# TEC TIMES

5/3/'90

Hi again, sorry for the delay in this newsletter but as you will see it was totally unavoidable.
For the last 3 Months my computer has been lock up at the Federal Police Headquarters. This came about when the cops raided TE because for alleged violations of the phone interception act.
My claims of total non-involvement were not accepted by the cops and they took the computer away to have a look at what was on the disks. "You should have it back by next Tuesday," they said.
It wasn't until the internal investigation branch of the Feds were called in, to the computer was returned, the next day, surprise, surprise!, just over three months from when they took it.
For three months I had no income and a lot of the work I had done for the TEC is a write off. I am out of pocket thousands of dollars and guess what? It is doubtful that I will see a cent in compensation despite the shabby way I was treated by these people, some of whom range from totally inapt to utterly despicable.
So on with the TEC news.
Issue 16 looks a fair way off yet. In fact it is not the next publication due. Because I have so much ready for the TEC, I have decided to put it in a booklet and sell it to interested TEC owners. The booklet is made-up of pages written for issue 16.
Apart from the booklet, I have some other offerings that I'll explain about after the description of the booklet.
SIMON fills the first 4 pages. At last this much talked about program is published. SIMON is fun to play and the program is fully explained so you can understand what is happening. There is good value in SIMON as it contains many useful routines.
The second program is SMON.
The S(imple)MON as its name suggests, is a very simple monitor. In fact it is only just enough to allow you to view, alter and run your programs. The SMON tutorial is a stepping stone to understanding (and writing) more advanced programs like JMON.
The SMON listing is more in the form of an assembler output listing. This means that it is symbolic and includes labels. You will be able to follow the program by reading English not HEX! The typesetting is more open and you will find it easy to read through.
The first hardware project is the TURBO OSCILLATOR.
We have been surprised with the amount of users who have modified their oscillator. Most TEC users want to be able to run their TECs at full crystal speed.
My new TURBO OSCILLATOR allows you to run your TEC at the full crystal speed and the normal half speed. The AMAZING thing about TURBO OSCILLATOR is the syncro switch allows you to happily change speed at any time and not causes the TEC to blink an eyelid. In fact you can switch between high and low speed all day and the TEC won't skip a beat.
FAST FORWARD is one of my earlier programs for the MON-1. Most of the programs I've written are programs to aid in programming. This is no exception. It's one of my first programs of any complexity. FAST FORWARD is a program designed to automatically step through the memory and display the address and data on the TEC LED display. The purpose of this is to allow you to write down your program without having to hit the "+" key all the time.
FAST FORWARD can step through the memory both forwards and backwards at slow speed and also at high speed.
THE 8255 has two pages dedicated to it. Of all the peripheral chips that can be connected to the Z80 this is by far the most handy. It is a parallel input/output interface that provides 24 I/O lines and easily interfaces to the TEC. A circuit diagram is provided to show the interconnections to the Z80.
A follow-up 2/4k eprom programmer is in the winds.
HL-TO-LED Display is a tutorial showing the concepts of how to take a 16 bit value and display it on the TEC LED display. One the basis is understood, you will easily be able to write programs that output figures to the LED display.
CRASH PROTECTOR is another hardware project to build.
How many times have you spent ages typing in a program only to have it wiped out in milliseconds by the TEC? Almost all crashes are caused when a program goes into a loop that continuously pushes onto the stack. The result is the stack quickly wraps-around and every thing in RAM is written over.
The idea of CRASH PROTECTOR is to detect when a write operation occurs at the address 07FF. This is the highest byte before the RAM and in normal operation a write to this address never happens. When the stack runs out of RAM it will try to PUSH into this ROM address.
If program execution is halted at this point we have avoided a disastrous crash.
JMON MENU DRIVER AND PERIMETER HANDLER are described in detail. All the relevant information required to operate each is given. Also featured is a powerful block relocation routine that uses the PERIMETER HANDLER to gather the start end and destination addresses. This block relocater is clever enough to work out if the destination falls within the source block and therefore take appropriate action. You'll be amazed at how complicated a block relocation can get!
Next there is a page containing a few reader send-ins. This page was added to later JMON listings so some of you may have it already. It is included for those who bought the earlier listings.
The final page is a guide to 16 bit compares.
These always cause difficulties as the Z80 doesn't have any instructions designed for the purpose. The article describes how to use the SBC HL,XX instruction correctly and how to interpret the results correctly. Routines are provided for the all the possible conditions you can test for.

## THE DISASSEMBLER ROM

Below is a description of my disassembler ROM for the TEC:

* DISASSEMBLES ALL Z80 INSTRUCTIONS.
* CALCULATES AND OUTPUTS THE TARGET ADDRESS OF RELATIVE JUMPS.
* DISPLAYS OPERATIONAL CODES AS WELL AS MNEMONICS.
* CLOSELY FOLLOWS ZILOG ASSEMBLER SYNTAX
* DISASSEMBLER ROUTINE CAN BE CALLED AS A SUB-ROUTINE.
* 16 BIT VALUES DISPLAYED IN MNEMONICS COLUMN ARE IN PROPER ORDER.
* ROUTINE IS LESS THAN 2k LONG.
* CAN USE JMON's PERIMETER HANDLER TO ENTER START AND END ADDRESS.
* TWO VERSIONS: VER 1.0P FOR PRINTER; VER 1.0D FOR LCD MODULE.
* LCD AND PRINTER OUTPUT ROUTINES TO SUPPLEMENT EACH VERSION.
* INSTRUCTIONS INCLUDED.

## NEW SALES OUTLET

Both the TEC-pack and the disassembler are only available from the Talking Electronics Shop. Please send orders there, not to Rosewarne Ave. This is the only way I can keep track of things. When sending-in, please include a Self-addressed envelop so I can mail you news of further up-dates.

## PRICES

The TEC-pack is $12.00 ($14.00 posted), and the Disassembler ROM is 22.00 (23.50 posted) for either version, (please specify). Order both together for only $30.00 ($2.00 extra for post)

[TE]

# SIMON

By J. Robertson.

This program is very similar to a commercially available game of the same name.

Generally the game consists of 4 coloured buttons that must be pressed in a particular order as defined by the running of the game.

The game starts off simple as only a short sequence must be remembered but as the entries increase, the game speeds up so that not only does the pattern need to be remembered but the speed of playing increases.

By the time 12-15 tones are played, your recall is stretched to the limit. We got up to 20 but this was exceptional.

This game is played on the TEC (also TEC 1 and TEC 1B) and uses keys 0, 4, 8 and C to play and "GO" to restart. The readout is on segments "g" of the 4 right hand displays and the tone comes from the speaker.

The game increases by one after each successful sequence and if a wrong key is pressed, a "you lose" tone is emitted and your score is displayed.

The program is entered at 0A00 to 0AFF and a full page of random numbers are placed at 0B00 to 0BFF by a very clever means.

The game requires 4 different values 1, 2, 4 and 8 to be entered at a completely random manner into the 0B00 page but the production of these numbers is better left to a computer.

So we have created a random number generator that uses the delay between successive button presses to produce the numbers.

The program for doing this is located at 0AE5 and is accessed by addressing 0AE5 and pressing GO. This is the first thing that you must do before running SIMON and the only monotonous part is to press any key 256 times!

When this is done, the program jumps out of the loop and resets.

If you look at 0B00 to 0BFF you will find your efforts recorded as 1, 2, 4 and 8.

The program produces patterns that you would never produce yourself - like 3 or 4 of the same number in a row.

The table is accessed at the beginning of the game via the "L" register and the value in this register is obtained from the refresh register "R."

The program runs independent of the monitor as the program polls the keyboard (monitors the output of the keyboard encoder) and has its own independent look-up table for the hex display code and separate sound routine for the tones.

This subroutine actually produces the tones and display values at the same time (refer to 0AA2 to 0AC9) and this is quite a brilliant piece of programming.

The program contains other clever ideas and is fully documented to assist in learning machine code.

## HARDWARE MODIFICATION

A hardware modification is required to run SIMON, IF THE LCD DISPLAY INTERFACE BOARD IS NOT FITTED.

The modification is to add a 4k7 between NMI of the Z-80 (pin 17) and data line D6.


Here is an overview of the technical side of the program.

## THE SET UP

At the beginning of the game, the frame counter is set to 1 and a random number is taken from the refresh register.

A short delay is then called to allow the player to release his hand and get ready.

## THE COMPUTERS LOOP

This loop plays the sound and lights and the player must repeat this sequence to gain a point.

## THE PLAYERS LOOP

The first part consists of a keyboard read loop. When a key is pressed, it is checked to see if it is valid and then converted as described below. It is then compared to the random table entry

and if right, the player's choice is echoed and the player's loop is repeated until the sequence is complete.

Then the count is increased by one and the program jumps back into the computer loop.

If incorrect, the program jumps to the error/end-of-game-loop.

## THE KEYBOARD DISCRIMINATION

During the game, only the keys 0, 4, 8 and "C" are accepted as valid. To do this, the input values are rotated right TWICE with branch carry. Bit 0 falls into the carry and also bit 7. Therefore zero remains zero, 4 becomes 1, 8 becomes 2 and "C" becomes 3. Then the result is compared to 4 and if it is lower than (carry generated) it is accepted as valid, otherwise it is ignored and the program jumps back and clears the input buffer and waits for a new input.

## THE KEYBOARD CONVERSION

This is done just after the keyboard discrimination and the input value has been altered to one of four values: 0, 1, 2, and 3. These are then used as displacements added to "HL." The result is HL points to a table with values as follows: 8, 4, 2, and 1. The entry is now compared to the random page table entry and if the player has correctly selected the right key, the two values will be equal. Then the pointer is incremented to the next entry and the count in "B" is decremented and if not zero, the player loop is re-executed.

## ERROR/END-OF-GAME LOOP

This consists of a "YOU LOSE" tone, a score display and a random number generator using the D register.

# SIMON

Created by Jim Robertson

| | | | | |
|---|---|---|---|---|
| SET UP | 0A00 | ED 5F | LD A,R | Load A with random number from refresh register. |
| | 0A02 | 6F | LD L,A | Place in "L" register. |
| | 0A03 | 0E 01 | LD C,01 | "C" counts the number of frames. |
| | 0A05 | CD 8E 0A | CALL 0A8E | Call delay before start to give operator time to remove hand. |
| | 0A08 | 26 0B | LD H,0B | H is high address byte of random look-up table. |
| | 0A0A | 41 | LD B,C | B is the working counter |
| | 0A0B | E5 | PUSH HL | Save the random table pointer. |
| START COMPUTER LOOP | 0A0C | 7E | LD A,(HL) | Get random value and load into "A". |
| | 0A0D | CD A2 0A | CALL 0AA2 | Call sound and lights routine. |
| | 0A10 | 2C | INC L | Increment random number pointer. |
| | 0A11 | CD 8E 0A | CALL 0A8E | Call the delay that shortens on each frame. |
| | 0A14 | 10 F6 | DJNZ, 0A0C | Decrement B and jump if not zero. |
| | 0A16 | E1 | POP HL | Get original starting point in look-up table. |
| | 0A17 | 41 | LD B,C | Load frame number into "B". |
| | 0A18 | E5 | PUSH HL | Save random pointer number on stack. |
| START PLAYER LOOP | 0A19 | CD CA 0A | CALL 0ACA | Go to key handler routine. |
| KEY PRESSED VALID? | 0A1C | 20 FB | JR NZ, 0A19 | Loop key handler routine until a key is pressed. |
| | 0A1E | 0F | RRCA | Check for 0,4,8 or C by shifting twice to right. |
| | 0A1F | 0F | RRCA | For the above keys, the result will be 0=0, 4=1, 8=2 and C=3. |
| | 0A20 | FE 04 | CP 04 | Compare the accumulator with 4 |
| VALID KEY | 0A22 | 30 F5 | JR NC, 0A19 | Jump relative to 0A19 if no carry is produced. |
| | 0A24 | E5 | PUSH HL | Save pointer again on stack, once for player, & for computer. |
| | 0A25 | 21 D7 0A | LD HL,0AD7 | Load HL with starting address of display table. |
| | 0A28 | 85 | ADD A,L | Add corresponding discrete key value to get display value. |
| | 0A29 | 6F | LD L,A | |
| | 0A2A | 7E | LD A,(HL) | Load the value pointed at by HL, in the accumulator. |
| | 0A2B | E1 | POP HL | Recover pointer and compare players answer |
| CORRECT? NO | 0A2C | BE | CP (HL) | to entry via the CP (HL) command. |
| | 0A2D | 20 12 | JR NZ, 0A41 | Jump if player wrong. |
| | 0A2F | CD A2 0A | CALL 0AA2 | Call sound and lights routine to echo players choice. |
| | 0A32 | 2C | INC L | Increment pointer. |
| | 0A33 | CD CA 0A | CALL 0ACA | Call key handler routine |
| PLAYER PUSHED ALL OK | 0A36 | 28 FB | JR Z, 0A33 | Loop until key released. |
| | 0A38 | 10 DF | DJNZ, 0A19 | Jump back to players loop until B is zero. |
| | 0A3A | E1 | POP HL | Pop the start of random pointer |
| | 0A3B | CD 8E 0A | CALL 0A8E | Call delay that shortens on each frame. |
| | 0A3E | 0C | INC C | Increment the counter for the number of frames. |
| END OF GAME | 0A3F | 18 C9 | JR, 0A0A | Jump to start of computer loop. |
| ERROR ROUTINE | 0A41 | 3E 30 | LD A,30 | Load "A" with "you lose" tone. |
| | 0A43 | CD A2 0A | CALL 0AA2 | Call sound and lights. |
| | 0A46 | 41 | LD B,C | Start to convert count to decimal. |
| HEX TO BCD CONVERSION | 0A47 | AF | XOR A | Zero the accumulator. |
| | 0A48 | 3C | INC A | Increment A |
| | 0A49 | 27 | DAA | Decimal adjust the accumulator. |
| | 0A4A | 10 FC | DJNZ, 0A48 | Loop until B is zero. |
| | 0A4C | 3D | DEC A | Subtract 1 from the accumulator. |
| | 0A4D | 27 | DAA | Decimal adjust the accumulator. |
| MULTIPLEX STARTS HERE | 0A4E | 4F | LD C,A | Store the decimal number in "C" |
| | 0A4F | AF | XOR A | Zero the accumulator. |
| | 0A50 | D3 02 | OUT (02),A | Output the accumulator to port 2. |
| | 0A52 | 3E 04 | LD A,04 | Load A with 4 for first display multiplex. |
| BCD TO DISPLAY CODE AND MULTIPLEX LOOP | 0A54 | D3 01 | OUT (01),A | Output to port 1 to turn on the right hand display |
| | 0A56 | 79 | LD A,C | Get decimal value |
| | 0A57 | CD 83 0A | CALL 0A83 | Call BCD to hex display code and output routine. |
| | 0A5A | 06 00 | LD B,00 | Load B to create a delay value |
| | 0A5C | 10 FE | DJNZ, 0A5C | Execute the delay via the DJNZ instruction. |
| | 0A5E | AF | XOR A | Clear the display |
| | 0A5F | D3 02 | OUT (02),A | Output to port 2. |
| | 0A61 | 3E 08 | LD A,08 | Select the left hand display. |
| | 0A63 | D3 01 | OUT (01),A | Output to port 1. |
| | 0A65 | 79 | LD A,C | Get BCD score. |

