Tower of Eightness Reference Manual.

Author: Duncan Gunn

Revised 31 October 2021.

Contents:

Chapter 1

The Zero Page

This chapter describes the zero-page memory usage of the ToE.

LAB_WARM	$00	BASIC warm start entry point
Wrmjpl	$01	BASIC warm start vector jump low byte
Wrmjph	$02	BASIC warm start vector jump high byte
Usrjmp	$0A	USR function JMP address
Usrjpl	$0B	USR function JMP vector low byte
Usrjph	$0C	USR function JMP vector high byte
Nullct	$0D	nulls output after each line
TPos	$0E	BASIC terminal position byte
TWidth	$0F	BASIC terminal width byte
Iclim	$10	Input column limit
Itempl	$11	Temporary integer low byte
Itemph	$12	Temporary integer high byte
nums_1	$11	Number to bin/hex string convert MSB
nums_2	$12	Number to bin/hex string convert
nums_3	$13	Number to bin/hex string convert LSB
Srchc	$5B	Search character
Temp3	$5B	Temp byte used in number routines
Scnquo	$5C	Scan-between-quotes flag
Asrch	$5C	Alt search character
XOAw_l	$5B	eXclusive OR, OR and word low byte
XOAw_h	$5C	eXclusive OR, OR and AND word high byte
Ibptr	$5D	Input buffer pointer
Dimcnt	$5D	# of dimensions
Tindx	$5D	Token index
Defdim	$5E	Default DIM flag
Dtypef	$5F	Data type flag, $FF=string, $00=numeric
Oquote	$60	Open quote flag (b7) (Flag: DATA scan; LIST quote; memory)
Gclctd	$60	Garbage collected flag
Sufnxf	$61	Subscript/FNX flag, 1xxx xxx = FN(0xxx xxx)
Imode	$62	Input mode flag, $00=INPUT, $80=READ
Cflag	$63	Comparison evaluation flag
TabSiz	$64	TAB step size (was input flag)
next_s	$65	Next descriptor stack address

These two bytes form a word pointer to the item currently on top of the descriptor stack.

last_s	$66	Last descriptor stack address low byte

last_sh	$67	Last descriptor stack address high byte (always $00)
des_sk	$68	Descriptor stack start address (temp strings)
	$70	End of descriptor stack
ut1_pl	$71	Utility pointer 1 low byte
ut1_ph	$72	Utility pointer 1 high byte
ut2_pl	$73	Utility pointer 2 low byte
ut2_ph	$74	Utility pointer 2 high byte
Temp_2	$71	Temp byte for block move
FACt_1	$75	FAC temp mantissa1
FACt_2	$76	FAC temp mantissa2
FACt_3	$77	FAC temp mantissa3
dims_l	$76	Array dimension size low byte
dims_h	$77	Array dimension size high byte
TempB	$78	Temp page 0 byte
Smeml	$79	Start of mem low byte (Start-of-Basic)
Smemh	$80	Start of mem high byte (Start-of-Basic)
Svarl	$7B	Start of vars low byte (Start-of-Variables)
Svarh	$7C	Start of vars high byte (Start-of-Variables)
Sarryl	$7D	Var mem end low byte (Start-of-Arrays)
Sarryh	$7E	Var mem end high byte (Start-of-Arrays)
Earryl	$7F	Array mem end low byte (End-of-Arrays)
Earryh	$80	Array mem end high byte (End-of-Arrays)
Sstorl	$81	String storage low byte (String storage (moving down))
Sstorh	$82	String storage high byte (String storage (moving down))
Sutill	$83	String utility ptr low byte
Sutilh	$84	String utility ptr high byte
Ememl	$85	End of mem low byte (Limit-of-memory)
Ememh	$86	End of mem high byte (Limit-of-memory)
Clinel	$87	Current line low byte (Basic line number)
Clineh	$88	Current line high byte (Basic line number)
Blinel	$89	Break line low byte (Previous Basic line number)
Blineh	$8A	Break line high byte (Previous Basic line number)
Cpntrl	$8B	Continue pointer low byte
Cpntrh	$8C	Continue pointer high byte
Dlinel	$8D	Current DATA line low byte
Dlineh	$8E	Current DATA line high byte

Dptrl	$8F	DATA pointer low byte
Dptrh	$90	DATA pointer high byte
Rdptrl	$91	Read pointer low byte
Rdptrh	$92	Read pointer high byte
Varnm1	$93	Current var name 1st byte
Varnm2	$94	Current var name 2nd byte
Cvaral	$95	Current var address low byte
Cvarah	$96	Current var address high byte
Frnxtl	$97	Var pointer for FOR/NEXT low byte
Frnxth	$98	Var pointer for FOR/NEXT high byte
Tidx1	$97	Temp line index
Lvarpl	$97	Let var pointer low byte
Lvarph	$98	Let var pointer high byte
Prstk	$99	Precedence stacked flag
comp_f	$9B	compare function flag, bits 0,1 and 2 used Bit 2 set if > Bit 1 set if = Bit 0 set if <
func_l	$9C	Function pointer low byte
func_h	$9D	Function pointer high byte
garb_l	$9C	Garbage collection working pointer low byte
garb_h	$9D	Garbage collection working pointer high byte
des_2l	$9E	String descriptor_2 pointer low byte
des_2h	$9F	String descriptor_2 pointer high byte
g_step	$A0	Garbage collect step size
Fnxjmp	$A1	Jump vector for functions
Fnxjpl	$A2	Functions jump vector low byte
Fnxjph	$A3	Functions jump vector high byte
g_indx	$A2	Garbage collect temp index
FAC2_r	$A3	FAC2 rounding byte
Adatal	$A4	Array data pointer low byte
Adatah	$A5	Array data pointer high byte

Nbendl	$A4	New block end pointer low byte
Nbendh	$A5	New block end pointer high byte
Obendl	$A6	Old block end pointer low byte
Obendh	$A7	Old block end pointer high byte
Numexp	$A8	String to float number exponent count
Expcnt	$A9	String to float exponent count
Numbit	$A8	Bit count for array element calculations
Numdpf	$AA	String to float decimal point flag
Expneg	$AB	String to float eval exponent -ve flag
Astrtl	$AA	Array start pointer low byte
Astrth	$AB	Array start pointer high byte
Histrl	$AA	Highest string low byte
Histrh	$AB	Highest string high byte
Baslnl	$AA	BASIC search line pointer low byte
Baslnh	$AB	BASIC search line pointer high byte
Fvar_l	$AA	Find/found variable pointer low byte
Fvar_h	$AB	Find/found variable pointer high byte
Ostrtl	$AA	Old block start pointer low byte
Ostrth	$AB	Old block start pointer high byte
Vrschl	$AA	Variable search pointer low byte
Vrschh	$AB	Variable search pointer high byte
FAC1_e	$AC	FAC1 exponent
FAC1_1	$AD	FAC1 mantissa1
FAC1_2	$AE	FAC1 mantissa2
FAC1_3	$AF	FAC1 mantissa3
FAC1_s	$B0	FAC1 sign (b7)
str_ln	$AC	String length
str_pl	$AD	String pointer low byte
str_ph	$AE	String pointer high byte
des_pl	$AE	String descriptor pointer low byte
des_ph	$AF	String descriptor pointer high byte
mids_l	$AF	MID$ string temp length byte
negnum	$B1	String to float eval -ve flag
numcon	$B1	Series evaluation constant count

