INTRODUCTION

A discussion on Talking Electronics latest monitor for the TEC computer

Article and monitor by
Jim Robertson

JMON is a big step ahead for the TEC.

Some of the contents of JMON are: a highly improved Monitor Program, a versatile Tape Storage Program, software for driving a liquid crystal display, a Menu Driver for utilities, a Perimeter Handler, User Reset Patch, Single Stepper and Break Pointer with register display software, and simplified access to utilities and user routines.

JMON also uses indirect look-up tables stored in RAM. This idea leaves the door open for many possibilities.

All the above and more is contained in 2k bytes.

The following is a description of the major blocks in the ROM.

THE MONITOR PROGRAM

To support new features added to the TEC, a new interactive monitor program has been written. The new monitor is, by itself, a considerable upgrade over previous monitors and when combined with other software in the monitor ROM, gives great features for the TEC user.

Major improvements have been made in the MONitor software, to allow quicker entry and editing of code. This has been achieved by adding such features as auto key repeat and auto increment. If you add the LCD, its larger display and cursor control software open up a second level of improvement.

THE TAPE SOFTWARE

The TAPE SAVE facility is versatile and reliable.

Some of the functions include: 300 and 600 baud tape SAVE, auto execution, LOAD selected file, LOAD next file, LOAD at optional address, TEST tape to memory block and TEST tape check sum. Both tests may be combined with other options.

The tape software uses the universal MENU driver and perimeter handler. These routines allow easy selection of cassette functions (e.g. Load, Save, etc.) and easy passing of variables to the tape software.

The tape software contains check-sum error detection that allows the user to know if the load has failed. A check-sum compare is performed after every page (256 bytes) and also after the leader is loaded. This means the user does not need to wait until the end of a load or test for error detection.

Each full page to be loaded, tested or saved, is displayed on the TEC LED display. Up to 16 pages are displayed.

Upon completion of a tape operation, the MENU is re-entered with an appropriate display showing:-END -S (END SAVE); PASS CS (CHECK SUM); PASS Tb (TEST BLOCK); PASS Ld (LOAD); FAIL CS (CHECK SUM); FAIL Tb (TEST BLOCK); FAIL Ld (LOAD).

The one exception is when an auto execute is performed after a successful load.

The tape software will display each file as it is found and also echo the tape signal.

LIQUID CRYSTAL SOFTWARE

This software is called from the monitor program. It is possible to deselect this software to allow the liquid crystal display to perform a user-defined purpose while the monitor is being used.

The Liquid Crystal Display is being accessed as a primary output device to the user during the execution of the monitor. Eight data bytes are displayed at a time and a space between each for the prompt (it appears as a "greater than" sign). Four digits in the top left hand corner show the address of the first byte.

In the bottom left hand corner is a current mode indicator and this lets you know which particular mode JMON is in. E.g. Data mode, Address mode etc.

The prompt points to the next location to have data entered, or if at the end of the 8 bytes being displayed, the prompt parks at the top left corner indicating a screen change will occur on the next data key press. This allows revision before proceeding.

It is possible to use the monitor with only the LCD unit, the only drawback being the actual current value of the address pointer is not displayed (the value shown in the address portion of the LED display). However this is only minor.

MENU DRIVER

This is a universal routine used to select various utilities routine from JMON. It is already used by the tape software and the utilities ROM. It may also be easily used by the TEC user.

The Menu Driver displays names of functions in the TEC LED display. The number of different names is variable and may be user defined. It is possible to step forward and backward through these names.

A 3-byte jump table with an entry for each name provides the address of the required routine. A jump is performed upon "GO."

To have a look, call up the cassette software by pressing SHIFT and ZERO together. If you have not fitted a shift key, the cassette software can be addressed by pressing the address key, then the plus key, then zero.

To move forward through the MENU, press "+". To move backward, press "-". Notice the automatic FIRST-TO-LAST, LAST-TO-FIRST wrap around.

Pressing "GO" will take you into the perimeter handler.

PERIMETER HANDLER

Like the Menu Driver, this is a universal program and may be easily used by the user.

This routine allows variables to be passed to routines in an easy manner. The variables are typically the start and end address of a block of memory that is to be operated on, such as a load, shift, copy, etc.

A 2-character name for each 2-byte variable is displayed in the data display while the actual variable is entered and displayed in the address display.

The number of variables may be from 1 to 255 and is user definable.

The data display is also user definable.

It is possible to step forwards and back through the perimeter handler in the same fashion as the MENU driver.

When a "GO" command is received, control is passed to the required routine

via a 2-byte address stored at 0888 by the calling routine.

The SINGLE STEPPER and BREAK POINT handler.

A single stepper program can be important when de-bugging a program. It effectively "runs" the program one step at a time and lets you know the contents of various registers at any point in the program.

If you have ever produced a program that doesn't "run", you will appreciate the importance of a single stepper. Many times, the program doesn't run because of an incorrect jump value or an instruction not behaving as the programmer thinks.

The single stepper runs through the program one instruction at a time and you can halt it when ever you wish. By looking at the contents of the registers, you can work out exactly what is happening at each stage of the program.

The single stepper operates by accessing a flip flop connected to the Maskable Interrupt line of the Z-80. It can be operated in the manual mode, in which a single instruction is executed after each press of the "GO" key. In the auto mode, 2 instructions are executed per second.

BREAK POINTS

Break points work with groups of instructions. They allow register examination in the same way as a single stepper. The advantage of break points is that there is no time wasted stepping slowly through a program. This is particularly important as some programs contain delay loops and they may take weeks to execute at 2Hz!

