How To Program the

Z80 Periphery Tutorial

Doc. Version 2013-08-23-1
Author: Mario Blunk

Abstract: Guideline to program Z80 family devices in a tutorial like manner. Assembly program code examples given.

Keywords: CPU, SIO, CTC, PIO, counter, timer, I²C, serial, terminal, null-modem, flow control, baud rate, vector table, interrupt service routine, mainline program, loop, instruction, execution time, mathematics, pulse width, jitter, oscilloscope, PWM, modulation, equivalents table, minicom, hyper terminal, time constant
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 Programming Pulse Width Modulation (PWM)..........................................38
  4.1 The Program Algorithm..........................................................................39
  4.2 The Math......................................................................................................42
  4.3 The Z80 Assembly Code..........................................................................44
  4.4 Results..........................................................................................................44
  4.5 Integrating Circuitry..................................................................................47
5 Z80 IC equivalents table..............................................................................48
6 References.......................................................................................................49
7 Useful Links.....................................................................................................50
8 Further Reading...............................................................................................53
9 Disclaimer........................................................................................................53
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.

This document does not aim to bring back “good old times”.

The official ZiLOG-datasheets [1] and [2] 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 to 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 (taken from [2]) shows the block diagram of the device with the blocks and signals needed for our example outlined in red. Figure 2 (taken from [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

Channel Clocks: TxC 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.

Blank electronic at www.train-z.de
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 shown 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”.

<table>
<thead>
<tr>
<th>SIO_A_RESET:</th>
<th>;set up TX and RX:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld a,00110000b</td>
<td>;write into WR0: error reset, select WR0</td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,018h</td>
<td>;write into WR0: channel reset</td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,004h</td>
<td>;write into WR0: select WR4</td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,44h</td>
<td>;44h write into WR4: clkx16, 1 stop bit, no parity</td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,005h</td>
<td>;write into WR0: select WR5</td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,0E8h</td>
<td>;DTR active, TX 8bit, BREAK off, TX on, RTS inactive</td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,01h</td>
<td>;write into WR0: select WR1</td>
</tr>
<tr>
<td>out (SIO_B_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,00000100b</td>
<td>;no interrupt in CH B, special RX condition affects vect</td>
</tr>
<tr>
<td>out (SIO_B_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,02h</td>
<td>;write into WR0: select WR2</td>
</tr>
<tr>
<td>out (SIO_B_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,0h</td>
<td>;write into WR2: cmd line int vect (see int vec table)</td>
</tr>
<tr>
<td>;bits D3,D2,D1 are changed according to RX condition</td>
<td></td>
</tr>
<tr>
<td>out (SIO_B_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,01h</td>
<td>;write into WR0: select WR1</td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,00011000b</td>
<td>;interrupt on all RX characters, parity is not a spec RX condition</td>
</tr>
<tr>
<td>;buffer overrun is a spec RX condition</td>
<td></td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIO_A_EI:</th>
<th>;enable SIO channel A RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld a,003h</td>
<td>;write into WR0: select WR3</td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
<tr>
<td>ld a,0C1h</td>
<td>;RX 8bit, auto enable off, RX on</td>
</tr>
<tr>
<td>out (SIO_A_C),A</td>
<td></td>
</tr>
<tr>
<td>;Channel A RX active</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td></td>
</tr>
</tbody>
</table>

**Text 3: configure the SIO**

*Blunk electronic at www.train-z.de*
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,0E4h    ;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 by 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         ; select RR1
  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 5 and 1.1.2 on page 6.

A web link to the description of the X-Modem protocol can be found in section 7 on page 50.

**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 6.**
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.

```assembly
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.

<table>
<thead>
<tr>
<th>sub</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>(temp0),A</td>
</tr>
<tr>
<td>ld</td>
<td>C,1h</td>
</tr>
<tr>
<td>ld</td>
<td>HL,8000h</td>
</tr>
<tr>
<td>call</td>
<td>SIO_A_EI</td>
</tr>
<tr>
<td>call</td>
<td>A_RTS_ON</td>
</tr>
<tr>
<td>call</td>
<td>TX_NAK</td>
</tr>
</tbody>
</table>

**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 ; await first rx char
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
out (SIO_A_C),A

ld a,10h
out (SIO_A_C),A

ld a,(temp0)
incc
jp REC_BLOCK ; clear bad block counter

l_210: call TX_ACK ; when no error
inc c ; prepare next block to receive
sub A
ld (temp0),A
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
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:
  l_205:
    in  A,(SIO_A_D) ;read RX byte into A
    cp  01h ;check for SOH
    jp  z,EXP_BLK_NR
    cp  04h ;check for EOT
    jp  nz,l_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,l_2020

    ;await complement of block number
  ld  A,C ;copy block nr to expect into A
  CPL
  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,l_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 ;update
  add  A,D ;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,l_2021
  ld  e,3h
  reti

l_2020:
  ld  E,1h
  RETI

l_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 display-less 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 (taken from [2]) shows the block diagram of the CTC device with its 4 timer/counter channels and the two channels we need marked red. Figure 4 (taken from [2]) 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

Tree 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.