FAC1_o	$B2	FAC1 overflow byte
FAC2_e	$B3	FAC2 exponent
FAC2_1	$B4	FAC2 mantissa1
FAC2_2	$B5	FAC2 mantissa2
FAC2_3	$B6	FAC2 mantissa3
FAC2_s	$B7	FAC2 sign (b7)
FAC_sc	$B8	FAC sign comparison, Acc#1 vs #2
FAC1_r	$B9	FAC1 rounding byte
ssptr_l	$B8	String start pointer low byte
ssptr_h	$B9	String start pointer high byte
sdescr	$B8	String descriptor pointer
csidx	$BA	Line crunch save index
Asptl	$BA	Array size/pointer low byte
Aspth	$BB	Array size/pointer high byte
Btmpl	$BA	BASIC pointer temp low byte
Btmph	$BB	BASIC pointer temp low byte
Cptrl	$BA	BASIC pointer temp low byte
Cptrh	$BB	BASIC pointer temp low byte
Sendl	$BA	BASIC pointer temp low byte
Sendh	$BB	BASIC pointer temp low byte
LAB_IGBY	$BC	Get next BASIC byte subroutine
LAB_GBYT	$C2	Get current BASIC byte subroutine
Bpntrl	$C3	BASIC execute (get byte) pointer low byte
Bpntrh	$C4	BASIC execute (get byte) pointer high byte
	$D7	end of get BASIC char subroutine
Rbyte4	$D8	Extra PRNG byte
Rbyte1	$D9	Most significant PRNG byte
Rbyte2	$DA	Middle PRNG byte
Rbyte3	$DB	Least significant PRNG byte
NmiBase	$DC	NMI handler enabled/setup/triggered flags <u>bit function</u> 7 interrupt enabled 6 interrupt setup 5 interrupt happened
	$DD	NMI handler addr low byte

	$DE	NMI handler addr high byte
IrqBase	$DF	IRQ handler enabled/setup/triggered flags
	$E0	IRQ handler addr low byte
	$E1	IRQ handler addr high byte
	$E2	TPB card temporary location
	$E3	TPB card temporary location
	$E4	TAPE temporary location.
	$E5	TAPE BlockLo
	$E6	TAPE BlockHi
	$E7	TOE_MemptrLo low byte general purpose pointer
	$E8	TOE_MemptrHi high byte general purpose pointer.
	$E9	unused
	$EA	unused
	$EB	unused
	$EC	unused
	$ED	unused
	$EE	unused
Decss	$EF	Number to decimal string start
Decssp1	$F0	Number to decimal string start
	$FF	Decimal string end

Chapter 2

OS Memory Map

Operating systems memory is necessary, or the OS would not work. Some benefit to the user is afforded by the clear documentation of this memory below.

Memory used by EhBASIC not in the zero page.

$200	ccflag	Control-C flag. 0=Enabled, 1=Disabled.
$201	ccbyte	Control-C character. One may define whatever one chooses, but with great power comes great RESETs!
$202	ccnull	Timeout time for the Control-C character.
$203-$204	VEC_CC	Control-C check vector.
$205-$206	VEC_IN	EhBASIC Input vector.
$207-$208	VEC_OUT	EhBASIC output vector.
$209-$20A	VEC_LD	EhBASIC LOAD vector.
$20B-$20C	VEC_SV	EhBASIC SAVE vector.
$20D-£20E	VEC_VERIFY	EhBASIC VERIFY vector.

THE Next two vectors (Greyed out) are no longer in use.

$F9F3-$F9F4	IRQ_vec	Interrupt ReQuest vector. (Currently patched out, see Chapter 12, The IRQ Sub-System.)
$20F-$210	NMI_vec	Non Maskable Interrupt vector.

Memory used by the ToE Monitor top level components.

$5E0	os_outsel	Output stream selection bitfield. See table at end of chapter.
$5E1	os_infilt	Input stream filter selection bitfield. See table at end of chapter.
$5E2	RESERVED	Reserved for forthcoming os_insel. This variable will contain an input stream selection bitfield.

Memory used by the Tower Peripheral Bus.

$5F2	TPB_curr_dev	TPB Currently selected device ID.
$5F3	TPB_dev_type	TPB device type.
$5F4	TPB_last_read	Last read byte from the TPB bus.
$5F5	TPB_BUS_status	Status word for the TPB bus. Subject to change.
$5F6	TPB_BUS_tries	Bus device counter, this ensures fewer hangs.
$5F7	TPB_BUS_lim	Bus countdown timer limit. (Reload value).
$5F8	TPB_BUS_lenlo	Low byte of the length of the block in or out.

$5F9	TPB_BUS_lenhi	High byte of the length of the block in or out.
$5FA	TPB_BUS_stlo	Low byte of the start of the block in or out.
$5FB	TPB_BUS_sthi	High byte of the start of the block in or out.
$5FC	TPB_BUS_blk_type	Type of block transfer. See Table below.
$600	TPB_Dev_table	Device descriptor table.
$610-$6FF	TPB_BUS_IO_buff	Buffer for IO operations on the TPB bus.
$700-$7FF	TPB_BUFFER	Buffer for TPB block operations.

Memory used by the TowerTAPE Filing System.

$900-$901	V_TAPE_BlockSize	Size of block to transfer to or from tape.
$902	TAPE_Temp2	Temporary memory location for TFS internals.
$903	TAPE_Temp3	Temporary memory location for TFS internals.
$904	TAPE_Temp4	Temporary memory location for TFS internals.
$905	TAPE_LineUptime	Number of passes the tape system superloop has made since the tape line rose.
$906	TAPE_Demod_Status	Demodulated bit status.
$907	TAPE_Demod_Last	Previous demodulated bit status.
$908	TAPE_StartDet	Start bit detection status.
$909	TAPE_RX_Status	Receive engine status bitfield.
$90A	TAPE_BitsToDecode	Down counter of remaining bits to decode.
$90B	TAPE_ByteReceived	Last byte received by the TFS.
$90C	TAPE_Sample_Position	Countdown timer for bit engine sample synchronization.
$90D	TAPE_BlockIn_Status	Status register for the F_TAPE_BlockIn function.
$90E-929	TAPE_Header_Buffer	This is where the tape header information is stored for use when **`SAVE`**`ing` and **`LOAD`**ing.
$92A	TAPE_CS_AccLo	Tape checksum accumulator low byte.
$92B	TAPE_CS_AccHi	Tape checksum accumulator high byte.
$92C	V_TAPE_Phasetime	Tape phasetime variable.

The following variables are allocated for future upgrades to the tape system and ignored for now.

$92D	V_TAPE_Sample_Offset	Sample offset variable. This determines how far into a bit the sample is taken.
$92E	V_TAPE_Bitlength	How many bits in a word stored on tape.

$92F	V_TAPE_bitcycles	Number of cycles of the super-loop to a bit.
$930	V_TAPE_Verify_Status	Stores the verify status bits used by the TAPE_VERIFY_vec ($FFD2)
$931-$942	V_TAPE_Fname_Buffer	Working file name buffer. Stores the null terminated file name specified in **LOAD**, **SAVE** and **VERIFY** commands.
$943	V_TAPE_LOADSAVE_Type	Temporary store of what file type is being worked on by the TAPE file system.
$944-$945	V_TAPE_Address_Buff	Temporary store of the starting address being worked on by the TAPE file system.
$946-$947	V_TAPE_Size_Buff	Temporary store of how big the file being worked on is.

AY Soundcard V2 Memory Locations

$A00	AY_Reg	Register to write to
$A01	AY_Data	Contents to be transferred between the system and the AY-3-8912A when calling AY_Userwrite_vec ($FFD4) or AY_Userread_vec ($FFD7).