| | | | | |
|---|---|---|---|---|
| | 0A66 | 0F | RRCA | Shift most significant nibble 4 places right. |
| | 0A67 | 0F | RRCA | " " |
| | 0A68 | 0F | RRCA | " " |
| | 0A69 | 0F | RRCA | " " |
| | 0A6A | CD 83 0A | CALL 0A83 | Call convert and display routine. |
| | 0A6D | 06 00 | LD B,00 | Load B with a delay value |
| | 0A6F | 10 FE | DJNZ, 0A6F | Delay routine |
| | 0A71 | CD CA 0A | CALL 0ACA | Call Key Handler routine |
| | 0A74 | 14 | INC D | Increment Random number, created by the random |
| KEY | 0A75 | FE 12 | CP 12 | time when player restarts game. Compare with 12. |
| TEST | 0A77 | 20 D6 | JR NZ, 0A4F | Loop if not "GO" key. |
| | 0A79 | 6A | LD L,D | Put random number into register L. |
| GAME | 0A7A | AF | XOR A | Clear accumulator |
| RESTART | 0A7B | D3 01 | OUT (01),A | and output to display before restart. |
| | 0A7D | CD A2 0A | CALL 0AA2 | Call sound and lights to signify start of new game. |
| | 0A80 | C3 03 0A | JP 0A03 | Jump to start. |
| BCD TO HEX | 0A83 | E6 0F | AND 0F | Mask off high nibble. |
| DISPLAY | 0A85 | 21 DB 0A | LD HL,0ADB | Table at 0ADB is display table. Load HL with start |
| CODE | 0A88 | 85 | ADD A,L | of display table. ADD A to base and create a new |
| | 0A89 | 6F | LD L,A | pointer value. |
| | 0A8A | 7E | LD A,(HL) | Load the value pointed to by HL into the |
| | 0A8B | D3 02 | OUT (02),A | accumulator and output to port 2. |
| | 0A8D | C9 | RET | Return. |
| | 0A8E | 11 00 40 | LD DE,4000 | This is the delay that shortens on each new frame. |
| DELAY | 0A91 | 79 | LD A,C | Load the count register with 4,000 and load the |
| | 0A92 | 07 | RLCA | frame counter into A. Shift "A" left to increase |
| | 0A93 | 07 | RLCA | its value 4 times. Decrement the count register |
| | 0A94 | 15 | DEC D | by an amount equal to 4 times the number of frames |
| | 0A95 | 3D | DEC A | This produces the speed-up on each game. |
| | 0A96 | 20 FC | JR NZ, 0A94 | |
| | 0A98 | 1B | DEC DE | Actual delay routine starts here. Decrement the count |
| | 0A99 | 7A | LD A,D | register, load the high nibble into A, OR with E |
| | 0A9A | B3 | OR E | and loop until the count register is zero. |
| | 0A9B | 20 FB | JR NZ, 0A98 | |
| | 0A9D | 3E 04 | LD A,04 | Load "A" with 4 to turn on segment "g" and output |
| | 0A9F | D3 02 | OUT (02),A | to the display. |
| | 0AA1 | C9 | RET | return. |
| | 0AA2 | E5 | PUSH HL | SOUND AND LIGHTS ROUTINE |
| SOUND | 0AA3 | D5 | PUSH DE | Save registers |
| AND | 0AA4 | C5 | PUSH BC | |
| LIGHTS | 0AA5 | 4F | LD C,A | A= display value |
| | 0AA6 | 07 | RLCA | Double it for more significance in tone |
| | 0AA7 | C6 18 | ADD A,18 | Add 18 to "A". This is how a discrete tone is |
| | 0AA9 | 21 E0 01 | LD HL,01E0 | produced for each display. HL = No of cycles. |
| | 0AAC | 47 | LD B,A | |
| | 0AAD | 79 | LD A,C | |
| | 0AAE | 11 01 00 | LD DE,0001 | |
| | 0AB1 | 48 | LD C,B | |
| | 0AB2 | E6 0F | AND 0F | Mask off unwanted bits from display value. |
| | 0AB4 | D3 01 | OUT (01),A | Output to port 1. |
| | 0AB6 | 41 | LD B,C | Short "tone" delay. |
| | 0AB7 | 10 FE | DJNZ, 0AB7 | |
| | 0AB9 | EE 80 | XOR 80 | Toggle speaker |
| | 0ABB | ED 52 | SBC HL,DE | Subtract one from count. |
| | 0ABD | 20 F5 | JR NZ, 0AB4 | Loop until zero. |
| | 0ABF | C1 | POP BC | |
| | 0AC0 | D1 | POP DE | Recover registers |
| | 0AC1 | E1 | POP HL | |
| | 0AC2 | 3E 04 | LD A,04 | Keep "g" segment ON. |
| | 0AC4 | D3 02 | OUT (02),A | Output to port 2. |
| | 0AC6 | AF | XOR A | Clear display |
| | 0AC7 | D3 01 | OUT (01),A | |
| | 0AC9 | C9 | RET | Return. |

```
0ACA   DB 03       IN A,03      This is the KEY HANDLER ROUTINE
0ACC   CB 77       BIT 6,A      Input from key encoder or latch chip
0ACE   C0          RET NZ       Bit low + key press
0ACF   DB 00       IN A,00      Return if no key pressed with "ZERO" flag set
0AD1   E6 1F       AND 1F       Input from key encoder
0AD3   5F          LD E,A       Mask unwanted bits and save in "E"
0AD4   AF          XOR A        Xor A to clear flags to signal "key pressed"
0AD5   7B          LD A,E       Recover key value
0AD6   C9          RET          Return

0AD7  08 04 02 01
0ADB  EB 28 CD AD 2E A7 E7 29 EF 2F

0AE5   06 00       LD B,00      B is page counter (256 Bytes)
0AE7   21 00 0B    LD HL,0B00   HL=start of random page
0AEA   16 11       LD D,11      Random number is
0AEC   CB 02       RLC D        produced by rotating D
0AEE   CD CA 0A    CALL 0ACA    Call KEY CHECK
0AF1   20 F9       JR NZ 0AEC   Loop until key pressed
0AF3   7A          LD A,D       Get random value
0AF4   E6 0F       AND 0F       Mask off high order nibble
0AF6   77          LD (HL),A    Store on random page
0AF7   23          INC HL       Inc HL to next location
0AF8   CD CA 0A    CALL 0ACA    Wait until key
0AFB   28 FB       JR Z 0AF8    is released
0AFD   10 ED       DJNZ 0AEC    Decrement byte counter and loop if not done
0AFF   C7          RST 00       Return to MONitor
```

This loop also poles the keyboard for the "GO" key. When pressed, the program jumps back to the start.

## THE SCORE

The score indicates the number of correct frames. It is one less than the current frame number (as the frame counter is incremented before the start of the current frame).

## THE HEX-TO-BCD CONVERSION

This conversion calculates the correct score in the accumulator. The BCD score is then preserved in the C register where it is continuously used in the BCD-to-HEX display code and multiplexing routine.

## THE BCD-TO-DISPLAY CODE/MULTIPLEXING ROUTINE

The BCD score is converted one nibble at a time. Immediately after each nibble is converted, it is outputted onto a display for a short period. The other nibble is then converted and displayed. The program loops continuously converting and displaying each nibble until "GO" is pressed. During this loop, the D register is constantly being incremented to create a random value for which to restart the game.

## THE DELAY

This delay separates the tones and also gives the player time to get ready after the start of the game.

As the number of frames increases, the delay between each decreases. This is done by subtracting from the most significant byte of the delay value.

## THE SOUND AND LIGHTS ROUTINE.

This routine simultaneously plays the tones and turns on the required LEDS. At its heart, is a simple tone generator that has its frequency altered by the LED display value. This gives the unique tones during the game and the "YOU LOSE" tone as well as the restart tone.

## THE KEYBOARD HANDLER

In order to allow complete compatibility with any MONitor, Simon has its own keyboard routine. This routine inputs from port 3 and if there is no input device on this port, the data available line is able to take bit 6 high or low. If the input latch is fitted on port 3, the data available bit is latched in through this.

Either way works and provided one or the other is fitted, Simon will work regardless of what MONitor is used.

## SIMON UP-DATE

One thing not pointed out very well in the simon text is that Simon will run with any MONitor or other program that may be at 0000 BUT THE HARDWARE MOD MUST BE DONE ONLY IF THE DAT BOARD IS NOT FITTED.

While the description strongly hints at this, I have found that you cannot count on all the people picking it up.

The description of the hardware mod refers to the LCD display interface board. This is the DAT BOARD. This description was written before the DAT BOARD was named.

The original text and diagram suggests that one end of the 4k7 resistor go to pin 17. I now recommend that instead you connect it to pin 15 of the 4049. The other end of the resistor is connected to pin 10 of the Z80 (D6). This frees the NMI pin up so that it is available for future use.

It is ok to have both the DAT BOARD and the resistor mod fitted but the resistor is not necessary in this case.

The original computer file of SIMON was lost during a computer malfunction and as a result this up-date has to been tacked onto the end.

# INTRODUCING SMON
## A TUTORIAL MONITOR

The S(imple)MON as its name suggests, is a very simple monitor. In fact it is only just enough to allow you to view, alter, and run your own programs.

The blandness of SMON is by design. This MONitor has been designed purely as a easy-to-understand tutorial, not as a fully functioned MONitor like JMON.

The JMON description lacked ease-of-understanding because JMON was not designed to be an easily read program. The SMON tutorial is a stepping stone to understanding (and writing) more advanced programs like JMON.

The SMON listing is more in the form of an assembler output listing. This means that it is symbolic and includes labels. You will be able to follow the program by reading English not HEX!

The typesetting is more open and you will find it a pleasure to read through.

### RUNNING SMON

Before you can fully understand how the SMON program works, you need to see it working so you can understand the action of the program. To get a working SMON requires either a NVR or an extra RAM chip and a TEC mod that allows you to switch the expansion port to the 0000 (this is a highly recommended mod).

You also are required to sit down and type it in. This should not be much of a problem as it is less than a page (256) bytes long. JMON owners are laughing as they don't have to increment the RAM pointer after each byte and also can save it on tape for future use.

Once you have SMON up and running, get to know its actions. Notice that there are only two modes, the ADDRESS and DATA modes. Notice that the ADDRESS mode only effects the operation of the DATA keys not the control keys. Also notice that you can run a program straight from the ADDR mode by hitting "GO" and notice there is no auto zeroing as with MON-1.

Make sure you get to know the outside actions of SMON fully before trying to understand the inside working of the software.

### SMON OVERVIEW

There are 3 main sections to SMON. The first is the main program. It is responsible for the set-up of variables, the calling of sub-routines and the processing of key strokes. The main body is located from 0000 to 0065 (it fits neatly under the NMI handler).

There is a slight difference between the main body of JMON and SMON. The difference is JMON calls a subroutine that scans the keyboard and display and returns when it finds a key press. SMON calls the scan routine and then looks for a key press itself.

The second section is the NMI handler at 0066. This is pinched straight from MON-1 except that SMON pushes AF and MON-1 doesn't.

The third section is collectively the sub-routines. The actions of the subroutines are to produce the tones, scan the display, set the dots and convert HEX values into display code. The subroutines are located at 0070.

A classic strategy is employed by SMON. Those of you who have read through the JMON listing will be familiar with it. The strategy is that control byte(s) are used to flag the current operating modes. Outside appearances, (Eg. the displays), are set-up by the master routine by referring to these bytes. This means that routines can change the operating mode of the software simply by changing a byte. The routines do not need to be concerned with up-dating the outside appearance, they leave this for the master.

Applying this to SMON, the initial variables are set-up from address 0000 through to 0014. The display up-dating is performed by the sub-routines called by the instructions at 0017 to 001D.

The key handler section, which alters these variable bytes starts at 0026. After the key handler is finished it jumps to 0010 (or 000D if the AD key was pressed as it must store the new control byte) and displays are up-dated.

### SMON VARIBLES

SMON has only two varibles. They are the CONTROL BYTE and the RAM pointer. The control byte flags between the two operating modes, ADDRESS and DATA, by the state of bit 4. If bit 4 is a logic 1 then SMON is in the ADDRESS mode. If it is a zero then SMON is in the DATA mode. No other bits are used in the control byte. The control byte is stored at 0F08.