```
INICTC:

;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
            ; TO2 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 interrups
  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.
3.2 The Open-Drain-Problem

A typical I²C master has open-drain pins. The Z80 PIO does not have open-drain outputs on its ports A and B. Instead they are of push-pull characteristic which requires two additional series resistors R3 and R4 as shown in Figure 5. R1 and R2 are mandatory for an I²C master².

R3 and R4 serve as overload protection for the PIO and the slaves in case both the master and one of the slaves drive onto the SCL or SDA net. The values of 10k for R1/R2 and 160 Ohms for R3/R4 are only rough estimations and should be modified according to your application.

² Some I²C masters may have these resistors built in.

Figure 5: external resistors

Blunk electronic at www.train-z.de
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.

```
INIPIO: ;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

L_77:
    Id B,0Ah ; do 10 SCL cycles while SDA is H
    call SCL_CYCLE
    djnz L_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 \texttt{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 \texttt{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.

\begin{verbatim}
  \textbf{l2C\_tx:}
  \begin{Verbatim}
  ;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
  \end{Verbatim}
\end{verbatim}

\textit{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
  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 sampled.³

```
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`.

<table>
<thead>
<tr>
<th>SDA_IN:</th>
</tr>
</thead>
<tbody>
<tr>
<td>;modifies A</td>
</tr>
<tr>
<td>;reloads PIO B mode</td>
</tr>
<tr>
<td>ld A,(PIO_B_MODE)</td>
</tr>
<tr>
<td>out (PIO_B_C),A</td>
</tr>
<tr>
<td>;change direction of SDA to input</td>
</tr>
<tr>
<td>ld A,(PIO_B_IO_CONF)</td>
</tr>
<tr>
<td>set 1,A</td>
</tr>
<tr>
<td>out (PIO_B_C),A</td>
</tr>
<tr>
<td>ld (PIO_B_IO_CONF),A</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SDA_OUT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>;modifies A</td>
</tr>
<tr>
<td>;reloads PIO B mode</td>
</tr>
<tr>
<td>ld A,(PIO_B_MODE)</td>
</tr>
<tr>
<td>out (PIO_B_C),A</td>
</tr>
<tr>
<td>;change direction of SDA to output</td>
</tr>
<tr>
<td>ld A,(PIO_B_IO_CONF)</td>
</tr>
<tr>
<td>res 1,A</td>
</tr>
<tr>
<td>out (PIO_B_C),A</td>
</tr>
<tr>
<td>ld (PIO_B_IO_CONF),A</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>