IRQ Handler Subsystem Locations

$A20-$A2F	IRQH_CallList	Table of 8 addresses for the IRQ Handlers
$A30-$A31	IRQH_CallReg	Address being worked on by set or clear calls.
$A32	IRQH_ClaimsList	bitfield showing which IRQ's claimed an interrupt. LSb is IRQ0, MSb is IRQ7
$A33	IRQH_MaskByte	Selection switch for interrupts. 1 means on and the order is LSb for IRQ0 through MSb for IRQ7.
$A34	IRQH_WorkingMask	Used internally when enumerating IRQs.
$A35	IRQH_CurrentEntry	Convenience variable informing IRQs etc which IRQ is currently being handled.
$A36-$A46	IRQH_Command_Table	16-byte table consisting of a parameter followed by the command code. Ordered by ascending IRQ number.

Countdown Timer IRQ Locations

$A46	CTR_V	Counter. This is the countdown timer's present value, decremented each GPIO card Timer 1 IRQ.
$A48	CTR_LOAD_VAL_V	Counter Load Value. Reload value for the GPIO card Timer 1 IRQ. Decrements once per PHI2 clock. Refer to your CPU jumper setting.

System Vector Locations

$FF60	TOE_PrintStr_vec	Prints a null terminated string to the currently selected outputs.
$FF90	ANSI_init_vec	Initialises the ANSI card.
$FF93	ANSI_write_vec	Writes whatever is in the accumulator to the ANSI card.
$FF96	TPB_init_vec	Initialises the Tower Peripheral Bus card.
$FF99	TPB_LPT_write_vec	Writes the contents of A to the Centronics port.
$FF9C	TPB_tx_byte_vec	Writes a byte to the tower peripheral bus.
$FF9F	TPB_block_vec	Writes a block of bytes to the tower peripheral bus.
$FFA2	TPB_ATN_handler_vec	Processes ATN signals generated by TPB peripherals.
$FFA5	TPB_rx_byte_vec	Reads a byte from the TPB bus.
$FFA8	TPB_rx_block_vec	Reads a block from the TPB bus.
$FFAB	TPB_Dev_Presence_vec	Checks for the presence of a given device on the bus.
$FFAE	TPB_Req_Dev_Type_vec	Requests the device report its device type.
$FFB1	TPB_dev_select_vec	Selects a device on the TPB. A device must not respond unless selected.
$FFB4	TPB_Ctrl_Block_Wr_vec	Writes to a devices control block.
$FFB7	TPB_Ctrl_Block_Rd_vec	Reads a devices control block.
$FFBA	TAPE_Leader_vec	Generates a tape leader signal.
$FFBD	TAPE_BlockOut_vec	Transmits a block of bytes.
$FFC0	TAPE_ByteOut_vec	Transmits a byte.
$FFC3	TAPE_BlockIn_vec	Reads a block of bytes from tape.
$FFC6	TAPE_ByteIn_vec	Reads a byte from tape.
$FFC9	TAPE_init_vec	Initialises the tape system.
$FFCC	TAPE_SAVE_BASIC_vec	**`SAVE`**s a program to tape.
$FFCF	TAPE_LOAD_BASIC_vec	**`LOAD`**s a program from tape.
$FFD2	TAPE_F_TAPE_VERIFY_BASIC	**`VERIFY`**s a program in memory against what is stored on tape. Can be accessed from EhBASIC using the **`VERIFY`** command.
$FFD5	AY_Userwrite_vec	Writes to the specified AY register.
$FFD8	AY_Userread_vec	Reads from the specified AY register.
$FFDB	IRQH_Handler_Init_vec	Initialises the IRQ Handler sub-system.
$FFDE	IRQH_SetIRQ_vec	Atomically sets an IRQ address in the handler table.
$FFE1	IRQH_ClrIRQ_vec	Atomically clears an IRQ to the null IRQ handler in the specified handler table.
$FFE4	IRQH_SystemReport_vec	Returns the IRQ Handler subsystem version and base address of the handler data structure.

System Softswitches

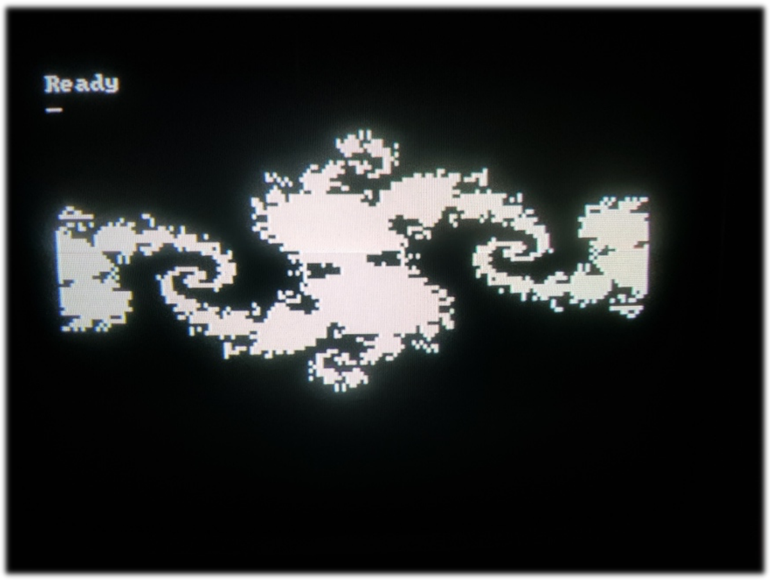
System soft switches are bits stored within system variables that control characteristics of the system such as output streams and input filtering. This section is likely to grow with revision changes.

os_outsel ($5E0)								
Bit	7	6	5	4	3	2	1	0
Function	Reserved for future uses.			TAPE	ACIA2	TPB LPT	ANSI	ACIA1

os_infilt ($5E1)								
Bit	7	6	5	4	3	2	1	0
Function	Reserved for future uses.						ACIA2 LF Filter	ACIA1 LF Filter.

Chapter 3

The ANSI Card


Ready
_

ANSI stands for American National Standards Institute, and in this case, we're dealing with ANSI terminal output.

Each and every character is put into the card's internal memory by means of addressing it's W65C22S-TPG14 versatile interface adapter. The ANSI card must be set to base address C000-C00F by setting the selector wheel to 0 if one wants to access it via the TowerOS. To read the cards address, just use the number on the wheel as the most significant digit of a two-digit hexadecimal number and add the nIOSEL base address to it.

Refer to the Western Design Center datasheet for operation of the VIA, something which cannot be understated in its importance on the ToE as it is extensively utilised on several add-on cards.

The Avail line is connected to PA6 and indicates the cards availability for use. PA7 is the acknowledge line and flips upon receipt of any byte placed upon port B.

Therefore, the relevant addresses are as per the below table.

Address	ANSI Function
0	Output byte (Output Register B)
1	PA7 = Ack, PA6 = Avail, PA5-0 not used. (Output Register A)
2	Data Direction Register B. Should be set to $FF
3	Data Direction Register A. Should be set to $40

To write directly to the ANSI processor, one checks the Ack bit and current Avail bit are matching, only changing the content of the data byte when they do. Then, one flips the Avail line. When the ANSI processor has accepted it, both the Ack and Avail lines will match. One can go about one's merry way but may not change the contents without first checking for agreement.

The nIRQ line of the VIA is *not* connected to the system bus. If one wishes to use the VIA internal hardware to generate an interrupt, one must use a Schottky diode to wire-or to the appropriate interrupt line.

Access to the ANSI card through the TowerOS is the recommended method and the following functions below are how this is achieved.