Break points are one of the most effective ways to debug a program!

STARTING WITH JMON

JMON is straight forward to use. Some new habits must be learnt, however they are all quite easy.

JMON has 4 modes of operation. They are:

DATA MODE, ADDRESS MODE, SHIFT MODE and FUNCTION MODE.

The data address and shift modes are not new but have been, in part, changed in their operation. The function mode is new to the TEC and I am sure you will find it useful. Below is a description of each mode.

THE DATA MODE

The data mode is used to enter, examine and edit, hex code into RAM memory. It is identified by one or two dots in the data display and the word "DATA" in the bottom left hand corner of the LCD display. It is similar to the data mode on all previous MONitors.

The data mode has a sub-mode called AUTO INCREMENT. This is a default setting, meaning that it is set to auto increment on reset. The user may turn off the auto increment sub-mode if desired.

When in the auto increment mode, the current address pointer in the address display is automatically pre-incremented on each third data key press.

A SINGLE DOT in the RIGHTMOST LED display indicates the current address will be incremented BEFORE the next nibble received from the keyboard is stored.

This allows the user to review the byte just entered. If an incorrect nibble is entered, the internal nibble counter MUST BE RESET by pressing the ADDRESS KEY TWICE. Then two nibbles may be entered at that location. This is a slight annoyance at first, but it is a small price to pay for such a powerful feature as auto increment!

After two nibbles have been entered, the prompt on the LCD is IMMEDIATELY updated and points to the next memory location, or in the case of the last byte on the LCD, the prompt PARKS AT THE TOP LEFT CORNER signifying an entire screen update UPON THE NEXT DATA KEY PRESS.

This allows the user to revise the entered code before continuing.

You must be in the data mode to perform a program execution with the "GO" key. (Actually, you can be in the SHIFT mode also.)

Because of the auto key repeat, and "auto increment", it is possible to fill memory locations with a value by holding down a data key. This may be useful to fill short spaces with FF's or zero's.

Because the LCD prompt is advanced immediately after the second nibble being entered while the LED display is advanced on the third nibble received, the "+" key will advance only the LED display while the "-" key will shift the LCD prompt back two spaces, if either are pressed immediately after the second nibble is entered. This may seem strange but is the result of a clever design which allows for revision of entered code on either display before proceeding.

ADDRESS MODE

This is identified by 4 dots appearing in the address display of the LED display and "ADDR" in the LCD bottom corner.

The address key is used to toggle in and out of this mode.

The address mode will be entered by an address key press from either the data or function mode. An address key pressed while in the address mode will result in a return to the data mode.

While in the address mode, data keys are used to enter an address while the control keys (+, -, GO) are used to enter the function mode. No auto zeroing has been included, therefore 4 keystrokes are required to enter any address.

SHIFT MODE

This mode allows easy manual use of the cursor. The shift works by holding down the shift key and at the same time, pressing a data key.

The monitor must be in the data mode and only data keys work with the shift.

Sixteen functions are available but only ten have been used in this monitor.

The shifts are:

Shift-zero: Cassette MENU is displayed.

Shift-one: Cursor back one byte.

Shift-two: Start single stepper at current address.

Shift-four: Cursor forward 4 bytes.

Shift-five: Break from shift lock (see function mode).

Shift-six: Cursor back 4 bytes.

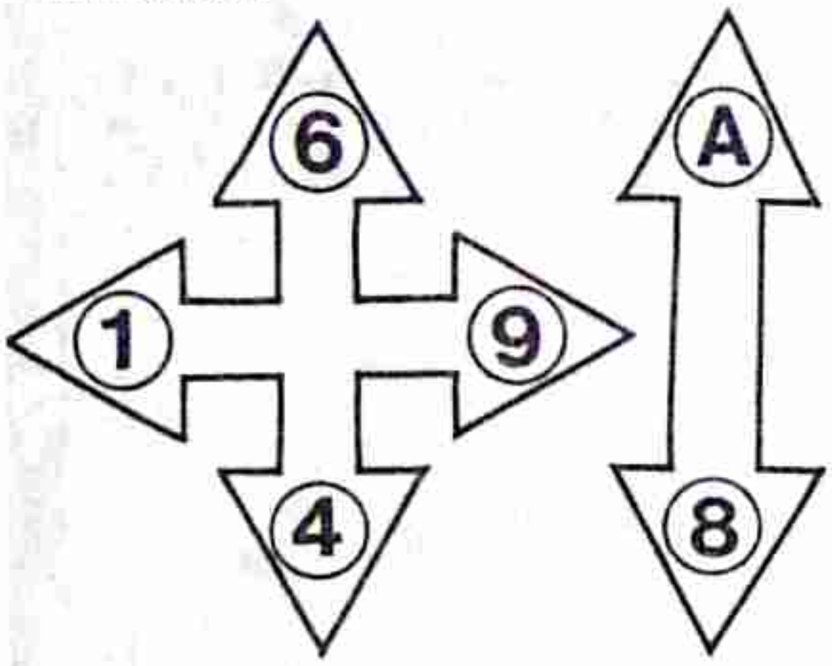
Shift-seven: Enter register examination routine.

Shift-eight: Cursor forward 8 bytes.

Shift-nine: Cursor forward 1 byte.

Shift-A: Cursor back 8 bytes.

Note that 1, 4, 6 and 9 form a cross and 8 and A form an arrow and each is positioned to correspond to their cursor movement.



Keys 1, 4, 6 and 9 move the cursor LEFT, RIGHT, UP AND down on the LCD.