**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*.

```assembly
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
1_74:
   sla C ; shift MSB of C into carry
   jp c, SDA_H ; when L
SDA_L: ; pull SDA low
   call SDA_OUT
   call SDA_IN
1_75:
   call SDA_IN ; release SDA to let it go high
   call SCL_CYCLE ; do SCL cycle (LHL)
   djnz 1_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 Programming Pulse Width Modulation (PWM)

Pulse width modulation (PWM) if based on a digital hardware in general can be regarded as a Digital-Analog-Converter (DAC): A digital value, lets say a byte, is converted into a PWM signal where the pulse width represents the byte being input.

Figure 6 depicts two examples, one for the value 32d/20h (dashed line) and another for value 224d/E0h (continuous line). The signal period ends (at T) where the sample count reaches 256d (or 100h), whereupon the cycle starts all over again.

With an integrator circuitry this signal can easily be “smoothed” to a real analog voltage or current (see section 4.5 on page 47).

The following discussion is more of theoretical nature with limited practical benefits as there are smarter solutions available today to generate a PWM signal i.e. CPLDs programmed in Verilog HDL. The basic questions of this discussion are:

- Is it possible to generate a PWM entirely CPU based?
- Which accuracy can be achieved?

The CPU used here is of course the famous Z80 processor together with its peripheral CTC and PIO units.
4.1 The Program Algorithm

As a program algorithm describes a procedure on a high level, it can be applied to every hardware platform and every programming language. Here some examples to outline the range:

- Z80 CPU programmed in assembly
- mid range processors/microcontrollers and programming languages
- TSC695F SPARC processor programmed in Ada

Table 1 lists the variables used for a CPU driven PWM. Figure 7 shows what to do to initialize, Figure 8 shows the essential interrupt service routine (ISR).

But first the init sequence:

1) Upon CPU reset the peripherals and the CPU interrupt mode need to be initialized (yellow box).

2) One pin of the PIO or GPIOs must be set as output.

3) The CTC must be set to request an interrupt at a constant rate every $x$ milli- or nanoseconds. Each requests represents a time slot wherein a sample counter gets incremented (see Figure 6 on page 38).

4) Then the values belonging to the PWM need to be assigned with a init value. So we have a safe starting point when the interrupt gets enabled (green box).

5) Usually the mainline program loop is to be entered afterward.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>dac_ch0</td>
<td>value to be converted into an analog pulse width</td>
<td>integer (i.e. 8 bit)</td>
</tr>
<tr>
<td>sample_counter</td>
<td>counts the samples of a period</td>
<td>modulus number, overflow indicates a new period</td>
</tr>
<tr>
<td>buffer</td>
<td>output buffer</td>
<td>latches the output signal between updates</td>
</tr>
<tr>
<td>out_port</td>
<td>output port</td>
<td>usually a real hardware output like GPIO or PIO</td>
</tr>
</tbody>
</table>

Table 1: variables used for a PWM

Every time, the counter of the CTC unit reaches zero count, the ISR shown in Figure 8 gets executed. Usually the CTC signals this event by asserting an interrupt request signal, whereupon the CPU starts executing the ISR.
Important to point out:

1. The CTC must have the highest interrupt priority as possible.\(^4\)
2. The ISR must finish before the next zero count of the CTC.

Figure 8:

1. The ISR execution time is not constant, as it contains branches depending on the contents of some values being tested. So it is reasonable to update `out_port` right at the beginning of the ISR (yellow box). The `out_port` signal is updated in real time.
2. Now if the `samples_counter` equals the `dac_ch0` variable (red diamond) a bit \(x\) is set in the buffer. If there is no match, the buffer bit is untouched. The `samples_counter` starts with zero, so the greater `dac_ch0` is, the later the buffer bit will be set.
3. Next the `samples_counter` is incremented (gray box). If the `samples_counter` overflows, means all its bits flip back to zero, the buffer bit gets cleared. This is the moment where the PWM signal period ends. Otherwise the buffer bit is left as it is.
4. Return from interrupt (RETI).
5. The next time the ISR gets executed, the `out_port` will be updated by the most recent state of the buffer bit. See action point 1.

