

How To Program The Z80 Periphery Document Version 1.0

Date: 2011-03-01

Contents

1 The Z80 SIO.....	4
1.1 Terminal Mode.....	4
1.1.1 Desired Communication Mechanism.....	4
1.1.2 SIO Device Structure and external wiring.....	5
1.1.2.1 Wiring.....	5
1.1.3 Programming.....	7
1.1.3.1 Header.....	8
1.1.3.2 Interrupt Vector Table.....	8
1.1.3.3 Initializing the SIO.....	9
1.1.3.4 Initializing the CTC.....	10
1.1.3.5 Initializing the CPU.....	10
1.1.3.6 Hardware Flow Control.....	11
1.1.3.7 Disabling SIO RX-channel.....	11
1.1.3.8 Interrupt Service Routines.....	12
1.1.3.9 Transmission of a character to the host.....	14
1.2 File Transfer Mode.....	15
1.2.1 Desired Communication Mechanism.....	15
1.2.2 Programming.....	16
1.2.2.1 Header.....	16
1.2.2.2 Interrupt Vector Table.....	16
1.2.2.3 Initializing the CTC.....	17
1.2.2.4 Initializing the CPU.....	17
1.2.2.5 X-Modem File Transfer.....	17
1.2.2.5.1 Host triggered transfer and setup.....	17
1.2.2.5.2 Subroutines.....	21
2 The Z80-CTC.....	22
2.1 Desired Mechanism.....	22
2.2 CTC Device Structure and external wiring.....	22
2.2.1 Wiring.....	22
2.3 Programming.....	24

2.3.1 Header.....	24
2.3.2 Interrupt table.....	24
2.3.3 Initializing the CTC.....	25
2.3.4 Initializing the CPU.....	26
2.3.5 Interrupt routine.....	26
3 The Z80 PIO.....	27
3.1 What do we need for the I ² C protocol ?.....	27
3.2 The Open-Drain-Problem.....	28
3.3 Wiring.....	29
3.4 Programming.....	29
3.4.1 Header.....	29
3.4.2 Initializing the PIO.....	30
3.4.3 Main Routines.....	31
3.4.3.1 Bus Reset.....	31
3.4.3.2 Bus Start and Stop.....	31
3.4.3.3 Sending.....	32
3.4.3.4 Receiving.....	33
3.4.4 Subroutines.....	34
3.4.4.1 SCL Cycle.....	34
3.4.4.2 Set SDA as input or output.....	35
3.4.4.3 Set SCL as output or input.....	36
3.4.4.4 Send a byte.....	37
4 Z80 IC equivalents table.....	38
5 Useful Links.....	39
6 Further Reading.....	39
7 Disclaimer.....	39

Preface

This document aims to make the Z80 processor system popular again since a lot of valuable literature and expertise has vanished from the public because of more sophisticated processor architectures of the present.

The official ZiLOG-datasheets give a good overall view of all the features of the peripheral devices but lack a tutorial like approach and programming examples in assembly language.

Remarkably ZiLOG still produces the ICs of the Z80 family – since the late seventies !

There is no special focus on hardware issues like device selection, pin characteristics or ratings. Please refer to the official ZiLOG datasheets at www.zilog.com or www.z80.info .

Special thanks go to ZiLOG for their datasheets I used for graphical illustrations within this document.

The code examples shown here provide by far not the best performance and robustness. Therefore I appreciate every hint or critics to improve the quality of this document.

1 The Z80 SIO

The Z80 SIO is the most powerful I/O device of the Z80 product family. Part one of this section describes how to program the SIO so that it communicates with a PC in asynchronous terminal mode whereas part two focuses on the block transfer mode used for file transmission. Also slightly touched in this document is the CTC programming and implementation of an interrupt mechanism.

1.1 Terminal Mode

1.1.1 Desired Communication Mechanism

We want to program the SIO for asynchronous RS232 terminal mode with these parameters:

Baudrate: 9600 Baud/sec

Stopbits: 1

Startbits: 1

Character length: 8 bit

Parity: none

In terminal mode the host computer (in our case the PC with any terminal program like *Minicom* or *Hyper Terminal* and an RS232 interface) communicates with the client (the Z80-SIO) **character based** via a so called Null-Modem-Cable. The host transmits one or more characters to the client, whereupon the client echoes this character back to the host and processes it. The host displays the echoed character on its screen. If the host does not "hear" the echoed character the communication is faulty.

Special attention is to be paid to the flow control scheme which is hardware based. In general this is called "RTSCTS" or just "hardware flow control". This method allows transmission of all 8-bit-characters (so called binary mode) and prevents overrunning of one of the peers in case one of them is too slow. In this document I assume the host PC is much faster than the client.

The wiring of the null modem cable used here has following connections between its female 9 pin D-Sub connectors:

1 – 4 / 4-1	cross wired DTR and DCD
2 – 3 / 3 – 2	cross wired TxD and RxD
5 – 5	signal ground (GND)
7 – 8 / 8 – 7	cross wired RTC and CTS
9 – 9	ring indicator (RI, not used here)

1.1.2 SIO Device Structure and external wiring

Figure 1 shows the block diagram of the device with the blocks and signals needed for our example outlined in red. Figure 2 shows the data paths within the SIO. Marked in red are the blocks we need for asynchronous mode.

1.1.2.1 Wiring

Data and control:	These are the Z80 bus signals D[7:0], A[1:0], /RD, /IOREQ, /RESET, /CE and CLK.
Interrupt Control Lines:	/M1, /INT connected to CPU, IEI and IEO daisy chained to other periphery
Serial Data:	TxD and RxD going towards host computer ¹
Channel Clocks:	TxCA and RxCA driven by CTC channel output TO0
Modem or other Controls:	CTS, RTS, DTR, DCD used for hardware flow control
miscellaneous:	/SYNC not used, pulled high by 10k resistor /Wait/Ready comes out of the device. It is to be connected to the WAIT-Input of the Z80-CPU. By asserting this signal the SIO tells the CPU to wait until the SIO has completed a character transfer. For this example we do not make use of this connection.

¹ Usually these signals are not connected directly to the host but via diver devices like MAX232, 1488, 1489 or similar level converters.

