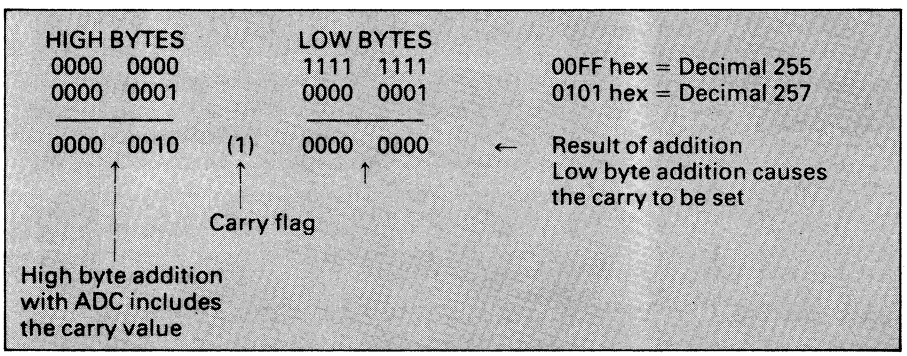


Fig 1 Z80 'add with carry' instruction

```
LD    HL,SECOND$NUMBER    ;HL points to low byte of second number
LD    A,FIRST$NUMBER      ;Get low byte of first number in Acc
ADD   A,(HL)              ;Add low bytes
LD    (RESULT),A          ;Store low byte of result
LD    A,FIRST$NUMBER+1    ;Get high byte of first number
INC   HL                  ;Now points to high byte of second number
ADC   A,(HL)              ;Add high bytes + carry
LD    (RESULT+1),A        ;Store high byte of result
```

Fig 2 Z80 16-bit addition

```
LD    DE,(FIRST$NUMBER)   ;Load DE with first number
LD    HL,(SECOND$NUMBER)  ;Load HL with second number
ADD   HL,DE               ;Performs HL ← HL + DE
LD    (RESULT),HL         ;Store result
```

Fig 3 Z80 alternative 16-bit addition

Addition 8080

Immediate loading of 8080 register pairs uses a LXI instruction. LXI H, SECOND$NUMBER will load the HL pair with the 16-bit address equivalent to the label SECOND$NUMBER. LDA is a direct loading of the accumulator from the byte whose address is FIRST$NUMBER. 'M' is the 8080 assembler convention to specify an

```
LXI   H,SECOND$NUMBER    ;HL points to low byte of second number
LDA   FIRST$NUMBER       ;Get low byte of first number in Acc
ADD   M                  ;Add low bytes
STA   RESULT             ;Store low byte of result
LDA   FIRST$NUMBER+1     ;Get high byte of first number
INX   H                  ;Now points to high byte of second number
ADC   M                  ;Add high bytes + carry
STA   RESULT+1           ;Store high byte of result
```

Fig 4 8080 16-bit addition

```
LHLD  FIRST$NUMBER       ;Load HL with first number
XCHG                     ;Swap to DE
LHLD  SECOND$NUMBER      ;Load HL with second number
DAD   D                  ;Performs HL ← HL + DE
SHLD  RESULT             ;Store result
```

Fig 5 8080 alternative 16-bit addition

```
CLC                      ;Clear.carry flag
LDA   FIRST$NUMBER       ;Low byte of first number
ADC   SECOND$NUMBER      ;Add low bytes
STA   RESULT             ;Store low byte of result
LDA   FIRST$NUMBER+1     ;High byte of first number
ADC   SECOND$NUMBER+1    ;Add high bytes
STA   RESULT+1           ;Store high byte of result
```

Fig 6 6502 16-bit addition

```
LD    HL,SECOND$NUMBER   ;HL points to low byte of second number
LD    A,FIRST$NUMBER     ;Get low byte of first number in Acc
SUB   (HL)               ;Subtract low bytes
LD    (RESULT),A         ;Store low byte of result
LD    A,FIRST$NUMBER+1   ;Get high byte of first number
INC   HL                 ;Now points to high byte of second number
SBC   A,(HL)             ;Subtract high bytes with borrow
LD    (RESULT+1),A       ;Store high byte of result
```

Fig 7 Z80 16-bit subtraction

```
LD    DE,(FIRST$NUMBER)  ;Load DE with first number
LD    HL,(SECOND$NUMBER) ;Load HL with second number
AND   A                  ;Clear the carry flag
SBC   HL,DE              ;Equivalent to HL ← HL + DE
LD    (RESULT),HL        ;Store result
```

Fig 8 Z80 alternative 16-bit subtraction

```
LXI   H,SECOND$NUMBER    ;HL Points to low byte of second number
LDA   FIRST$NUMBER       ;Get low byte of first number in Acc
SUB   M                  ;Subtract low bytes
STA   RESULT             ;Store low byte of result
LDA   FIRST$NUMBER+1     ;Get high byte of first number
INX   H                  ;Now points to high byte of second number
SBB   M                  ;Subtract high bytes with borrow
STA   RESULT+1           ;Store high byte of result
```

Fig 9 8080 16-bit subtraction

```
SEC                      ;Set carry flag
LDA   FIRST$NUMBER       ;Low byte of first number in accumulator
SBC   SECOND$NUMBER      ;Subtract low bytes
STA   RESULT             ;Store low byte of result
LDA   FIRST$NUMBER+1     ;High byte of first number in accumulator
SBC   SECOND$NUMBER+1    ;Subtract high bytes
STA   RESULT+1           ;Store high byte of result
```

Fig 10 6502 16-bit subtraction

indirectly addresed memory location, and it refers to the byte whose address is contained in the HL register pair. Thus, ADD M on the 8080 is performing the same function as ADD A,(HL) on the Z80. STA is the 8080 'store accumulator direct', the contents of the accumulator are stored at the address specified. INX is a 'double register increment'. After the INX H instruction, HL is pointing to the byte after that labelled SECOND$NUMBER — that is, it is pointing to SECOND$NUMBER+1. Typical 8080 code is shown in Fig 4.

An equivalent version of the second Z80 form using the HL and DE register pairs can be written, the only difference being that on the 8080 it's not possible to load the DE pair directly. Instead, we load HL with the contents of the byte labelled FIRST$NUMBER, then use an exchange instruction XCHG to 'swap' the contents of the HL and DE registers. The first number is therefore placed into DE, leaving us free to re-load HL with the second number. A double register DAD D instruction is then used to perform the function HL ← HL+DE. The instruction SHLD will store the contents of the HL register pair in the two bytes RESULT and RESULT+1 (Fig 5).

Addition 6502

The only addition instruction available on the 6502 is an 'add with carry' (the mnemonic is ADC). This is no real disadvantage, but it does mean that if you wish to perform 'normal addition' you must 'clear' the carry flag before using ADC. The 6502 can be conditioned to operate in one of two modes, Binary or Decimal. The operations we are discussing are related to normal binary operation and we'll assume that the processor has been placed in binary mode by using a CLD (clear decimal mode) instruction (Fig 6).

Z80 subtraction

As with the addition instructions, it's useful to have two types of subtraction — normal subtraction and 'subtraction with borrow'. Normal subtraction (mnemonic SUB) is used for the 'low bytes' (least significant bytes), and subtraction with borrow (mnemonic SBC) is used for the succeeding bytes (most of the instructions in Fig 7 are identical to the earlier addition program). If, after the subtraction of the least significant bytes the carry flag has been set, this indicates that the value subtracted from the accumulator is greater than the accumulator value itself — a borrow has occurred. The SBC instruction allows for this 'borrow' by including the carry flag in the subtraction.

A more compact version using HL and DE can also be written. The only subtraction instruction available for the

double register operations is a subtract with carry. This being so, we clear the carry flag by ANDing the accumulator with itself, thus producing a 'normal subtraction' (there is no explicit 'clear carry Z80 instruction' that could be used). The code in Fig 8 gives the general idea.

Subtraction 8080

The mnemonics are SUB and SBB. The 8080 does not have double register subtraction instructions, and the example in Fig 9 uses the accumulator as in the first 8080 addition example.

Subtraction 6502

The 'subtract with borrow' instruction on the 6502 performs the function A ← A − operand − $\overline{\text{Carry}}$, with the bar over the carry indicating the 'complement' of the carry. Borrow is thus defined as the carry flag complemented. The 6502 equivalent for a 16-bit subtraction starts by SETTING the carry flag using a SEC instruction. As with Z80 and 8080 forms, the least significant bytes are dealt with first. The equivalent 6502 program for a 16-bit subtraction is shown in Fig 10.

These ideas can be expanded to any number of bytes and the general principles remain unchanged, but for now we'll turn our attention to the slightly more complicated problem of multiplication and division.