ANSI_init_vec *Call Address: $FF90*

Calling this address initialises the ANSI card registers.

ANSI_write_vec *Call Address: $FF93*

Whatever is in the Accumulator is written to the ANSI card. Note that if the ANSI card is still busy, this will function will block until it has completed. There is no timeout or error condition.

System Soft switch Control

The system soft switch for this card is in os_outsel ($5E0) and is bit 1. Setting this bit causes the system to send output to the ANSI card and is on by default unless you use the ACIA build of the ToE ROM.

Controlling the Display

To control the display, one sends control codes which cause the video processor to update the display contents. There is a full set of ASCII characters including extended PC DOS characters and some limited graphics.

Other effects include doubling the width, height of the characters, and whether it is bold.

To control the character width, height and whether it is bold, one sends a character control attribute $18 (24) followed by a byte containing the following bitfield:

Character Control Attribute $18 (24) followed by the binary below: -							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Graphics	Spare	Spare	Spare	Spare	Double Height	Bold	80 Columns

Positioning the Cursor

Issuing control code $0E (14) (set column) or $0F (15) (set row) followed by the desired position achieves this goal.

Redefining the Cursor

Control code $02 (2) followed by the ASCII character desired defines the cursor. If the cursor is set to 0 then it is turned off. Leaving the cursor over a graphical area causes the undesirable effect of making graphical area to flash.

Accessing the PC DOS Characters (0 (0) – $1F (31))

To use these extra characters, send $1A (26) first, this causes the next character to be displayed as the specified PC DOS character.

Simple Graphics

The pixel resolution is 160 across by 100, the origin of which is top left at 0,0 up to 159,99 in the bottom right.

One may issue a SetPixel command $5 (5) or ClearPixel $6 (6) command followed by x then y.

Please note that there is currently no way to read back what you have on the screen so you will need to keep track of the salient details.

Writing a Pixel Pattern Directly

Internally, each character cell can be loaded with a 2 by 4 pattern by first sending $80 (128), then the bit pattern is defined by the bit position of the character cell.

Video Control Codes

$01 (01)	Cursor home	*(Standard ASCII)*
$02 (02)	Define cursor character (2nd byte is the curs character, or 0 to turn off)	
$03 (03)	Cursor blinking	
$04 (04)	Cursor solid	
$05 (05)	Set graphics pixel (next two bytes = x,y)	
$06 (06)	Reset graphics pixel (next two bytes = x,y)	
$08 (08)	Backspace	*(Standard ASCII)*
$09 (09)	Tab	*(Standard ASCII)*
$0A (10)	Linefeed	*(Standard ASCII)*
$0C (12)	Clear screen	*(Standard ASCII)*
$0D (13)	Carriage return	*(Standard ASCII)*
$0E (14)	Set column 0 to 79 (2nd byte is the column number) or 0 to 39 for a 40 char line	
$0F (15)	Set row 0 to 24 (2nd byte is the row number)	
$10 (16)	Delete start of line	
$11 (17)	Delete to end of line	
$12 (18)	Delete to start of screen	
$13 (19)	Delete to end of screen	
$14 (20)	Scroll up	
$15 (21)	Scroll down	
$16 (22)	Scroll left	
$17 (23)	Scroll right	
$18 (24)	Set font attribute for the current line	
$1A (26)	Treat next byte as a character (to allow PC DOS char codes 1 to 31 to be displayed on screen)	
$1B (27)	ESC - reserved for ANSI sequences	
$1C (28)	Cursor right	
$1D (29)	Cursor Left	
$1E (30)	Cursor up	
$1F (31)	Cursor down	
$20 (32) to...		
$7E (126)	Standard ASCII codes	
$7F (127)	Delete	
$80 (128) to...		
$FF (255)	PC (DOS) extended characters	

Chapter 4

The Single ACIA Card

Basic Description

Serial communications are provided to permit headless usage, for serial terminals, modems and many other devices to be connected to the Tower of Eightness. Most notably is keyboard input, the PS2 to Serial interface being the easiest way to control the ToE. Without some way to communicate with the ToE it would be useless and so this is one of the essential interfaces. The default port setting is 9600 baud, 1 start bit, 1 stop bit, no parity and RTS/CTS hardware handshaking.

The 65C51-4P or equivalent Asynchronous Communications Interface Adapter which ACIA stands for, bridges the gap between the system bus and RS232 serial. This card provides a 9 pin RS232 port at the correct signalling levels.

Mark is -Ve and Space is +Ve. Serial data always starts with a start bit, 5 to 8 data bits depending on the setting and at least one stop bits at a regular rate known as the bitrate. The bit rate is not the baud rate as can be realised by considering that the start bit and stop bits take up time also, therefore the baud rate is less than the bitrate by necessity.

Provided below is the pinout of the serial IO DB9 connector which is wired as Data Terminal Equipment. The only handshaking lines provided however are Request To Send and Clear To Send.

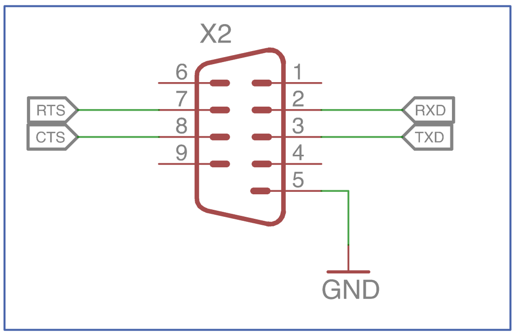


Figure 1

Setting the IO Address

To set the IO address one moves jumpers on JP3 to either closed or open positions. Open is a 1 and closed is a 0. There are six of them and they form the top six bits of the IO address offset from nIOSEL which is at $C000. This card occupies four locations in IO address space. The default base address and the one the Tower OS will use is at $C010 meaning that all the jumpers should be set to CCOCCC from back edge to front.

The least significant bit is nearest the bus connector and the most significant furthest away. Whatever binary value is jumpered, just multiply it by four and add the nIOSEL base address.

It is possible to add several ACIA cards, and even mix with dual ACIA cards. but they must not share IO addresses with anything else and at least *one* must be at $C010 as it receives keystrokes.

System Soft switch Control

The system soft switch for output to this card is in os_outsel ($5E0) and is bit 0. Setting this bit causes the system to send output to the ACIA card and is off by default unless you use the ACIA build of the ToE ROM.

There is also a system soft switch for input filtering, which is os_infilt ($5E1). Bit 0 when set to 1 (default is set) strips out linefeed characters ($A).

Chapter 5

The Dual ACIA Card

Basic Description

Serial communications are provided to permit headless usage, for serial terminals, modems, and many other devices to be connected to the Tower of Eightness. Most notably is keyboard input, the PS2 to Serial interface being the easiest way to control the ToE. Without some way to communicate with the ToE it would be useless and so this is one of the essential interfaces. The default port setting is 9600 baud, 1 start bit, 1 stop bit, no parity and RTS/CTS hardware handshaking.

The 65C51-4P or equivalent Asynchronous Communications Interface Adapter which ACIA stands for, bridges the gap between the system bus and RS232 serial. This card provides a 9 pin RS232 port at the correct signalling levels.

Mark is -Ve and Space is +Ve. Serial data always starts with a start bit, 5 to 8 data bits depending on the setting and at least one stop bits at a regular rate known as the bitrate. The bit rate is not the baud rate as can be realised by considering that the start bit and stop bits take up time also, therefore the baud rate is less than the bitrate by necessity.

Provided below is the pinout of the serial IO DB9 connectors which are wired as Data Terminal Equipment. The only handshaking lines provided however are Request To Send and Clear To Send.