The RAM pointer holds the address of the RAM location the SMON is currently displaying. The RAM pointer is stored at 0F06.

### POINTS OF INTEREST

If you are wondering why the stack pointer is loaded at 1000 and not 0FFF, the reason is the stack pointer is decremented by one BEFORE anything is pushed onto it. Therefore the first value pushed onto it will be stored at 0FFE and 0FFF.

Another point of interest is that there is a "SETDOTS" routine but no "RESET" dots routine, so how do the dots get erased from the display?

The answer is the dots are removed from the display buffer when a new value is written in by the HEX-to-display code routine. The HEX-to-display code routine is always called before the SETDOTS routine so when the SETDOTS routine is reached all the dots are cleared.

Have a look at the AD key handler at 0048. Address 0F08 and press the AD key. Watch what happens to the value at 0F08 each time you hit the AD keythat . You should now see how the ADDR and DATA modes are toggled between.

Now look at the set dots routine at 00AB. Do you see how the difference between the ADDR and DATA modes is detected and then acted upon?

Another feature to look closely at is the method used to shift a new data nibble into the current RAM location. The code to do this is at 0062.

The code that enters a new nibble into the RAM pointer buffer, as required when in the ADDR mode, is at 005C.

Grab your copy of issue 13 and turn to page 16. You will find my three digit counter (I wrote the counter program but not the comments).

Look at convert to display code subroutine at 0A00. It is identical in operation to the one used here in SMON. If you look closely, you will notice the CALL and RETurn instructions at 0A09 are missing from SMON. I will leave it to you to discover why the CALL and RETurn are not needed. (Hint: look at the CALL address).

Also look at the differences between the scanning routines, SMON's scans 6 digit while the three digit counter only scans 3. Do you see how this difference is achieved without the use of a counter?

Finally, a slight change to the SMON's tone routine makes SMON's shorter and easier to understand than JMON's.

There is a lot of information and reference material packed into this simple-to-understand program.

```
0000  31 00 10                        LD SP,1000            ;set the stack to top of RAM+1
0003  21 00 08                        LD HL,0800            ;load HL with first RAM location
0006  22 06 0F                        LD (PTR_BUFF),HL      ;and put it in RAM pointer buffer
0009  CD C1 00                        CALL RESET_TONES      ;call double reset tone
000C  AF          CLR_CON_BYT:        XOR A                 ;clear control byte
      ;When the AD key is pressed, the MONitor jumps to here to store a new control byte provided by the AD key handler.
000D  32 08 0F    STR_CON_BYT:        LD (CONT_BUFF),A      ;store control byte
      ;MON jumps here to clear the key buffer in the interrupt reg after key processing (except for AD see above).
0010  3E FF       CLR_KEY_FLG:        LD A,FF               ;set interrupt vector register to FF to signify no
0012  ED 47                           LD I,A                ;key press:
0014  2A 06 0F                        LD HL,(PTR_BUFF)      ;put RAM pointer into HL and call
0017  CD 8B 00                        CALL CON_HL_A         ;HL and (HL) to display code conversion
001A  CD AB 00                        CALL SET_DOTS         ;call set dots
001D  CD 70 00    KEY_LOOP:           CALL SCAN             ;call scan: Key loop jumps here until key press
0020  ED 57                           LD A,I                ;get byte from interrupt register
0022  FE FF                           CP FF                 ;test for FF: If it is then no key is
0024  28 F7                           JR Z, KEY_LOOP        ;pressed so keep looping else continue
0026  F5                              PUSH AF               ;on and process key: save key value
0027  CD C4 00                        CALL KEY_TONE         ;call key press tone
002A  F1                              POP AF                ;recover key value
002B  2A 06 0F                        LD HL, (PTR_BUFF)     ;put RAM pointer into HL
002E  CB 67                           BIT 4,A               ;if the key +, -, go or AD then bit 4 is
0030  28 1D                           JR Z, DAT_KEY_PROC    ;set: jump if data key
0032  FE 10                           CP 10                 ;else process control key here: Is it "+"
0034  20 06                           JR NZ, CP_MINUS       ;jump if not
0036  23                              INC HL                ;else increment RAM pointer
0037  22 06 0F    PTR_UPDATE:         LD (PTR_BUFF), HL     ;place new RAM pointer in its buffer
003A  18 D4                           JR CLR_KEY_FLG        ;jump to set key buffer and up-date display
003C  FE 11       CP_MINUS:           CP 11                 ;is key "-"?
003E  20 03                           JR NZ, CP_GO          ;jump if not
0040  2B                              DEC HL                ;decrement RAM pointer
0041  18 F4                           JR PTR_UPDATE         ;jump to up-date its buffer
0043  FE 12       CP_GO:              CP 12                 ;is key the "GO" key?
0045  20 01                           JR NZ, AD_KEY         ;jump if not
0047  E9                              JP (HL)               ;else jump to current RAM location
0048  3A 08 0F    AD_KEY:             LD A, (CONT_BUFF)     ;key MUST BE "AD": GET control byte in A
004B  EE 10                           XOR 10                ;toggle the mode Eg. if ADDR now DATA and
004D  18 BE                           JR STR_CON_BYT        ;vica-versa: Jump to store new control byte
004F  47          D_KEY_PROC:         LD B,A                ;DATA KEY HANDLER: save key value in B
0050  3A 08 0F                        LD A, (CONT_BUFF)     ;get control byte in A
0053  CB 67                           BIT 4,A               ;test for which mode
0055  78                              LD A,B                ;put key value back in A
0056  20 04                           JRNZ, D_KEY_AD_MD     ;jump if ADDR mode
0058  ED 6F                           RLD                   ;else shift nibble into RAM location
005A  18 B4                           JR CLR_KEY_FLG        ;jump to up-date display
005C  21 06 0F    D_KEY_AD_MD:        LD HL, PTR_BUFF       ;DATA key in ADDR mode: point HL at RAM
005F  ED 6F                           RLD                   ;pointer buffer and shift
0061  23                              INC HL                ;the new nibble in
0062  ED 6F                           RLD                   ;and shift the carry out nibble into second
0064  18 AA                           JR CLR_KEY_FLG        ;byte: Jump to up-date displays
0066  F5          NMI_HANDLER:        PUSH AF               ;NMI HANDLER: Save A
0067  DB 00                           IN A,(00)             ;get key value
0069  E6 1F                           AND 1F                ;mask off junk bits
006B  ED 47                           LD I,A                ;save it in interrupt vector register
006D  F1                              POP AF                ;recover AF
006E  C9                              RET                   ;return
006F  FF                              RST 38                ;ignore
0070  06 20       SCAN:               LD B,20               ;SCAN: load B with scan bit
0072  21 00 0F                        LD HL, DISP_BUFF      ;point HL at display buffer
0075  7E          SCAN_LOOP:          LD A,(HL)             ;get first display digit
0076  D3 02                           OUT (02),A            ;output to port 2
0078  78                              LD A,B                ;put scan bit in A
0079  D3 01                           OUT (01),A            ;output to port 1
007B  06 80                           LD B,80               ;use B for a short
007D  10 FE       D_LOOP:             DJNZ, D_LOOP          ;display delay
007F  23                              INC HL                ;point HL to next display byte
0080  47                              LD B,A                ;put scan bit back into B
0081  AF                              XOR A                 ;clear the last port
```

```
0082  D3 01                            OUT (01),A            ;switched on to prevent "ghosting"
0084  CB 08                            RRC B                 ;shift scan bit to next digit
0086  30 ED                            JR NC SCAN_LOOP       ;jump if scan bit didn't "fall" into carry
0088  D3 02                            OUT (02),A            ;else all digits scanned: (unnecessarily)
008A  C9                               RET                   ;clear port 2 and return
```

;HL is converted to display code via the convert A routine. After H is converted the corresponding two display bytes are at the lower end of the display buffer. The next two bytes in the display buffer are for the display codes for L.

```
008B  01 00 0F    CON_HL_A:            LD BC, DISP_BUFF      ;HL CONVERT: point BC to display buffer
008E  7C                               LD A,H                ;get high byte
008F  CD 97 00                         CALL CON_A            ;call convert A
0092  7D                               LD A,L                ;get low byte
0093  CD 97 00                         CALL CON_A            ;call convert A
```

;After HL is converted, the byte at (HL) is converted. This is the value that appears on the DATA displays.

```
0096  7E                               LD A,(HL)             ;get contents of RAM location
0097  F5          CON_A:               PUSH AF               ;save byte for second nibble convert
0098  07                               RLCA                  ;shift
0099  07                               RLCA                  ;high nibble to
009A  07                               RLCA                  ;low nibble spot
009B  07                               RLCA                  ;for ease of convertion
009C  CD A0 00                         CALL CON_NIBBLE       ;call nibble convert
009F  F1                               POP AF                ;recover A for second nibble Convert
00A0  E6 0F       CON_NIBBLE:          AND 0F                ;mask off unwanted bits
00A2  11 E0 00                         LD DE, DISP_COD_TAB   ;point DE to conversion table
00A5  83                               ADD A,E               ;add nibble value to table pointer
00A6  5F                               LD E,A                ;put new table pointer low byte into DE
00A7  1A                               LD A,(DE)             ;get display code and put in display buffer
00A8  02                               LD (BC),A             ;any set dot is cleared by this operation
00A9  03                               INC BC                ;point BC to next display buffer
00AA  C9                               RET                   ;done
```

The set dots routine causes either the DATA or ADDR dots to be set on the LED display. This is achveived by setting bit 4 of the DATA or ADDR section of the display buffer. Because the DATA section is at the higher end of the display buffer, HL is loaded to point the end of the diplay buffer and is decremented down two bytes in the case of ADDR mode. This is more efficent than loading HL several times.

```
00AB  21 05 0F    SETDOTS:             LD HL, DISP_BUFF_END  ;point HL to data end of display buffer
00AE  06 02                            LD B,02               ;set B for 2 dots
00B0  3A 08 0F                         LD A, (CONT_BUFF)     ;get control byte
00B3  CB 67                            BIT 4,A               ;test mode
00B5  28 04                            JR Z, DO_SET          ;jump if it is DATA mode
00B7  2B                               DEC HL                ;else move HL to lowest
00B8  2B                               DEC HL                ;ADDR display buffer
00B9  06 04                            LD B,04               ;set B for 4 dots
00BB  CB E6       DO_SET:              SET 4,(HL)            ;set dot
00BD  2B                               DEC HL                ;point to next digit
00BE  10 FB                            DJNZ, DO_SET          ;do until B=0
00C0  C9                               RET                   ;done
```

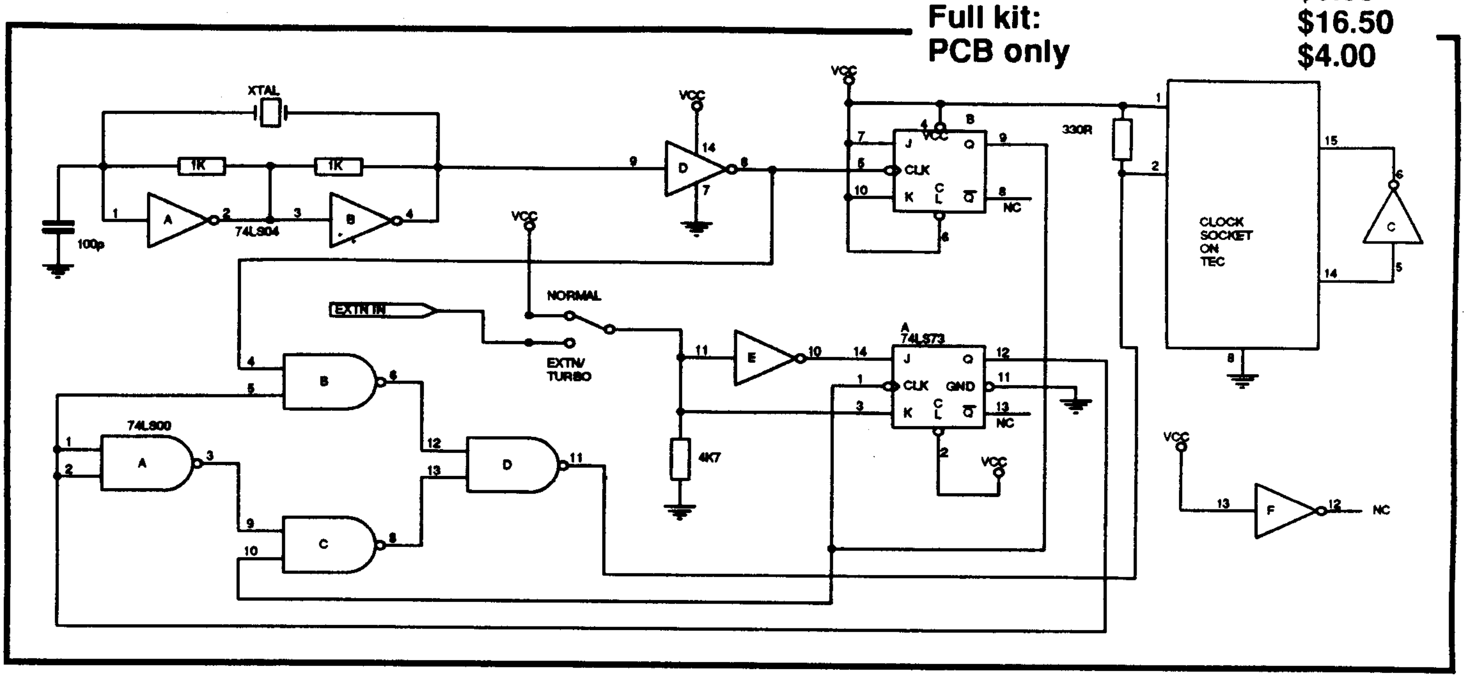
The double reset tone is created by calling the key tone then returning to the key press tone.

```
00C1  CD C4 00    RESET_TONES:         CALL KEY_TONE         ;call key tone
00C4  0E 40       KEY_TONE:            LD C,40               ;set C for half cycle count
00C6  AF                               XOR A                 ;turn off speaker bit
00C7  D3 01       TONE_LOOP:           OUT (01),A            ;on bit 7 of port 1
00C9  06 40                            LD B,40               ;put delay period into B
00CB  10 FE       TONE_DELAY:          DJNZ, TONE_DELAY      ;and do delay
00CD  EE 80                            XOR 80                ;toggle speaker bit in A
00CF  0D                               DEC C                 ;count down cycles
00D0  20 F5                            JR NZ, TONE_LOOP      ;toggle speaker bit until L is 0
00D1  C9                               RET                   ;done

00E0              DISP_COD_TAB         DEFB EB, 28, CD, AD   ;display codes for 0, 1, 2, 3
00E4                                   DEFB 2E, A7, E7, 29   ;display codes for 4, 5, 6, 7
00E8                                   DEFB EF, 2F, 6F, E6   ;display codes for 8, 9, A, B
00EC                                   DEFB C3, EC, C7, 47   ;display codes for C, D, E, F

                  PTR_BUFF             EQU 0F06              ;set PTR_BUFF to 0F06
                  CONT_BUFF            EQU 0F08              ;set CONT_BUFF to 0F08
                  DISP_BUFF            EQU 0F00              ;set DISP_BUFF to 0F00
                  DISP_BUFF_END        EQU 0F05              ;set DISP_BUFF_END to 0F05
```