Multiplication

Consider the base 10 product shown below:

```
    2 5     ← Multiplicand
    1 2     ← Multiplier
-----------
  2 5       ← Partial products
    5 0
-----------
3 0 0       ← Result
-----------
```

Let's take this simple product and do the same calculation using base 2 — that is, binary arithmetic:

```
          1 1 0 0 1   ← Multiplicand (25)
            1 1 0 0   ← Multiplier (12)
---------------------
  1 1 0 0 1           ← Partial products
    1 1 0 0 1
      0 0 0 0 0
        0 0 0 0 0
---------------------
1 0 0 1 0 1 1 0 0     ← Result (300)
---------------------
```

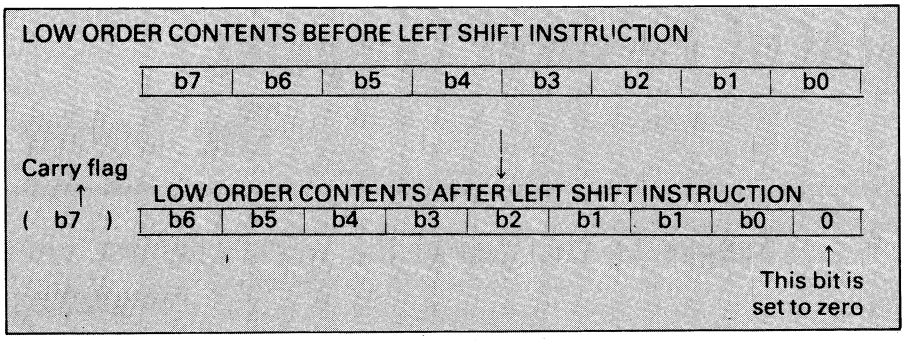


Fig 11 Normal left shift on low order byte

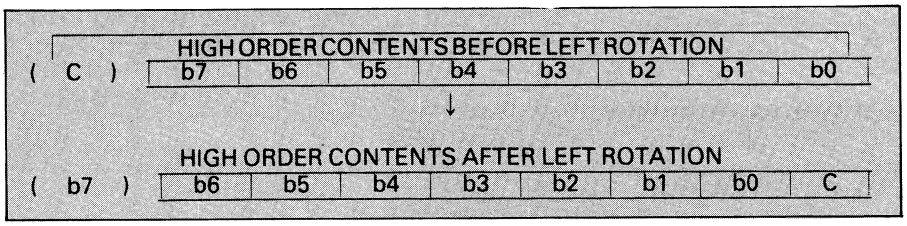


Fig 12 Rotation to the left

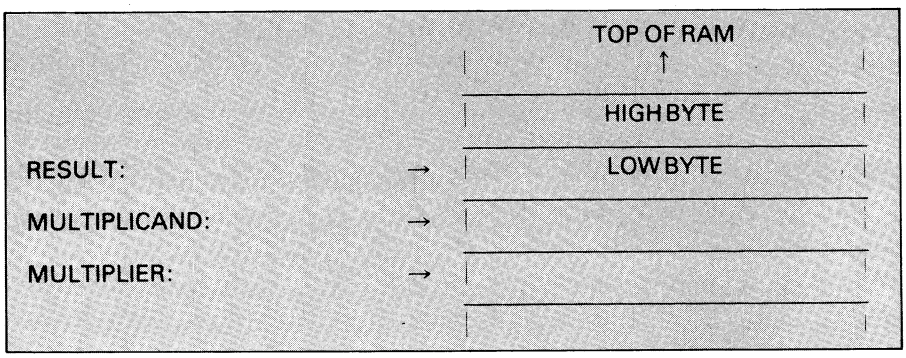


Fig 13 Layout in memory of 8-bit multiplication

The important point is that the partial products are either zeros, or a 'shifted' version of the multiplicand; we can use this knowledge to devise an algorithm for binary multiplication. For each 'Bit' in the multiplier, we ask: 'Is this bit set to 1?' If it is, we add the shifted equivalent of the multiplicand to the result. Two approaches are possible: we can either 'left shift' the multiplicand during the operations, or we can 'right shift' the bytes or registers that are storing the result.

Before showing some typical code for an 8-bit multiplication, we need to understand the general ideas behind creating '16-bit shifts'. Generally, the left shift operations available on our microprocessors will push bit7 into the carry flag. When attempting to left shift a 16-bit (2-byte) value, we can use a normal left shift on the low order byte as shown in Fig 11.

Bit7 falls into the carry flag, and to obtain a 16-bit shift we must shift this bit, now in the carry flag, into bit8 of the 16-bit number. In other words, we want to push this carry value into bit 0 of the high order byte. We need an instruction that performs a left shift and includes the carry, and the most commonly implemented instructions that perform this are rotation instructions. Rotation to the left has the effect shown in Fig 12.

By utilising a combination of left shift on the low order byte and a left rotation (through the carry) on the high order byte, we can left shift a 16-bit number held in two bytes or in two 8-bit registers; the principles can be extended to any number of bytes as required. Instructions are usually available for the equivalent right shifts and right rotations. Occasionally, you will find 'tricks' being used to create 16-bit left shifts. One favourite on the Z80 is to use the double-register addition instructions to add a register pair to itself. For example, ADD HL,HL results in a 16-bit arithmetic left shift.

Let's see how these ideas help to produce a simple multiplication program that takes an 8-bit number held in a location labelled MULTIPLICAND, mul-

```
         LD     HL,MULTIPLIER     ;HL points to multiplier
         LD     C,(HL)            ;Get multiplier in C register
         LD     B,8               ;B is used as a 'bit' counter
         INC    HL                ;Now HL points to
                                   multiplicand
         LD     E,(HL)            ;Get multiplicand in E
                                   register
         LD     D,0               ;Now DE = multiplicand !
         LD     HL,0              ;HL will be used to hold
                                   result
-------------------------------------------------------------
MULTIPLY: SRL   C                 ;Least sig but (multiplier)
                                   into carry
         JR     NC,SKIP           ;Indicates least sig bit is
                                   zero
         ADD    HL,DE             ;Add partial product to
                                   result
SKIP     SLA    E                 ;Left shift multiplicand low
                                   byte
         RL     D                 ;Left rotate high byte
                                   through carry
         DEC    B                 ;Decrease bit counter
         JP     NZ,MULTIPLY       ;Do next bit
         LD     (RESULT),HL       ;Store result
```

Fig 14 Z80 8-bit multiplication

```
         LD     HL,(MULTIPLIER—1) ;Get multiplier in H register
         LD     L,0               ;Clear to zero
         LD     B,8               ;B is used as a 'bit' counter
         LD     DE,MULTIPLICAND   ;Get multiplicand in E
                                   register
         LD     D,0               ;Now DE = multiplicand !
-------------------------------------------------------------
MULTIPLY: ADD   HL,HL             ;16-bit left shift
         JR     NC,SKIP           ;Indicates least sig bit is
                                   zero
         ADD    HL,DE             ;Add partial product to
                                   result
SKIP:    DJNZ   MULTIPLY          ;Do next bit
         LD     (RESULT),HL       ;Store result
```

Fig 15 Z80 8-bit multiplication version two

```
         LXI    H,(MULTIPLIER—1)  ;Get multiplier in H register
         MVI    L,0               ;Clear to zero
         MVI    B,8               ;B is used as a 'bit' counter
         LXI    D,MULTIPLICAND    ;Get multiplicand in E
                                   register
         MVI    D,0               ;Now DE = multiplicand!
-------------------------------------------------------------
MULTIPLY: DAD   H                 ;16-bit left shift
         JNC    SKIP              ;Indicates least sig bit is
                                   zero
         DAD    D                 ;Add partial product to
                                   result
SKIP:    DCR    B                 ;Decrease counter
         JNZ    MULTIPLY          ;Do next bit
         SHLD   RESULT            ;Store result
```

Fig 16 8080 8-bit multiplication

tiplies it by a second number held in location MULTIPLIER, and places the result into the two bytes starting from the lowest byte, which has been labelled RESULT (Fig 13).

Z80 multiply

The code in Fig 14 is split into two parts. Firstly, we load the registers with the following data: HL is loaded with the address of the multiplier, and register C is then loaded with the multiplier itself (using indirect addressing through HL). A 'bit count' of eight is loaded into the B register, and this will be used to count how many times we have gone through the 'multiplication loop'. The HL pair are then incremented so that they point to the multiplicand, which is placed in the E register using a LD E,(HL) instruction. Register D is set to zero because, although the multiplicand is only eight bits, we'll need 16 bits available as in the 16-bit left shift operation explained earlier. Finally, HL is set to zero and will be used to collect the result prior to storing it in locations RESULT and RESULT+1.

The second section of code is the actual multiplication. We use a right shift operation on the C register so that the least significant bit goes into the carry. This means that if the carry becomes 'set', then the least significant bit was a '1'. The carry flag is tested and if it has not been set, the partial product is zero and we skip the addition. Before moving on to the start of the loop again, the DE pair are shifted using a left shift followed by a left rotation, and the 'bit counter' B is decreased. If B is not zero we repeat the loop again, otherwise the final result is stored in RESULT and RESULT+1.