\(^4\) A lower priority is possible, but decreases the output accuracy significantly.
**Figure 7: general init sequence**

INIT PIO  
INIT CTC  
INIT Interrupts

INIT VARIABLES:
- dac_ch0 = 0
- sample_counter = 0
- buffer = 0
- out_port = 0

ENABLE INTERRUPTS

Main Line Program

**Figure 8: general ISR**

out_port := buffer

sample_counter = dac_ch0 ?

set bit x in buffer

samples_counter + 1

sample_counter = 0 ?

clear bit x in buffer

return from interrupt
4.2 The Math

Life without mathematics is fuzzy and boring.

The most important constraints of the PWM output signal are:

1. Frequency \( f_o \) and Period \( T_o \)
2. Resolution \( R \)
3. Stability/Accuracy

In this example \( f_o \) is given with 50 Hz. \( R \) is to be 256 steps (8 bit).

1. This yields a sampling frequency of
   \[
   f_s = 50 \text{ Hz} \times 256 \\
   f_s = 12.8 \text{ kHz}
   \]

2. So the CTC must be programmed to reach zero count at a rate of:
   \[
   t_s = \frac{1}{f_s} \\
   t_s = \frac{1}{12.8} \text{ kHz} \\
   t_s = 78.1 \mu s
   \]
   So the ISR will be executed every 78.1 micro seconds. This implies that the total ISR execution time itself must be less than 78.1 micro seconds.

3. Usually the time \( T_m \) required to execute a machine instruction is expressed as a multiple of the CPU clock period \( T_{cpu} \). For example the Z80 instruction SET 0, (IX+0) takes 23 CPU clocks. Hence \( T_m \) for this instruction is 23.

   To express the maximum total execution time of the ISR these equations apply:
   \[
   T_{cpu} = \frac{1}{f_{cpu}}
   \]
   The ISR in worst case must require less than \( N_C \) periods:
   \[
   N_C < \frac{t_s}{T_{cpu}}
   \]
   When programming the ISR, the sum of the individual \( T_m \) values of the machine instructions must not exceed \( N_C \).

4. The stability is impaired by the fact, that the CPU services an interrupt not immediately but finishes the instruction of the mainline program being executed currently first. So the delay \( t_d \) between the interrupt requested (by the CTC) and
the update of the PWM output (by the ISR) varies. The shortest instruction of the CPU instruction set dictates the minimum delay, the longest instruction the maximum delay. See the CPU instruction set for CPU specific ratings.

As far as the Z80 CPU is concerned, \( T_m \) ranges from 4 (shortest instruction) to 23 (longest instruction).

\[ 4 \leq T_m \leq 23 \]

So the delay in seconds ranges:

\[ 4 \times T_{cpu} < t_d < 23 \times T_{cpu} \]

If the Z80 CPU clock is 5 Mhz the PWM signal update will jitter by 3.8\( \mu \)s between 0.8 and 4.6\( \mu \)s, indicated with a red arrows in Figure 9 and Figure 10.

\[ \text{Figure 9: PWM output jitter} \]
4.3 The Z80 Assembly Code

The program code used for this experiment can be found here:
http://www.train-z.de/train-z/sw/applications/pwm/dac_1_channel.asm

This tiny program has a main loop where the shortest and the longest machine instructions of the Z80 are used in order to expand the stochastic jitter to its maximum.

The ISR requires tuning, in order to reduce the total execution time. The less $N_C$ gets, the higher $f_o$ may become. In the current state of this small test program the total execution time $t_{ISR}$ of the ISR is 198 cycles, or in other words 198 times the CPU clock period $T_{cpu}$:

\[
t_{ISR} = 198 \times T_{CPU}
\]

\[
t_{ISR} = 198 \times \left( \frac{1}{5MHz} \right)
\]

\[
t_{ISR} = 39.6\mu s
\]

which is well below the maximum allotted time $t_S$ of 78.8\mu s.

4.4 Results

Table 2 shows some results of the PWM experiment with the given parameters on light blue background. The measurements of $t_{width}$ confirm the expected jitter of about 3.8\mu s (see section 4.2, page 42, point 4). Noteworthy is that the pulse width $t_{width}$ is always a multiple of $t_s=78.1\mu s$ which is easy to see with very small pulse widths.

The measurements on gray background are of limited accuracy since the oscilloscope used here rounds up to tenths of milliseconds. However, the achievable mean voltage $U_{mean}$ results are most accurate.

The PWM output frequency $f_o$ is 50.9 Hz.

The resolution $R$ of this quasi DAC is 8 bit.
Figure 10 shows a scope screenshot with the jitter marked by red arrows. The yellow pointers indicate the start of the signal period.

<table>
<thead>
<tr>
<th>dac_ch0</th>
<th>( t_{\text{width.min}} ) [ms]</th>
<th>( t_{\text{width.max}} ) [ms]</th>
<th>( U_{\text{mean.min}} ) [V]</th>
<th>( U_{\text{mean.max}} ) [V]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0743</td>