Key "A" shifts the screen back to display the previous eight bytes.

Key "8" shifts the screen forward eight bytes.

When editing a program, the shift enables fast movement through the memory. Data entry is achieved by releasing the shift key.

The shift mode is not identified explicitly on either display.

THE FUNCTION MODE

This has been provided to enable a quick way to call commonly used routines. Only three keystrokes are required invoke up to 48 different routines.

The function mode is broken up into 3 sections.

They are: Function select-1, Function select-2 and Function select-3.

Each is identified by a single dot in the address display: right-most for function 1, second right for function 2 and third right for function 3. On the LCD display, the functions are identified by: Fs - 1, Fs - 2, or Fs - 3 in the bottom left corner.

Fs - 1, Fs-2 and Fs-3 are entered FROM THE ADDRESS MODE by pressing the "+" key for Fs-1, the "-" key for Fs-2, the "GO" key for Fs-3.

It is possible to swap between sections without coming out of the current function mode by pressing the required function select key. After entering the required section, A DATA KEY IS THEN USED TO SELECT ONE OF SIXTEEN ROUTINES.

The address of these routines are stored in a look-up table.

SECTION-1 - the SHIFT-LOCK FEATURE.

Section-1 is selected FROM THE ADDRESS MODE by pushing the "+" key. The keys 0, 1, 2, 4, 5, 6, 7, 8, 9 and A then have the functions as listed in the shift mode. (Key 5 has the function of returning to the data mode.)

Cursor control routines return back to section-1 to enable continuous cursor movement (shift-lock).

The look-up table for the jump addresses for section-1 is at 07E0.

SECTION-2

Section-2 is selected from the address mode by pushing the "-" key. This is unused by any existing software and is available to the user.

HERE'S HOW TO USE IT:

Using the section-2 is very easy. All that is required is to enter the address(es) of the required routines in a table. The table begins at 08C0. The first two bytes at 08C0 correspond to the zero key in section 2. While the second two (08C2) correspond to key one etc.

Here is a short program as an example:

08C0: 00 09 04 09 08 09

(These are the addresses of the routines).

0900: 3E EB 18 06 3E 28 18 02

0908: 3E CD D3 02 3E 01 D3 01

0910: 76 C9

Now push ADdress, "-", "0" and the routine at 0900 will be CALLED from the MONitor. Reset the TEC and try ADdress, "-", "1" and ADdress, "-", 2.

Because these routines are CALLED from the MONitor, you may use a return (RET, C9 or RET NZ etc.) instruction to re-enter the MONitor in the same state as you left it. e.g. in the function select-2 mode.

SECTION-3

This has been reserved for the utilities ROM at 3800. The table for Section-3 is at 3820.

USING THE SINGLE STEPPER

Getting the single stepper to work is simple enough, however there is some skill required to understand its limitations and knowing how to avoid them.

To start with, you need a program that you require to be SINGLE STEPPED.

This program may be anywhere in memory except in the lowest 2k (the MON ROM).

This is because the MON select line is used as part of the timing. You may call into the MON ROM but only the first instruction will single step, and when returning out of the ROM, the next instruction will also not be stepped. (However they will be executed at normal speed.)

Programs that use the TEC's keyboard require careful attention as you cannot step them in the normal way. This is because there is no way to distinguishing between key-presses for the single stepper and those for the subject program.

This reduces the usefulness of the single stepper a little however thoughtful software design enables a fair degree of flexibility and this problem may be side-stepped.

The key use of the single stepper is as a de-bugging aid. When you are writing programs. effective use of the single stepper usually requires that while writing your programs, you allow for the use of the single stepper by leaving room to place one byte instructions that turn ON and OFF the single stepper.

Programs using the keyboard may be stepped by turning OFF the stepper. This allows areas requiring use of the keyboard to run in real time while other areas may be single stepped. This applies only to programs that use the keyboard routines provided inside JMON.

The only disadvantage here is that after completing your program you may have NOPs left.(from where you blanked over the single stepper control bytes).

The keyboard controls for the single stepper are as follows:

To start single stepping from the current address, this is what to do: From the data mode, press shift-2. This will start the single stepper. The first instruction will be performed and the address will be displayed as "PC" (Program Counter) on the single stepper. To examine the registers, press "+". The left two nibbles correspond to the high order byte and in the case of register pairs, the left-hand register. You may go backwards also by pressing "-". The registers displayed are : PC, AF, BC, DE, HL, IX, IY, AF', BC', DE', HL' and SP, in this order.

To step the next instruction, press GO. You can also step continuously at about 2Hz by pressing any data key.

When in the auto step mode, you can stop at any time and examine the registers by pressing "+" or "-", or bring it back to the manual mode by pressing GO.

The address key resets back to the MONitor unconditionally. The control bytes for the single stepper are as follows:

To stop single stepping in a program: F3 (disable interrupt).

To restart in a program: EF (restart 28). This causes a restart to 0028 where a routine passes the start address (which is actually the return address of the restart 28 instruction) to the single stepper. It also enables the interrupts and then returns to the next instruction which is then single stepped.

This SINGLE STEPPER is only a first model. Hopefully, when more room is available, some improvements can be added. One improvement on the "cards," is allowing it to be interfaced with a utilities ROM. This ROM will extend the display capabilities, allow editing while stepping and to disassemble on the LCD each instruction as it is stepped. If you have any ideas or requirements, write in and tell us.

BREAK POINTS

Break points are locations in a program where execution is stopped and the registers are examined in the same fashion as the single stepper. The advantages over single stepping include real time execution and less or no control bytes in a program. They also usually allow much quicker fault finding.