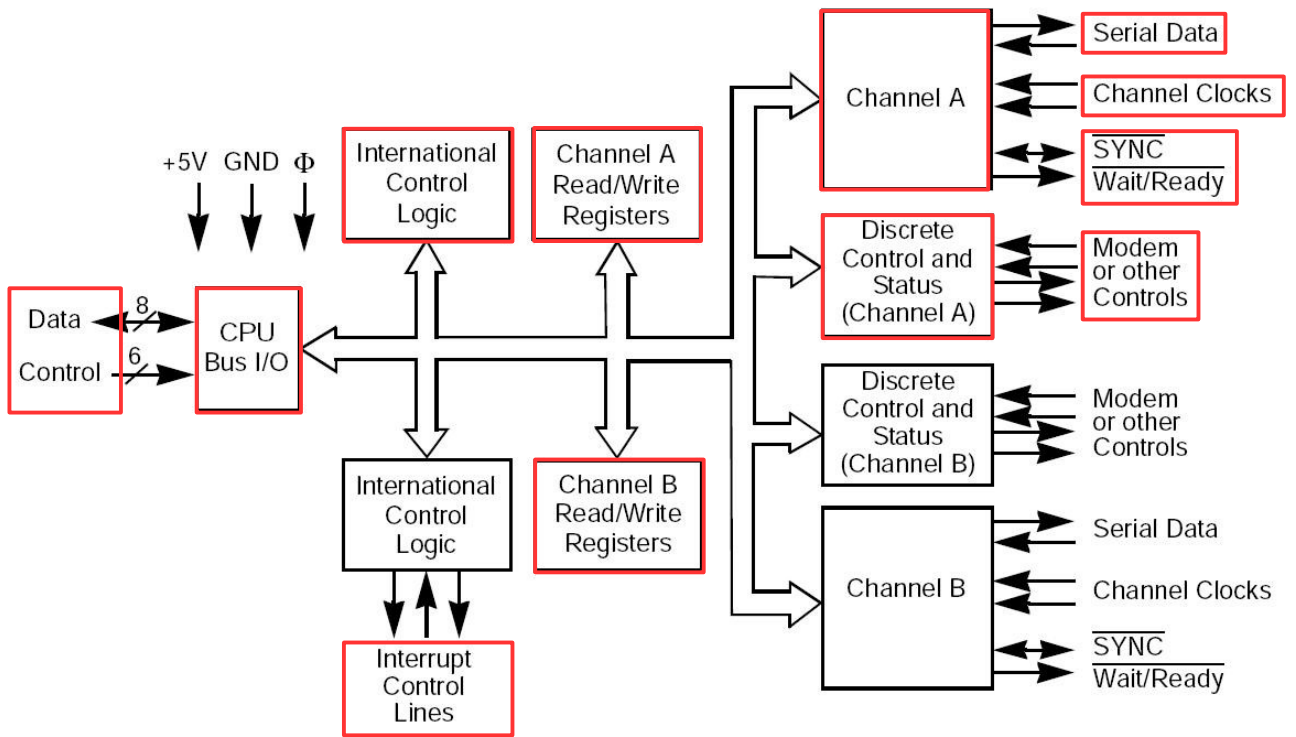


Figure 1: Block Diagram

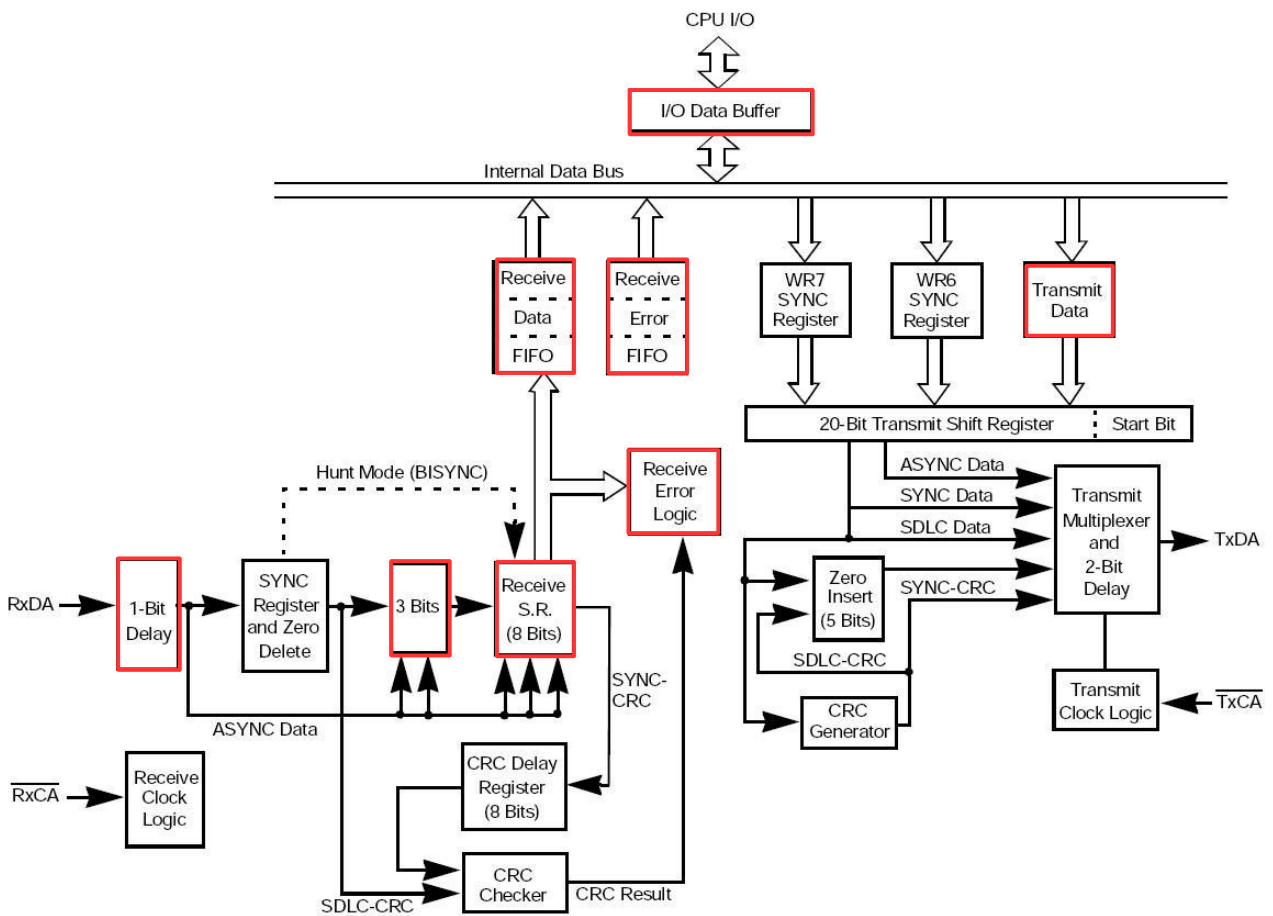


Figure 2: Data Path

1.1.3 Programming

Three problems have to be solved:

- ◆ initializing the SIO
- ◆ implementing the interrupt mechanism
- ◆ echoing the received character
- ◆ transmitting a character
- ◆ turning on/off the SIO RX-channel in certain situations

1.1.3.1 Header

The header show below defines the hardware addresses of the data and control port of **your** SIO and the address of **your** CTC channel 0. My hardware here uses the addresses 0x4, 0x6 and 0x0.

```
SIO_A_D    equ    4h
SIO_A_C    equ    6h
CH0        equ    0h
```