# TURBO OSCILLATOR

## With syncro-switch

**Short form kit & PCB: $8.00**
**Full kit: $16.50**
**PCB only $4.00**




## TURBO OSCILLATOR CIRCUIT DIAGRAM

The crystal oscillator has been the most popular TEC add-on. Since the arrival of the DAT BOARD with its tape system, the demand has soared. We have also been surprised with the amount of users who have modified their oscillator. Most TEC users want to be able to run their TECs at full crystal speed.

To do this you can fit a switch onto the oscillator board that switches the undivided clock to the output. Not only will you have to cut tracks, but you also will have to fit a switch on the board which is a bit difficult as there is no where to solder it to.

Apart from the construction hassles, the above idea fails as the TEC will crash each time you change speed and has to be reset. Alternatively, you could hold down the reset key when you change the speed. Either way, the software is interrupted.

This means that if you over-estimate a software delay you can't "TURBO" pass it, or speed through JMON's single stepper etc.

The oscillator is also hacked to provide the 3.58MHz for the TEC speech.

My new TURBO OSCILLATOR with syncro switch solves all these problems and can be expanded to add new features.

Naturally enough TURBO OSCILLATOR allows you to run your TEC at the full crystal speed and the normal half speed. The AMAZING thing about TURBO OSCILLATOR is the syncro switch allows you to happily change speed at any time and not causes the TEC to blink an eyelid. In fact you can switch between high and low speed all day and the TEC won't skip a beat.

To achieve this unique ability, two extra stages have been added to the old oscillator. These are a synchronizer and a data selector.

The SYNCHRONIZER is made-up of the unused JK flip-flop and an unused inverter. Together they form a D flip-flop and the role of this D flip-flop is to hold back the select signal from the DATA SELECTOR until the both the clock lines are at a suitable level. The falling edge of the lower speed clock is used in the TURBO OSCILLATOR as you can be sure that the faster clock is also on its falling edge by virtue of it being an even multiple of the lower speed clock.

This means the clocks are swapped over just as both have gone low and the transition is glitch free. The Z80 carries on quietly about its business.

The actual gating is done by feeding the lower speed clock to the clock input of the JK cum D flip-flop. The flip-flop transfers the speed select signal to the data selector on the falling edge. By the time the new signal arrives at the data selector, the clocks have gone low.

The role of the DATA SELECTOR is to choose one of the two clocks and pass it to the TEC. It is a simple two line to one line selector and has been made-up of the four NAND gates from the added 74LS00.

The oscillator section is identical to the original.

TURBO OSCILLATOR has pads for connecting devices to the clock outputs.

The LS family of chips are specified in preference to the standard TTL family for TURBO OSCILLATOR as they use less power. If you have a standard 7473 chip from the old oscillator, feel free to use it in TURBO OSCILLATOR. The only difference will be a slight increase in current consumption. The 74LS00 can also be a standard TTL chip but it's best to use a 74LS04 for the oscillator.

While at the moment the speed selection is a manual operation done via the slide switch, the TURBO OSCILLATOR has been designed to be extended to allow automatic software selection or push button operation.

Just think of it, software like the tape system can select their required speed, load or save a file and then switch the TEC back to TURBO without you having to lift a finger.

My (yet unpublished) Mk3 2/4/8k Eprom programmer would love this as it is software timed and I am constantly having to stop the burning half way through after realizing that the TEC is on the wrong speed!

### Z80 REQUIREMENTS

Before TURBO OSCILLATOR is of any use you must have a Z80A. Most TEC kits were supplied with a Z80A, but some earlier kits had the standard Z80.

The current batch at TE are the standard Z80's so if you want a Z80A you will have to try the local electronics store for the time being.

### KITS

Two different kits are available for the TURBO OSCILLATOR. The first is a short form "make-up" kit that contains the additional parts required to make the TURBO OSCILLATOR up from the original oscillator kit. This kit includes the PCB, replacement sockets, resistors, caps and the extra chip. (No crystal).

The second kit is the full kit and contains all the parts including the crystal.

### CONSTRUCTION

Construction is straight forward and presents no problems. The only bit to watch is soldering the dip-header to the underside of the board. The crystal in TURBO OSCILLATOR stands upright but don't have it too high up as there is a second story to go on at a later stage.

Like the original oscillator the chips are around the wrong way when compared to the rest of the TEC. Keep this in mind when inserting them. There are a few links to go onto the board and apart from fitting these first, you can fit the other components in any order.

### TESTING

Once you have it built and fitted, switch on the TEC. Hopefully the TEC will spring to life but if not then recheck the board for bad joints etc. The TE logic probe can be used to check the static logic levels, but is no good for the clock signals as they are to fast for CMOS at 5v. Don't be fooled if the probe doesn't detect a clock, its just not able to pick it up.

When the TEC is running, hold down the "+" key and let the TEC step through the memory. While holding the "+" key flick the TURBO switch across to the opposite side and hopefully the TEC will continue on stepping through the memory at either half or double the speed as before. If this doesn't happen re-check your construction and eliminate the problem.

If you don't have a JMON you can test it by flicking the switch across and seeing that the TEC doesn't crash. After this reset the TEC and see if the reset tone changes pitch dramatically each time the switch is in a different position.

Give the switch a good work-out to make sure that your TURBO OSCILLATOR works correctly all the time.

Don't forget that TE has a repair service if you cant' get it going.

### THE OUTPUT TAPS

There are 3 output taps on the turbo oscillator. They are: FULL, the full crystal speed output, HALF, the divide by two output and SELECTED, the clock speed that is outputted from the TURBO OSCILLATOR board to the TEC.

The FULL output is the one you use for the SPEECH project while the other two are for possible future projects.

### THE EXTN SWITCH INPUT

The EXTN SWITCH INPUT is an optional input that allows the TURBO OSCILLATOR to be controlled by an external source. This external source could be a latched output controlled by the TEC software or just a push button.

The EXTN SWITCH input is selected when the manual speed switch on the board is at the EXTN/TURBO position. In this position, the speed is TURBO by default by virtue of R3 which pulls the data selector input high. The slow speed is selected by the external device taking this input low.

### FULL KIT PARTS LIST

1 - 330R
2 - 1k
1 - 3k3

3 - 100n monoblocks
1 - 100p ceramic

1 - 74LS04
1 - 74LS00
1 - 74LS73

3 - 14 pin IC sockets
1 - 16 dip header
1 - 5cm tinned wire
1 - mini SPDT switch
1 - 3.58MHz crystal
1 - TURBO PCB

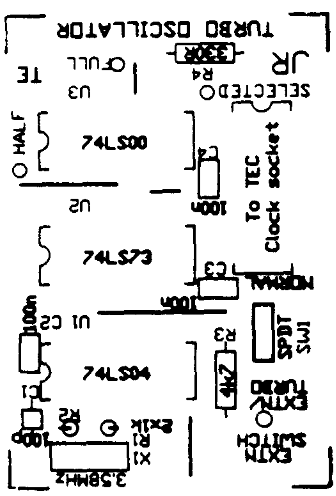
### MAKE-UP KIT

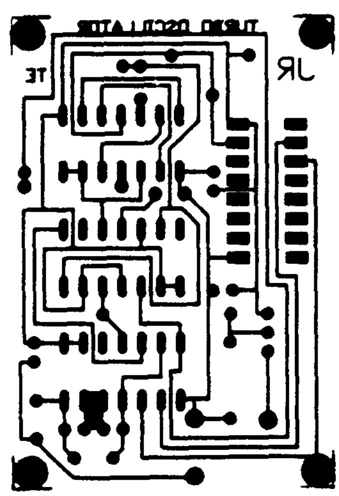
1 - 330R
2 - 1k
1 - 3k3

3 - 100n monoblocks
1 - 100p ceramic

1 - 74LS00

3 - 14 pin IC sockets
1 - 16 Dip header
1 - 5cm Tinned wire
1 - Mini SPDT switch
1 - TURBO PCB



### ACTUAL SIZE ARTWORK

**The writing on the overlay appears right-side-up when the board is fitted to the TEC (and the three chips are fitted). The chip numbers are upside down so as they read correctly when you go to fit the chips. This will help avoid putting the chips in the wrong way around.**

# FAST FORWARD

By Jim

Most of the programs I've written are programs to aid in programming. This is no exception. It's one of my first programs of any complexity.

It is written for the MON-1 series of MONitors and as a result will not run with JMON or the MON-2 series.

FAST FORWARD is a program designed to automatically step through the memory and display the address and data on the TEC LED display.

The purpose of this is to allow you to write down your program without having to hit the increment key all the time.

FAST FORWARD can step through the memory both forwards and backwards at slow speed and also at high speed. It does not, however, have a manual step feature. (strangely, I did not think it necessary at the time).

Each time FAST FORWARD changes to a new address, a beep is sounded to let you know. This way you can keep in time with the changes.

The following keys are used:

A is the fast forward key
B is the fast reverse key
C is the stop key
D is the slow reverse key
ANY other key for slow forward

Two MONitor routines are called by fast forward. These are the LED scan routine at 0140 and the beep routine at 018E.

FAST FORWARD has its own binary-to-display code conversion as I must not of known where this routine was located in the MON-1 ROM at the time.

The conversion routine for FAST FORWARD is at 0A00.

FAST FORWARD also has its own display code table located at 0B00.

The speed of the stepping is controlled by the values loaded into the BC register pair at addresses 0906, 0950 and 095A. Depending on your oscillator, you may need to alter this values.

The purpose of presenting FAST FORWARD is to allow you to play around with the program for whatever purposes you like.

As an interesting challenge, see if you can add a manual step function, it shouldn't be too hard.

FAST FORWARD is hardly a example of good programming, but it does produce the results!

```
0900  21 00 00     LD HL,0000
0903  01 0F 04     LD BC,040F
0906  11 00 0B     LD DE,0B00
0909  DD 21 00 0C  LD IX,0C00
090D  7E           LD A,(HL)
090E  CD 00 0A     CALL 0A00
0911  7D           LD A,L
0912  CD 00 0A     CALL 0A00
0915  7C           LD A,H
0916  CD 00 0A     CALL 0A00
0919  C5           PUSH BC
091A  E5           PUSH HL
091B  CD 8E 01     CALL 018E
091E  E1           POP HL
091F  23           INC HL
0920  DD 21 00 0C  LD IX,0C00
0924  D1           POP DE
0925  CD 40 01     CALL 0140
0928  ED 57        LD A,I
092A  FE 0A        CP 0A
092C  CA 50 09     JP Z 0950
092F  FE 0B        CP 0B
0931  CA 5A09      JP Z 095A
0934  FE 0C        CP 0C
0936  CA 65 09     JP Z 0965
0939  FE 0D        CP 0D
093B  CA 6A09      JP Z 096A
093E  1B           DEC DE
093F  7B           LD A,E
0940  B2           OR D
0941  20 E2        JR NZ 0925
0943  C3 03 09     JP 0903

0950  01 00 01     LD BC,0100
0953  C3 6F09      JP 096F

095A  01 00 01     LD BC,0100
095D  2B           DEC HL
095E  2B           DEC HL
095F  C3 6F 09     JP 096F

0965  C3 25 09     JP 0925
0968  FF           RST 38
0969  FF           RST 38
096A  01 00 04     LD BC,0400
096D  2B           DEC HL
096E  2B           DEC HL
096F  C5           PUSH BC
0970  D1           POP DE
0971  CD 40 01     CALL 0140
0974  1B           DEC DE
0975  7A           LD A,D
0976  B3           OR E
0977  20 F8        JR NZ 0971
0979  C3 06 09     JP 0906

0A00  E5           PUSH HL
0A01  F5           PUSH AF
0A02  E6 0F        AND 0F
0A04  26 00        LD H,00
0A06  6F           LD L,A
0A07  19           ADD HL,DE
0A08  7E           LD A,(HL)
0A09  DD 77 00     LD (IX+00),A
0A0C  DD 23        INC IX
0A0E  F1           POP AF
0A0F  E6 F0        AND F0
0A11  1F           RRA
0A12  1F           RRA
0A13  1F           RRA
0A14  1F           RRA
0A15  26 00        LD H,00
0A17  6F           LD L,A
0A18  19           ADD HL,DE
0A19  7E           LD A,(HL)
0A1A  DD 77 00     LD (IX+00),A
0A1D  DD 23        INC IX
0A1F  E1           POP HL
0A20  C9           RET
```

### DISPLAY TABLE

```
0B00: EB 28 CD AD 2E A7 E7 29
0B08: EF 2F 6F E6 C3 EC C7 47
```

### MON-1 SCAN ROUTINE

```
0140  DD E5        PUSH IX
0142  01 01 06     LD BC,0601
0145  DD 7E 00     LD A,(IX+00)
0148  D3 02        OUT 02,A
014A  DD 23        INC IX
014C  79           LD A,C
014D  D3 01        OUT 01,A
014F  CB 27        SLA A
0151  4F           LD C,A
0152  3E 0A        LD A,0A
0154  3D           DEC A
0155  C2 54 01     JP NZ 0154
0158  D3 01        OUT 01,A
015A  10 E9        DJNZ 0145
015C  DD E1        POP IX
015E  C9           RET
```

### MON-1 BEEP ROUTINE

```
018E  0E 0A        LD C,0A
0190  21 50 00     LD HL,0050
0193  29           ADD HL,HL
0194  01 01 00     LD DE,0001
0197  AF           XOR A
0198  D3 02        OUT 02,A
019A  3D           DEC A
019B  D3 01        OUT 01,A
019D  41           LD B,C
019E  10 FE        DJNZ 019E
01A0  EE 80        XOR 80
01A2  ED 52        SBC HL,DE
01A4  20 F5        JR NZ 019B
01A6  C9           RET
```

# THE 8255 PPI

The 8255 is one of the handiest peripheral chips to attach to the Z80. It is a Parallel Interface Chip that provides 3 eight bit ports. One of the ports can be split into 2 four bit ports so actually there are four ports on the 8255.