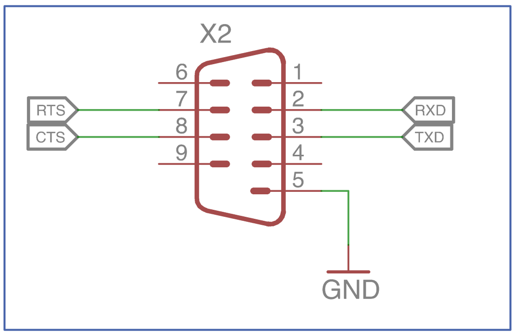


Figure 1

Setting the IO Address

To set the IO address one sets the DIP switches to the appropriate binary address. There are six of them and they form the top six bits of the IO address offset from nIOSEL which is at $C000. This card occupies two sets of four locations in IO address space. The default base address and the one the Tower OS will use for primary input is at $C010 meaning that the appropriate DIP switch setting is 00100. The second ACIA port should be set to $C014 with the DIP switches set to 00101.

The least significant bit is nearest the bus connector and the most significant furthest away. Whatever binary value is jumpered, just multiply it by four and add the nIOSEL base address.

It is possible to add several ACIA cards, but they must not share IO addresses with anything else and at least *one* must be at $C010 as it receives keystrokes upon start up.

System Soft switch Control

The system soft switches for output to this card are in os_outsel ($5E0). Bit 0 controls the primary ACIA and bit 3 controls the secondary ACIA. Setting these bits causes the system to send output to the selected ACIA and ACIA 1 is on off default unless you use the ACIA build of the ToE ROM.

There are also system soft switches for input filtering, which are in os_infilt ($5E1). Bit 0 when set to 1 (default is set) strips out linefeed characters ($A) on ACIA 1 and bit 1 (on by default) filters ACIA 2 in the same manner.

Chapter 6

The GPIO Card

General Description

This card is basically both ports, complete with CA1, CA2, BA1 and BA2 broken out into two identical sockets. The IRQ line is also connected to assist the programmer with its use.

For those wishing to attach an external peripheral, be it home-made or otherwise, the GPIO card implements not one, but two BBC Micro equivalent user ports. Port B is notably however, used for the cassette and joystick interface and if used for other things may clash with other things. This is true only of this card if it is mapped into a base address of $C040.

General Configuration

To set its address, one set the jumpers in accordance with the upper nybble of the lower byte of its address and then adds on the nIOSEL offset which is $C000.

Placing the card on a flat surface with the bus connector to the left, moving a jumper to the left represents a binary 0 and to the right is binary 1. CID1 is the least significant bit and CID4 is the most significant bit so the default from CID1 to 4 is left, left, right, left.

Usage

There are no calls for this card except for what the standard cassette interface uses. To use these, please read the chapter entitled Cassette and Joystick Interface. To take full advantage of this card one needs to write directly to the registers of its VIA chip and so one is directed to the WDC65C22 datasheet. This is especially important with regards to the VIAs electrical specification as exceeding those may lead to damage to the card and even in some cases the ToE itself.

Pinout

Port A. Base address offset 1, Data Direction Register Offset 3.

19	17	15	13	11	9	7	5	3	1
0V	0V	0V	0V	0V	0V	0V	0V	+5V	+5V
20	18	16	14	12	10	8	6	4	2
PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	CA2	CA1

Port B. Base address offset 0, Data Direction Register Offset 2.

19	17	15	13	11	9	7	5	3	1
0V	0V	0V	0V	0V	0V	0V	0V	+5V	+5V
20	18	16	14	12	10	8	6	4	2
PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	CB2	CB1

Be aware that neither of these ports is hot pluggable or protected by any kind of static, or out of specification protection except for the VIAs internal diodes. Also, there is no fuse or other kind of current limiting on the 5V lines. It is up to the user to ensure that the bus is not abused.

Chapter 7

The AY-3-8912A

Sound Card V2

Description

> *Music speaks what cannot be expressed,*
> *sooths the mind and gives it rest,*
> *heals the heart and makes it whole,*
> *and flows from heaven to the soul.*

From the above prose, you might gather that this is a *sound card* and so it is that you may, with minimal skill produce at least some beeps and squarks. With great skill the chip this sound card is based on has produced some very joyous tunes rejoiced by many a kid of the 80's.

The Tower of Eightness has a maximum clock speed primarily limited by its peripheral set. As the processor has a design limit of 14MHz and the backplane is good for perhaps 10MHz, only cards that can handle the system clock frequency and do not unduly load the bus may operate correctly as part of the system. One such device to limit the system speed is the AY-3-8912A sound chip which here is clocked locally at 1.842MHz. To that end, this card isolates the sound chip from the main system bus to permit maximum system operating speed. This isolation is achieved by the use of an on card W65C22S VIA chip.

This VIA is mapped into $C0E0 to $C0EF by convention, thereby allowing programs to directly drive it for maximum performance. This is necessary as sound is popular for games amongst other things.

The nIRQ line is connected so that one may use interrupts.

Port B (Offset 0) is connected to the data lines of the AY-3-8912A, and Port A (Offset 1) is used for control lines. PA0 is connected to BC1 whilst PA1 is connected to BDIR. BC2 is tied high therefore creating a simplified bus for the AY chip.

PA0 and PA1 should be set as outputs and their logic should be driven as listed below.

It is important to note that there is currently no logic protection to prevent bus contention between the AY and the VIA chip and the onus is on the programmer to maintain harmony and protect the logic from stressful conditions. More advanced logic will be brought forward at a later date but for now this is all that is required. This has been done with the Oric-1 and Oric Atmos computers and many of those have survived more than 30 years so this is a proven adequate solution.

When BDIR is low, the AY bus is readied for output of its internal registers, and when high it is ready to receive from the bus.

BC1 should remain low until Port A is configured as an input with BDIR low or an output with appropriate data on it and BDIR high. BC1 should be strobed to make the transfer.

The following table should clarify this: -

Table of Bus States		
BC1	BDIR	State
0	0	Inactive.
1	0	Read from AY. The AY selected register is on the data lines.
0	1	Write to AY. The data lines are transferred to the AY
1	1	Latch Address. Write register address to AY from data lines.

For the full AY-3-8912A hardware specification, refer to the Microchip datasheet.

Configuration

The only configuration that can be done to this card is to set its base address offset by means of the provided jumpers. The four jumpers are arranged such that the least significant bit is nearest the VIA and the most significant nearest the edge of the card. These jumpers form a binary address that is the most significant four bits of the offset from nIOSEL. Leave the base address at $C0E0 unless this is an additional sound card since software is being targeted at the above agreed address and firmware is being written to use it too.

Usage from EhBASIC

There is an easy call provided for writing directly to the AY registers at $FFD5. This takes two parameters, AY_Reg ($A00) contains the register address and AY_Data ($A01) contains the data to be written.

One simply pokes the appropriate register and data to the above addresses and calls the AY_Userwrite function at $FFD5.

To read a register on the AY-3-8912A, one specifies the register of interest by writing it to AY_Reg ($A00), then calls AY_Userread_vec ($FFD8). The registers contents will then be found at AY_Data ($A01).

Chapter 8

The Tape &

Joystick Interface

Description

Tape loading, saving and dual Atari style joystick interface are supplied by the use of this interface. It is designed to be connected to a GPIO port and the TowerOS provides support for it through Port B of the GPIO card mapped to the base address of $C040.

One should keep bits 0 and 7 of DDRB $C042 set when using it as these are used to drive the tape output and select which joystick will be read.