This 'first attempt' code can be shortened and improved in several ways. The Z80 has a combined 'decrement and relative jump on not zero' instruction. It operates using the B register as the counter and decreases the B register by 1, and if B<>0, the relative jump is performed. Another improvement is also possible, but is less obvious. If the Multiplier is placed

```
         LDA  #0
         STA  RESULT
LDX      #8
MULTIPLY: LSR MULTIPLIER
         BCC  SKIP
         CLC
         ADC  MULTIPLICAND
SKIP:    ROR  A
         ROR  RESULT
         DEX
         BNE  MULTIPLY
         STA  RESULT+1
```

Fig 17 6502 8-bit multiplication

in the H register and the L register set to zero, the instruction ADD HL,HL will perform a 16-bit left shift. As the multiplier is shifted out during processing, we create room to store the result in HL.

To take advantage of this arrangement we must shift the multiplier to the LEFT, meaning that we deal with the most significant partial product first. We can also 'tighten up' the initial loading code by loading HL as a register pair starting one byte *below* the multiplier (so that the multiplier goes into the H register). The L register can be cleared after this 16-bit load in readiness for receiving the result. A similar 'trick' can be used to load the multiplicand into the E register.

These improvements have been made in the version shown in Fig 15.

Multiplication 8080

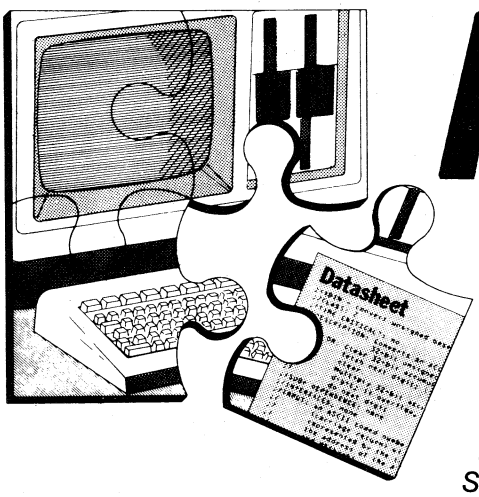
Translation to 8080 form is straightforward. All the improvements made in the second Z80 version can be implemented on the 8080 except for the automated DJNZ instruction. Relative jumps are not supported, so normal jump instructions are used in the loop (Fig 16).

6502 multiply

On the 6502, we cannot use any 16-bit 'paired registers', but we can create similar effects by considering the accumulator as the high byte of such a pair, and a memory location as the equivalent low byte. Such a combination can be shifted in the same way as explained earlier. The X register can be utilised as a 'bit counter', and an LSR (logical shift right) instruction can be used to push the least significant bits of the multiplier into the carry flag; this is used to decide whether or not to add the multiplicand.

In the example shown in Fig 17 the multiplicand is not shifted, it is just added to the accumulator. We right shift the 'accumulator memory byte' 16-bit pair using ROR instructions, and this provides an equivalent alternative.

Did you try the left shift experiment suggested last month? If you did, you will have found that shifting a number to the left is equivalent to multiplying the number by 2. Similarly, two left shifts are equivalent to multiplying by 4. In general, an 'n bit' left shift will multiply the value by 2 raised to the power 'n'. END



APC SUBSET

Alan Tootill and David Barrow present more useful assembler language subroutines. This is your chance to build a library of general-purpose routines, documented to the standard we have developed together in this series. You can contribute a Datasheet, improve or develop one already printed or translate the implementation of a good idea from one processor to another. APC will pay for those contributions that achieve Datasheet status. Contributions (for any of the popular processsors) should be sent to SUB SET, 77 Glenhuntly Road, Elwood, Vic 3184.

Z80 block relocation

Several of you liked R Cath's 6502 block relocation routine RRL (misprinted as 6802 Block relocation, November '83) well enough to implement it for other processors. First, a couple of errors. LDA M9 should be inserted at the 17th. instruction. The 41st. instruction should have been BBCRRL3. We apologise to the BBC.

For the Z80, we like this effort from Mark Wenham. His ROLR moves a block to either a lower or higher memory address using the Z80 block move instructions. One of its best features is its simple, unambiguous input requirements; the address of the first byte to be moved, the address of the last byte to be moved and the address to which the block is to be moved. Where the block is moved to a higher memory address, memory below the destination is moved down to make way for it and where the block is moved to a lower memory address, memory above the destination is moved up.

DATASHEET

```
;= ROLR—Block relocation.
;/CLASS: 2  (Could be made CLASS 1 by preserving the A
            register and flags, since they are not needed to
            convey information to or from the routine.)
;/ TIME CRITICAL ?: No
;/ DESCRIPTION: Rotates a block of memory up or down
                 memory.
;/ ACTION: Not given
;/ SUBr DEPENDENCE: None
;/ INTERFACES: None
;/ INPUT:  BC = address of first byte of block to be moved
;/         HL = address of last byte of block to be moved
;/         DE = address block to be moved to. (HL = BC
            is OK)
;/ OUTPUT:  The block is rotated.
;/ REGs USED:  AF, BC, DE, HL
;/ STACK USE:  6
;/ LENGTH:  68
;/ PROCESSOR:  Z80
ROLR: AND A       ;clear carry                    A7
      SBC HL,DE   ;compare last address of        ED 52
      ADD HL,DE   ;block with destn. address      19
      JR C,UP     ;if destination higher, jump    38 OE
DOWN: PUSH HL     ;else save end of block         E5
      SBC HL,BC   ;get length of block            ED 42
      PUSH HL     ;and save it                    E5
      ADD HL,BC   ;                               09
      SBC HL,DE   ;get total bytes between        ED 52
      EX (SP),HL  ;end of block & destination     E3
      POP BC      ;into BC                        C1
      POP DE      ;DE = address of end of blk     D1
      INC HL      ;HL = length of blk to be       23
      SCF         ;moved. Set 'down' flag         37
      JR MOVE     ;                               18 11
UP:   AND A       ;clear carry                    A7
      INC HL      ;                               23
      SBC HL,BC   ;get length of block            ED 42
      PUSH HL     ;save it                        E5
      EX DE,HL    ;destination = destination +    EB
      ADD HL,DE   ;length of blk to be moved      19
      EX DE,HL    ;                               EB
      ADD HL,BC   ;HL = end of block              09
      EX DE,HL    ;HL = destination               EB
      SBC HL,BC   ;get total no. of bytes         ED 42
      LD D,B      ;between destn. & blk start     50
      LD E,C      ;DE = start of block            59
      EX (SP),HL  ;HL = length of block           E3
      POP BC      ;BC = total bytes +1            C1
      DEC BC      ;                               0B
MOVE: DEC HL      ;decrement counter              2B
      PUSH HL     ;save counter                   E5
      PUSH DE     ;save start or end of block     D5
      PUSH BC     ;save no. of bytes to be        C5
      LD H,D      ;moved                          62
      LD L,E      ;                               6B
      LD A,(DE)   ;get byte                       1A
      JR C,LWR    ;jump if 'down' flag set        38 05
      INC HL      ;else move rest of block &      23
      LDIR        ;bytes between block and        ED B0
      JR INS      ;destination down               18 03
LWR:  DEC HL      ;or move rest of blk & bytes    2B
      LDDR        ;from blk & destination up      ED B8
INS:  LD (DE),A   ;insert byte                    12
      POP BC      ;                               C1
      POP DE      ;                               D1
      POP HL      ;                               E1
      PUSH AF     ;save flags                     F5
      LD A,H      ;test counter                   7C
      OR L        ;                               B5
      JR Z,END    ;                               28 03
      POP AF      ;restore flags                  F1
      JR MOVE     ;and continue                   18 E3
END:  POP AF      ;restore stack                  F1
      RET         ;and return                     C9
```

```
  : DR.75*SINM,75*COSM
  : P.TAB(30,8)h":"jmlM
120
130 PRINT''"Clock set"''"Press KEY0 to display clock"
140 PRINT''"You may delete the program."
```

HIRES SCREEN DUMPS FOR 1520 PLOTTER

Hires screen dumps are possible on a 1520 plotter, with this routine, using a VIC 20 and super expander. It should also work with a Commodore 64 and Simon's Basic if it has the RDOT function or something similar to test whether a point is on or off.

S Mills

```
63010 FOR Y=6 TO 1023 STEP 6
63020 FOR X=6 TO 1023 STEP 6
63030 IF RDOT(X,Y)<>1 AND F=0 THEN F=1:SX=X:SY=Y
63040 IF RDOT(X,Y)=1 AND F=1 THEN F=0:GOSUB 63200
63050 NEXT
63060 IF F=1 THEN GOSUB 63200:F=0
63070 NEXT
63080 END
63200 OPEN 1,6,1:PRINT#1,"M",SX/3,-SY/3
63210 PRINT#1,"D",X/3,-Y/3:CLOSE1:RETURN
```

BACKGROUND VZ

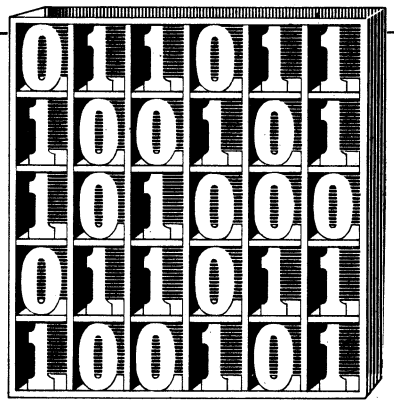
One of the limitations of the VZ-200 is that it has only two background colours in each mode: green and orange in mode 0, buff and green in mode 1. This short machine code program fills the screen with any desired character in either mode 0 or 1, making any of the eight foreground colours available as a background.