Text 1: header

1.1.3.2 Interrupt Vector Table

Every time the SIO receives a character it requests an interrupt causing the CPU to jump to the memory address specified by the term RX_CHA_AVAILABLE. Special receive conditions like receiver buffer overrun cause a jump to location SPEC_RX_CONDITION.

```
INT_VEC:
    org          0Ch
    DEFW        RX_CHA_AVAILABLE
    org          0Eh
    DEFW        SPEC_RX_CONDITION
```

Text 2: SIO interrupt vector table

1.1.3.3 Initializing the SIO

First we have to configure the SIO using the sequence shown in Text 3. For detailed information on the purpose of certain registers and control bits please read the SIO datasheet. We operate the SIO in interrupt mode *“interrupt on all received characters”*.

```
SIO_A_RESET:
    ;set up TX and RX:
    ld    a,00110000b    ;write into WR0: error reset, select WR0
    out   (SIO_A_C),A

    ld    a,018h        ;write into WR0: channel reset
    out   (SIO_A_C),A

    ld    a,004h        ;write into WR0: select WR4
    out   (SIO_A_C),A
    ld    a,44h         ;44h write into WR4: clkx16,1 stop bit, no parity
    out   (SIO_A_C),A

    ld    a,005h        ;write into WR0: select WR5
    out   (SIO_A_C),A
    ld    a,0E8h        ;DTR active, TX 8bit, BREAK off, TX on, RTS inactive
    out   (SIO_A_C),A

    ld    a,01h         ;write into WR0: select WR1
    out   (SIO_B_C),A
    ld    a,00000100b   ;no interrupt in CH B, special RX condition affects vect
    out   (SIO_B_C),A

    ld    a,02h         ;write into WR0: select WR2
    out   (SIO_B_C),A
    ld    a,0h          ;write into WR2: cmd line int vect (see int vec table)
                                ;bits D3,D2,D1 are changed according to RX condition
    out   (SIO_B_C),A

    ld    a,01h         ;write into WR0: select WR1
    out   (SIO_A_C),A
    ld    a,00011000b   ;interrupt on all RX characters, parity is not a spec RX condition
                                ;buffer overrun is a spec RX condition
    out   (SIO_A_C),A

SIO_A_EI:
    ;enable SIO channel A RX
    ld    a,003h        ;write into WR0: select WR3
    out   (SIO_A_C),A
    ld    a,0C1h        ;RX 8bit, auto enable off, RX on
    out   (SIO_A_C),A
    ;Channel A RX active
    RET
```

Text 3: configure the SIO

1.1.3.4 Initializing the CTC

The CTC channel 0 provides the receive and transmit clock for the SIO.

```
INI_CTC:

;init CH0
;CH0 provides SIO A RX/TX clock

ld    A,00000111b    ; int off, timer on, prescaler=16, don't care ext. TRG edge,
                    ; start timer on loading constant, time constant follows
                    ; sw-rst active, this is a ctrl cmd

out   (CH0),A
ld    A,2h          ; time constant defined
out   (CH0),A      ; and loaded into channel 0

                    ; TO0 outputs frequency=CLK/2/16/(time constant)/2
                    ; which results in 9600 bits per sec
```

Text 4: configuring the CTC channel 0

1.1.3.5 Initializing the CPU

The CPU is to run in interrupt mode 2. See Text 5 below. This has to be done **after** initializing SIO and CTC.

```
INT_INI:

ld    A,0
ld    I,A          ;load I reg with zero
im    2            ;set int mode 2
ei                    ;enable interrupt
```

Text 5: set up the CPU interrupt mode 2

1.1.3.6 Hardware Flow Control

In order to signal the host whether the client is ready or not to receive a character the RTS line coming out of the client (and driving towards the host) needs to be switched. As earlier said I assume the host is much faster than the client, that why I do not implement a routine to check the CTS-line coming from the host.

```
A_RTS_OFF:
    ld    a,005h        ;write into WR0: select WR5
    out   (SIO_A_C),A
    ld    a,0E8h        ;DTR active, TX 8bit, BREAK off, TX on, RTS inactive
    out   (SIO_A_C),A
    ret

A_RTS_ON:
    ld    a,005h        ;write into WR0: select WR5
    out   (SIO_A_C),A
    ld    a,0EAh        ;DTR active, TX 8bit, BREAK off, TX on, RTS active
    out   (SIO_A_C),A
    ret
```

Text 6: signaling the host go or nogo for reception

1.1.3.7 Disabling SIO RX-channel

When certain conditions arise it might be important to disable the receive channel of the SIO (see routine in Text 7).

```
SIO_A_DI:
    ;disable SIO channel A RX
    ld    a,003h        ;write into WR0: select WR3
    out   (SIO_A_C),A
    ld    a,0C0h        ;RX 8bit, auto enable off, RX off
    out   (SIO_A_C),A
    ;Channel A RX inactive
    ret
```

Text 7: Disabling the SIO

1.1.3.8 Interrupt Service Routines

Upon reception of a character the routine RX_CHA_AVAILABLE shown in Text 8 is executed. Here you get the character set by the host.

Note: In this example we backup only register AF. Depending on your application you might be required to backup more registers like HL, DE, CD, ...

Routine SPEC_RX_CONDITION is executed upon a special receive condition like buffer overrun. In my example the CPU is to jump at the warmstart location 0x0000.

RX_CHA_AVAILABLE:

```
push    AF           ;backup AF
call    A_RTS_OFF
in      A,(SIO_A_D)  ;read RX character into A
```

;examine received character:

```
cp      0Dh          ;was last RX char a CR ?
jp      z,RX_CR
cp      08h          ;was last RX char a BS ?
jp      z,RX_BS
cp      7Fh          ;was last RX char a DEL ?
jp      z,RX_BS
```

;echo any other received character back to host

```
out     (SIO_A_D),A
```

;do something useful with the received character here !

```
call    TX_EMP
call    RX_EMP       ;flush receive buffer
jp      EO_CH_AV
```

RX_CR:

;do something on carriage return reception here

```
jp      EO_CH_AV
```

RX_BS:

;do something on backspace reception here

```
jp      EO_CH_AV
```

EO_CH_AV:

```
ei      ;see comments below
call    A_RTS_ON    ;see comments below
pop     AF          ;restore AF
Reti
```

SPEC_RX_CONDITION:

```
jp      0000h
```

Text 8: character received routine

Note: The code written in red might be required if you want the CPU to be ready for another interrupt (ei) and to give the host a go for another transmission (call A_RTS_ON).

I recommend to put these two lines not here but in your main program routine that processes the characters received by the SIO. This way you process one character after another and avoid overrunning your SIO RX buffer.