<td>0.077</td>
<td>0.0084</td>
<td>0.022</td>
</tr>
<tr>
<td>2</td>
<td>0.152</td>
<td>0.155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.23</td>
<td>0.233</td>
<td>0.053</td>
<td>0.068</td>
</tr>
<tr>
<td>20</td>
<td>2.45</td>
<td>2.5</td>
<td>0.727</td>
<td>0.734</td>
</tr>
<tr>
<td>40</td>
<td>4.9</td>
<td>4.95</td>
<td>1.46</td>
<td>1.475</td>
</tr>
<tr>
<td>80</td>
<td>9.8</td>
<td>9.85</td>
<td>2.48</td>
<td>2.49</td>
</tr>
<tr>
<td>C0</td>
<td>14.7</td>
<td>14.75</td>
<td>3.5</td>
<td>3.516</td>
</tr>
<tr>
<td>FE</td>
<td>19.5</td>
<td>19.55</td>
<td>4.93</td>
<td>4.94</td>
</tr>
<tr>
<td>FF</td>
<td>19.55</td>
<td>19.6</td>
<td>4.96</td>
<td>4.97</td>
</tr>
</tbody>
</table>

\( f_{\text{cpu}} = 5 \text{ MHz} \)
\( f_{o} = 51 \text{ Hz} \)
\( U_{h} = 4.96 \text{ V} \)

Table 2: Results
For applications where accuracy and stability are no critical issues, this approach may serve for control of:

- Brightness of LEDs or lamps
- Motors
- Brakes
- Magnets
- ...

Figure 10: Oscilloscope Screenshot (PWM input value 32d/20h)
4.5 Integrating Circuitry

The mean voltage $U_{\text{mean}}$ in Table 2 page 45 has been computed by the oscilloscope (see also Figure 10 page 46). In order to obtain a real analog output signal from the PWM signal, an integrator circuitry is required. This circuitry basically must have a switched reference voltage and at least a simple RC-Low-Pass filter (or better an op-amp based integrator) with a sufficient high time constant.

Figure 11 depicts a possible circuitry taken from [3]. The time constant here is

$$t = R7 \times C4$$

$$t = 1 \text{MOhms} \times 680 \text{nF}$$

$$t = 0.68 \text{s}$$

and applies for the application described in [3]:

- $R = 4096$, (12 bit)
- $f_s = 70 \text{kHz}$
- $T_o = 60 \text{ms}$

So as a rule of thumb, the filter time constant $t$ should be 10 times the PWM signal period $T_o$. To accommodate to our figures given in section 4.2 page 42, the R7 and C4 should be replaced by 1 MOhms and 0.2µF which equals a 0.2s time constant.

The greater the time constant, the greater the settle time for the output!

Figure 11: Integrator Circuitry

The ICs used here are the classical voltage regulator µA723 and the FET op-amp TL080 (additional offset compensation may be required).
## 5 Z80 IC equivalents table

An overview of ICs of the famous Z80 family gives Table 3.

<table>
<thead>
<tr>
<th>device</th>
<th>equivalent type</th>
</tr>
</thead>
</table>
| 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)  
Z8440006 (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 3: Z80 equivalents*
6 References

1. ZiLOG, Z80 Family CPU User Manual UM008005-0205
2. ZiLOG, Z80 Family CPU Peripherals User Manual UM008101-0601

Blunk electronic at www.train-z.de
7 Useful Links

(1) Find updates of this tutorial at [http://www.train-z.de](http://www.train-z.de)

(2) *CadSoft EAGLE Training and Consulting* – a reasonable way to reasonable work at [http://www.train-z.de](http://www.train-z.de)

(3) An *EAGLE* configuration script `eagle.scr`. Units, grid, line with, text size, font, drills and more – well defined and cleaned up …

(4) *EAGLE* - an affordable and very efficient schematics and layout tool at [http://www.cadsoftusa.com](http://www.cadsoftusa.com)

(5) A Gerber Data Viewer and Editor at [http://www.pentalogix.com](http://www.pentalogix.com)

German Sales and Support Office:

Helmut Mendritzki
Software-Beratung-Vertrieb
Dahlienhofer 1
25462 RELLINGEN
GERMANY
Tel.: +49 (0) 4101 - 20 60 51
Fax: +49 (0) 4101 - 20 60 53
Mobile: +49 (0) 171 - 2155852
eMail: mendritzki@aol.com
What is Boundary Scan?

Looking for a lean **Boundary Scan Test System**? Please have a look [here](http://www.train-z.de/logic_scanner/Logic_Scanner_UM.pdf)!

Debug SPI, I²C, Boundary Scan/JTAG and other hardware with the **Logic Scanner** at [http://www.train-z.de/logic_scanner/Logic_Scanner_UM.pdf](http://www.train-z.de/logic_scanner/Logic_Scanner_UM.pdf)
(9) The office alternative: *LibreOffice* at http://www.libreoffice.org

(10) A complete embedded Z80 system plus assembler for Linux and UNIX can be found at http://www.train-z.de/train-z

(11) The powerful communication tool Kermit at http://www.columbia.edu/kermit/


(13) The Z80 interrupt structure at http://www.train-z.de/train-z/doc/z80-interrupts_rewritten.pdf


(15) More Z80 stuff at http://www.z80.info
8 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)

9 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.

My Boss is a Jewish Carpenter