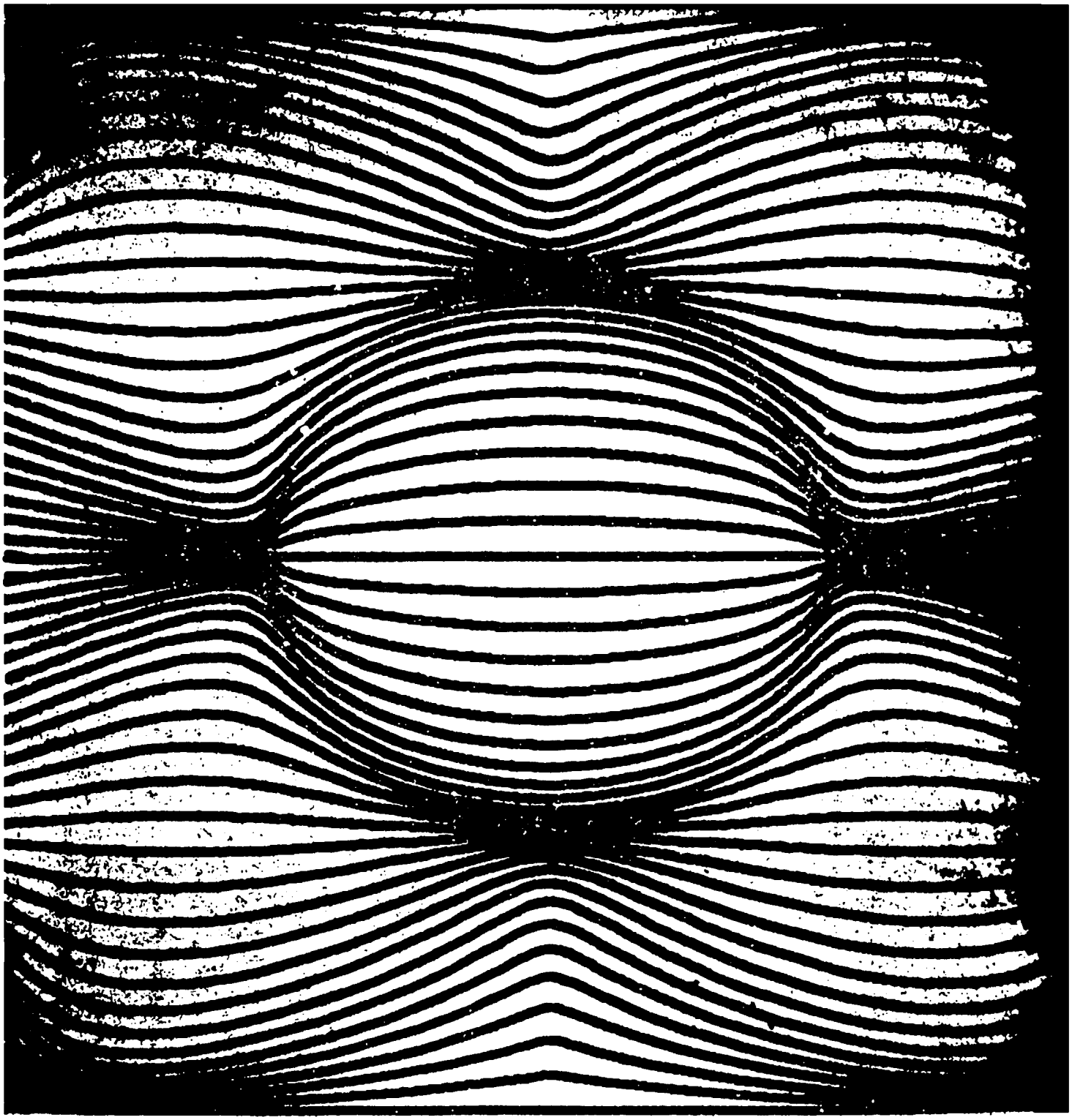


CHIPOS

MICHAEL J BAUER

C H I P O S

Software Manual for the D.R.E.A.M-6800
VIDEO COMPUTER.

by Michael J. BAUER

published by

DREAMWARE
P.O. Box 343
Belmont VIC 3216
Australia

Deakin University Printery 1979

PREFACE

This booklet is a programmers' reference manual for a home/school computer known as the DREAM-6800: that's Domestic Recreational & Educational Adaptive Microcomputer; based on the world's most powerful truly 8-bit microprocessor ... the Motorola M6800! Details of the DREAM-6800 system are to be found in 'Electronics Australia' magazine; May, June, July and August issues, 1979. The information contained in this manual is supplementary to these articles. The reader is referred to the Motorola M6800 Applications Handbook (for advanced users), and (for beginners) the text: "Basic Microprocessors and the 6800" by Ron Bishop (Hayden, 1979) available from Rank Industries (Aust) Pty Ltd and Motorola Semiconductor Inc. An in-depth treatment of CHIP-8 programming can be found in an article by the originator Joe Weisbecker in "An Easy Programming System" (BYTE, Dec. '78).