Text 9 shows the routine to flush the receive buffer. This is important if the host sends more than one character upon pressing a key like ESC or cursor up/down keys. The routine of Text 8 echoes just the first received character back to the host, but by calling RX_EMP all characters following the first one get flushed into the void.

```
RX_EMP:
    ;check for RX buffer empty
    ;modifies A
    sub    a                ;clear a, write into WR0: select RR0
    out    (SIO_A_C),A
    in     A,(SIO_A_C)      ;read RRx
    bit    0,A
    ret    z                ;if any rx char left in rx buffer
    in     A,(SIO_A_D)      ;read that char
    jp     RX_EMP
```

Text 9: flushing the receive buffer

1.1.3.9 Transmission of a character to the host

In general transmitting of a character is done by the single command
out (SIO_A_D),A

as written in Text 8. To make sure the character has been sent completely the transmit buffer needs to be checked if it is empty. The general routine to achieve this is shown in Text 10.

```
TX_EMP:
    ; check for TX buffer empty
    sub    a
    ;clear a, write into WR0: select RR0
    inc    a
    ;select RR1
    out    (SIO_A_C),A
    in     A,(SIO_A_C)
    ;read RRx
    bit    0,A
    jp     z,TX_EMP
    ret
```

Text 10: transmitting a character to host

1.2 File Transfer Mode

1.2.1 Desired Communication Mechanism

We want to program the SIO for asynchronous RS232 **X-Modem** protocol with these parameters:

Baudrate: 9600 Baud/sec

Stopbits: 1

Startbits: 1

Character length: 8 bit

Parity: none

In difference to the **character** based mode described in section 1.1 (Terminal Mode) **blocks** of 128 byte size are to be transferred over the Null-Modem-Cable from the host PC to the client, the Z80-machine. I choose the X-Modem protocol due to its robustness and easy feasibility. Typical terminal programs like *HyperTerminal*, *Kermit* or *Minicom* do support the X-Modem protocol.

Of course you can also transfer a file via character based mode but the transfer will take much more time.

Regarding the device structure, Null-Modem-Cable, wiring and flow-control please refer to section 1.1.1 on page 4 and 1.1.2 on page 5.

A web link to the description of the X-Modem protocol can be found in section 5 on page 39.

Note: For this mode the connection of the CPU pin /WAIT and the SIO pin /Wait/Ready is required. Please see section 1.1.2.1 on page 5.

1.2.2 Programming

Four problems have to be solved: initializing the SIO, implementing the interrupt mechanism, requesting the host to start the X-Modem transfer and load the file to a certain RAM location.

1.2.2.1 Header

The header show below defines the hardware addresses of the data and control port of **your** SIO and the address of **your** CTC channel 0. My hardware here uses the addresses 0x4, 0x6 and 0x0. Further on there is a RAM locations defined for counting bad blocks while the file is being transferred.

```
SIO_A_D    equ    4h
SIO_A_C    equ    6h
CH0        equ    0h

temp0      equ    1015h ;holds number of
                       ;unsuccessful block transfers/block during download

Text 11: header
```

1.2.2.2 Interrupt Vector Table

Every time the SIO receives the **first** byte of a block it requests an interrupt causing the CPU to jump to the memory address specified by the term BYTE_AVAILABLE. This is the interrupt mode: **interrupt on first character**. Special receive conditions like receiver buffer overrun cause a jump to location SPEC_BYTE_COND. The latter case aborts the transfer.

```
INT_VEC:
    org    1Ch
    DEFW   BYTE_AVAILABLE
    org    1Eh
    DEFW   SPEC_BYTE_COND

Text 12: SIO interrupt vector table
```

1.2.2.3 Initializing the CTC

Please read section 1.1.3.4 on page 10.

1.2.2.4 Initializing the CPU

Please read section 1.1.3.5 on page 10.

1.2.2.5 X-Modem File Transfer

The assembly code of this module is described in the following sections. Due to its complexity I split it into parts shown in Text 13, 14 and 15 whose succession **must not** be mixed. For detailed information on the purpose of certain registers and control bits please read the SIO datasheet.

1.2.2.5.1 Host triggered transfer and setup

The host PC initiates the transfer. Using *Minicom* for example you press CTRL-A-S to get into a menu where you select the x-modem protocol and afterward into the file menu to select the file to be sent to the client. The procedure is similar with *HyperTerminal*.

After that the host waits for a NAK character sent by the client.

Now you should run the code shown below in Text 13 on your Z80-machine. This code initializes the SIO for interrupt mode *"interrupt on first received character"*.

```
;set up TX and RX:

ld    a,018h          ;write into WR0: channel reset
out   (SIO_A_C),A

ld    a,004h          ;write into WR0: select WR4
out   (SIO_A_C),A
ld    a,44h           ;44h write into WR4: clkx16,1 stop bit, no parity
out   (SIO_A_C),A

ld    a,005h          ;write into WR0: select WR5
out   (SIO_A_C),A
ld    a,0E8h          ;DTR active, TX 8bit, BREAK off, TX on, RTS inactive
out   (SIO_A_C),A

ld    a,01h           ;write into WR0: select WR1
out   (SIO_B_C),A
ld    a,00000100b     ;no interrupt in CH B, special RX condition affects vect
out   (SIO_B_C),A

ld    a,02h           ;write into WR0: select WR2
out   (SIO_B_C),A
ld    a,10h           ;write into WR2: cmd line int vect (see int vec table)
out   (SIO_B_C),A    ;bits D3,D2,D1 are changed according to RX condition
```

Text 13: setup 1

Now we do some settings for bad block counting, the first block number to expect and the RAM destination address of the file to receive from the host. See Text 14. The destination address setting is red colored. From this RAM location onwards the file is to be stored. In my example I use address 0x8000. Depending on your application you should change this value.

```
sub    A
ld     (temp0),A           ;reset bad blocks counter
ld     C,1h                ;C holds first block nr to expect
ld     HL,8000h           ;set lower destination address of file

call   SIO_A_EI
call   A_RTS_ON

call   TX_NAK              ;NAK indicates ready for transmission to host
```

Text 14: setup 2

Text 15 shows the code section that prepares the CPU for the reception of the first byte of a data block. The line colored red makes the CPU waiting for an interrupt which is caused by the SIO. The belonging interrupt service routine is shown in Text 16.

Once a block has been received, the checksum is verified and possible bad blocks counted. The same data block is transferred maximal 10 times whereupon the transfer is aborted.