As a trade-off move, only a simple (but effective!) form of break-point is available with JMON. This allows for more MONitor functions and also eliminates the need for extra hardware.

More complex methods automatically remove the break-point control byte and re-insert the correct op-code and allows re-entry to the program.

USING JMON BREAK-POINTS

Break points are achieved by using a restart 38 instruction. The op-code for this is FF and all that is required is for it to be placed where ever you require your break point.

Before running your program, make sure the TEC is reset to 0900. This is necessary to clear the auto-repeat on the stepper/break-point register display. (This is explained in the LCD section).

Simply run your program as normal. When the break point is reached, the register display routine is entered. The value of the program counter display WILL NOT BE VALID on the first occurring break unless you provided the address of the break point at 0858. This minor flaw was unavoidable without considerable additional software which would have "eaten" memory like there's no tomorrow!

If you allowed for break commands in your program, you may then have multiple breaks and step to the next break with the GO key.

However if you placed a break command over an existing instruction then no further breaks will be valid and you should never try multiple breaks in this case AS YOU MAY CRASH THE MEMORY.

In the above case, make a note of the contents of the registers and return to the monitor via the address key and then examine memory locations, if required. (You may enter the register examination routine via shift-7). Further breaks should be done by removing the existing break and placing it where required and re-executing the program from the start.

Some other good ideas are to load the stack away from the MONitor's RAM area. (08F0 is good but make certain that 08FF does not contain AA - as this prevents the MONitor rebooting its variables on reset and your stack may have accidentally crashed these variables.) Also, if you are using the LED display scan routine in the MONitor ROM, shift your display buffer to 08F0 by putting this address into 082C/2D. Now you can examine your stack and display values after returning to the MONitor.

There is a conditional way to cause breaks. To do this requires a conditional jump relative with FF as the displacement. If the condition is met, the jump is made and jumps back onto the displacement which then becomes the next op-code! Remember this as it is a very useful idea. You cannot continue on with multiple breaks after a break caused by this method.

Break points are a quick way to debug a program. It is very important that you familiarize yourself with them. They have been the single most important programming aid used when writing most of JMON and the utilities ROM.

SUMMARY:

Clear the auto-repeat via the reset.

: Use FF to cause a break

: PC not valid on first break.

: For multiple breaks, provide spaces for the break control byte.

: Shift stack and display buffer (optional)

: Use FF as displacement for conditional breaks.

Finally, make sure you write down when, where and why, each time you insert a break-point.

ACKNOWLEDGMENT

Thanks to MR. C PISTRIN of Traralgon VIC. His SINGLE STEPPER program for the MON-1B inspired me to include one in JMON and provided me with a circuit for the hardware section.

See page 47 for the circuit

Programs which have neither a HALT or LD A,I instruction cannot be altered by any of the above methods because they enter a continuous loop and require the interrupt to force an input value into the accumulator. A classic example of this is the "space invaders shooting" on page 14, issue 14. This above loop is located at 0821. (while you're looking at this, grab a pen and change the byte at 0812 from 02 to 01, at least it will run correctly with MON-1)! All the above types are among those listed as not being suitable for modification via these methods.

FINALLY

If you find a program which doesn't work (we haven't tried them all) or something else interesting, please write and let us know.

USING THE KEYBOARD IN YOUR PROGRAMS

The new keyboard set-up is no more difficult to use now than before. In actual fact it is easier and requires less bytes than before thanks to the use of the RST instructions.

Four RST's are provided to handle the keyboard in different ways. The first RST we shall look at is RST 08 (CFH). This RST is a "loop until a NEW key press is detected" routine. If you refer to the section on running old programs, you will see that this RST is used to simulate/replace the HALT instruction. (You know how to use it Already!)

An important feature of this RST is that it ignores any current key PRESSED, that is if a key is being pressed when this RST is performed, it will not be recognized. This mimics the NMI which only recognized a key press once. (This is why the auto-repeat feature could not be done with the keyboard hooked up to the NMI).

When this RST detects a valid key press, it inputs the value from the key encoder and masks the unwanted bits and stores the input in the interrupt vector register (as did the MON-1 series). The input value is also returned in the accumulator. The shift key can not be read from this (or any other MONitor keyboard routine) as the shift input bit (bit 5) is masked off.

Here is an example of its use:

```
0900  CF      RST 08
0901  FE 12   CP 12
0903  20 04   JR NZ,0909
0905  3E EA   LD A,EA
0907  18 06   JR 090F
0909  FE 01   CP 01
090B  20 F3   JR NZ,0900
090D  3E 28   LD A,28
090F  D3 02   OUT (02),A
0911  3E 01   LD A,01
0913  D3 01   OUT (01),A
0915  18 E9   JR 0900
```

The first thing you should notice when you enter and run the above, is that the "go" key is not detected when the routine is first started, even though it is being pressed. This is because the first part of the RST loops until the key being pressed is released. The RST then loops until a new key press is detected. When the RST returns, the input value is both in the interrupt vector register and the accumulator. The rest of the routine tests for either a 01 or "GO" key and outputs to the display.

Use this RST when ever you want the TEC to go "dead" and wait for a key press.

The second RST is RST 10 (D7). This is similar to the first RST but has one very important difference. The difference is that this RST DOES NOT wait for a key being pressed to be released before returning. While this is not as likely to be used as much as the first RST, it does have some good uses. Any program which requires some action to take place while there is a key pressed, but do nothing when there is not, may make good use of this RST. Some possible uses include random number generation on the time the key is held down; count while a key is pressed; turn on a relay while a key is pressed etc. As you can see, this RST simulates momentary action switches.