To use the program just type in the listing, either at the start of another program or on its own, and CSAVE it. RUN the program and, to fill the screen, POKE the code for the desired character into location 28672 (start of screen address) and enter PRINT USR(0). In mode 1 and colour 0, 0 gives a green background, 85 gives yellow, 170 blue and 255 gives a red background. In mode 1, colour 1, buff = 0, cyan = 85, 170 = orange and 255 = magenta.

I Williams

Basic listing:

```
10 TM=PEEK(30897)+256*PEEK(30898)-20
20 POKE 30897,TM-INT(TM/256)*256:POKE
   30898,INT(TM/256)
30 TM=TM-1:A=TM-65536
40 FOR I=0 TO 15
50 READ D:POKE I+A,D
60 NEXT I
70 POKE 30862,TM-INT(TM/256)*256:POKE
   30863,INT(TM/256)
80 DATA 58,0,112,71,33,0,112,17,0,120,112,35,
   223,32,251,201
```



SUBSET

David Barrow presents more documented machine code routines and useful information for the assembly language programmer. If you have a good routine, an improvement or conversion of one already printed, or just a helpful programming hint, then send it in and share it with other programmers. Subroutines for any of the popular processors and computers are welcome but please include full documentation. All published code will be paid for. Send your contributions to Sub Set, APC, 77 Glenhuntly Road, Elwood, Victoria 3184.

Z80 FLOATING POINT

As a somewhat delayed follow-up to his conversion of a 32-bit integer arithmetic suite from Z80 to 6502 code, Dennis May provides Subset with a Z80 suite to perform floating point binary arithmetic.

The suite acts on arguments anywhere in memory, indexed by the Z80's two 16-bit index registers, IX and IY. It returns the result in the six registers BCDEHL, ready for storage anywhere. The zero and carry flags are used to convey division by zero and overflow error information.

Each number in the format required by the suite is contained in six bytes. The first byte is the exponent and the second is the sign byte of which only bit 7 is actually used. The remaining four bytes form the mantissa, or fraction. The 32-bit precision is about the equivalent of 9.5 decimal digits. The range of magnitude made available by the format is $1 * 2^{-128}$ (about 0.29E-38) to 4,294,967,295 * 2^{95} (about 0.17E39).

To facilitate the possible use of 16-bit arithmetic, the mantissa is stored with the low-order byte in the third byte of the number, high order in the sixth byte. The mantissa is normalised so that for a non-zero number, bit 31 is always set. (Anyone wishing to try and improve upon the suite might like to utilise this fact — the sign bit can be stored in the high order bit of the mantissa, cutting variable storage by 1/6.)

The exponent is held as an 'excess 128' value; this is similar to the 'two's complement' notation with which you should be familiar but has 128 (80H) as the zero exponent value with lower values negative and higher ones positive. The advantage of excess 128 over the familiar method of indicating signed numbers is that zero — the easiest value to test — signifies either overflow or underflow. Dennis also uses a zero exponent to indicate that a number is zero, making it unnecessary to clear the mantissa and rest the sign.

33-BIT SUBTRACTION

One elegant feature of the suite worth commenting upon is the implementation of 33-bit arithmetic in the division routine FPDIV. If, in any iteration, the 32-bit subtraction of divisor from dividend fails then a borrow is generated. However, the previous iteration shifts the highest dividend bit out to carry and the subtraction would be possible if this bit were set. In fact the borrow in 33-bit arithmetic will be the inverse of the shifted out bit, and the algorithm used by Dennis simply replaces the 32-bit borrow by the complement of the shifted bit.

DATASHEET 1

```
;=============================================================
;= FETCH    Fetch arguments to stack.
;  MOPUP    (MOPDZ, MOPNR, MOPOV, MOPUF) Exit correctly.
;=============================================================
;JOB        FETCH: To copy two arguments to stack work area and
;                  clear result flags.
;           MOPUP: To set correct flags, zeroise result if
;                  necessary, pull results and exit.
;ACTION     See comments.
;-------------------------------------------------------------
;CPU        Z80
;HARDWARE   FETCH: arguments in RAM. MOPUP: none.
;SOFTWARE   FETCH: none. MOPUP: "NORM" and "ROUND".
;-------------------------------------------------------------
;INPUT      FETCH: IX and IY address the two arguments
;           MOPUP: Result in 6-bytes on stack addressed by IX.
;OUTPUT     FETCH: IX, IY, AF on stack. Arguments on stack
;                  indexed by IX. All flags cleared.
;                  A, B, DE and HL changed.
;           MOPUP: Result in BCDEHL. IX, IY and A restored.
;                  Z = 1: division by zero (FPDIV).
;                  Z = 0: C = 1: overflow. C = 0: no error.
;ERRORS     None.
;REG USE    F BC DE HL IX IY (A changed in FETCH).
;STACK USE  16 (MOPUP: -16).
;RAM USE    None.
;LENGTH     FETCH: 48.  MOPUP: 39.
;CYCLES     Not given.
;-------------------------------------------------------------
;CLASS 2    -discreet      *interruptable      *promable
;-*****     *reentrant     *relocatable        *robust
;=============================================================
;
FETCH  POP  HL          ;Get return address in HL.      E1
       PUSH IX          ;Save IX, IY and AF on stack    DD E5
       PUSH IY          ;before stacked numbers, other  FD E5
       PUSH AF          ;registers used for output.     F5
       LD   IX,-12      ;Index stack area below         DD 21 F4 FF
       ADD  IX,SP       ;12-byte workspace and set      DD 39
       LD   SP,IX       ;Stack pointer accordingly.     DD F9
       LD   D,(IX+17)   ;Get stacked IX to DE for       DD 56 11
       LD   E,(IX+16)   ;transfer loop source.          DD 5E 10
       LD   B,6         ;Count for 6-byte numbers.      06 06
;
FETCHL LD   A,(DE)      ;Move byte from 1st number to   1A
       LD   (IX+0),A    ;workspace low address.         DD 77 00
       LD   A,(IY+0)    ;Move byte from 2nd number to   FD 7E 00
       LD   (IX+6),A    ;workspace high address.        DD 77 06
       INC  IX          ;Bump workspace pointer         DD 23
       INC  DE          ;and 1st number pointer         13
       INC  IY          ;and 2nd number pointer.        FD 23
       DJNZ FETCHL      ;Repeat for 6-byte numbers.     10 EF
;
       LD   DE,-6       ;Reset IX to point to start     11 FA FF
       ADD  IX,DE       ;of stack workspace area.       DD 19
       LD   (IX+12),B   ;Clear Z and Cy flags (B=0).    DD 70 0C
       JP   (HL)        ;Return by jumping.             E9
;-------------------------------------------------------------
;
MOPUP                   ;Exit routine with entry points for
                        ;division by zero, overflow, underflow or
                        ;zero result, normalise and round.
;
MOPDZ  SET  6,(IX+12)   ;Set stacked Z flag to show     DD CB 0C F6
       JR   MOPPOP      ;division by zero, and exit.    18 10
MOPNR  CALL NORM        ;Normalise if necessary,        CD lo hi
       CALL ROUND       ;then round, and go exit if     CD lo hi
       JR   NZ,MOPPOP   ;result exponent not zero.      20 08
MOPOV  SET  0,(IX+12)   ;Set stacked Cy= overflow       DD CB 0C C6
MOPUF  LD   (IX+0),0    ;Set exponent 0, zero result.   DD 36 00 00
;
MOPPOP LD   SP,IX       ;Set SP at result, losing any   DD F9
       POP  BC          ;values on stack, and pick up   C1
       POP  HL          ;result in BCDEHL, B = sign,    E1
       POP  DE          ;C = exp, DEHL = mantissa.      D1
       POP  IX          ;Pop 6 bytes of 2nd argument    DD E1
       POP  IX          ;into IX to discard it and      DD E1
```

```
        POP   IX          :move Stack Pointer up to      DD E1
        POP   AF          :saved registers. Restore A    F1
        POP   IY          :and result flags. Restore     FD E1
        POP   IX          :argument pointers IX and IY.  DD E1
        RET               :Exit floating point suite.    C9
;=============================================================
```