REC_BLOCK:

```
    ;set block transfer mode
    ld    a,21h          ;write into WR0 cmd4 and select WR1
    out   (SIO_A_C),A
    ld    a,10101000b   ;wait active, interrupt on first RX character
    out   (SIO_A_C),A   ;buffer overrun is a spec RX condition

    ei
    call  A_RTS_ON
    halt
    call  A_RTS_OFF

    ld    a,01h         ;write into WR0: select WR1
    out   (SIO_A_C),A
    ld    a,00101000b   ;wait function inactive
    out   (SIO_A_C),A

    ;check return code of block reception (e holds return code)
    ld    a,e
    cp    0              ;block finished, no error
    jp    z,l_210
    cp    2              ;eot found
    jp    z,l_211
    cp    3              ;chk sum error
    jp    z,l_613
    ld    a,10h
    jp    l_612

l_210: call  TX_ACK      ;when no error
       inc  c           ;prepare next block to receive
       sub  A
       ld   (temp0),A   ;clear bad block counter
       jp  REC_BLOCK

l_211: call  TX_ACK      ;on eot
       ld   A,01h
       jp  l_612

l_613: call  TX_NAK      ;on chk sum error
       scf
       ccf          ;clear carry flag
       ld   DE,0080h   ;subtract 80h
       sbc  HL,DE      ;from HL, so HL is reset to block start address

       ld   A,(temp0)  ;count bad blocks in temp0
       inc  A
       ld   (temp0),A
       cp   09h
       jp  z,l_612     ;abort download after 9 attempts to transfer a block
       jp  REC_BLOCK   ;repeat block reception

l_612:
DLD_END:

    ret
```

Text 15: Receive Data Block

```

BYTE_AVAILABLE:

EXP_SOH_EOT:
    in     A,(SIO_A_D)           ;read RX byte into A
I_205:   cp     01h              ;check for SOH
        jp     z,EXP_BLK_NR
        cp     04h              ;check for EOT
        jp     nz,I_2020
        ld     e,2h
        reti

        ;await block number
EXP_BLK_NR:
    in     A,(SIO_A_D)           ;read RX byte into A
    cp     C                    ;check for match of block nr
    jp     nz,I_2020

        ;await complement of block number
    ld     A,C                  ;copy block nr to expect into A
    CPL                                ;and cpl A
    ld     E,A                  ;E holds cpl of block nr to expect

EXP_CPL_BLK_NR:
    in     A,(SIO_A_D)           ;read RX byte into A
    cp     E                    ;check for cpl of block nr
    jp     nz,I_2020

        ;await data block
    ld     D,0h                 ;start value of checksum
    ld     B,80h                ;defines block size 128byte

EXP_DATA:
    in     A,(SIO_A_D)           ;read RX byte into A
    ld     (HL),A
    add    A,D                  ;update
    ld     D,A                  ;checksum in D
    inc    HL                   ;dest address +1
    djnz  EXP_DATA             ;loop until block finished

EXP_CHK_SUM:
    in     A,(SIO_A_D)           ;read RX byte into A
;    ld     a,045h              ;for debug only
    cp     D                    ;check for checksum match
    jp     z,I_2021
    ld     e,3h
    reti

I_2020:  ld     E,1h
        RETI
I_2021:  ld     E,0h
        RETI                    ;return when block received completely

;-----Int routine on RX overflow-----

SPEC_BYTE_COND:                ;in case of RX overflow prepare abort of transfer
    ld     HL,DLD_END
    push  HL
    reti

```

Text 16: Interrupt Service Routine

1.2.2.5.2 Subroutines

Important for the X-Modem protocol is the sending of the *Acknowledge* and the *Not-Acknowledge* character to the host machine. For all other routines used in the code above please refer to sections 1.1.3.3 on page 9 and 1.1.3.6 on page 11.

```
TX_NAK:
  ld    a,15h    ;send NAK 15h to host
  out   (SIO_A_D),A
  call  TX_EMP
  RET
```

```
TX_ACK:
  ld    a,6h    ;send AK to host
  out   (SIO_A_D),A
  call  TX_EMP
  RET
```

Text 17: Acknowledge / Not-Acknowledge

2 The Z80-CTC

The Z80 CTC provides features to realize various counting and timing mechanisms within the Z80 computer system. The datasheet gives a good overall view of all the features of this device but lacks a tutorial like approach and programming examples in assembly language.

This section describes how to program the CTC and the associated interrupt structure so that a kind of heartbeat is generated. This beat can be used to make a displayless embedded computer giving a life sign every couple of seconds. Furthermore the code examples shown here can be improved to make a system clock.

2.1 Desired Mechanism

Imagine an embedded computer, based on the famous Z80 CPU, which has no display means but a single LED. Immediately after power up the board has to give a life sign by flashing the LED every t seconds as a kind of a heartbeat.

The main program running on the board shall not be affected by the heartbeat function, except it's interruption every t seconds of course.

2.2 CTC Device Structure and external wiring

Figure 3 shows the block diagram of the CTC device with its 4 timer/counter channels and the two channels we need marked red. Figure 4 shows the structure of a single channel.

Note: The output of channel 3 is not connected to any pin.

2.2.1 Wiring

Data and control:	These are the Z80 bus signals D[7:0], A[1:0] or CS[1:0], /RD, /IOREQ, /CE, /RESET and CLK.
Interrupt Control Lines:	/M1, /INT connected to CPU, IEI and IEO daisy chained to other periphery
Outputs:	zero count signals TO[2:0], TO2 is wired to TRG3
Inputs:	counter inputs TRG[3:0]