The philosophy of the DREAM-6800 video computer is simplicity. There is much that can be done with such a simple system, without any add-ons. Many constructors will be tempted to add 16K RAM boards, ASCII keyboards, printers... you name it. But the real purpose is to stretch the imagination to see what can be done with the system as it stands: with one-K RAM, CHIPOS EPROM, 64x32 dot video display, HEX keypad, timer, bleeper and cassette. Only when you've exhausted the capabilities of the minimum system do you have an excuse for adding extra memory, relays, flashing lights, sound generators.....

Readers who develop interesting programs and applications for the DREAM-6800 are encouraged to forward their ideas to DREAMWARE (an M6800 software house). If the response is good, these ideas will be printed and distributed to subscribers of 'DREAMer' (the DREAM-6800 User Group journal, TO BE ANNOUNCED.)

(M.J. Bauer)

CHIPOS Software Design and System Integration

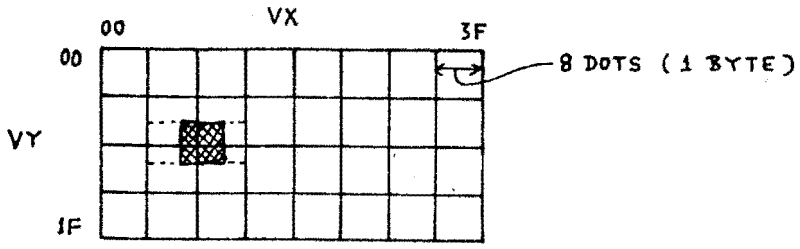
CHIPOS (Compact Hexadecimal Interpretive Programming and Operating System) has been written as an integrated package of subroutines, for two purposes:

1. To allow system routines to be shared by the CHIP-8 Interpreter, monitor, and utility programs; and,
2. To make the system routines available to the machine-code programmer.

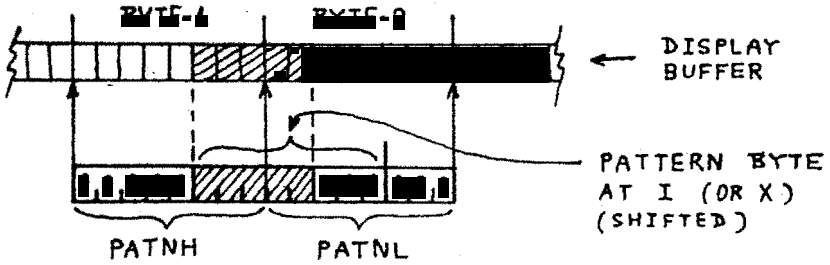
These routines, although compact and efficient, are reasonably well "structured" for ease of use by the caller. Thus, a lot of memory space and effort can be saved by utilising CHIPOS subroutines for such tasks as: Displaying symbols (e.g. hex digits), Inputting from the keypad , Performing serial I/O, Timing, etc. Before examining each subroutine in detail, a few words of explanation about the trickier routines might be in order.

How the Display Driver Routines work:

The subroutines SHOW, SHOWI, SHOWX, SHOWUP, DISLOC etc, perform the task of writing a symbol pattern from anywhere in memory into the Display Refresh Buffer (0100-0200) at a specified location given by a coordinate pair (VX and VY). This task is made awkward by the fact that the symbol will usually overlap 2 horizontally adjacent bytes in the buffer, thus:



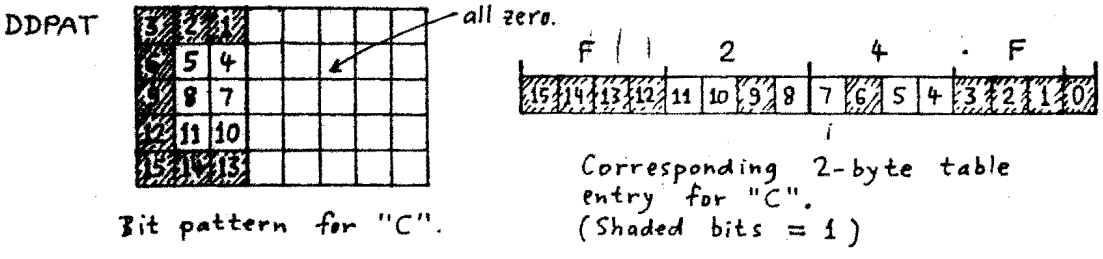
Consider the top 2 bytes only, in detail:



Assume we want to show a pattern 5 dots wide, as shown above. The SHOW routine must first determine the addresses of byte-1 and byte-2, given the x and y coordinates. This is done by subroutine DISLOC. Note that if byte-1 had been the rightmost byte on the screen, then byte-2 would be the leftmost ON THE SAME LINE, in order to achieve the "wrap-round" feature required by CHIP-8. Next, the pattern byte must be shifted and split into two (PATNH and PATNL) so that each buffer byte can be updated separately. This is done by the loop at SHOW3, using the 3 LS bits of VX to determine the number of bit positions to be shifted. The subroutine SHOWUP then XORs each pattern byte with the corresponding display byte. (Only 5 display bits will be affected, in the above example.) To test for any overlapping bits (i.e. both 1) SHOWUP also ORs the bytes (in acc.A) and compares: if the XORed result differs from the ORed result, then the overlap flag (VF) is set to 01.

Displaying a 3 x 5 dot symbol (from a table of 16bit patterns)

The system needs a way of storing patterns for the symbols: 0123456789 and ABCDEF (i.e. hex digits). The most straightforward way would be to use 5 bytes for each symbol. This would require 16 X 5 = 80 bytes of storage, most of which is wasted (only the first 3 bits are needed).
To save space, a more compact method was devised. Since 15 bits are needed for each symbol, only 2 bytes are necessary to store each digit pattern:-



16 such 2-byte words are stored in a table (HEXTAB) in CHIPOS to hold the symbols 0 thru F (total: 32 bytes). The subroutine LETDSP takes the hex digit in acc-A and looks up the table to get the appropriate pattern, which is re-formatted into a 5-byte pattern (DDPAT) suitable for use with the SHOWI routine. The pointer I is set to the location of DDPAT.
LETDSP is written in such a way that a programmer can construct his own table of 3 x 5 symbols in RAM and use LDSP1 and SHOWI to display them.
The "T.V. Typewriter" game uses this technique.

The Interrupt Structure

CHIPOS initializes the system so that only the RTC (timer pulses) can cause interrupts. The interrupt service routine (ISR) at C3E9 merely decrements 2 scratchpad parameters (TIME and TONE) which is a requirement of the CHIP-8 Interpreter and the BLEEP routine. Should the user require a different interrupt structure, the program must store the address of the ISR into locations 0000-0001. (N.B: GETKEY calls BLEEP which needs the DEC TONE and TST PIAB instructions in the ISR.)

> Example: A certain program requires a "live" keypad, i.e. a key depression will interrupt the execution of the main program and store the inputted digit(s) in some memory location(s).
>
> The main program must first write the address of the ISR into the IRQ vector (0000-1), and it must set up the PIA so that CA1 causes an interrupt. The ISR may call PAINZ and KEYINP to input a digit, then store it, then re-initialize the PIA before the RTI.

WARNING! The main program and the ISR must not share a common subroutine (e.g. SHOW) unless IRQ is disabled prior to the JSR, because the mainline might be using the subroutine at the time the IRQ occured, and the ISR will then corrupt the scratchpad bytes that the mainline was using!

Software interrupts are easier to use. If you have a subroutine in your program which is frequently accessed, or uses nearly all the MPU registers, why not use SWI/RTI instead of JSR/RTS ? Note that SWI stacks ALL registers and status, not just the PC. Put this subroutine (ISR) at 0080, or put a JMP at 0080. SWI is also very useful for diagnostic purposes, by using SWIs as breakpoints (which can be replaced with NOPs when the program works). Refer to the "DREAMBUG" routine given elsewhere.

What about NMI ? Forget it !! If you can't do it without NMI, it's not worth doing. (The ONLY sensible use for NMI is as a power-fail battery backup service routine, which is what was put there for.)
Interrupts, in general, are avoided by designers unless the advantages of using them are paramount.