Inside the 8255 are three registers that do all the communicating between the Z80 and the external ports. There is a fourth register called the control register and it is used to set the various operating modes of the 8255. The controlling computer sends a byte to this register to initialize the operating conditions of the ports. The ports are defined as either inputs or outputs by the bits within the control register.

Once the three ports are defined, communication to the outside world is done through the three port registers.

## THE PORTS

Each of the ports has a letter assigned to it for identification. The lowest decoded port is assigned as port A while the next port is port B and then port C. The control register is decoded on the fourth input/output address.

Port C can be split into 2 groups of four with either group defined as an input or output. In this case port C is referred to as port C upper and port C lower.

## INTERFACING

Interfacing the 8355 to the Z80 is a very simple matter as most of the fifteen connections go directly to the Z80 with no extra logic required. In fact the only parts required in addition to the Z80 and 8255 is the optional decoding and either an inverter between the two resets or a RC network on the 8255's reset pin.

The decoding is only required if there are multiple input/output devices on the Z80 as in the TEC environment.

If decoding is required it is a little tricky as the ports must be spaced four apart. This is to allow the 8255 its internal decoding that picks one of four ports depending on the state of the two address lines fed to it.

The normal I/O decoding on the TEC is unsuitable as the ports are decoded consecutively.

A simple way around this is to memory map the 8255. This takes care of any decoding problems but is wasteful as an entire 2k is used for only four address locations.

Apart from the advantage of simpler decoding, the memory map arrangement provides another advantage. This advantage is that two of the ports may be simultaneously accessed by a sixteen bit memory reference instruction.

## OPERATING MODES

There are three operating modes to the 8255 giving it flexibility to perform a variety of roles. These modes are described below. In these notes only mode 0 will be dealt with in detail as modes 1 and 2 are not likely to be of use on the TEC.

### MODE 0

This is the most basic mode and the most widely used. In mode 0 the 8255 is configured to have three input/output ports. No handshaking is provided or needed.

In this mode any of the ports may be either an input or an output and port C can be split into halves. There are Sixteen combinations of input/output configurations in this mode.

### MODE 1

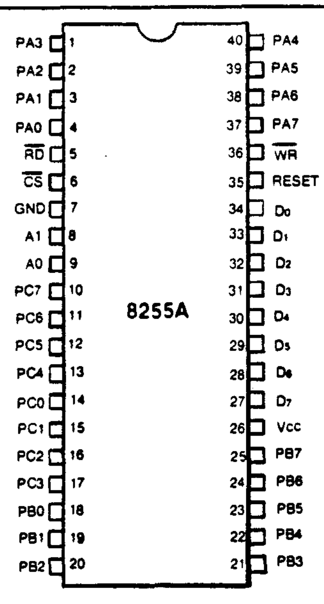
Mode 1 provides a strobed input/output function. In this mode only ports A and B are available for input and output operations. Port C is divided into two and used to provide handshaking for ports A and B. The inputs and outputs are latched.

### MODE 2

Mode 2 is the most complex of the three. Mode 2 provides one Bi-directional 8 bit port on port A. Five bits of port C are used for handshaking while port B is not available for use.

## PINOUTS

Below are the 8255 pinouts.




## PIN DESCRIPTION

**D7-D0** This is the Bi-directional data bus and is connected directly to the Z80 data bus.

**RESET** Active high, sets all ports to inputs. Requires an inverter from the Z80 reset or an RC network to reset the 8255 on power up.

**$\overline{CS}$** Chip Select is an active low input that enables all communication with the chip. It is connected to the memory decoder IC (74LS138) on the TEC.

**$\overline{RD}$** Active low read enable that allows the 8255 to send data to the TEC. This connects directly to the pin of the same name on the Z80.

**$\overline{WR}$** Active low write. This enables the Z80 to send data to the 8255. Connect this to the Z80 $\overline{WR}$ or pin 21 of the expansion socket.

**A0, A1** These select one of the four ports. They are usually connected to the Z80 A0 and A1 and are in the following project.

**VCC and GND** these are the power connections. The chip requires a 5V 10% power rail.

**PA0-7** These are the port A input/output pins.

**PB0-7** Port B input/output pins.

**PC0-3** Low half of port C. These can be configured as inputs or outputs independent of the upper half.

**PC4-7** the second group of pins that make up port C. Like the first group these may be either inputs or outputs.

## PROGRAMMING THE 8255

Only mode 0 is discussed in these notes as modes 1 and 2 are quite complex and would take pages to explain. Given that they are not used except for more advanced interface arrangements, there is nothing to gain by doing so.

The three modes of operation and the direction of all the ports is selected by a byte that is written into the control register.

Of the 8 bits in the control register only 5 concern us when using mode 0. Four of these bits select between the 16 possible input/output configurations and the fifth bit is the mode select bit.

Bit 7 is the mode select bit and is ALWAYS HIGH.

Bit 0 sets PC0-3 (port C lower) as inputs if set high.

Bit 1 sets all of port B as inputs if set high.

Bit 3 sets PC4-7 (port C upper) as inputs if set high.

Bit 4 sets all of port A as inputs if set high.

The following table contains the summary of the above:

| Lines | Control Word 80H | 82H | 81H | 83H | 88H | 8AH | 89H | 8BH |
|---|---|---|---|---|---|---|---|---|
| PA7–PA0 | out | out | out | out | out | out | out | out |
| PC7–PC4 | out | out | out | out | in | in | in | in |
| PC3–PC0 | out | out | in | in | out | out | in | in |
| PB7–PB0 | out | in | out | in | out | in | out | in |
| **Lines** | **90H** | **92H** | **91H** | **93H** | **98H** | **9AH** | **99H** | **9BH** |
| PA7–PA0 | in | in | in | in | in | in | in | in |
| PC7–PC4 | out | out | out | out | in | in | in | in |
| PC3–PC0 | out | out | in | in | out | out | in | in |
| PB7–PB0 | out | in | out | in | out | in | out | in |

## CONNECTING THE 8255 TO THE TEC

The connection is as straight forward as you get.

There are 15 lines to go to the TEC and 14 of these go directly to the pin of the same name on the Z80!

The odd one out is the $\overline{CS}$ and this goes to $\overline{CS}$ 1000 or pin 13 of the right-most 74LS138 chip. If you are wiring it up through the TEC expansion port then it goes to pin 18 of the expansion port.

We need to do something with the reset line and a simple RC network that will spike it high on power-up is all that is required. In fact if you really want to get lazy, you can just wire it straight to ground!

Of the 15 lines that go to the TEC, 14 are available through the TEC expansion port. The one that misses out is the $\overline{RD}$. This must be connected to pin 21 of the Z80.

You CANNOT simply invert the $\overline{WR}$ to get a $\overline{RD}$ signal, it just doesn't work like that. $\overline{RD}$ must go to $\overline{RD}$.

## ONE HASSLE

One problem with the 8255 is that if you change the in/out configuration, any output port that remains as an output port will have its high bits reset low.

For instance, say that all the ports are outputs, if you change port A to an input all the output lines on ports B and C go low.

Keep this fault in mind when using the 8255, It may explain any mysterious happenings.

## THE NEW DESIGNER BOARD AND EPROM PROGRAMMER

TE have a new designer board that is just perfect for bird-nesting this project. It is called the designer board MK2 (how unimaginative) but if you ask for a Jim's designer board I'm sure they'll know what you mean.

The board is designed for larger chips than the previous boards and it doesn't have any holes to contend with. The pads are uniform in width and there is a lot of copper there so the board will stand-up to a fair amount of soldering and re-soldering, It's reusable! The cost is $5.50 per board.

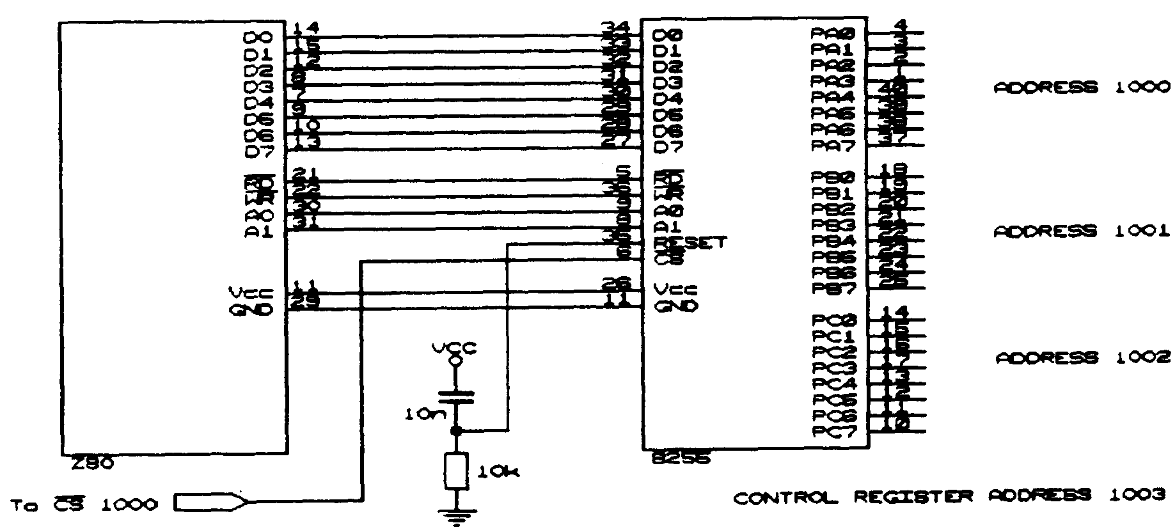
If you do use this board, and I recommend that you do, place the 8255 toward the left-hand end. This will leave space for a 24 pin socket and half-a-dozen transistors and resistor so the board can be converted into a simple but versatile 2/4k EPROM programmer /copier/reader and expansion port simulator.

## CONCLUSION

If you want additional input/output on the TEC the 8255 is far better than the old TEC input/output board (issue 14) and the 8255 is a better choice than the Z80 PIO (parallel Input Output) for most applications.

The advantages the 8255 has over the PIO are: one additional port, much simpler hardware interfacing and software control.

The 8255 is disadvantaged when compared to the Z80 PIO in that the Z80 PIO has a sophisticated interrupt system and bit input/output. Generally though, the features provided by the Z80 PIO are not required and the advantages of the 8255 far out-weigh the disadvantages.




### TEST SOFTWARE

```
0900  LD A,80      SET MODE 0
0902  LD (1003),A  ALL OUTPUTS
0905  LD A,55      BIT PATTERN
0907  LD (1000),A  OUT TO A
090A  LD (1001),A  OUT TO B
090D  LD (1002),A  OUT TO C
0910  RST 00
```

Test the outputs with a logic probe. They should alternate high and low.

### EXPERIMENT

Use the table above to set port C upper as an output and C lower as an input. Load the accum with different values. Find out where the nibble must be in A to appear at port C upper.

# DISPLAYING HL ON THE LED DISPLAY

This chapter describes how to display a 16 bit number in HL on the TEC LED display. There are two major blocks in this program, one to convert HL to display code and the other to scan the display. We will be using fastscan (it has been documented elsewhere in this book). The conversion routine is quite simple. This is the way it works: To display a digit on the TEC'S seven segment display, a full byte is used. Because each byte in HL contains two digits, it will be converted to two display bytes for the display. To put this another way, each nibble in HL must be converted to one display byte.

The conversion routine must be able to split each byte into two nibbles and convert each separately. This operation is done for each byte in HL.

Because there is no logical relationship between the value of a nibble and its display code, a look-up table must be used. This look up table will have entries one byte long, and contain 16, one for each HEX digit.

Once each nibble is converted, it must be stored somewhere or outputted to the display. For reasons of efficiency, the 2nd idea is not a good one as the display brightness will suffer if half the "on time" for each display has been used up in an unnecessary conversion!! This is compounded if, as in all the TEC MONitors, a further operation, such as dot setting, is required. (Not withstanding this, MON-2 does use this poor arrangement. depending on your TEC speed, the difference in brightness can be quite noticeable).