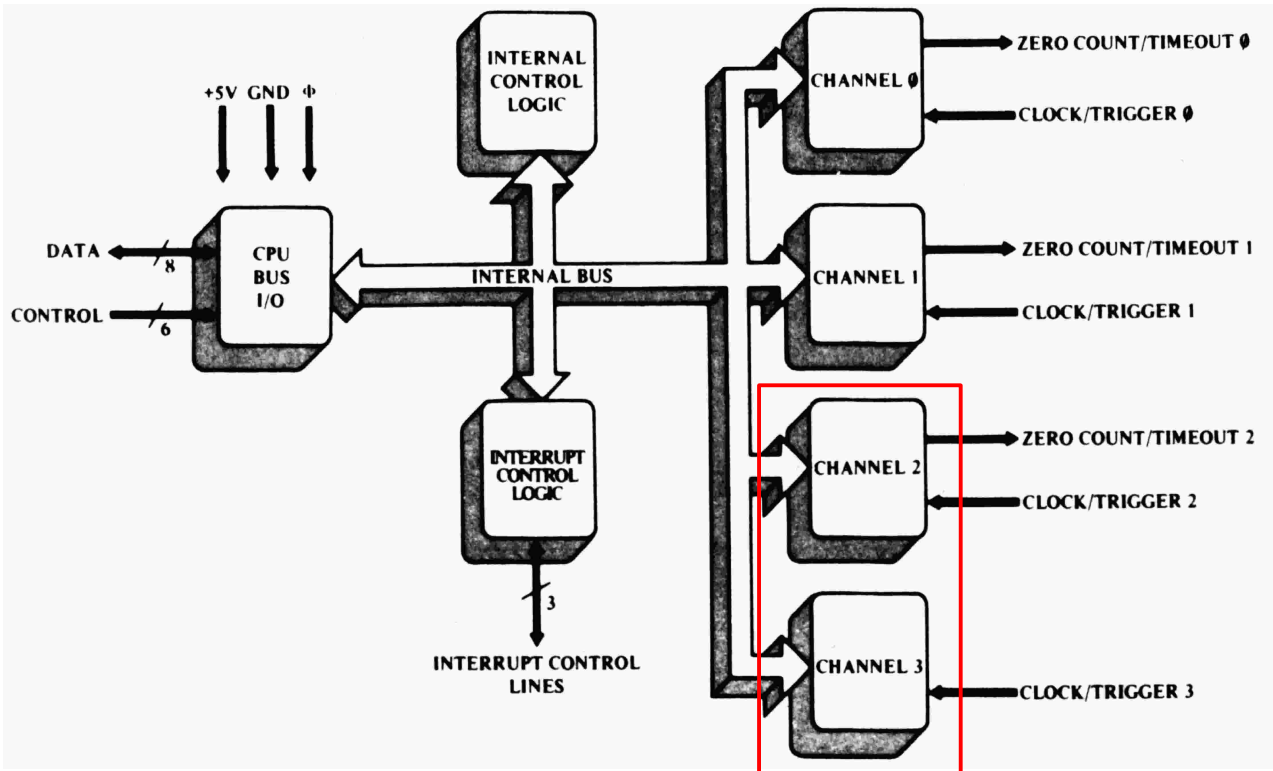


Figure 3: CTC Block Diagram

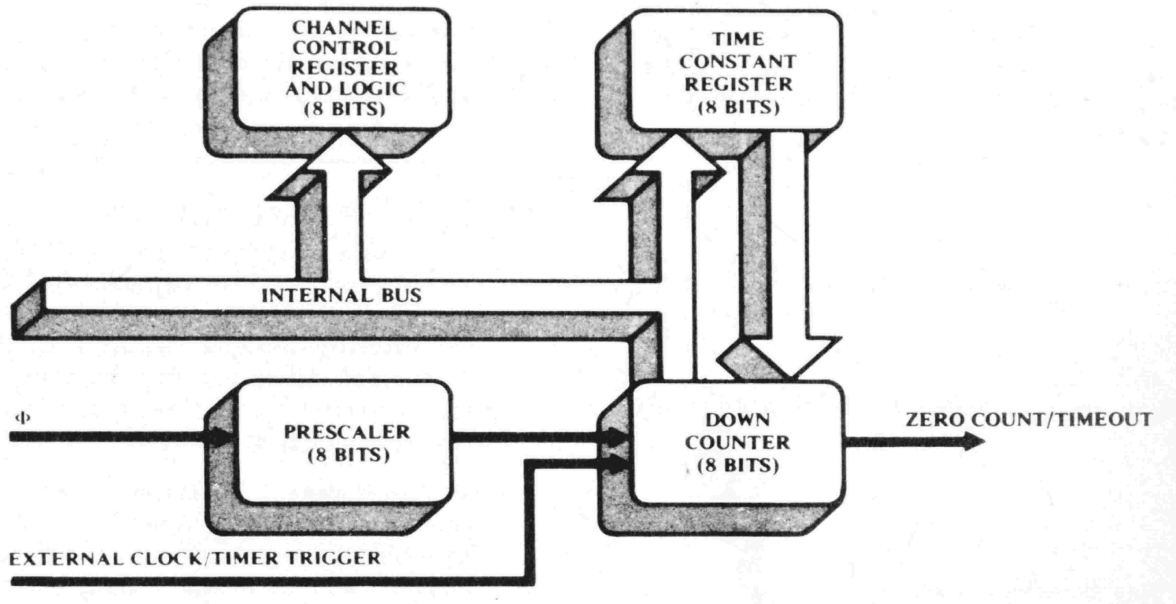


Figure 4: Channel Structure

2.3 Programming

Three problems have to be solved:

- ◆ initializing the CTC
- ◆ implementing the interrupt mechanism
- ◆ writing an interrupt service routine that handles the flashing of the LED

2.3.1 Header

The header shown below defines the hardware addresses of the control port of **your** four CTC channels 0 to 3. In my case here the addresses equal the channel number.

```
CH0 equ 0h
CH1 equ 1h
CH2 equ 2h
CH3 equ 3h
```

Text 18: header

2.3.2 Interrupt table

Every time CTC channel 3 count equals zero an interrupt is triggered causing the CPU to jump to the memory address specified by the term CT3_ZERO.

```
org 16h
DEFW CT3_ZERO
```

Text 19: CTC interrupt vector table

2.3.3 Initializing the CTC

First we have to configure the CTC channels as shown in Text 20. We don't need channel 0 and 1. They are on hold. We operate the CTC channel 2 as frequency divider which scales the CPU clock of 5 MHz down by factor 256×256 . The output TO2 of channel 2 drives input TRG3 of channel 3 which further divides by factor AFh. This causes channel 3 to zero count at a frequency of approximately 0.44Hz. So the interrupt occurs every 2.3 seconds.

For detailed information on the purpose of certain registers and control bits please read the CTC datasheet.

```
INI_CTC:
;init CH 0 and 1
ld      A,00000011b    ; int off, timer on, prescaler=16, don't care ext. TRG edge,
                    ; start timer on loading constant, no time constant follows
                    ; sw-rst active, this is a ctrl cmd
out     (CH0),A        ; CH0 is on hold now
out     (CH1),A        ; CH1 is on hold now

;init CH2
;CH2 divides CPU CLK by (256*256) providing a clock signal at TO2. TO2 is connected to TRG3.
ld      A,00100111b    ; int off, timer on, prescaler=256, no ext. start,
                    ; start upon loading time constant, time constant follows
                    ; sw reset, this is a ctrl cmd
out     (CH2),A
ld      A,0FFh         ; time constant 255d defined
out     (CH2),A        ; and loaded into channel 2
                    ; T02 outputs f= CPU_CLK/(256*256)

;init CH3
;input TRG of CH3 is supplied by clock signal from TO2
;CH3 divides TO2 clock by AFh
;CH3 interrupts CPU appr. every 2sec to service int routine CT3_ZERO (flashes LED)
ld      A,11000111b    ; int on, counter on, prescaler don't care, edge don't care,
                    ; time trigger don't care, time constant follows
                    ; sw reset, this is a ctrl cmd
out     (CH3),A
ld      A,0AFh         ; time constant AFh defined
out     (CH3),A        ; and loaded into channel 3

ld      A,10h          ; it vector defined in bit 7-3,bit 2-1 don't care, bit 0 = 0
out     (CH0),A        ; and loaded into channel 0
```

Text 20: configure the CTC

2.3.4 Initializing the CPU

The CPU is to run in interrupt mode 2. See Text 21 below. This setting has to be done **after** initializing the CTC.

```
INT_INI:
    ld    A,0
    ld    I,A    ;load I reg with zero
    im    2      ;set int mode 2
    ei                    ;enable interrupt
```

Text 21: set up the CPU interrupt mode 2

2.3.5 Interrupt routine

Upon zero count of channel 3 the routine CT3_ZERO as shown in Text 18 is executed. Here you put the code that switches the LED on and off.