DATASHEET 2

```
;=============================================================
:= NORM      Normalise floating point number.
:> SWAP      Swap two contiguous floating point numbers.
:> ROUND     Round floating point number.
:> RSHIFT    Shift floating point number justification.
;=============================================================
:JOB         Set of four utility subroutines acting on floating
:            point mantissa and exponent.
:ACTION      See individual routine comments.
:-------------------------------------------------------------
:CPU         Z80
:HARDWARE    RAM indexed by IX (Stack workspace within suite).
:SOFTWARE    None.
:-------------------------------------------------------------
:INPUT       IX addresses floating point number.
:            C = rounding byte. D = byte above mantissa.
:            B = bit count for RSHIFT.
:OUTPUT      Number normalised, rounded or right shifted (or
:            exchanged with succeeding number in SWAP).
:            C, D and exponents are corrected. B & A changed.
:            Z set to show exponent overflow in ROUND & RSHIFT.
:ERRORS      None.
:REG USE     NORM:  AF C IX.   SWAP:   F B DE IX.
:            ROUND: AF C IX.   RSHIFT: AF B C D IX.
:STACK USE   SWAP: 2. Others: none.
:RAM USE     None.
:LENGTH      NORM: 33. SWAP: 24. ROUND: 36. RSHIFT: 39.
:CYCLES      Not given.
:-------------------------------------------------------------
:CLASS 2     -discreet      *interruptable      *promable
:-*****      *reentrant     *relocatable        *robust
;=============================================================
:
NORM   LD   A(IX+5)   :Test high order bit of        DD 7E 05
       RLA            :mantissa and exit if set      17
       RET  C         :with number normalised.       D8
       LD   A,(IX+0)  :Test input or adjusted        DD 7E 00
       OR   A         :exponent and exit if zero as  B7
       RET  Z         :either zero or underflow.     C8
       SLA  C         :Else shift mantissa           CB 21
       RL   (IX+2)    :left one bit with bit from    DD CB 02 16
       RL   (IX+3)    :rounding byte going in to     DD CB 03 16
       RL   (IX+4)    :mantissa lowest bit.          DD CB 04 16
       RL   (IX+5)    :Then subtract 1 from exponent DD CB 05 16
       DEC  (IX+0)    :to show shift made and loop   DD 35 00
       JR   NORM      :until normalised or zero.     18 DF
:-------------------------------------------------------------
:
SWAP   LD   B,6       :Count for 6-byte numbers.     06 06
SWAPLP LD   E,(IX+0)  :Starting with lowest          DD 5E 00
       LD   D,(IX+6)  :addressed bytes, exchange     DD 56 06
       LD   (IX+0),D  :corresponding bytes in 1st    DD 72 00
       LD   (IX+6),E  :and 2nd arguments, index      DD 73 06
       INC  IX        :next bytes higher and repeat  DD 23
       DJNZ SWAPLP    :for all six bytes.            10 F0
       LD   DE,-6     :Restore IX to point to        11 FA FF
       ADD  IX,DE     :lowest workspace byte.        DD 19
       RET            :Exit, numbers swapped.        C9
:-------------------------------------------------------------
:
ROUND  LD   A,(IX+0)  :Test exponent and             DD 7E 00
       OR   A         :if not zero then okay         B7
       JR   NZ,ROUNDB :but if zero then              20 02
       LD   C,0       :zeroise rounding byte C.      0E 00
ROUNDB LD   A,80H     :Get complement of rounding    3E 80
       XOR  C         :bit so exit made with Z = 0   A9
       RET  M         :if now set (no rounding).     F8
:
       INC  (IX+2)    :Else round up mantissa,       DD 34 02
       RET  NZ        :incrementing each byte in     C0
       INC  (IX+3)    :turn if last byte overflowed  DD 34 03
       RET  NZ        :else exit Z reset if          C0
       INC  (IX+4)    :at any stage the increment    DD 34 04
       RET  NZ        :does not produce zero (i.e.   C0
       INC  (IX+5)    :no carry to next byte).       DD 34 05
       RET  NZ        :If 4th byte incremented to    C0
       SET  7,(IX+5)  :zero then normalise, set top  DD CB 05 FE
       INC  (IX+0)    :bit and increment exponent.   DD 34 00
       RET            :Exit Z set if exp. overflow.  C9
:-------------------------------------------------------------
:
RSHIFT LD   A,B       :Test input shift count        78
       OR   A         :and exit immediately if zero  B7
       JR   Z,RSHEND  :but with Z reset.             28 21
       LD   A,(IX+0)  :Test exponent and exit        DD 7E 00
       OR   A         :immediately if zero but       B7
       JR   Z,RSHEND  :with Z reset.                 28 1B
:
RSHLP  SRL  D         :Else shift mantissa right     CB 3A
       RR   (IX+5)    :by one bit, bit from D        DD CB 05 1E
       RR   (IX+4)    :going to high order bit       DD CB 04 1E
       RR   (IX+3)    :of mantissa and low order     DD CB 03 1E
       RR   (IX+2)    :bit of mantissa shifted       DD CB 02 1E
       RR   C         :out to rounding byte C.       CB 19
       INC  (IX+0)    :Adjust exponent for shift,    DD 34 00
       RET  Z         :exit Z = 1 on overflow,       C8
       DJNZ RSHLP     :else repeat for count.        10 E6
       LD   A,B       :Ensure Z reset on exit.       78
:
RSHEND INC  A         :Make A non-zero so Z reset    3C
       RET            :to show non-overflow on exit. C9
;=============================================================
```

DATASHEET 3

```
;=============================================================
:= ADDM      Add mantissas of two floating point numbers.
:> SUBM      Subtract mantissas of two floating point numbers.
:> NEGM      Negate mantissa of a floating point number.
;=============================================================
:JOB         Set of three arithmetic utility routines acting on
:            the 4-byte mantissas of floating point numbers.
:ACTION      See individual routine comments.
:-------------------------------------------------------------
:CPU         Z80
:HARDWARE    RAM indexed by IX (Stack workspace within suite).
:SOFTWARE    None.
:-------------------------------------------------------------
:INPUT       IX addresses floating point number(s).
:            (C = rounding byte in NEGM).
:OUTPUT      IX unchanged. A changed. Cy = carry or borrow out.
:            ADDM: Sum stored in 1st number mantissa.
:            SUBM: Difference stored in 1st number mantissa.
:                  B = 0 if difference = 0.
:            NEGM: 2's complement negation stored in 1st number
:                  mantissa and rounding byte C.
:                  Cy = 0 if negation = 0, else Cy = 1.
:ERRORS      None.
:REG USE     AF B IX (also C in NEGM).
:STACK USE   4
:RAM USE     None.
:LENGTH      ADDM: 26. SUBM: 32. NEGM: 27.
:CYCLES      Not given.
:-------------------------------------------------------------
:CLASS 2     -discreet      *interruptable      *promable
:-*****      *reentrant     *relocatable        *robust
;=============================================================
:
ADDM   PUSH DE        :Save DE for use in ADDM.      D5
       LD   B,4       :Set 4-byte mantissa count.    06 04
       OR   A         :Clear Cy initially.           B7
:
ADDMLP LD   A,(IX+2)  :Loop adding corresponding     DD 7E 02
       ADC  A,(IX+8)  :bytes of mantissa, storing    DD 8E 08
       LD   (IX+2),A  :result in number at IX+0.     DD 77 02
       INC  IX        :Repeat for all four bytes     DD 23
       DJNZ ADDMLP    :of mantissa.                  10 F3
:
       PUSH AF        :Save flags while              F5
       LD   DE,-4     :restoring IX to point to      11 FC FF
       ADD  IX,DE     :1st byte of 1st number.       DD 19
       POP  AF        :Restore flags,                F1
       POP  DE        :and DE then exit with         D1
       RET            :mantissa added.               C9
:-------------------------------------------------------------
:
SUBM   PUSH DE        :Save DE for use in SUBM.      D5
       LD   B,4       :Set 4-byte mantissa count.    06 04
       XOR  A         :Clear borrow initially and    AF
       LD   D,A       :clear zero indicator D.       57
:
SUBMLP LD   A,(IX+2)  :Subtract corresponding bytes  DD 7E 02
       SBC  A,(IX+8)  :of 2nd number mantissa from   DD 9E 08
       LD   (IX+2),A  :1st with result to 1st.       DD 77 02
       PUSH AF        :Save borrow and               F5
       OR   D         :merge any set bits into       B2
       LD   D,A       :zero indicator.               57
       POP  AF        :Restore borrow, then index    F1
       INC  IX        :next bytes and repeat for     DD 23
       DJNZ SUBMLP    :all four bytes of mantissa.   10 EF
:
       PUSH AF        :Save flags and transfer zero  F5
       LD   B,D       :indicator to B. Then          42
       LD   DE,-4     :restore IX to point to        11 FC FF
       ADD  IX,DE     :1st byte of 1st number.       DD 19
       POP  AF        :Restore flags,                F1
       POP  DE        :and DE then exit with         D1
       RET            :mantissa subtracted.          C9
:-------------------------------------------------------------
:
NEGM   PUSH DE        :Save DE for use in NEGM.      D5
       LD   B,4       :Set 4-byte mantissa count.    06 04
       XOR  A         :Clear Accumulator and Cy.     AF
       SUB  C         :Negate rounding byte in C     91
       LD   C,A       :with possible carry to loop.  4F
:
NEGMLP LD   A,0       :Loop subtracting each byte    3E 00
       SBC  A,(IX+2)  :of mantissa and carry from    DD 9E 02
       LD   (IX+2),A  :previous byte from zero.      DD 77 02
```

```
        INC  IX         ;2's complement result back     DD 23
        DJNZ NEGMLP     ;to number at IX+0.             10 F4

        PUSH AF         ;Save flags while               F5
        LD   DE,-4      ;restoring IX to point to       11 FC FF
        ADD  IX,DE      ;1st byte of 1st number.        DD 19
        POP  AF         ;Restore flags,                 F1
        POP  DE         ;and DE then exit with          D1
        RET             ;mantissa negated.              C9
;***************************************************************
```