The better idea is to store the display information in a buffer and scan it all in one hit. This is the approach used in this routine. A 16 bit register pair will be used to point to the current display buffer.

Ok, enough of my rambling on, here is the actual conversion routine:

```
0A00  01 00 08   LD BC,0800    ;BC is the display buffer
0A03  7C         LD A,H        ;Put first byte to convert in A
0A04  E5         PUSH HL       ;Save HL for second byte conversion
0A05  CD 0A 09   CALL 090A     ;Call sub-routine to convert A
0A08  E1         POP HL        ;Recover HL
0A09  7D         LD A,L        ;Put second byte in A
```

Byte conversion

```
0A0A  F5         PUSH AF       ;Conversion starts here: Save A
0A0B  0F         RRCA          ;Shift the high order nibble
0A0C  0F         RRCA          ;Down to the low order position
0A0D  0F         RRCA          ;By rotating right left
0A0E  0F         RRCA          ;Four times
0A0F  CD 13 09   CALL 0913     ;call nibble conversion
0A12  F1         POP AF        ;Recover A for second nibble
```

Nibble conversion

```
0A13  E6 0F      AND 0F        ;Mask off unwanted bits
0A15  21 00 0B   LD HL,0B00    ;Load HL with table base
0A18  85         ADD A,L       ;Add nibble to table base
0A19  6F         LD L,A        ;And put entry address in HL
0A1A  7E         LD A,(HL)     ;Put display byte into A
0A1B  02         LD (BC),A     ;Store in display buffer
0A1C  03         INC BC        ;Next display buffer location
0A1D  C9         RET           ;Return to calling routine
```

- Display code Look-up table -

```
0B00: EB 28 CD AD 2E A7 E7 29 EF AF 6F E6 C3 EA C7 47
```

Now, because the above is a subroutine, we need to have a master routine to call it.
The master routine must also load HL with some (any) value, clear the unused display buffer locations (so the right most displays remain off), and finally provide a never-ending loop that calls fastscan.

The routine below is the master:

```
0900   2A 00 00    LD HL,(0000)  ;The address we want into HL
0903   CD 00 0A    CALL 0A00     ;Call HL conversion
0906   AF          XOR A         ;Clear A to
0907   02          LD (BC),A     ;Clear unused
0908   03          INC BC        ;Display buffer
0909   02          LD (BC),A     ;Locations
090A   CD 80 0A    CALL 0A80     ;Call fastscan
```

In the case of a MONitor program, we would look for a key press here.

```
090D   18 FB       JR 090A       ;Loop forever
```

If the data at the address HL is pointing to is continuously changing (e.g. an interrupt routine alters it or HL is pointing to a memory mapped peripheral like a 8255 PPI chip, the previous jump should jump back to address 0900 so that the displays are constantly up- dated.

Now, put fastscan at 0A80

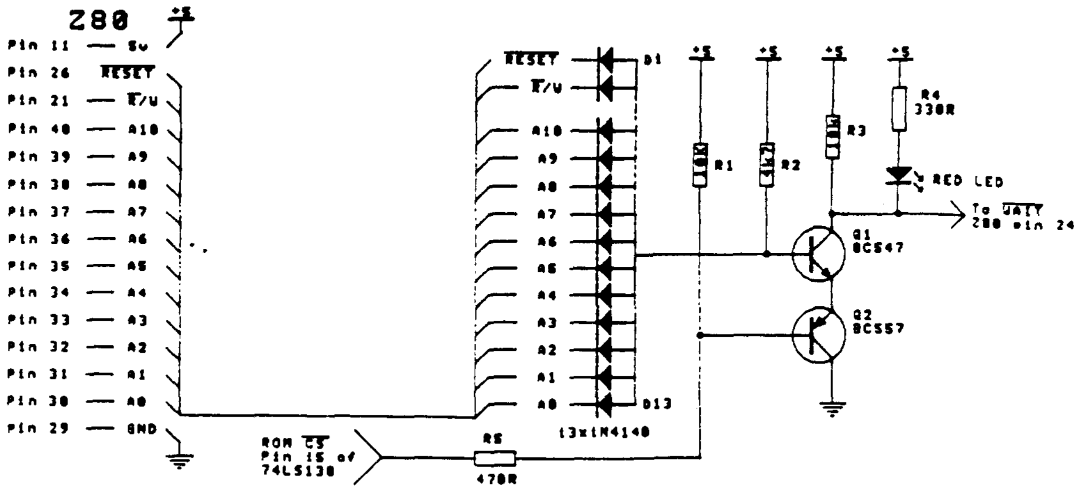
```
0A80   21 00 08    LD HL,0800
0A83   06 20       LD B,04
0A85   7E          LD A,(HL)
0A86   D3 02       OUT (02),A
0A88   78          LD A,B
0A89   D3 01       OUT (01),A
0A8B   06 50       LD B,50
0A8D   10 FE       DJNZ 0A8D
0A8F   47           LD B,A
0A90   AF          XOR A
0A91   D3 01       OUT (01),A
0A93   23          INC HL
0A94   CB 00       RLC B
0A96   30 ED       JRNC 0A85
0A98   C9          RET
```

While fastscan is configured to scan left-to-right, only the four right-most displays will be on as the right-most two have their display buffers cleared by the master routine.
Because fastscan scans Left to Right, the display buffer pointer is set at the low address end of the buffer and moves upward through the table. (The display codes are lined up with the highest order display byte at the lower address). To put this another way, the highest display byte, which is stored at the lowest address, is outputted first because its corresponding common cathode is turned on first.
What next?
From here the next step is logical. Now we have the contents of HL converted and displayed, we should now also convert and display the contents of the memory location addressed by HL. This will complete a working model of how MONitor programs display the address and data information. This I shall leave as an interesting challenge for you. If you understand the above thoroughly, it will not be difficult as only the master routine at 0900 needs to be altered!
As a future project, I hope to produce an article on a simple MONitor and explain each stage step-by-step. Such a project will give a good understanding of Z80 programming techniques and also help you design your own programs.

# *CRASH* PROTECTOR

-Note the play on words-

PARTS & PCB: $6.70




## CRASH PROTECTOR CIRCUIT DIAGRAM

How many times have you spent ages typing in a program only to have it wiped out in milliseconds by the TEC? I have suffered this fate many times. Finally I did something about it.

Almost all crashes are caused when a program goes into a loop that continuously pushes onto the stack. The result is the stack quickly wraps-around and everything in RAM is written over.

If you could prevent the stack passing a certain point, the crash would be avoided.

Unfortunately the Z80 doesn't include a hardware watch-dog for the stack and so we are left to our own ingenuity to devise a suitable hardware circuit.

Fortunately the TEC hardware and MONitor software environment provide us with a simple way where we can detect a dangerous stack operation.

We can detect when the stack has over-stepped the mark when it has ran out of RAM and tries to write into ROM. Both JMON and MON-2 put the stack in the lowest RAM page (0800) and so only the non-essential MONitor variables will be over-written before the stack crash is detected. In fact nothing of any value will be lost as the MONitor variables are re-booted on reset.

### MON-1

MON-1 is different as it loads the stack at 0FF0. This means that everything in RAM between 0800 and 0FF0 will be lost if you don't shift the stack pointer to the first RAM page.

Shifting the stack pointer is your only option if you wish to protect the lower 2k of RAM. Your program should then start at 0900 with the stack somewhere in the 0800 page.

### BROAD CIRCUIT OVERVIEW

The idea of the circuit is to detect when a write operation occurs at the address 07FF. This is the highest byte before the RAM and in normal operation a write to this address never happens. When the stack runs out of RAM it will try to PUSH into this ROM address. This illegal operation is a good warning that a stack crash is imminent and if program execution is halted at this point we have avoided a disastrous crash.

Apart from detecting the imminent crash, the circuit must have a means to stop the program. The method chosen is to put the Z80 into WAIT. WAIT has been chosen over RESET and INT as these two may be activated by other means. The WAIT is exclusive to the crash protector and so you know why program execution was halted. A pad has been provided for connecting the protector to INT but at this stage there is no software to take advantage of it. DO NOT connect the protector to INT. It should go to WAIT until further advised.

### CIRCUIT THEORY

The circuit is simply a 14 input AND gate with an inverter on the output.

As we are at the computer level, I have taken for granted that you understand the theory of the diode AND gate. If you are unsure of its workings then refer to page 43 of NOTEBOOK 2 published by Talking Electronics, for a description of its theory.

To keep the discussion simple I have separated the diode AND gate from the transistor inverter on its output and regarded each as a separate entity. I could have collectively referred to them as a NAND gate but the operation of an AND gate is much easier to visualize.

Before you can understand the circuit operation you must know the conditions the circuit will cause the WAIT state. These conditions are listed below:

Firstly there must be a memory access to the MONitor ROM. This means the ROM select must be low.

The second condition is that there must be a memory access to address 7FF. This is the last address of the ROM and is detected by the row of diodes from pin 30 to pin 40 on the Z80.

The third condition is that it must be a write cycle.

Finally, the RESET must be inactive. This feature is to prevent the TEC becoming jammed in WAIT.

### HOW THE CONDITIONS ARE GATED

The MONitor ROM SELECT is gated by the BC557 transistor. If the ROM is not being accessed then the ROM select will be high and the BC557 will be turned off via the 10k pull-up resistor on its base. This then prevents the BC547's emitter lead from going to ground and the BC547's collector cannot pull the WAIT low. The 10k pull-up on the base lead ensures that when the ROM select is high, the base is pulled up to full rail voltage. Keep in mind that if the base is just 0.6v lower than the emitter the BC557 transistor will (just) turn on.

The memory access to address 7FF condition is gated by the eleven diodes connected to the lowest 11 address lines. These address lines will ALL BE HIGH when an address with 7FF as the lower three nibbles is accessed. If there is a low on any of these address lines then the low is transferred through the diode and the output of the AND gate is low. This low holds the BC547 off and the active low WAIT is held high by the 10k pull-up resistor. Only when all these address lines are high will the AND gate go high to provide a turn-on voltage to the BC547.

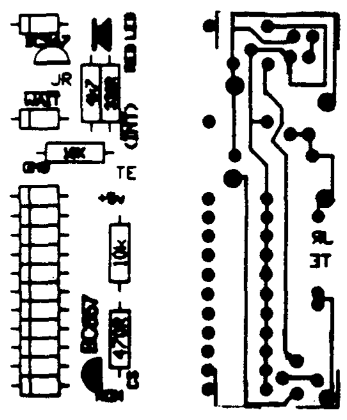
The write condition is gated exactly the same way as the address condition via the diode AND gate. The active low read signal is fed to the diode AND gate and if a READ operation is taking place then this signal is low. The low passes through the diode and disables the output section in the same manner as described above.

Finally the RESET signal is gated in the same fashion as above so when the TEC is reset the RESET input to the AND gate is taken low and thus disabling the transistor switching.

The BC557 also has a hidden role. If it was not there the BC547 transistor will be just switched on by the 0.6v drop across the diodes of the AND gate. The BC557 provides about a 0.2v drop between the emitter of the BC547 and ground and therefore the base voltage must rise to 0.8v before the transistor begins to turn on.

## THE BOARD

The board is designed to be mounted on top of the Z80. To do this a row of diodes hang over the edge and are soldered to the Z80. Each diode is in line with the pin it is soldered to.




**ACTUAL SIZE ARTWORK**

On the other side of the board there are pads for jumpers that are soldered onto the Z80. There is only one external connection and this is to the memory decoder IC. This connection goes to pin 15 of the right-hand 74LS138.

## PUTTING IT TOGETHER

Solder all the parts onto the board and take care with the orientation of the diodes. The bands are marked on the overlay so you shouldn't make any mistakes. Solder short lengths of tinned copper wire to the pads marked WAIT, +5v and GND. These are bent down and soldered to the Z80 pins they line up with. Solder a piece to the pad marked INT but don't solder it to the Z80. This is for future use but must be soldered to the board before it is soldered in place.

Also solder and 5cm length of hook-up wire to the pad labeled ROM CS. Actually, you should solder the female matrix connector to the other end before soldering it to the board.

Because the board sits across the Z80, cut the leads to a consistent height so that the board sits flat.

When all the parts and jumpers are mounted place the board over the Z80. The two end diodes will line-up with pins 21 and 40 of the Z80. Bend the diode leads to 90 degrees or until they make contact with the Z80 pins when the board is central on the Z80. Carefully tack the end diodes in place. Now you can position the board correctly and solder the jumpers onto the Z80 pins they line up with. Finish soldering all the diodes and jumpers when you are happy with the board's placement.

All that is left is to solder the male matrix pin to pin 15 of the 74LS138 adjacent to pin 1 of the Z80.

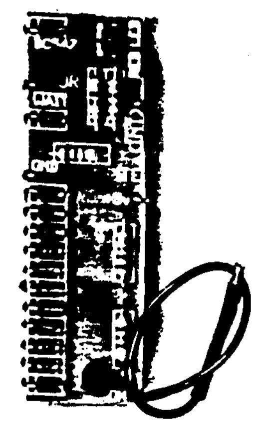
Fit the female matrix connector over the male pin and we're ready to test it.

## TESTING

The first test is to see weather the TEC still runs with the board fitted.

If not, it is most likely there's a short between pins. Use a continuity tester to find any shorts. Also check the orientation of the diodes as a diode around the wrong way will stop the TEC dead. Another possibility is that a diode has been "cooked" when soldered and has shorted. Checked this with a multimeter as it will also find open diodes.

Once you have the TEC running, address 07FF. If the TEC goes into WAIT as soon as you address 07FF then the active low READ is not getting through to the AND gate. Investigate the diode and connection on pin 21.



**Finished board ready to go**

When you have the TEC looking at 07FF, press a data key. This will cause the MONitor to try to write into this location. the TEC should go dead and the WAIT LED should come on.