Note: In this example we backup only register AF. Depending on your application you might be required to backup more registers like HL, DE, CD, ...

```
CT3_ZERO:
    ;flashes LED
    push  AF          ;backup registers A and F

    ; now address your periphery that turns the LED on/off e.g. a D-Flip-Flop

    pop   AF          ;restore registers A and F
    EI                    ;re-enable interrupts
    reti
```

Text 22: CT3 zero count routine

3 The Z80 PIO

The I²C protocol allows the communication of various devices like ADC, DAC, expanders, memories and a lot more via a 2 wire bus which saves board space to a great extent. The Z80 PIO device can be programmed so that it becomes the I²C bus master. The details of the I²C protocol can be found at <http://en.wikipedia.org/wiki/I²C>.

Within this document there is no special focus on programming of the PIO nor on hardware issues like device selection, pin characteristics or ratings. The Z80 PIO device and is well documented by the official ZiLOG datasheets at www.zilog.com or www.z80.info. Please read also the datasheets provided by respective I²C device manufacturers.

3.1 What do we need for the I²C protocol ?

The aim are some major routines written in assembly code:

- ◆ sending any byte onto the I²C bus
- ◆ receiving any byte from the bus
- ◆ resetting the bus
- ◆ starting and stopping the bus

The I²C bus is connected to PIO port B with B0 driving SCL, and B1 driving and reading SDA as shown in Figure 5.

3.3 Wiring

The general rules of the Z80 bus system apply as follows:

Data and control: These are the Z80 bus signals D[7:0], A[1:0] or CONTSEL and PORTSEL, /RD, /IOREQ, /CE and CLK.

Interrupt Control Lines: /INT connected to CPU, IEI and IEO daisy chained to other periphery

In/Outputs: A[7:0], B[7:0], /ASTRB, /BSTRB, ARDY, BRDY

Note 1: I recommend the AND-ing of the CPU Reset and the CPU M1 signal to form the PIO M1 signal. This makes the PIO starting up properly upon system reset (please see Figure 5).

Note 2: All unused pins of port A and B should be pulled up by 10k resistors to avoid them floating when programmed as inputs.

3.4 Programming

3.4.1 Header

The header shown below defines the hardware addresses of the control and data port of **your** PIO. In my case here they are 9 and Bh. Furthermore there are two RAM locations reserved to store current mode and I/O configuration.

```
PIO_B_D          equ    9h
PIO_B_C          equ    0Bh

PIO_B_MODE       equ    1005h ;holds current PIO B mode
PIO_B_IO_CONF    equ    1006h ;holds current IO configuration of PIO B
```

Text 23: header

3.4.2 Initializing the PIO

Text 24 shows the actions needed:

- ◆ setting port B in bit mode.
- ◆ setting pins B0 and B1 in input mode.
- ◆ loading the output register with FCh (a binary 11111100).

Later in the program we do the following:

Every time pin B0 or B1 is set to output mode the values zero held by the output register is passed through to the pin. This causes a hard low on the pin. If the pin B0 or B1 is set to input mode, it releases the line whereupon it is pulled high by the pull resistors R1 or R2. So we control the logical level of the SDA or SCL lines **only** by the I/O configuration register of the PIO port B.

```
INI_PIO:

    ;init PIO port B

    ld    A,0CFh          ; set PIO B to bit mode
    ld    (PIO_B_MODE),A ; update global PIO B mode status variable
    out   (PIO_B_C),A

    ld    a,0FFh         ; set D7..0 to input mode
    ld    (PIO_B_IO_CONF),A ; update global PIO B IO status variable
    out   (PIO_B_C),A    ; write IO configuration into PIO B

    ld    A,0FCh         ; if direction of B1 or B0 changes to output
                        ; the pin will drive L
    out   (PIO_B_D),A    ; load PIO B output register
```

Text 24: configure the PIO port B

3.4.3 Main Routines

3.4.3.1 Bus Reset

In order to get all slaves proper reset, the following code is recommended. SCL is clocked 10 times while SDA is held H.

```
RST_I2C:
    ;modifies A, B, D
    ;leaves SDA = H and SCL = H

    ld    B,0Ah        ; do 10 SCL cycles while SDA is H
I_77:   call   SCL_CYCLE
        djnz  I_77
        call   SCL_IN
        ret

Text 25: bus reset
```

3.4.3.2 Bus Start and Stop

The I²C bus protocol requires a certain start and stop sequence. Text 26 shows the code.

```
I2C_START:
    ;starts I2C bus

    call   SDA_OUT    ;SDA = L
    call   SCL_OUT    ;SCL = L
    ret

I2C_STOP:
    ;stops I2C bus

    call   SDA_OUT
    call   SCL_IN
    call   SDA_IN
    ret

Text 26: start and stop routines
```

3.4.3.3 Sending

The code shown in Text 23 is the routine *I2C_tx* which sends a byte onto the bus. The byte to send has to be in the accumulator (or CPU register A) prior to calling this routine. This routine first sends the sata byte (by calling *send_byte*), then checks for the acknowledge bit sent by a slave. If no acknowledge bit is found the routine leaves the carry flag set.

```
I2C_tx:

    ;byte to send provides accumulator
    ;returns with carry cleared if ackn bit not found
    ;modifies A,B,C,D,HL

    call    send_byte
    bit     1,D           ; test D register for acknowledge bit
    scf
    ret     z             ;return if akn bit = L with carry set

    ;when ACK error - stop bus
    call    I2C_STOP
    scf
    ccf
    ret                     ;return if akn bit = H with carry cleared
```

Text 27: send routine

3.4.3.4 Receiving

The routine to receive a byte from a slave is shown below. The byte received from the slave is returned in the accumulator.

```
I2C_RX:
    ;modifies A, B, D
    ;returns with slave data byte in A
    ;leaves SCL = L and SDA = H

    ld    B,8h
I_66:   in    A,(PIO_B_D)
    scf
    bit   1,A
    jp   nz,H_found
L_found: ccf
H_found: rl    C
    call SCL_CYCLE
    djnz I_66
    call SCL_CYCLE    ;send NAK to slave

    ;slave byte ready in C
    ld   A,C
    ret

Text 28: receive routine
```

3.4.4 Subroutines

The main routines described above frequently call other code which we see in the following sections. These routines are written with these primary objectives:

- ◆ memory saving
- ◆ modularity
- ◆ easy to understand (hopefully)

The execution speed is of secondary importance here.