This RST exits with the input stored in the same fashion as the above RST.

The third RST is RST 18 (DF). This is a LED scan loop and keyboard reader. The scan routine will scan the 6 TEC LED displays once with the display codes addressed by the address at 082A. (0800 is stored here by JMON. You can leave it as it is, just store what ever you want at 0800 before using this RST). After the scanning routine is done, the keyboard routine is called. The keyboard routine is actually called from RST 20. What happens is this. After the scan has been called from the RST 18, the program continues on at 0020, which is the start address for RST 20. So the RST 18 is the same as RST 20 EXCEPT THAT RST 18 CALLS FASTSCAN. Therefor the description below applies to BOTH RST 18 AND RST 20.

This keyboard routine is very intelligent and is able to detect several different conditions.

One important feature is that it "remembers" if it has already detected the one key press and it ignores it if it has. This provides us with a "ONE AND ONLY ONE" key recognition for each key press. Each key press is "recognized" on the first detection.

The key is checked for a "FIRST KEY PRESS" by the use of a flag byte. When the routine is entered AND NO KEY IS PRESSED, this flag byte is CLEARED. When a key is detected, the flag byte is checked. If zero, the key is accepted as a "FIRST KEY PRESS." The flag byte is then set to stop further "validating" of the same key press. The input value is then masked and returned in the Accumulator (only).

If the flag byte IS NOT CLEAR, then the key is not recognized as "valid."

Careful consideration was giving to the interaction of the MONitor and user routines so that the "GO" command from the MONitor WILL NOT BE TREATED AS THE FIRST KEY PRESS of a user routine. (This was achieved by using the same flag byte for both JMON and any user routine).

HOW TO INTERPRET THIS RST

If a key is recognized as a "FIRST KEY PRESS" then the ZERO FLAG will be set to its active state (a logic 1) and the MASKED KEY INPUT will be returned in the accumulator.

If the key is NOT valid then the ZERO FLAG will be clear AND the accumulator WILL HAVE ALL ITS BITS SET (FF).

(FOR ADVANCED PROGRAMMERS)

In addition to the zero flag being conditionally set, the RST 20 (E7) also sets the carry conditionally, according to the following conditions:

If there is a key pressed then the carry will be SET REGARDLESS of whether it is a "first key pressed" or NOT. If NO key is pressed then the carry is cleared.

This allows you to interpret the keyboard the way you want, while still giving you the convenience of using the RST to do some of the work.

Jim's section cont P 47.

JMON'S PERIMETER HANDLER AND MENU DRIVER

THE PERIMETER HANDLER.

The PERIMETER HANDLER is a useful MONitor routine.

It allows you to enter the start and end address of blocks to have an operation performed on them. Just imagine how much more convenient Ken's MON-2 utilities and the original printer software would be if you could easily type in the start and end of blocks.

The ease of use of the tape software demonstrates the value of the PERIMETER HANDLER.

THE MENU DRIVER

This also adds convenience as it allow functionally related routines to be grouped together as an organized "smorgasbord." You can then pick a routine for execution by using "+" and "-" to scroll through the entries and then hit "GO" when the required routine is displayed.

Both the MENU DRIVER and the PERIMETER HANDLER are universal. This means that they are not tied to the tape software only, but can be utilized by other utilities supplied by myself or used in your own routines.

Even if you don't find any use for either, it would be remiss of me not to explain how simple they are to use. After all, you have paid for the software and shouldn't be intimidated by its imaginary complexity.

USING THE PERIMETER HANDLER

To use the PERIMETER HANDLER in JMON you only need to provide the following information.

A 10 byte command string and display codes for your chosen data displays.

After setting-up the above information, you then jump to the PERIMETER HANDLER.

THE PERIMETER HANDLER'S COMMAND STRING

The command string consists of the following information.

The first two bytes are OPTIONAL signature bytes.

These bytes identify the calling routine for the use of other routines in the program. The PERIMETER HANDLER DOES NOT need any particular values placed here, these bytes are entirely optional.

The next two bytes are the address of the DATA DISPLAY CODES. This address is stored in normal Z80 format with the low byte first.

Following the display address is the FIRST INPUT WINDOW POINTER. This is set to point to the HIGH ORDER BYTE of each two byte entry. An area at 0898 is set a side for the PERIMETER HANDLER's input window(s), so usually you will enter 99 08 in that order for these two bytes. That then means the first two byte value entered will be stored at 0898, the second at 089A, third at 089C etc.

The next byte is the number of the first window to be displayed when the PERIMETER HANDLER is first entered. This will be ZERO in just about all cases.

Following the first window number is the "number of windows" byte -1. If you want one window, set this byte to 00. For two windows, set this byte to 01 and for three windows set it to 02 etc.

The last two bytes on the command string are the jump address of the routine you want to be executed immediately after the PERIMETER HANDLER. This address is stored in normal Z80 format with the low byte first.

THE COMMAND STRING LOCATION

The start of the command string is fixed at 0880. Every routine that uses the PERIMETER HANDLER will put its command string at 0880. This means you must have your command string stored in its own area and copy it across before jumping to the PERIMETER HANDLER.

THE DISPLAY CODES

The display codes produce the shapes on the data displays. Each window needs two display bytes, one for the left-hand display and one for the right-hand display.

The format for the display codes is the following:

The codes for the first window are stored at the lowest end of the display table. The codes for the second window are stored in the next two bytes higher etc.

The code for the left-hand DATA display is stored in the lower memory address of each two byte entry.