The interface is driven by modulating a bit to generate a tape signal for **SAVE** operations. Conversely, a separate bit is used to monitor the state of the audio coming in for **LOAD** operations.

The tape signal coming in is cleaned up by a biased Schmitt trigger buffer amplifier and fed back to the VIA on pin 6.

The TowerOS provides several system vectors that can be called and has extensive memory locations associated with this interface which will need to be used to make the most of it and are documented later.

To use the joystick interface, one either clears bit 0, selecting joystick port 1 (left) or set it, selecting joystick port 2 (right). The state of the joystick can then be read on Port B bits 1 through 5.

An example fire button read would go something like this: -

```
10 F = NOT(BITTST $C040,5)
```

The observant amongst you will already have noticed the **NOT** in that little snippet of EhBASIC. That is because the joystick ports supply negative logic. A 0 means that switch is pressed.

One should never make the joystick bits outputs as that would prevent access to *both* joysticks on the affected bits.

Below is a table of bits associated with the joystick port: -

Tape and Joystick Bit Usage							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Tape Out	Tape In	Fire	Right	Left	Down	Up	JS Select

Tape Support in TowerOS

TowerOS supports loading and saving of EhBASIC from simple **`LOAD`** and **`SAVE`** commands, but for the advanced user there are system calls and memory locations associated.

To load an EhBASIC program simply type '**`LOAD`** `"<optional filename>"¶`' and play the cassette. Specifying the file name ensures that *only* that file will be loaded

Under normal circumstances, the display will tell you '**`Searching…`**' whilst it is looking for a header, '**`Found BASIC:`** `"<Filename>"`' when it finds the specified file and '**`Loading.`**' whilst it is loading and '**`Ready`**' when it has completed loading. Not giving a file name causes it to load the *first* file it finds.

To save an EhBASIC program one types '**`SAVE`** `"<filename>"¶`' whilst the cassette is recording. The system will tell you it is '**`Saving…`**' and when done will drop you to the **`Ready`** prompt.

Given the nature of cassette storage, errors are a concern and to ensure the integrity of the saved file, the '**`VERIFY`** `"<optional filename>"¶`' command is included. This will operate very much like the **`LOAD`** `""` command, but instead reports on the consistency of the **`SAVE`**d content with that in RAM.

When one gives a specific file name to **`LOAD`** or **`VERIFY`**, the system ignores any files that do not match the name given or of another file type.

File name support is limited to 16 characters.

If you should receive an error message, then remedial action will be required. The following error messages and their meanings are listed below.

- **`Header error. Retrying.`**
  - This means loading error has occurred with the header. TowerTAPE filing system lets you know so you can rewind to the start and try again or escape back to EhBASIC if you give up.
- **`Tape loading error.`**
  - This means loading error has occurred with the file block. TowerTAPE filing system drops you back into the EhBASIC **`Ready`** prompt letting you know of the failure.
- **`Verify Error.`**
  - Receiving this indicates that there is something wrong with the program stored on the cassette and that it will need to be re-done. Consider the quality of the equipment and media in use if this becomes too much of a nuisance.

To save binary data one does as follows: -

‘**SAVE** “<optional filename>” **$A000, $100¶**’

This would save $100 bytes of data starting at address $A000. To load in binary data one must specify a load address as in the below example: -

Loading is accomplished as follows: -

‘**LOAD** “<optional filename>” **$A000¶**’

And to verify the integrity of the data saved: -

‘**VERIFY** “<optional filename>” **$A000¶**’

Note that there is no comma separating the address from the filename, it is not needed. One does have to specify the load address at present, as there is currently no way to indicate that the binary file should be loaded at its original address. This is a feature that is being looked into.

List of TowerTAPE Filing System Calls.

$FFBA	TAPE_Leader_vec
$FFBD	TAPE_BlockOut_vec
$FFC0	TAPE_ByteOut_vec
$FFC3	TAPE_BlockIn_vec
$FFC6	TAPE_ByteIn
$FFC9	TAPE_init_vec
$FFCC	TAPE_SAVE_BASIC_vec (to be used from BASIC)
$FFCF	TAPE_LOAD_BASIC_vec (to be used from BASIC)

$FFBA TAPE_Leader_vec

Calling this vector causes the generation of a leader tone.

$FFBD TAPE_BlockOut_vec

This is the vector call address for F_TAPE_BlockOut. V_TAPE_BlockSize ($900) contains the number of bytes to write and TAPE_BlockLo ($E5) and TAPE_BlockHi($E6) contain the block pointer.

This function returns having modified TAPE_BlockLo ($E5) and TAPE_BlockHi ($E6).

$FFC0 TAPE_ByteOut_vec

Whatever byte is in the accumulator when this function is called is output.

$FFC3 TAPE_BlockIn_vec

Calls to this function require parameters to be loaded into specific memory locations. point to the start of the block to be read. V_TAPE_BlockSize ($900-$901) specifies how many bytes will be read and TAPE_BlockLo ($E5) and TAPE_BlockHi ($E6) contain the write pointer used by this function. No additional information is read.

This function drops through before completion if any character is received from the ACIA card or an overrun error occurs and the state of the engine upon exit is reported in **TAPE_BlockIn_Status** ($90D)

This function returns having modified TAPE_BlockLo ($E5) and TAPE_BlockHi ($E6) which it uses these incrementally point to the byte it is writing to memory at any given moment.

$FFC6 TAPE_ByteIn

Attempts to read a byte from the tape and return it in **TAPE_ByteReceived** ($90B). If a byte is received from the ACIA whilst it is in progress or if an overrun error occurs this function will exit reporting its status in **TAPE_RX_Status** ($909).

$FFC9 TAPE_init_vec

Initialises the TowerTAPE filing system. This is called by TowerOS on bootup and only needs to be called if the TowerTAPE filing system needs to be re-initialised.

$FFCC TAPE_SAVE_BASIC_vec

Causes the EhBASIC program to be saved. This should not be called from anywhere except EhBASIC as it is carefully designed to work as part of EhBASIC.

$FFCF TAPE_LOAD_BASIC_vec

Causes the attempted loading of an EhBASIC program from tape. This should not be called from anywhere except EhBASIC as it is carefully designed to work as part of EhBASIC.

TowerTAPE Filing System, System Variables

Below is a list of system variables with their associated address and size. These were grabbed direct from a spreadsheet which is also available.

System Variable	Address	Number of Bytes
V_TAPE_BlockSize	$900	2
TAPE_temp2	$902	1
TAPE_temp3	$903	1
TAPE_temp4	$904	1
TAPE_LineUptime	$905	1
TAPE_Demod_Status	$906	1
TAPE_Demod_Last	$907	1
TAPE_StartDet	$908	1
TAPE_RX_Status	$909	1
TAPE_BitsToDecode	$90A	1
TAPE_ByteReceived	$90B	1
TAPE_Sample_Position	$90C	1
TAPE_BlockIn_Status	$90D	1
TAPE_Header_Buffer	**$90E**	**24**
TAPE_HeaderID	$90E	4
TAPE_FileType	$912	1
TAPE_FileSizeLo	$913	1
TAPE_FileSizeHi	$914	1
TAPE_LoadAddrLo	$915	1
TAPE_LoadAddrHi	$916	1
TAPE_FileName	$917-927	17
TAPE_ChecksumLo	$928	1
TAPE_ChecksumHi	$929	1
TAPE_CS_AccLo	$92A	1
TAPE_CS_AccHi	$92B	1
V_TAPE_Phasetime	$92C	1
V_TAPE_Sample_Offset	$92D	1
V_TAPE_Bitlength	$92E	1
V_TAPE_bitcycles	$92F	1
V_TAPE_Verify_Status	$930	1
V_TAPE_Fname_Buffer	$931-$942	17
V_TAPE_LOADSAVE_Type	$943	1
V_TAPE_Address_Buff	$944-$945	2
V_TAPE_Size_Buff	$946-$947	2

TowerTAPE File Modulation Scheme (Under review)

Each byte is encoded a series of bursts of carrier. A presence of carrier signifies a 1 and an absence a 0. Each burst of carrier is 16 cycles at 4KHz.