DATASHEET 4

```
;***************************************************************
:= FPSUB    Floating point subtraction.
:> FPADD    Floating point addition.
;***************************************************************
;JOB         To perform addition or subtraction on two floating
;           point numbers held in memory, returning a valid
;           result in registers or error information in flags.
;ACTION      Move numbers to stack workspace.
;           IF either number = 0 THEN: [ Return other number.]
;           Equalise exponents, justifying mantissas.
;           Add/subtract mantissas with any necessary negation,
;           justification, sign change or exponent zeroising.
;           Exit, result to registers, setting correct flags.


;---------------------------------------------------------------
;CPU         Z80
;HARDWARE    Memory containing the two numbers.
;SOFTWARE    FETCH MOPUP SWAP RSHIFT ADDM SUBM NEGM
;---------------------------------------------------------------

;INPUT       IX addresses 1st number, IY 2nd number in memory.
;OUTPUT      IX, IY and indexed numbers unchanged. Z = 0.
;           Cy = 0: Result in BCDEHL. Cy = 1: Overflow.
;ERRORS      None.
;REG USE     F BC DE HL IX IY
;STACK USE   22
;RAM USE     None.
;LENGTH      95
;CYCLES      Not given.

;---------------------------------------------------------------
;CLASS 2     -discreet       *interruptable      *promable
;-*****      *reentrant      *relocatable        *robust
;***************************************************************

;
FPADD   CALL FETCH      ;Move arguments to workspace   CD lo hi
        JR   FPAS       ;on stack. Go to common part.  18 0B
;

 FPSUB  CALL FETCH      ;Get arguments to workspace    CD lo hi
        LD   A,-1       ;and negate sign of second     3E FF
        XOR  (IX+7)     ;number so add/sub operation   DD AE 07
        LD   (IX+7),A   ;is essentially the same.      DD 77 07
;
FPAS    LD   A,(IX+0)   ;Get and test 1st exponent     DD 7E 00
        OR   A          ;for number = 0. If so, move   B7
        CALL Z,SWAP     ;2nd number to result place.   CC lo hi
        LD   A,(IX+6)   ;Get and test 2nd exponent.    DD 7E 06
        OR   A          ;If zero, return 1st number    B7
        JP   Z,MOPPOP   ;as sum or difference.         CA lo hi
;
        SUB  (IX+0)     ;Subtract exponents to give    DD 96 00
        JR   NC,FPASEQ  ;shifts needed to equalise     30 05
        NEG             ;exponents, swapping numbers   ED 44
        CALL SWAP       ;if necessary.                 CD lo hi
FPASEQ  LD   C,0        ;Clear rounding byte and       0E 00
        LD   D,C        ;shift-in byte.                51
        LD   B,A        ;Move shift count to B and     47
        CALL RSHIFT     ;shift if needed.              CD lo hi
;
        LD   A,(IX+1)   ;Compare signs. If same then   DD 7E 01
        XOR  (IX+7)     ;addition of mantissas is      DD AE 07
        RLA             ;needed but subtraction if     17
        JR   C,FPMSUB   ;signs are different.          38 10
;
        CALL ADDM       ;Add mantissas. Prepare for    CD lo hi
        LD   B,1        ;shifting carry out of add     06 01
        LD   D,1        ;back into mantissa.           16 01
        CALL C,RSHIFT   ;Shift any carry back in.      DC lo hi
        JP   Z,MOPOV    ;Exit if exponent overflow.    CA lo hi
        JP   MOPNR      ;Normalise, round & exit.      C3 lo hi
;
FPMSUB  CALL SUBM       ;Subtract mantissas and,       CD lo hi
        PUSH AF         ;saving borrow, test           F5
        LD   A,B        ;if difference is zero,        78
        OR   A          ;if so then exit clearing      B7
        JP   Z,MOPUF    ;exponent to show number = 0.  CA lo hi
        POP  AF         ;restore borrow and negate     F1
        CALL C,NEGM     ;result if borrow generated.   DC lo hi
        RRA             ;Use borrow to set correct     1F
        XOR  (IX+1)     ;result sign, negating sign    DD AE 01
        LD   (IX+1),A   ;only if Cy was set.           DD 77 01
        JP   MOPNR      ;Normalise, round & exit.      C3 lo hi
;***************************************************************
```

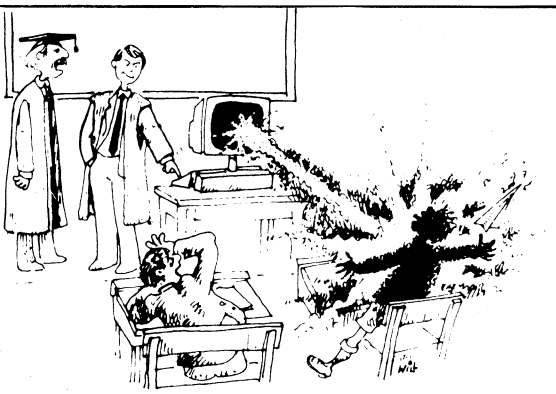
DATASHEET 5

```
:============================================================
:= FPMUL    Floating point multiplication.
:============================================================
:JOB        To perform multiplication on two floating point
:           numbers held in memory, returning a valid product in
:           registers or error information in flags.
:ACTION     Move numbers to stack workspace.
:           IF either number = 0 THEN: [ Return zero.]
:           Add exponents, exiting on overflow or underflow.
:           Product sign = multiplier XOR multiplicand signs.
:           Multiply mantissas.
:           Normalise, round and exit, product to registers.
:------------------------------------------------------------
:CPU        Z80
:HARDWARE   Memory containing the two numbers.
:SOFTWARE   FETCH MOPUP ADDM
:------------------------------------------------------------
:INPUT      IX addresses multiplier, IY multiplicand in memory.
:OUTPUT     IX, IY and indexed numbers unchanged. Z = 0.
:           Cy = 0: Result in BCDEHL. Cy = 1: Overflow.
:ERRORS     None.
:REG USE    F BC DE HL IX IY
:STACK USE  24
:RAM USE    None.
:LENGTH     110
:CYCLES     Not given.
:------------------------------------------------------------
:CLASS 2     -discreet      *interruptable      *promable
:-*****      *reentrant     *relocatable        *robust
:============================================================
:
FPMUL  CALL FETCH     :Move numbers to workspace on   CD lo hi
       LD   A,(IX+0)  :stack and test exponent of     DD 7E 00
       OR   A         :multiplier, returning it as    B7
       JP   Z,MOPPOP  :product if zero.               CA lo hi
       LD   A,(IX+6)  :Test multiplicand exponent     DD 7E 06
       OR   A         :and exit with zero product     B7
       JP   Z,MOPUF   :if it is zero.                 CA lo hi
:
       ADD  A,(IX+0)  :Else add exponents, use Cy     DD 86 00
       JR   C,FPMPOS  :and S to determine if add      38 05
       JP   P,MOPUF   :gives underflow or overflow    F2 lo hi
       JR   FPMEXS    :and if so, exit with correct   18 03
FPMPOS JP   M,MOPOV   :zero result/overflow flag.     FA lo hi
FPMEXS ADD  A,80H     :Restore excess 128 exponent    C6 80
       LD   (IX+0),A  :to product exponent byte.      DD 77 00
 :
       LD   A,(IX+1)  :Exclusive-OR multiplier        DD 7E 01
       XOR  (IX+7)    :and multiplicand signs to      DD AE 07
       LD   (IX+1),A  :give product sign.             DD 77 01
      LD   L,(IX+2)  :Move multiplier mantissa        DD 6E 02
      LD   H,(IX+3)  :low-word to (SP)                DD 66 03
      PUSH HL        :and high-word to HL             E5
      LD   L,(IX+4)  :for shifting in                 DD 6E 04
      LD   H,(IX+5)  :multiplication loop.            DD 66 05
:
       LD   C,0       :Zeroise rounding byte          0E 00
       LD   (IX+2),C  :and product accumulator        DD 71 02
       LD   (IX+3),C  :(replacing multiplier          DD 71 03
       LD   (IX+4),C  :in workspace).                 DD 71 04
       LD   (IX+5),C  :                               DD 71 05

       LD   D,32      :Set 32 bit count.              16 20
:
FPMLP  SRL  H         :Long multiplication by shift   CB 3C
       RR   L         :and addition at correct bit    CB 1D
       EX   (SP),HL   :place if multiplier bit set.   E3
       RR   H         :Shift right next multiplier    CB 1C
       RR   L         :bit through HL and (SP)        CB 1D
       EX   (SP),HL   :into Cy.                       E3
       CALL C,ADDM    :If set, add multiplicand       DC lo hi
       RR   (IX+5)    :and shift partial product      DD CB 05 1E
       RR   (IX+4)    :right to next bit place,       DD CB 04 1E
       RR   (IX+3)    :getting any carry from         DD CB 03 1E
       RR   (IX+2)    :addition. Rounding byte        DD CB 02 1E
       RR   C         :takes shift out bits.          CB 19
       DEC  D         :Repeat for all 32 bits of      15
       JR   NZ,FPMLP  :multiplier.                    D0 DE
       JP   MOPNR     :Normalise, round & exit.       C3 lo hi
:============================================================
```