This table may be anywhere in memory. It is pointed to by the display pointer address in the command string (see above).

ENTERING THE PERIMETER HANDLER.

After coping across your command string, the PERIMETER HANDLER is entered by JUMPING to 0044. A jump at 0044 then jumps to the PERIMETER HANDLER. NEVER jump directly to the PERIMETER HANDLER, always use the indirect "gate" at 0044. Otherwise up-ward compatibility with JMON up- grades cannot be assured.

EXAMPLE OF PERIMETER USE

This example shows a typical way to set-up the PERIMETER HANDLER. Apart from being an example of the use of the PERIMETER HANDLER, this routine is very useful and should be saved on tape. The following routine is adopted from the JMON utilities ROM.

The routine moves all the bytes in between and including the start and end address, to the destination address. The Move Routine allows the destination to be between the start and end so that the block may over-write itself. The routine has been deliberately placed at 0F80 to be out of the way of your usual program space.

Let's start.

The first thing we need is to create our command string.

EXAMPLE PERIMETER COMMAND STRING

As the first two bytes are not important, we will place FF FF here:

Command string:

0F80 FF FF

The next two bytes is the address, in Z80 format, of the data display codes. The address of the data displays in this example is 0F8A.

Command string:

0F80 FF FF 8A 0F

After the DATA display address comes the address of the first PERIMETER HANDLER window +1. There is an area specially set a side for the PERIMETER HANDLER's windows at 0898 and we will use this area.

So the address entered here is 0898+1 = 0899.

Command string:

0F80 FF FF 8A 0F 99 08

The next byte is the number of the first window to be displayed. This value is always ZERO (or at least until you know what you are doing).

Command string:

0F80 FF FF 8A 0F 99 08 00

Immediately after the "first window number" is the number of windows -1. As we need three windows, this value is 3-1 = 2.

Command string:

0F80 FF FF 8A 0F 99 08 00 02

The last two bytes are the jump address of the wanted routine.

Quick arithmetic shows the first available address for the block shift routine is 0F9E. This is where I have located the routine so put this address in the command string.

Completed command string:

0F80 FF FF 8A 0F 99 08 00 02 9E 0F

Now at 0F8A we put the data display codes. The displays we want are: -S, -E and -D for start, end and destination. The the left- most display byte for the first window is stored at the lowest location in the table. In this case it is the display code for "-", which is 04. The right-hand display for the first window is the next byte in the table. This is A7, the display value for "S". So far we have:

0F8A 04 A7

The left-hand display for the second window data display is next byte to be stored. This is 04 being the display code for "-". The display code for "E" is after that and is C7. Our data display table, when completed looks like this:

0F8A 04 A7 04 C7

And finally the "-d" for destination.

0F8A 04 A7 04 C7 04 EC

Ok, the data tables have been defined. Lets look at the program requirements.

The first thing we need to do is to move the command string down to 0880. To do this we will use the Z80's block load instruction in a short routine at 0F90. After the command string is loaded into place, we then jump to the PERIMETER HANDLER "gate" at 0044.

When GO is pressed in the PERIMETER HANDLER, HL is loaded from 0888. The address at 0888 is the address of the actual block move routine at 0F9E and was supplied in the command string as described above.

A JP (HL) is then performed and the move routine is executed.

Example perimeter command string:

0F80 FF FF 8A 0F 99 08 00 02 9E 0F

and data display codes:

0F8A 04 A7 04 C7 04 EC

Set-up routine and start location:

```
0F90  21 80 0F    LD HL,0F80      These instructions move the
0F93  11 80 08    LD DE,0880      command string from its
0F96  01 0A 00    LD BC,000A      storage buffer to the correct
0F99  ED B0       LDIR            working spot in RAM and jumps
0F9B  C3 44 00    JP 0044         to the PERIMETER HANDLER
```

Actual shift routine starts here:

```
0F9E  2A 98 08    LD HL,(0898)    HL= start of block
0FA1  ED 4B 9C 08 LD BC,(089C)    BC=destination
0FA5  ED 5B 9A 08 LD DE,(089A)    DE=end
0FA9  E5          PUSH HL         save start
0FAA  B7          OR A            clear carry flag
0FAB  ED 42       SBC HL,BC       is dest < start
0FAD  30 06       JR NC 0FB5      jump if it is
0FAF  C5          PUSH BC         swap dest
0FB0  E1          POP HL          and start/dest offset
0FB1  13          INC DE          DE = end+1
0FB2  B7          OR A            clear carry
0FB3  ED 52       SBC HL,DE       find diff between dest and end
0FB5  E1          POP HL          recover start in HL
0FB6  F5          PUSH AF         save flags
0FB7  E5          PUSH HL         save start
0FB8  EB          EX DE,HL        put start in DE, end in HL
0FB9  B7          OR A            clear carry
0FBA  ED 52       SBC HL,DE       is end lower than start?
0FBC  EB          EX DE,HL        put byte count in DE
0FBD  E1          POP HL          recover start
0FBE  30 04       JR NC 0FC4      jump if end higher than start
0FC0  F1          POP AF          clean up stack
0FC1  C3 4A 00    JP 004A         jump to display "Err In"
0FC4  F1          POP AF          recover flags
0FC5  D5          PUSH DE         swap count in DE
0FC6  C5          PUSH BC         and dest in BC
0FC7  D1          POP DE          with each other
0FC8  C1          POP BC
0FC9  30 08       JR NC 0FD3      jump if dest end or start
0FCB  EB          EX DE,HL        else calculate the address of the last
0FCC  09          ADD HL,BC       byte in destination block by adding
0FCD  EB          EX DE,HL        count and dest" DE= new dest
0FCE  09          ADD HL,BC       add start and count to get end
0FCF  03          INC BC          set BC to real count
0FD0  ED B8       LDDR            shift the block starting from the end
0FD2  C9          RET             done
0FD3  03          INC BC          set BC to real count
0FD4  ED B0       LDIR            shift block from the start first
0FD6  C9          RET             done
```

TO RUN THE EXAMPLE

Use the instructions in issue 15, page 20 to enter 0F90 in a FUNCTION-2 jump table. or if you don't have an issue 15 handy, you can run this by addressing 0F90 and hitting "GO".