If this doesn't happen then recheck your work for construction faults. and faulty components.

When you have it working as described above, enter this short routine (any MONitor):

```
0900  LD SP,0820   31 20 08
0903  PUSH HL      E5
0904  JR -3        18 FD
```

TEST DATA:
0906 01 02 03 04 05 06 07 08 09

Without the crash protector, this routine will wipe out the whole memory in a blink of an eyelid. If you don't believe me then desolder the jumper from the WAIT pin (pin 24) and run the program. Told you so!

With the crash protector enabled you could run this program over and over and never lose one byte from any address above 081F.

## PARTS LIST

1 - 330R
1 - 470R
1 - 4k7
2 - 10k

13 - 1N 4148
1 - BC 547
1 - BC 557

1 - 3mm Red LED
1 - Male matrix pin
1 - Female matrix pin
1 - 5cm hook-up wire
1 - Crash Protector PCB

# 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.

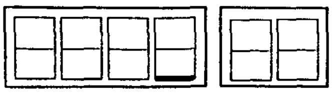
If this problem occurs, the only solution is to perform a "TEST CHECKSUM" on the tape before doing the BLOCK TEST. If the TEST CHECKSUM passes OK but the TEST BLOCK fails, then you know that the tape is good but its contents are not the same as the memory block.

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.

## SEGMENT TARGET GAME

By Mr. S Clarke, 2774

Segment Target is a simple game in which you must hit the moving segment in the bottom right of the address section. i.e.



**Shoot when the highlighted segment is illuminated.**

As each target is hit, the next one moves even FASTER! Any key can be used to shoot. Your score is stored at 08FF (in HEX)

************************************

SEGMENT TARGET, as presented below, has been written to run with the MON-1 series MONitors. By changing the LD A,I (ED 57) to RST 20/NOP (E7, 00) as described in the section on running old programs with JMON in issue 15, it will run equally as well with JMON.

Don't be content to just play SEGMENT TARGET GAME, see if you can improve on it!

-JIM

```
0900  11 00 38     LD DE,3800
0903  ED 53 A6 09  LD (09A6),DE
0907  3E 00        LD A,00
0909  32 FF 08     LD (08FF),A
090C  21 80 09     LD HL,0980
090F  7E           LD A,(HL)
0910  47           LD B,A
0911  23           INC HL
0912  7E           LD A,(HL)
0913  4F           LD C,A
0914  23           INC HL
0915  78           LD A,B
0916  FE FF        CP FF
0918  CA 6B 09     JP Z,096B
091B  D3 01        OUT (01),A
091D  79           LD A,C
091E  D3 02        OUT (02),A
0920  CD 2E 09     CALL 092E
0923  CD 3A 09     CALL 093A
0926  FE 12        CP 12
0928  CA 0C 09     JP Z,090C
092B  C3 0F 09     JP 090F
092E  ED 5B A6 09  LD DE,(09A6)
0932  1B           DEC DE
0933  7A           LD A,D
0934  FE 00        CP 00
0936  C8           RET Z
0937  C3 32 09     JP 0932
093A  ED 57/E7,00  LD A,I
093C  5F           LD E,A
093D  3E FF        LD A,FF
093F  ED 47        LD I,A
0941  7B           LD A,E
0942  FE FF        CP FF
0944  C8           RET Z
0945  78           LD A,B
0946  FE 04        CP 04
0948  C0           RET NZ
0949  79           LD A,C
094A  FE 80        CP 80
094C  C0           RET NZ
094D  3E 03        LD A,03
094F  D3 01        OUT (01),A
0951  3E FF        LD A,FF
0953  D3 02        OUT (02),A
0955  CD 2E 09     CALL 092E
0958  3A FF 08     LD A,(08FF)
095B  3C           INC A
095C  32 FF 08     LD (08FF),A
095F  ED 5B A6 09  LD DE,(09A6)
0963  15           DEC D
0964  ED 53 A6 09  LD (09A6),DE
0968  3E 12        LD A,12
096A  C9           RET
096B  11 00 BF     LD DE,BF00
096E  ED 53 A6 09  LD (09A6),DE
0972  3E FF        LD A,FF
0974  D3 01        OUT (01),A
0976  3E 85        LD A,85
0978  D3 02        OUT (02),A
097A  CD 2E 09     CALL 092E
097D  C7           RST 00

0980  20 01 10 01 08 01 04 01
0988  04 08 04 04 08 04 10 04
0990  20 04 20 40 20 80 10 80
0998  08 80 04 80 02 80 01 80
09A0  FF
```

## WHIRL

by Jeff Kennett 3218

This clever routine for the 8x8 display continuously rotates the display around 90 degrees and produces quite an interesting effect. After a while the eyes are fooled and it begins to look like anything other than a rotating arrow head. One staff member thought it looked like a plus sign trying to rap dance!!

Experiment with the values in the table at 0A00 and the delay at 0927/8 to see what dazzling effects you can produce!

```
0900  CD 27 09     CALL 0927
0903  11 08 0A     LD DE,0A08
0906  06 08        LD B,08
0A08  C5           PUSH BC
0909  06 08        LD B,08
090B  21 00 0A     LD HL,0A00
090E  AF           XOR A
090F  CB 06        RLC (HL)
0911  1F           RRA
0912  23           INC HL
0913  10 FA        DJNZ 090F
0915  12           LD (DE),A
0916  13           INC DE
0917  C1           POP BC
0918  10 EE        DJNZ 0908
091A  01 08 00     LD BC,0008
091D  11 00 0A     LD DE,0A00
0920  21 08 0A     LD HL,0A08
0923  ED B0        LDIR
0925  18 D9        JR 0900
0927  06 50        LD B.06
0929  C5           PUSH BC
092A  06 80        LD B,80
092C  21 00 0A     LD HL,0A00
092F  7E           LD A,(HL)
0930  D3 05        OUT (05),A
0932  78           LD A,B
0933  D3 06        OUT (06),A
0935  06 40        LD B,40
0937  10 FE        DJNZ 0937
0939  47           LD B,A
093A  AF           XOR A
093B  D3 06        OUT (06),A
093D  23           INC HL
093E  CB 08        RRC B
0940  30 ED        JRNC 092F
0942  C1           POP BC
0943  10 E4        DJNZ 0929
0945  C9           RET

0A00: 18 30 60 FF FF 60 30 18
```

## HEX TO BCD CONVERSION

By James Doran 3259

This SUB-ROUTINE will convert a hex number in A into its decimal equivalent and store the result in BC.

The hex number is held in A on entry.

The routine works by counting up in decimal while counting down the HEX number until zero.

This means that low numbers are converted quickly while larger numbers take longer.

The decimal counter is achieved by the use of the DECIMAL ADJUST ACCUMULATOR (DAA) instruction.

```
0900  06 00        LD B,00
0902  4F           LD C,A
0903  3E 00        LD A,00
0905  3C           INC A
0906  27           DAA
0907  30 02        JR NC,+2
0909  04           INC B
090A  3F           CCF
090B  0D           DEC C
090A  20 F7        JR NZ,-9
090C  4F           LD C,A
090D  C9           RET
```

Exit: BC = packed BCD equivalent of two hex digits in A.

The above routine is useful as a HEX to BCD conversion SUB-ROUTINE, but keep in mind the disadvantage of the length of time being very dependent on the magnitude of the HEX number to be converted.

# A GUIDE TO SIXTEEN BIT COMPARES

This discussion shows how to compare a 16 bit value in HL to one in DE (or BC). The basic idea can be applied to the IX and IY registers also.

The Z80 doesn't have any special 16 bit compare instructions so we are left to our own devices to find a suitable method to perform 16 bit compares.

The most common method in use is to subtract one value from another using the HL register pair. In fact, a compare is just a subtraction with the result discarded.

The first trap involved when using a subtract instruction is we lose the value in HL (it is replaced by the result).

This is not always a problem, but when it is we can get around it by PUSHing HL and POPing it later.

The second problem is the lack of subtract-without-carry instructions provided by the Z80.

This means we will have to use the subtract with carry and make sure the carry is clear before the instruction.

The Z80 doesn't have any instructions purely designed to clear the carry, but there are a couple of instructions that will clear the carry and actually do nothing else (Pity ZILOG didn't think to rename one of these instructions). These two instructions are:

AND A (A7) and OR A (B7)

The above two cause the ACCUMULATOR to be ANDed and ORed with itself which of course doesn't effect the value inside the ACCUMULATOR but the carry is cleared.

If the subtract instruction immediately follows either of these two instructions, you can be sure that the carry will be clear.

So a basic routine to compare HL to DE and preserve HL would look like this:

```
PUSH HL
OR A
SBC HL,DE
POP HL
```

Now we must examine the role of the flags. The two flags we are interested in are the ZERO FLAG and the CARRY FLAG.

The ZERO flag is set (to denote a zero condition) if the two numbers are the same. Conversely, if the numbers are not the same the zero flag will be clear.

The carry flag is useful to denote weather HL is less than DE or not less than (you cannot tell the difference between it being greater than or equal by just the carry flag alone).

By examining BOTH the ZERO AND CARRY FLAGS we can find out if HL is EQUAL TO, LESS THAN or GREATER THAN DE.

Great confusion exists about the condition denoted by the carry flag. Let me give you this simple explanation:

A carry will occur (thus the carry flag is set) if the number being subtracted is GREATER THAN THE NUMBER IT IS BEING SUBTRACTED FROM. Now, as DE is being subtracted from HL, IF HL IS LESS THAN DE THE CARRY WILL BE SET.

IF HL IS NOT LESS THAN DE THEN THE CARRY WILL BE CLEAR.

## SUMMARY

IF HL IS EQUAL TO DE THE ZERO FLAG IS SET (to denote a ZERO condition)

IF HL IS GREATER THAN DE THE CARRY FLAG IS CLEAR AND THE ZERO FLAG IS CLEAR.

IF HL IS LESS THAN DE THE CARRY FLAG IS SET AND THE ZERO FLAG IS CLEAR.

## SOME ROUTINES USING THE ABOVE EXAMPLES.

The jumps that occur when the condition been tested is met are in bold typeface. Any jump not in bold is purely to skip over the "condition true jump." Don't confuse the two.

***1, Jump if HL AND DE ARE EQUAL***

PUSH HL
OR A
SBC HL,DE
POP HL
**JR Z,** JUMPS HERE IF HL AND DE ARE EQUAL

***2, Jump if HL AND DE ARE NOT EQUAL***

PUSH HL
OR A
SBC HL,DE
POP HL
**JR NZ,** JUMPS HERE IF HL , DE NOT EQUAL

***3, Jump if HL IS LESS THAN DE***

PUSH HL
OR A
SBC HL,DE
POP HL
**JR C,** JUMPS HERE IF HL LESS THAN DE

***4, Jump if HL IS GREATER OR EQUAL TO DE***

PUSH HL
OR A
SBC HL,DE
POP HL
**JR NC,** JUMPS HERE IF HL = DE

***5, Jump if HL IS GREATER THAN DE***

PUSH HL
OR A
SBC HL,DE
POP HL
JR Z,02
**JR NC,** JUMPS HERE IF HL GREATER THAN DE

***6, Jump if HL IS EQUAL OR LESS THAN DE***

PUSH HL
OR A
SBC HL,DE
POP HL
**JR Z,** JUMPS HERE IF HL = DE
**JR C,** JUMPS HERE IF HL LESS THAN DE

Refer to these examples when you require a 16 bit compare or use these notes as a reference when following someone else's program.

Always think in terms of: is HL greater than or equal to DE? rather than: is DE less than or equal to HL?, etc. This, along with these notes will remove any confusion that may arise.

The same flag settings are used for the eight bit CP instruction. If you substitute A for HL and B, C, D, E, H, L or (HL) for DE, the above examples will then apply except that you don't need to save A when using the CP instruction as the result is discarded and does not overwrite the value in A.

The IX and IY resisters can be used instead of the HL register pair but keep in mind that they use at least one extra byte per instruction. TE

# SERIAL ROUTINES FOR THE TEC

## RECEIVER

This is the routine I use when I wish to down-load a file from the IBM. It's a simple routine that converts a serial stream into bytes and stores them in RAM starting at the address provided at 0898. The routine also has an end address to allow a maximum file length. This is in case something goes wrong with the data transfer. Anything important can be protected by placing it above the end address.

No hand-shaking is needed as the TEC can cope with the speed of the data stream. It is up to you to ensure the TEC is ready before you send the data. The serial input is bit 0 of PORT 3. The DAT BOARD has provision for 2 diodes and a resistor at this input to clip an incoming RS232 signal. In the RS232 format, a logic 1 is represented by a negative voltage while a logic 0 is a positive voltage. The clipper on the DAT BOARD changes an RS232 logic 0 (positive voltage) into a digital logic 1 while an RS232 logic 1 is clipped to zero volts and becomes a digital logic 0.