DATASHEET 6

```
:============================================================
:= FPDIV    Floating point division.
:============================================================
:JOB        To perform division on two floating point numbers
:           held in memory, returning a valid quotient in
:           registers or error information in flags.
:ACTION     Move numbers to stack workspace.
:           IF divisor = 0 THEN: [ Exit, division by zero. ]
:           IF dividend = 0 THEN: [ Return zero.]
:           Subtract exponents, exit on overflow or underflow.
:           Quotient sign = dividend XOR divisor signs.
:           Divide mantissas.
:           Normalise, round and exit, quotient to registers.
:------------------------------------------------------------
:CPU        Z80
:HARDWARE   Memory containing the two numbers.
:SOFTWARE   FETCH MOPUP SUBM ADDM
:------------------------------------------------------------
:INPUT      IX addresses dividend, IY divisor in memory.
:OUTPUT     IX, IY and indexed numbers unchanged.
:           Z = 1: division by zero.
:           Z = 0: Cy = 0: Result in BCDEHL. Cy = 1: Overflow.
:ERRORS     None.
:REG USE    F BC DE HL IX IY
:STACK USE  24
:RAM USE    None.
:LENGTH     114
:CYCLES     Not given.
:------------------------------------------------------------
:CLASS 2     -discreet      *interruptable      *promable
:-*****      *reentrant     *relocatable        *robust
:============================================================
:
FPDIV  CALL FETCH     :Move numbers to workspace on   CD lo hi
       LD   A,(IX+6)  :stack and test exponent of     DD 7E 06
       OR   A         :divisor. If zero then error    B7
       JP   Z,MOPDZ   :exit - division by zero.       CA lo hi
       LD   A,(IX+0)  :Test dividend exponent, if     DD 7E 00
       OR   A         :zero then return valid         B7
       JP   Z,MOPUF   :zero quotient.                 CA lo hi
:
       SUB  (IX+6)    :Subtract exponents, use Cy     DD 96 06
       JR   NC,FPDPOS :and S to test if subtraction   30 05
       JP   P,MOPUF   :gives underflow/overflow,      F2 lo hi
       JR   FPDEXS    :if so exit with correct        18 03
FPDPOS JP   M,MOPOV   :zero quotient/overflow flag.   FA lo hi
FPDEXS ADD  A,81H     :Correct excess 128 exponent    C6 81
       JP   Z,MOPOV   :+1 for bit normalisation,      CA lo hi
       LD   (IX+0),A  :exiting if overflow.           DD 77 00
:
       LD   A,(IX+1)  :Exclusive-OR dividend and      DD 7E 01
       XOR  (IX+7)    :divisor sign to give quotient  DD AE 07
       LD   (IX+1),A  :sign, replacing dividend.      DD 77 01
       LD   HL,0      :Clear quotient mantissa in     21 00 00
       PUSH HL        :(SP) and HL, clear rounding    E5
       LD   C,H       :byte C. Set D as 32-bit count  4C
       LD   D,34      :+ 1 normalisation + 1 round.   16 22
:
FPDLP  RL   E         :Save hi-bit shifted out of     CB 13
       CALL SUBM      :dividend and subtract divisor  CD lo hi
       JR   NC,FPDAD  :skip if subtraction went ok    30 03
       RR   E         :else correct for 33-bit        CB 1B
       CCF            :subtraction. If still no go    3F
FPDAD  CALL C,ADDM    :then add divisor back and      DC lo hi
       JR   C,FPDRS   :skip quotient bit setting.     38 02
       SET  5,C       :Set correct place bit.         CB E9
:
FPDRS  SLA  C         :Shift partial quotient and     CB 21
       ADC  HL,HL     :rounding byte in               ED 6A
       EX   (SP),HL   :(SP), HL and C one bit left    E3
       ADC  HL,HL     :to next bit place.             ED 6A
       EX   (SP),HL   :                               E3
       SLA  (IX+2)    :Shift dividend left            DD CB 02 26
       RL   (IX+3)    :by one bit to next bit place   DD CB 03 16
       RL   (IX+4)    :top bit out to Cy.             DD CB 04 16
       RL   (IX+5)    :Repeat for 34 bits             DD CB 05 16
       DEC  D         :needed to ensure correct       15
       JR   NZ,FPDLP  :normalised, rounded quotient.  20 D4
:
       LD   (IX+2),L  :Move quotient mantissa         DD 75 02
       LD   (IX+3),H  :from (SP) and HL to            DD 74 03
       POP  HL        :replace dividend in stack      E1
       LD   (IX+4),L  :workspace as raw quotient      DD 75 04
       LD   (IX+5),H  :then exit to normalise         DD 74 05
       JP   MOPNR     :and round.                     C3 lo hi
:============================================================
```



'In my day, Mr Hobson, we used to throw chalk at them!'

PRETTY PC PICS

This program draws seven mathematically based graphs which have been converted from an old text-based Basic to the graphics of the IBM PC. This program requires the color graphics adapter to work.

M Hindson

```
1  SCREEN 1,1:W=7:CLS:KEY OFF
2 FOR COUNT=1 TO W:ON COUNT GOSUB 12,13,14,15,16,17,18
3 FOR X=-30 TO 30 STEP 1.5
4 C=0
5 Y1=5*INT(SQR(900-X^2)/5)
6 FOR Y=Y1 TO -Y1 STEP -5
7 Z=INT(25+FNA(SQR(X^2+Y^2))-.7*Y)+50
8 IF Z<=C THEN 10
9 C=X:PSET (Z*2,(X+50)*2)
10 NEXT Y,X
11 LOCATE 20,27:PRINT"Pattern No.";COUNT:Q$=INPUT$(1):CLS:NEXT:END
12 DEF FNA(Z)=30*(COS(Z/16))^2:RETURN
13 DEF FNA(Z)=30*EXP(-Z*Z/100):RETURN
14 DEF FNA(Z)=SQR(900.01-Z*Z)*.9-2:RETURN
15 DEF FNA(Z)=30-30*SIN(Z/18):RETURN
16 DEF FNA(Z)=30*EXP(-COS(Z/16))-30:RETURN
17 DEF FNA(Z)=30*SIN(Z/10):RETURN
18 DEF FNA(Z)=EXP(COS(Z/4.8)*COS(Z/2))*30:RETURN
```

BIG BINOMIALS

Part of the formula for calculating a binomial distribution is N!/X!(N-X)!. However if N,X or N-X are much larger than 70 most computers will crash due to numeric overload. The following algorithm eliminates the need for calculating these large factorials. It assumes that N and X are positive intergers and that N is greater than X.

S Sullivan

```
100 INPUT N,X
110 T=1
120 FOR A=N-X+1 TO N
130 T=T*A/(A-N+X)
140 NEXT A
150 PRINT T
160 END
```

STOP CHATTERING WHILE READING PROGRAMS

When loading commercial software, the 1541 disk drive often makes a loud chattering noise while reading the program. This is usually caused by a copy protection technique, whereby the disk drive is forced to read a bad track or sector on the disk.

This repeated vibration of the drive's head could eventually damage the drive, or cause the stepper motor assembly to slip out of alignment.

To protect your disk drive, here is a short program to prevent bumping when going to track one. This is performed by the memory write command in the DOS.

To execute it from Basic, type in the one line program in command mode (ie without a line number) and press Return. It should be entered before loading a commercial program.

This program should be about 90 per cent effective, but there will be cases when it will not stop the chattering.

J Stromsnes

```
OPEN 1,8,15,"M-W"+CHR$(106)+CH
R$(0)+CHR$(1)+CHR$(133):CLOSE 1
```

PROMPT REMOVAL FROM COMMODORE 64 INPUT

Many ways have been suggested for removing the '?' prompt from the INPUT statement on the Commodore 64. Some involve rewriting the ROM, others opening channels from the keyboard for input. Easiest is a variation on the latter using a zero page memory location. For instance:

```
10 POKE 19,1
20 INPUT "ENTER X:";X
30 POKE 19,0
```

Note that location 19 must have its contents restored.

Some other tips are: the RUN/STOP key can be turned off using POKE 788,52. Unfortunately, the jiffy clock is also turned off. The RESTORE key can be disabled with POKE 808,234, and finally, the LIST command can be disabled with POKE 744,0. Now it will list only line numbers, but using other values than 0 yields other results (the normal value is 26).

J Hainsworth

VZ-200 BUG

To the VZ-200 hackers among us this short series of program statements crashes the VZ-200 (Version 2.0).

```
10 N=1 : INPUTS : FOR
   P=1 TO S : N=N*
   P/(P+1) : ? N; : NEXT :
   RUN
```

INPUT 23 twice and the second time round the machine goes crazy.

W Tritscher

P.S. If you pay me for the above, keep it and send it to the person who provides the ROM-patch routine.