3.4.4.1 SCL Cycle

Every bit transferred via the SDA line must be accompanied by a L-H-L sequence of the SCL line. The following routine accomplishes that. After SCL going H the SDA line is sampled³.

```
SCL_CYCLE:
    ;modifies A
    ;returns D wherein bit 1 represents status of SDA while SCL was H
    ;leaves SCL = L

    call    SCL_OUT
    call    SCL_IN

    ;look for ackn bit
    in     A,(PIO_B_D)
    ld     D,A
    call   SCL_OUT
    ret
```

Text 29: SCL cycle

³ Only the 9th sample of a byte transfer is important regarding the acknowledge bit.

3.4.4.2 Set SDA as input or output

As mentioned earlier the direction setting of B1 determines whether a H or L is driven on the line. So if you want a H on SDA run routine *SDA_IN* if you need an L run *SDA_OUT*.

```
SDA_IN:

    ;modifies A
    ;reloads PIO B mode

    ld    A,(PIO_B_MODE)
    out   (PIO_B_C),A

    ;change direction of SDA to input
    ld    A,(PIO_B_IO_CONF)
    set   1,A
    out   (PIO_B_C),A
    ld    (PIO_B_IO_CONF),A
    ret

SDA_OUT:

    ;modifies A
    ;reloads PIO B mode

    ld    A,(PIO_B_MODE)
    out   (PIO_B_C),A

    ;change direction of SDA to output
    ld    A,(PIO_B_IO_CONF)
    res   1,A
    out   (PIO_B_C),A
    ld    (PIO_B_IO_CONF),A
    ret

Text 30: set SDA as output or input
```

Note: If your SDA line is stuck at low or high for some reason, the routine shown here will **not** detect this malfunction. An immediate reading back of SDA can be implemented easily.

3.4.4.3 Set SCL as output or input

Similar to SDA the SCL line is controlled by the direction of pin B0. If you want a H on SCL run routine *SCL_IN* if you need an L run *SCL_OUT*.

```
SCL_IN:

    ;modifies A
    ;reloads PIO B mode

    ld    A,(PIO_B_MODE)
    out   (PIO_B_C),A
    ;change direction of SCL to input
    ld    A,(PIO_B_IO_CONF)
    set   0,A
    out   (PIO_B_C),A
    ld    (PIO_B_IO_CONF),A
    ret

SCL_OUT:

    ;modifies A
    ;reloads PIO B mode

    ld    A,(PIO_B_MODE)
    out   (PIO_B_C),A
    ;change direction of SCL to output
    ld    A,(PIO_B_IO_CONF)
    res   0,A
    out   (PIO_B_C),A
    ld    (PIO_B_IO_CONF),A
    ret
```

Text 31: set SCL as output or input

Note: If your SCL line is stuck at low or high for some reason, the routine shown here will **not** detect this malfunction. An immediate reading back of SCL can be implemented easily.

3.4.4.4 Send a byte

This routine performs the clocking out of the data byte.

Note: Do not confuse this routine with the one shown in section 3.4.3.3 . *Send_byte* is called by *2C_tx*.

```
send_byte:

    ;requires byte to be sent in A
    ;returns with bit 1 of D holding status of ACKN bit
    ;leaves SCL = L and SDA = H
    ;modifies A, B, C, D

    ld    B,8h          ; 8 bits are to be clocked out
    ld    C,A           ; copy to C reg
I_74:   sla    C         ; shift MSB of C into carry
        jp    c,SDA_H   ; when L
SDA_L:  call   SDA_OUT   ; pull SDA low
        jp    I_75
SDA_H:  call   SDA_IN    ; release SDA to let it go high
I_75:   call   SCL_CYCLE ; do SCL cycle (LHL)
        djnz  I_74     ; process next bit of C reg
        call  SDA_IN    ; release SDA to let it go high
        call  SCL_CYCLE ; do SCL cycle (LHL), bit 1 of D holds ackn bit
        ret
```

Text 32: send byte

4 Z80 IC equivalents table

An overview of ICs of the famous Z80 family gives Table 1.

device	equivalent type
Z80-CPU	BU18400A-PS (ROHM) D780C-1(NEC) KP1858BM1/2/3 / KR1858BM1/2/3 (USSR) LH0080 (Sharp) MK3880x (Mostek) T34VM1 / T34BM1 (USSR) TMPZ84C00AP-8 (Toshiba) UA880 / UB880 / VB880D (MME) Z0840004 (ZiLOG) Z0840006 (ZiLOG) Z80ACPUD1 (SGS-Ates) Z84C00AB6 (SGS-Thomson) Z84C00 (ZiLOG) Z8400A (Goldstar) U84C00 (MME)
Z80-SIO	UA8560 , UB8560 (MME) Z0844004 (ZiLOG) Z8440AB1 (ST) Z0844006 (ZiLOG) Z84C40 (ZiLOG) U84C40 (MME)
Z80-PIO	Z0842004/6 (ZiLOG) UA855 / UB855 (MME) Z84C20 (ZiLOG) U84C20 (MME)
Z80-CTC	Z84C30 (ZiLOG) U84C30 (MME) UA857 / UB857 (MME)

Table 1: Z80 equivalents

5 Useful Links

- ◆ Debug your hardware with the *Logic Scanner* at http://www.train-z.de/logic_scanner
- ◆ A complete embedded Z80 system can be found at <http://www.train-z.de/train-z>
- ◆ The Free and Open Productivity Suite OpenOffice at <http://www.openoffice.org>
- ◆ Z80 assembler for Linux and UNIX at <http://www.unix4fun.org/z80pack/>
- ◆ The powerful communication tool Kermit at <http://www.columbia.edu/kermit/>
- ◆ Z80 Verilog and VHDL Cores at <http://www.cast-inc.com> and <http://opencores.org>
- ◆ The X-Modem Protocol Reference by Chuck Forsberg at <http://www.train-z.de/train-z/pdf/xymodem.pdf>
- ◆ EAGLE - an affordable and very efficient schematics and layout tool at <http://www.cadsoftusa.com/>



6 Further Reading

I recommend to read these books:

“Using C-Kermit” / Frank da Cruz, Christine M. Gianone / ISBN 1-55558-108-0 (english)

“C-Kermit : Einführung und Referenz” / Frank da Cruz, Christine M. Gianone / ISBN 3-88229-023-4 (german)

7 Disclaimer

This tutorial is believed to be accurate and reliable. I do not assume responsibility for any errors which may appear in this document. I reserve the right to change it at any time without notice, and do not make any commitment to update the information contained herein.

Copyright 2011 for all sheets of this document except the graphics taken from ZiLOG datasheets.

My Boss is a Jewish Carpenter