This means that the inputted data must be inverted back into its true form. This is done with the CPL instruction at 092C. The format of the data is as follows: 2400 BAUD, NO PARITY, 8 BITS, STOP BITS OPTIONAL, TEC SPEED: 3.58/2

## SERIAL INPUT ROUTINE

```
0900  2A 98 08     LD HL,(0898)   START ADDRESS IN HL
0903  CD 12 09     CALL 0912      GET BYTE
0906  ED 4B 9A 08  LD BC,(089A)   PUT END ADDR IN BC
090A  B7           OR A           CLEAR CARRY FLAG
090B  E5           PUSH HL        SAVE HL
090C  ED 42        SBC HL,BC      SUB CURRENT ADDR FROM END
090E  E1           POP HL         RECOVER HL
090F  38 F2        JR C 0903      JUMP IF NOT DONE
0911  C9           RET            ELSE RETURN TO JMON
0912  DB 03        IN A,03        LOOK FOR START BIT
0914  07           RLCA           PUT IN CARRY
0915  30 FB        JR NC 0912     LOOP UNTIL START BIT FOUND
0917  06 40        LD B,40        DELAY TO HALF WAY IN
0919  10 FE        DJNZ 0919      FIRST CELL
091B  1E 00        LD E,00        E IS RECEIVER BYTE
091D  0E 08        LD C,08        C IS COUNT SET FOR 8 BITS
091F  DB 07        IN A,07        INPUT BIT
0921  07           RLCA           INTO CARRY FLAG
0922  CB 1B        RR E           THEN STORE IN E
0924  06 39        LD B,39        B=HALF CELL DELAY
0926  10 FE        DJNZ 0926      LOOP TO NEXT CELL ARRIVES
0928  0D           DEC C          DEC LOOP COUNTER
0929  20 F4        JR NZ 091F     JUMP IF 8 BITS NOT DONE
092B  7B           LD A,E         PUT INPUTTED BYTE IN A
092C  2F           CPL            AND INVERT TO TRUE FORM
092D  77           LD (HL),A      STORE IN MEMORY
092E  23           INC HL         INCREASE MEMORY POINTERS
092F  C9           RET            RETURN FROM SUBROUTINE
```

## SERIAL OUTPUT ROUTINE

```
0A00  2A 98 08     LD HL,(0898)   PUT START IN HL
0A03  CD 12 0A     CALL 0A12      OUTPUT BYTE
0A06  ED 4B 9A 08  LD BC (089A)   END IN BC
0A0A  B7           OR A           CLEAR CARRY
0A0B  E5           PUSH HL        SAVE START
0A0C  ED 42        SBC HL,BC      SUB END FROM START
0A0E  E1           POP HL         RECOVER START
0A0F  38 F2        JR C,0A03      JUMP IF END << THAN
0A11  C9           RET            START ELSE RETURN
0A12  3E 80        LD A,80        SET START BIT
0A14  D3 01        OUT (01),A     OUT START BIT
0A16  CD 2D 0A     CALL 0A2D      CALL DELAY
0A19  7E           LD A,(HL)      GET BYTE TO OUTPUT
0A1A  23           INC HL         POINT TO NEXT BYTE
0A1B  06 08        LD B,08        SET COUNT FOR 8 BITS
0A1D  0F           RRCA           PUT BIT INTO BIT 7
0A1E  EE 80        XOR 80         COMPLEMENT BIT
0A20  D3 01        OUT (01),A     OUTPUT IT
0A22  CD 2D 0A     CALL 0A2D      CALL DELAY
0A25  10 F6        DJNZ,0A1D      DO FOR 8 BITS
0A27  AF           XOR A          CLEAR FOR STOP BIT
0A28  D3 01        OUT (01),A     OUT STOP BIT x2
0A2A  CD 2D 0A     CALL 0A2D      FIRST STOP DELAY
0A2D  C5           PUSH BC        SAVE BIT COUNT
0A2E  06 36        LD B,36        LOAD B WITH DELAY
0A30  10 FE        DJNZ,0A30      DO DELAY
0A32  C1           POP BC         RECOVER BIT COUNT
0A33  C9           RET            DONE
```

## SERIAL OUTPUT ROUTINE

This is the complement routine of the serial receiver. It will send serial data through the TEC speaker bit. The data is taken from the latch side of the base resistor of the transistor inverter and inputted directly to an RS232 Rx input or the DAT BOARD serial input.

Strictly speaking the data stream is not RS232 compatible but in practice it works ok, although the occasional error may creep in.

Oh yes, before sending data, the key press beep must be turned off. To do this, place FF at 0822 and put AA at 08FF.

The serial sender uses the same start and end buffers as the receive described above with the same speed etc. Two stop bits are sent as this provides compatibility with all serial systems.

## IBM SOFTWARE

The software I used for receiving the serial is PROCOMM. It is a public domain program and can be purchased from the Talking Electronics Shop. Cat S-449.

The protocol to use is ASCII.

The sending software poses a few difficulties. One big problem is that some packages won't send the 1A character. Actually, I believe the problem is in the DOS serial interrupt and if the software uses it then it won't send the 1A character.

It is rare that I send anything back to the TEC and when I do, it's with a serial routine Craig wrote and probably won't work with all computers as it directly manipulates the hardware; not a recommended practice.

It is up to you to experiment around and find something that works.

I would like to hear from anyone who has found or written a good sending routine that doesn't have the 1A character problem.

Hardware wise, the CTS must be taken high before the IBM will send the data. This means that the IBM to TEC link consists of three wires: the ground, the serial data line and +5v.

Only ground and the serial data are required for the TEC to IBM link.

# ERROR FILE

## DAT BOARD ERROR

The transistor on the DAT BOARD is the wrong way around on the first 2 issues of the board. The error has been corrected on later boards.

To identify the error, look at the transistor. If the flat side faces to the right it is incorrect.

Interesting enough, I have yet to hear of anyone who could not get the tape system working! For some reason beyond me, the tape system does work with the transistor in the wrong-way-around but of course you should turn it around the right way. You will have to bend the base lead back to fit it in the right way.

## LCD PIN NUMBERS

The missing LCD pin numbers on the circuit diagram are the following:

Vcc - pin 2, Vo - pin 3, D3 - pin 10, D4 - pin 11, D5 - pin 12, D6 - pin 13 and D7 - pin 14.

## EPROM PROGRAMMER MODS

The 10n cap I recommended that you put across the 100k emitter resistor should not be put there. This cap forms a RC network with the 10k resistor shown in the bottom right corner of the circuit diagram on page 32 and delays the active low chip select signal for to long. Instead, put the cap between the collector and ground of the same transistor.

The description on the same page, second last paragraph middle column tells you to solder one end of a 10k resistor to the diode junction and the other to ground.

This should say: solder one end to the diode junction and the other end to 5v.

The corrected circiut diagram is below.

## PRINT-3 HEX DUMP

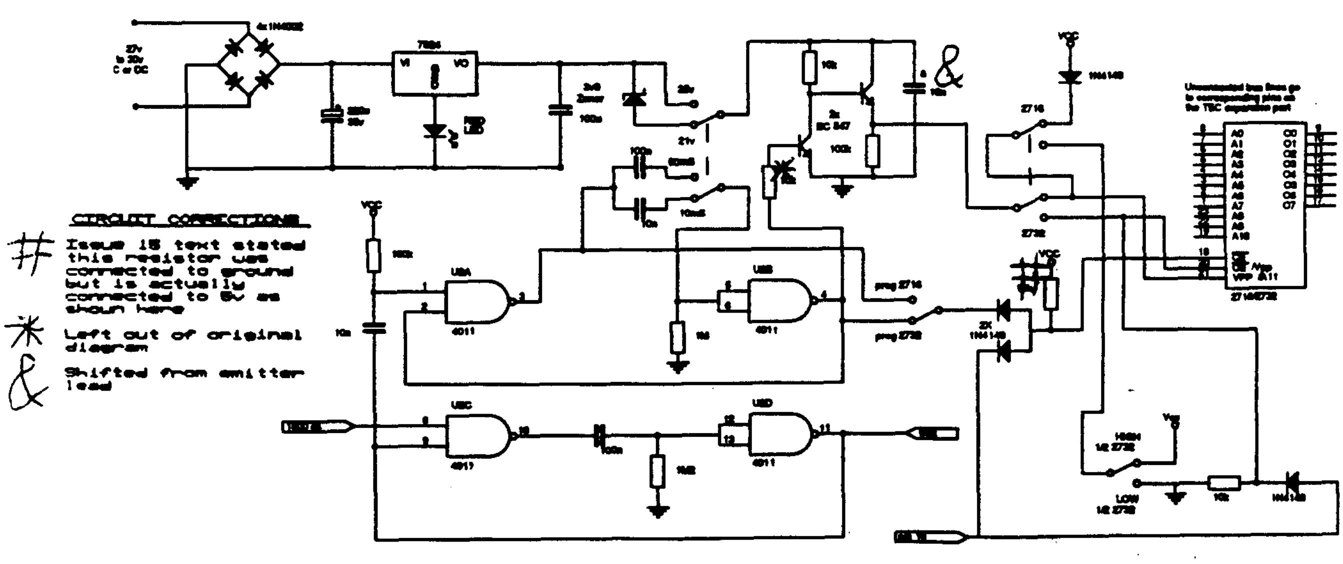
Two bytes in the hex dump were corrupted in the transfer between the TEC and the IBM clone. The first is at address 1A12. The listing contains FF, but the correct value is 01. The second wrong byte is at address 1A84. The listing has it at FF, but we know it is meant to be 0F (it is pretty obvious isn't it)?

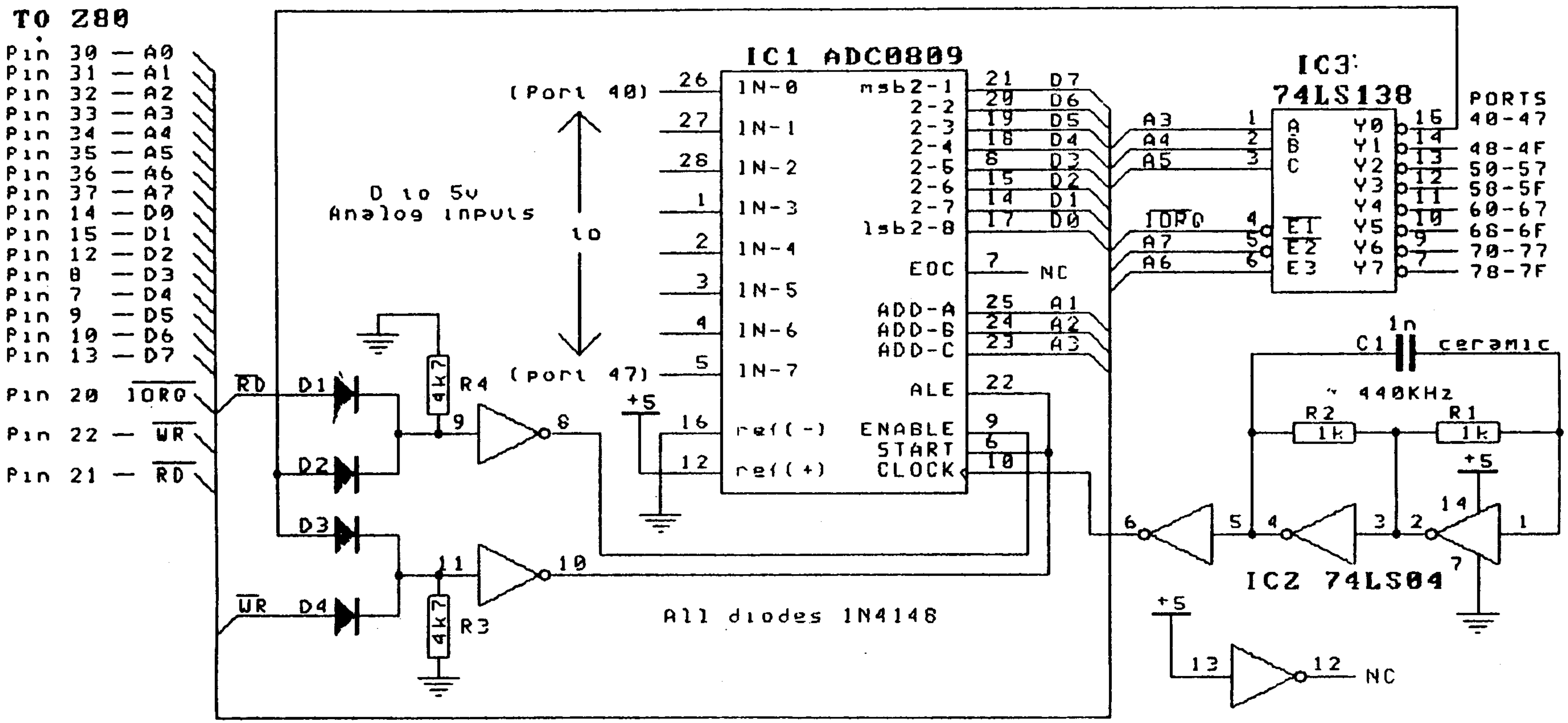
## TURBO OSCILLATOR PCB

Unfortunately the TURBO OSCILLATOR PCB's were made with the wrong artwork. The PCB's have one critical error, the power rails on the 74ls73 are transposed. To correct this pins 4 and 11 must be isolated and the tracks going to each re-routed to go to the other pin.

The price of the kit has been kept low to compensate for this inconvenience

This completes the list of known errors and omissions. Thanks to those who pointed them out, particularly David Smith who picked up on most errors (and let me know about them).




TO Z80
Pin 30 — A0
Pin 31 — A1
Pin 32 — A2
Pin 33 — A3
Pin 34 — A4
Pin 35 — A5
Pin 36 — A6
Pin 37 — A7
Pin 14 — D0
Pin 15 — D1
Pin 12 — D2
Pin 8 — D3
Pin 7 — D4
Pin 9 — D5
Pin 10 — D6
Pin 13 — D7
Pin 20 IORQ
Pin 22 — WR
Pin 21 — RD
IC1 ADC0809
IN-0
IN-1
IN-2
IN-3
IN-4
IN-5
IN-6
IN-7
ref(-)
ref(+)
msb2-1
2-2
2-3
2-4
2-5
2-6
2-7
lsb2-8
EOC
ADD-A
ADD-B
ADD-C
ALE
ENABLE
START
CLOCK
(Port 40)
to
(port 47)
0 to 5v
Analog inputs
NC
IC3:
74LS138
PORTS
40-47
48-4F
50-57
58-5F
60-67
68-6F
70-77
78-7F
C1 1n ceramic
~ 440KHz
R2 1k
R1 1k
IC2 74LS04
+5
NC
R4 4k7
R3 4k7
D1
D2
D3
D4
All diodes 1N4148