USING THE MENU DRIVER

To use the MENU DRIVER, you must supply the following information: A command string of 10 bytes, a jump vector or RETurn instruction for the data keys, the display codes for both the address and data displays and finally a jump table.

THE MENU COMMAND STRING

The command string has is similar in some respects to the command string in the PERIMETER HANDLER. The format for the command string is as follows:

Like the PERIMETER HANDLER, the first two bytes are optional and may be any value you like.

The next byte is the number of the current MENU display. THIS MUST BE SET TO ZERO IN THE COMMAND STRING.

Following this is the number of MENU displays -1. If you want 3 displays then this will be set to 02 etc.

The next two bytes are the base address of a jump table that is used to jump to the selected routine.

Next, we have a pointer that is the base of the ADDRESS DISPLAY TABLE for the MENU DRIVER.

The data display pointer is next and points to the base of the DATA DISPLAY TABLE.

LOCATION OF THE MENU COMMAND STRING

The command string for the MENU DRIVER is fixed at 088D and like the command string for the PERIMETER HANDLER, should be stored in its own area and shifted into RAM before you enter the MENU DRIVER.

DATA KEY RETurn

Following the command string is a jump or RETurn instruction for the data keys. If a data key is pressed while in the MENU DRIVER, the MENU DRIVER CALLS to a predetermined address in RAM (0897). This is where you put either a jump if you have a data key handler or a RETurn (C9) if you do not. Usually, it is not necessary to have a data key handler and you will place a RETurn here (actually, you can tack it on to the end of the command string and shift it across with the command string as it is the next byte after the command string).

THE MENU DISPLAY CODE TABLES

Tables are used to hold the display codes for the MENU displays. The address displays and the data displays are held in separate tables that may be anywhere in memory. The format for the data table is the same as it was for the DATA DISPLAY TABLE used by the PERIMETER HANDLER. The address table is different only in that the entries are FOUR display bytes long. So again the left-most display byte for the first address display is located at the lowest address of the table as it is for the data displays. The four bytes for the second entry follow immediately the four for the first entry etc.

THE MENU JUMP TABLE

The last requirement is a table of jumps which are arranged in the following order:

The first byte of each entry is ALWAYS C3. This is the OP-CODE for an unconditional jump. Following each jump OP-CODE is the address of a routine that is to be selected by the MENU DRIVER.

The first entry in the table is the jump vector for the first routine name displayed in the MENU DRIVER. All the entries are arranged in the order they appear on the MENU DRIVER and there should be one jump vector for each MENU ENTRY.

It is allowable and indeed sometimes desirable that several MENU entries have the same jump address value. The common routine selected can identify which MENU entry was selected by the value of the current MENU entry number at 088F.

This method has been used in the tape software in JMON. The high and low speed MENU displays both jump to the same routine. The routine then identifies the selected speed by the value at 088F.

ENTERING THE MENU DRIVER

The MENU DRIVER is entered by JUMPING to 0041. Before you jump, you must have set-up the command string.

NEVER JUMP DIRECTLY TO THE MENU DRIVER, ALWAYS USE THE INDIRECT "GATE" AT 0041.

EXAMPLE OF MENU USE

In this example, we will set the MENU to select between two routines. The first routine will be the tape software. The tape software will then set-up the MENU and enter it with its displays. This will show that you may have SUB-MENUS or MENUS off MENUS. Unfortunately, I did not consider this when designing the MENU and there is not currently an (easy) way to return from a SUB-MENU back to a higher level MENU. I do not think this is much of a problem at this stage though.

The second routine called from the MENU will be the block move described above.

Ok, let us start by building the command string.

EXAMPLE MENU COMMAND STRING

The first two bytes are optional signature bytes, as they were for the PERIMETER HANDLER. Any value may be used for these bytes as far as the MENU DRIVER is concerned. We will leave these at FF FF.

0F10 FF FF

The next byte is the number of the first menu entry. This MUST ALWAYS BE ZERO.

MENU command string:

0F10 FF FF 00

Immediately after the "initial MENU entry" byte, is the number of required MENU entries -1.

E.g. If you want 2 menu entries then this will be 01 and if you want 3 entries then it will be 02 etc.

As we require 2 MENU entries, the value we enter is 01.

0F10 FF FF 00 01

Next we have the address of the first byte of our JUMP TABLE. The jump table is at 0F20 so put this in the command string.

0F10 FF FF 00 01 20 0F

After the jump table base pointer is the base of the ADDRESS DISPLAYS table. You can have your ADDRESS DISPLAYS table anywhere in memory, ROM or RAM. Set this 16 bit value to point to the first byte of the TABLE. In this example, the table is at 0F30. Put this in the command string:

0F10 FF FF 00 01 20 0F 30 0F

The last two bytes is the base of the DATA DISPLAY TABLE. The table is at 0F40

0F10 FF FF 00 01 20 0F 30 0F 40 0F

This completes the command string.