'BEE AUTO-START

The following procedure creates a copy of a Basic program that will auto-start upon loading into the MicroBee. This saves the Basic program as a machine language file, with the auto-start address 801EH.

1 Save the program as per usual.
2 Type the following line in command mode:
a=PEEK(248)+PEEK(249)*256:b=PEEK(250)+PEEK(251)*256:PRINT b,b+a
3 The result is two numbers: b and b+a
4 Convert these two numbers to hexadecimal, aaaa and bbbb respectively.

required. If only one value is used, flashing with the value set previously will result.
HORIZONTAL SCROLLING — as display is controlled by the 6845 chip, OUT 256,No. will move the whole screen area with the edges wrapping round. Here, 'No' is a displacement from the right side, so that although OUT 256,20 moves the display by half a screen, OUT 256,39 will place the left margin at column two.
COLOUR MASKING — use of CHR$(22) with text and CHR$(23) with graphics is one of the most powerful of the Amstrad's features but difficult, perhaps, for beginners to understand in terms of the results achieved. Bearing in mind what was previously said about the PEN results of bit combination in each mode, remember that overlapping pixel colours arise from a logical combination of PEN colours. Try this:

```
10 MODE 1:INK 0,0:INK
    1,24:INK
    2,2:INK 3,15
15 GOTO 30
20 PRINT CHR$(23);
    CHR$(1)
30 FOR X=300 TO 350
    STEP 2:MOVE X,100:
    DRAW 0,150,1:NEXT
40 FOR Y=150 TO 200
    STEP 2:MOVE 250,Y:
    DRAWR 150,0,3:NEXT
```

Now delete line 15 and run again. Change INKs 1 & 3 from the keyboard and you'll see that the centre block of colour is unaffected, because the logic result from line 20 remains the same:
1XOR 3=3
(&X00000001 XOR &X00000011 = &X00000010)
(remember this is 'double width' mode — take the STEPs out of lines 30 & 40 and watch the result!).

Last, but not least, a demonstration of 'hidden line' drawing, using the same example. Having deleted line 15, EDIT line 10 and change INK 2,2 to INK 2,24 and run.

T Mayne

SIMPLE CODE CONFUSES PRYING DISASSEMBLERS

A neat trick for budding 6502 assemblers, to confuse the prying eyes of the disassemblers, is to put a few &80,&20, bytes strategically placed around your code.

This has no effect on the program at all but when the code is disassembled, the effect can be quite dramatic.

The &80 will cause the 6502 to skip the next byte, whereas the disassembler will either produce an error, or ignore it and produce the &20 as a JSR.

For example, if you had a piece of code such as:
LDA &FF[A5 FF]
and before it you had put &80,&20, the disassembled version would read:
JSR &FFA5
which is enough to confuse all but the very wary.

Note that extra instruction may not work on every 6502, as any deviations from the standard instruction set depend on the manufacturer of the chip.

D Barrett

VZ VARIABLE DEFINITION

The statements DEFINT, DEFSNG, DEFDBL and DEFSTR are not implemented in VZ-200 Basic (although the code for these is in ROM). A way of simulating these statements, without having to write great chunks of assembler, is to make use of the Variable Declaration Table located between 30977 and 31002 (7901-791AH).

The VDT is 26 bytes in length, one for each letter of the alphabet. Each location contains a code defining the status of variables beginning with each letter:

2 — integer
3 — string
4 — single precision
8 — double precision

On power up and whenever a program is RUN, the whole of the VDT is initialised to single precision (ie, each location contains a 4).

The values in the VDT may be altered to define different variable types. For example, if you wanted to define all A to Z variables as integers, you would put the following code at the start of your program:

```
10 FOR I = 30977 TO
    31002 : POKE I,2 :
    NEXT
```

This is equivalent to the 'DEFINT A-Z' statement in Level II Microsoft Basic.

Alternatively, the following formula could be used to define individual variables:

```
10 POKE 30912 +
    ASC("Q"),3
```

(This would define Q as a string as in 'DEFSTR Q'.)

Note that Basic will not accept double precision variables as counters in FOR-NEXT loops. Also note that it is no longer necessary to use a suffix of '$' or '%' after a string or integer variable has been defined.

C Stamboulidis

TJ'S WORKSHOP

Our monthly pot-pourri of hardware and software tips for popular micros. If you have a favourite tip to pass on, send it to 'TJ's Workshop', 2nd floor, 215 Clarence Street, Sydney 2000. Please keep your contributions as concise as possible. We will pay $10-$25 for any tips we publish. APC can accept no responsibility for damage caused by using these tips, and readers should be advised that any hardware modifications may render the maker's guarantee invalid.

ATARI TOUCH-TABLET

I have found a way to use my Atari touch-tablet in my own programs. The touch-tablet uses the paddle locations: Paddle(0) for horizontal movement; and Paddle(1) for vertical movement. The two buttons are treated as Paddle buttons, Trigger 0 is the same as the left button and Trigger 1 is the right button. The touch-tablet buttons can also be read from Stick(0).

J Risby

Stick(0)=15	No buttons pressed
Stick(0)=14	Touch-tablet stylus pressed
Stick(0)=11	Left button
Stick(0)=10	Stylus and left button
Stick(0)=7	Right button
Stick(0)=6	Stylus and right button
Stick(0)=3	Both buttons
Stick(0)=2	Stylus and both buttons

REAL TIME CLOCK

The following set of subroutines can be used to implement timing on any VZ-200.

```
100 'X=TIME & STOP
105 POKE 30845,201
110 X=PEEK(LC)+256*
    PEEK(LC+1)
120 RETURN
130 'ZERO & DISSABLE
140 POKE 30845,201:POKE
    LC,0:POKE LC+1,0
145 RETURN
150 'SET UP TIME ROUTINE
155 GOSUB 130:
    L=30816:RESTORE
160 READ X
165 IF X>0, POKE
    L,X:L=L+1:GOTO 160
170 POKE 30846,96:POKE
    30847,120
180 DATA 42,104,120,35,34,
    104,120,201,-1
185 LC=30824
190 RETURN
200 'START TIME
205 POKE 30845,195
210 RETURN
```

The subroutine at 150 is used to set up a simple machine code program which increments locations 30824 and 30825 every time the VZ-200 interrupt routine is executed, which is 50 times every second. When the time is read by calling the subroutine at line 100, the value returned in X should be divided by 50 to read the number of seconds since the timer was started.

To start timing, use GOSUB 200. To zero the timeclock, use GOSUB 130.

To read the time without stopping the clock, use GOSUB 110.

To read the time and stop the clock, use GOSUB 100.

Be sure that before you use any of these subroutines, you do a GOSUB 150 to set up the right routines. Your main program should not use the variable LC as this is used in these timing programs.

C Griffin

MEMOTECH GRAPHICS SCREENDUMP

This routine dumps a copy of the Memotech graphics screen to an Epson printer. It produces a double-sized copy of the original which can either be black on white or *vice versa.*

The easiest way to read the graphics screen on the Memotech is to use Control-B. This, when sent to the screen, has the same effect as GR$(x,y,n), and expects three more bytes where x,y are the coordinates of the point to be read and n is the number of bits to be read down.

Before the routine is called, VS4 or a graphics screen must be present or an error will occur.

C Amor

```
MTX TO EPSON SCREEN DUMP
The routine has four modes of operation.
The second byte holds the type in this case.
(0) This uses algorithm to match alternate points.
(1) Uses block of four points per screem pixel.
(2) Inverse of type 0.
(3) inverse of type 1.
writen by Chris Amor.     July 1984.
50 CODE

8648            LD A,0      ;TYPE OF PRINT
864D            EX AF,AF'
864E            LD D,191    ;TOP OF DUMP
8650            LD B,0      ;BOT
8652            LD E,0      ;LEFT
8654            LD C,255    ;RIGHT
8656            LD IX,£FD75         ;PRORPL (OUTPUT DEVICE)
865A            LD HL,£03FC         ;ADJUST SIZE OF DUMP
865D            LD A,B
865E            AND L
865F            AND 191
8661            LD B,A
8662            LD A,D
8663            OR H
8664            AND 191
8666            LD D,A
8667            LD HL,BYTE
866A            LD (HL),0   ;CLEAR FIRST POSITION
866C            PUSH BC
866D            POP HL
866E            LD (IX+0),1         ;SELECT PRINTER
8672            RST 10
8673            DB £83,27,"A",8     ;SET LINE FEED FOR EPSON
8677            LD B,D
8678            JR INSTART
867A ROW:       DEC B       ;SET UP NEXT ROW POINTER
867B            DEC B
867C            DEC B
867D            DEC B
867E INSTART:   PUSH BC
867F            LD A,L
8680            SUB E
8681            LD B,0
8683            LD C,A
8684            INC BC
8685            SLA C
8687            RL B
8689            RST 10      ;SELECT EPSON GRAPHICS MODE
868A            DB £A5,10,13,27,"*",4,£C0  ; FOR (BC) BYTES
8691            POP BC
8692            LD C,E
8693 COL:       LD (IX+0),0         ;SELECT SCREEN
```