Each byte is structured as follows: -

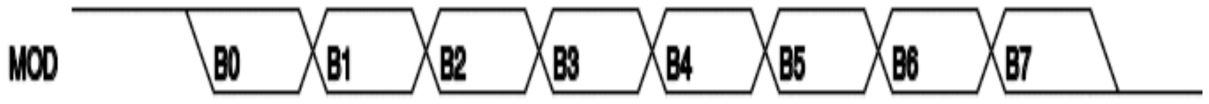


Figure 2

Chapter 9

The V2 Memory Board

Memory Bank Structure on the ToE

The ToE organises its memory into four groups of two banks of memory of 16 kibibytes each. Each bank group has a selection bit associated with it in a write only register located at address $C0FF. Writes to this register should be handled with great care as to not inadvertently page out something you are using at that time. Since many things you might need are located in the bottom and top banks, these are most likely to be of use to the machine code programmer and unlikely to be of use to anyone programming in EhBASIC. The banks with the best outcome to the user of EhBASIC are the middle ones as they can be excluded from use by BASIC at boot time or by careful reconfiguration later.

This register is initialised to 0 at reset placing the lower half of each memory IC into view. For now, the upper four bits should be set to 0 whenever writing to this register as this is reserved for later expansion.

Banks 0 through 2 are RAM and bank 3 is the ToEs ROM containing TowerOS and EhBASIC.

Although bank 3 consists of 32 kibibytes of space, the address range $C000 to $C0FF is put aside for memory mapped hardware and is inaccessible. This constitutes a loss of just half a kibibyte. In a system with 96 kibibytes of RAM and 32 kibibytes of ROM. A small price to pay for such a generous and fully decoded IO space!

Jumper Selection

There are two jumpers on this card, the one nearest the bus connector (JP4), is to allow the use of 27 series 16K EPROMs and should be set 1-2 (back) for 27 series, or 2-3 (front) for 28C256 EEPROMs.

The other jumper (JP5) is a write protect. One cannot state enough that this should be left write protected under normal circumstances to avoid crashes and corruption of the ROM. Position 1-2 (back) is write protected and 2-3 (front) allows writes to occur.

Firmware Update Guidance

To write to the EEPROM, one must use a bootloader as write times are somewhat in excess of normal access times and the data lines contain write status information whilst a write is being attempted. The CPU would attempt to execute this status information as instructions if code was being executed from the ROM at this time!!

Inadvertent writes will not cause a crash when write protect is on.

Chapter 10

The CPU Card (V2)

Central to the Tower of Eightness is its processor. The WDC65C22-TPG14 processor is a CMOS microprocessor with an eight-bit wide data bus and a sixteen-bit wide address bus. It executes single byte opcodes at typically two to three processor cycles per instruction and does this at up to 14MHz. The Tower bus being a backplane for various unknown addons has required the CPU to be limited to 4MHz and so the on-board oscillator produces 4, 2 or 1MHz as selected by a set of jumpers.

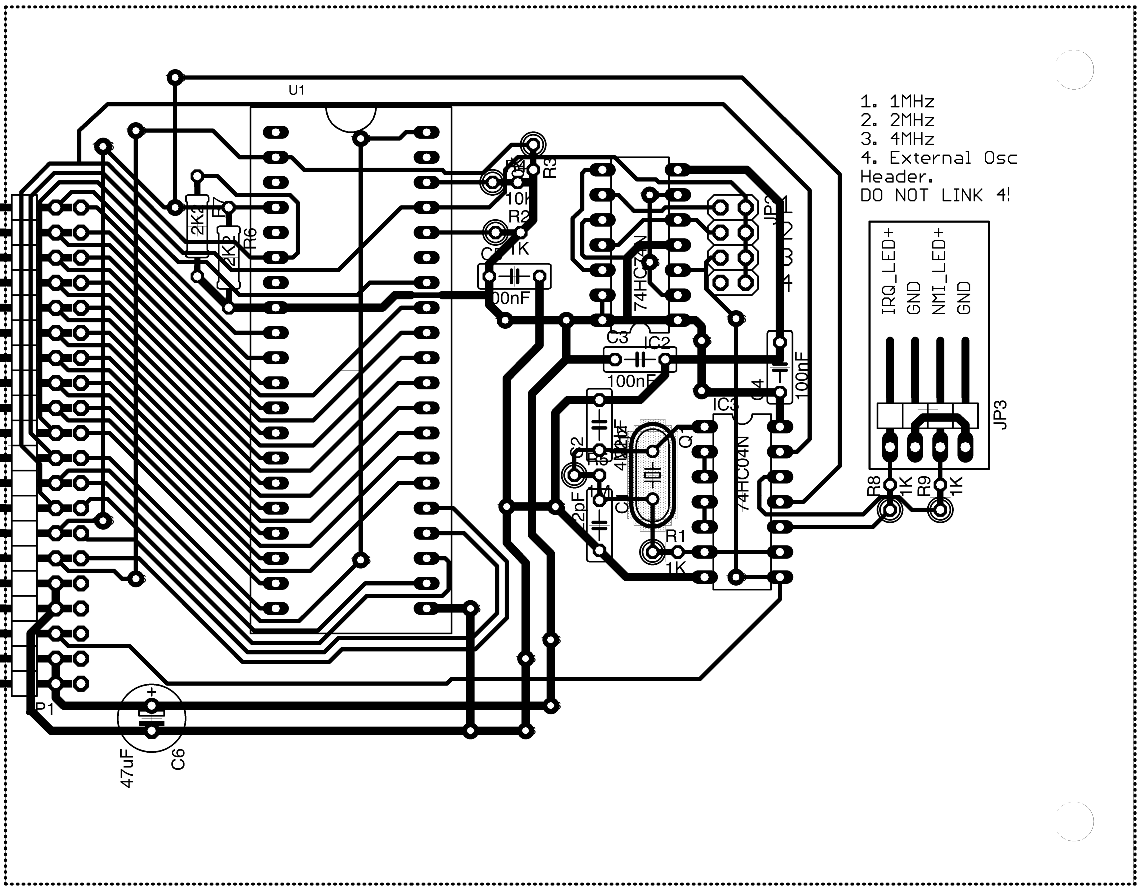


Figure 3

As can be seen from Figure 3, the board has four positions on a header labelled 1 through 4 alongside some text informing one of the available speeds. Connecting across jumper position 1 will give 1MHz operation, 2 will give 2MHz, and 3 will give 4MHz from the divider circuit. Position four is for connecting an external clock to the system bus and *must* not be shorted or the clock line will be tied to ground!

The recommended position is as fast as your hardware will allow. Most of the time this is 4MHz. There is also a header which provides LED output for the status of the IRQ and NMI lines to assist the programmer with interrupts. For the assembly language programmer or the hardware developer, the WDC65C02 datasheet is a must read and is included. For the EhBASIC user, it is usually enough to know how fast this processor is running.