EXAMPLE DATA KEY HANDLER (A RETurn)

Now that we have our command string, we must decide if we want the DATA KEYS to do anything.

In this example we do not, so we must provide a RETurn instruction at the DATA KEY INDIRECT JUMP BUFFER. This buffer is fixed at 0897. The last byte of the command string, when loaded into its fixed RAM location is 0898. This means we can tack the RETurn (or jump) instruction onto the end of the command string.

```
0F10  FF FF 00 01 20 0F 30 0F 40
0F18  F0 C9
```

EXAMPLE DISPLAY TABLES

Now we must create the DATA and ADDRESS DISPLAY TABLES.

The address table consists of two words: TAPE and BLOC (short for BLOCK).

The display codes for these are C6, 6F, 4F and C7 for TAPE and E6, C2, EB AND C3 for BLOC.

Put the above in the ADDRESS DISPLAY TABLE at 0F30

```
0F30  C6 6F 4F C7 E6 C2 EB C3
```

The DATA DISPLAYS will be "- -" for the tape display and "-S" for the user display. The "- -" symbolizes a SUB-MENU will be entered and the "-S" is to complete the heading "BLOC -S on the display for BLOCK SHIFT.

The DATA DISPLAY CODE TABLE at 0F40 is:

```
0F40  04 04 ("--") 04 A7 ("-S")
```

Finally, we must have a three byte jump table that is located at 0F20

The first jump instruction in this table corresponds to the first routine to be displayed on the MENU. This is the TAPE routine. The address of the routine is at 0F50 so the first entry is:

```
0F20  C3 50 0F
```

The second displayed routine's jump table entry is stored as the second jump entry in the table. The completed jump table looks like this:

```
0F20  C3 50 0F C3 90 0F
```

Ok, we have the BLOCK SHIFT set-up routine already at 0F90, now we must provide a short routine to call-up the TAPE software at 0F50.

```
0F50  2A E0 07  LD HL,(07E0)
0F53  E9        JP HL
```

The address at 07E0 is the FUNCTION-1 jump address which points to the start of the TAPE software.

(The first thing the TAPE software does is set-up the MENU driver. If you follow the instructions pointed to by the address at 07E0 you will see that the TAPE software sets up the MENU driver in the same fashion as we have done here).

Finally we move the command string and the DATA KEY RETurn instruction to 088D and jump to the MENU via its indirect "gate" at 0041.

```
0F00   21 10 0F     LD HL,0F10
0F03   11 8D 08     LD DE,088D
0F06   01 0B 00     LD BC,000B
0F09   ED B0        LDIR
0F0B   C3 41 00     JP 0041
```

The code for the MENU example is now complete.

To start execution: address 0F00 and "GO". Try selecting both routines to make sure everything works as expected.

There is not much more I can tell you about using the PERIMETER HANDLER and the MENU DRIVER except to experiment until you understand how it all goes together.

For those who have the JMON listing, the software will be easier to understand now that you know the role of the RAM variables.

The MENU DRIVER is more of a user's aid and probably won't find itself playing a starring role in your programs.

The PERIMETER HANDLER, on the other hand, could be used when ever variables are required to be passed to a routine. This is a quite common requirement. For this reason I strongly urge you to become familiar with its operation.

-Jim

BELOW IS THE DUMP OF THE ABOVE MENU EXAMPLE:

```
0F00  21 10 0F 11 8D 08 01 0B
0F18  00 ED B0 C3 41 00 FF FF
0F20  C3 50 0F C3 90 0F FF FF
0F28  FF FF FF FF FF FF FF FF
0F30  C6 6F 4F C7 E6 C2 EB C3
0F38  FF FF FF FF FF FF FF FF
0F40  04 04 04 A7 FF FF FF FF
0F48  FF FF FF FF FF FF FF FF
0F50  2A E0 07 E9 FF FF FF FF
```

TWO ANOMALIES IN THE TAPE SYSTEM

Since the release of JMON, I have realized that there a couple of unplanned side effects in the JMON tape software. The first anomaly is this:

If you load an auto-executing program in at an optional address it will execute. This will cause havoc with any routine that is not written in POSITION INDEPENDENT CODE (almost all).

This is an important point to keep in mind when loading in unknown files as the consequences of running a program in an incorrect position in memory could be a memory crash! nasty stuff!

So, if in doubt, don't use an optional load address!

The second anomaly is only minor but I thought I should um, confess.

The problem is when performing a BLOCK TEST with the tape system.

There are two possible ways the test can fail. It may fail because what is on the tape doesn't match the memory.

In this case the MENU DRIVER is re-entered with "FAIL tb" for FAIL TEST BLOCK. So far so good.

The test may also fail because the information on the tape is corrupted and as a result, the CHECKSUM on the tape will not match the added CHECKSUM. Now here is the anomaly:

The software doesn't tell you the reason for the failure was the checksum error. The menu is re-entered with "FAIL tb" for FAIL TEST BLOCK and not "FAIL CS" for fail CHECKSUM.

The end result is that you cannot be sure weather the tape is faulty or the code on the tape is not the same as the block in memory.

If this problem occurs, the only solution is to perform a "TEST CHECKSUM" on the tape before doing the BLOCK TEST. If the TEST CHECKSUM passes OK but the TEST BLOCK fails, then you know that the tape is good but its contents are not the same as the memory block.

If you keep good records of your files you won't have to perform a block test for ID purposes, and because the tape failures are rare you won't come across this problem very often. Still it's best you know and better yet, I get it off my chest.