Chapter 11

Tower Peripheral Bus & Centronics Interface

General Description

This card provides both multi-drop serial communications (Tower Peripheral Bus) and Centronics printer support. The TPB bus in particular, has very complex behaviour and many function calls. This bus is a work-in-progress at present, but the hardware is functional.

The Centronics Port.

This is modelled after the BBC micro implementation and may serve to output any characters as the user wishes via the OUTP_V, or more directly via either the vectored system call TPB_LPT_write_vec ($FF99) or by direct hardware access, though this is discouraged. To use TPB_LPT_write_vec, place the character to be written into the accumulator and call TPB_LPT_write_vec. This presently a blocking call and if the printer hangs the Centronics bus in a way you can't clear printer-side, you will have to perform a warm start.

To direct your output stream to also go to the LPT port, one usually sets os_outsel ($5E0) bit 2 to 1, everything then also going to the Centronics port. Clearing bit 2 stops this.

Chapter 12

The IRQ Handler

Sub-system.

General Description

Interrupts make it easier to handle outside events such as incoming serial data, timing, and software exceptions. The ToE's processor furnishes only rudimentary interrupt handling, and this sub-system is provided to flexibly manage multiple interrupts. The way this is achieved is by use of a table of eight vectors processed one by one, using a bitfield to select which one's are active. Initially, the table is populated with a null handler such that if one of its entries is inadvertently enabled, it is handled gracefully, rather than crashing the system.

The means by which one sets or clears an interrupt vector are by the provision of atomic calls which retain the interrupt handling state as much as possible so that other interrupts are delayed as little as possible.

IRQ System Call Vectors

$FFD8	IRQH_Handler_Init_vec
$FFDE	IRQH_SetIRQ_vec
$FFE1	IRQH_ClrIRQ_vec
$FFE4	IRQH_SystemReport_vec

$FFD8 IRQH_Handler_Init_vec

Calling this vector clears the interrupt vector table and resets the IRQ Handler sub-system. It may be that individual interrupt vectors get a reset vector table too, but this is not a given yet.

$FFDE IRQH_SetIRQ_vec

Atomically transfers an interrupt's vector from **IRQ_CallReg** to the table location specified in A. Locations available are 0 through 7 with 0 being handled first, subsequent locations being handled in sequential order. Note that this does not affect whether the interrupt vector is selected or not. It is the users' responsibility to manage this themselves.

$FFE1 IRQH_ClrIRQ_vec

Atomically transfers the null vector to the table location specified in the accumulator. Locations available are 0 through 7 with 0 being handled first, subsequent locations being handled in sequential order. Note that this does not affect whether the interrupt vector is selected or not. It is the users' responsibility to manage this themselves.

$FFE4 IRQH_SystemReport_vec

Calling this returns the base address of the IRQ handler table, including all variables. This is done so that the table location does not need to be known in advance. Given that this data structure is very likely to change, this will prevent breakage of code written for the ToE.

Upon return, X will contain **IRQH_Table_Base** low byte, Y will contain its high byte, and A will contain the IRQ handler version.

IRQH_Table_Base Structure

Offset	Name	Purpose
$0-$15	IRQH_CallList	Eight consecutive little-endian call addresses for the users' IRQ device handlers.
$16-$17	IRQH_CallReg	Intermediate register for atomically transferring addresses to the call list above.
$18	IRQH_ClaimsList	Bitfield showing which interrupts claimed and thus serviced an interrupt.
$19	IRQH_MaskByte	Bitfield for selecting which interrupts are to be serviced in the event of the IRQ vector being invoked.
$20	IRQH_WorkingMask	Internal variable, this walks from the LSb to the MSb and is used for various parts of the process. Do not write to this variable.
$21-$31	IRQH_CMD_Table	Table of IRQ Commands and parameter values.

Command Codes

Each IRQ has an associated parameter and command code byte entry in the IRQH_CMD_Table. These are sorted as follows. First is the parameter, followed by the command code.

Each IRQ must check on entry and before claiming IRQ for the following commands:

0.	IRQH_Service_CMD	Instructs the IRQ that service is required.
1.	IRQH_Shutdown_CMD	The IRQ must perform and orderly shutdown.
2.	IRQH_Reset_CMD	The IRQ must reset to an initial state.

Further commands may be implemented by the user but values below 8 are presently reserved for future upgrades.

Starting an IRQ Service.

Starting and IRQ consists of writing it's address to the IRQH_CallReg ($A30-$A31), Loading the accumulator with our chosen IRQH_CallList location (0-7), calling IRQH_SetIRQ_vec to load it atomically and then calling our IRQ's initialisation routine.

The IRQ initialisation routine should handle starting of the IRQ as necessary. To return the IRQ vector to the system, there is a separate call IRQH_ClrIRQ_vec ($FFE1).

Servicing an IRQ.

IRQ's running through the IRQ Handler must check upon entry, their associated command entry, pointed to by IRQH_CMD_Table + the X index register. The IRQ must process at least the minimum system command codes listed above. Parameters for the minimum set are not required.

Each time the IRQ is called, it must check that the hardware associated with it has generated an interrupt, set the appropriate claim bit in IRQH_ClaimsList ($A32) by ORing IRQH_WorkingMask ($A34) into it and clear the hardware IRQ signal so as not to cause nuisance IRQ calls and system hangs. IRQ's not generated by the associated hardware must not cause undue resource wastage or set the claim bit.

Exit from the IRQ routine is by RTS not RTI. This is so that other IRQ routine's may check their associated hardware and commands before the handler hands control back to the system.

Stopping an IRQ.

Sending the IRQH_Shutdown_CMD (1) to the IRQ will cause the IRQ to stop. This does not remove it from the IRQH_CallList, but does allow the IRQ the chance to stop in an appropriate manner. Alternatively, an atomic call may be created to do the same.

A stopped IRQ may be re-started at any time by either calling its initialisation routine or if appropriate, setting its associated bit in IRQH_MaskByte ($A33).

Removing an IRQ Service from the IRQH_CallList ($A20-$A30).

One must have first stopped the IRQ, then the accumulator must be loaded with the correct table entry. This is followed by calling IRQH_ClrIRQ_vec, which handles this atomically. After clearing, the table contains a safe dummy entry that points to a function that does nothing but ensure the IRQH_ClaimsList is correct before returning control to the IRQ handler.

The Countdown

IRQ Handler

Sub-system.

General Description.

Provides a countdown timer with programmable count rate. A GPIO card or other hardware containing a 6522 must be installed with its 6522 base address at $C020 for this to work as it relies on the 6522's Timer 1 to generate a regular stream of interrupts. This 6522 is chosen as it provides cassette storage signals and is normally expected to be present at this base address.

The IRQ is initialised at start-up in prime position at IRQH_CallList ($A20) position 0.

The default count interval is set at 39999 PHI2 ticks/count giving 10mS tick at 4MHz. It is the users' responsibility to ensure an appropriate reload value is set for proper operation.

There is only one vector associated with the countdown timer, INIT_COUNTDOWN_IRQ_vec ($FFE7). Calling this with its IRQ mask bit in the accumulator (1 unless the user moves the IRQ to a less prime location) initialises the countdown timer, which will update the countdown variable each count until it reaches 0, after which it shuts down.

It is possible to alter the IRQ reload value mid countdown, but one must either set the interrupt mask bit update.

System Variables for the Countdown IRQ.

($A46-$A47)	CTR_V	Counts down from its initial value until zero is reached.
($A48-$A50)	CTR_LOAD_VAL_V	The T1 reload value used upon initialisation. Changing this alters the IRQ rate. Caution is advised that setting this at or close to zero will cause the system to become unresponsive.