Application Note

Using the Z8 MCU as an I^2C Bus Master

AN003601-Z8X1199

ZiLOG WORLDWIDE HEADQUARTERS • 910 E. HAMILTON AVENUE • CAMPBELL, CA 95008
TELEPHONE: 408.558.8500 • FAX: 408.558.8300 • www.ZiLOG.com

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, contact:

ZiLOG Worldwide Headquarters
910 E. Hamilton Avenue
Campbell, CA 95008
Telephone: 408.558.8500
Fax: 408.558.8300
www.ZiLOG.com

Windows is a registered trademark of Microsoft Corporation.

Information Integrity

The information contained within this document has been verified according to the general principles of electrical and mechanical engineering. Any applicable source code illustrated in the document was either written by an authorized ZiLOG employee or licensed consultant. Permission to use these codes in any form, besides the intended application, must be approved through a license agreement between both parties. ZiLOG will not be responsible for any code(s) used beyond the intended application. Contact the local ZiLOG Sales Office to obtain necessary license agreements.

Document Disclaimer

Table of Contents

Acknowledgements

Project Lead Engineer

Jon Veres

Application and Support Engineers

Denny Hopp
Oscar Miramontes
J.R. Wang

System and Code Development

Jon Veres

Using the Z8 MCU as an I2C Bus Master

General Overview

The ZiLOG Z8 family features many integrated features that simplify system design requirements in embedded applications. These on-chip peripheral features include Power-On Reset (POR), Low Voltage Protection, Watch-Dog Timer (WDT), programmable I/O, comparators, multiple timer/counters (including external event counting and output waveform generation), and multiple external/internal interrupts. Some applications, however, may require additional external peripheral support, such as an I/O expander, specialty memory, LCD driver, or Analog-to-Digital (A/D) converter. To provide a standardized peripheral bus connection, a simple bi-directional two-wire serial interface called the Inter-IC or I2C bus can be used. While many Z8 MCU family members do not possess dedicated on-chip I2C support hardware, this serial bus can be simulated by software control of two Z8 MCU I/O pins.

This application note provides an overview of the I2C bus interface and the 24Cxxx EEPROM family, and details an application interfacing the Z8 MCU (I2C Master) to a two-wire 24Cxxx EEPROM (I2C Slave).

Discussion

I2C Bus Overview

The I2C bus uses a two-wire interface consisting of a serial data line (SDA) and a serial clock line (SCL) to exchange information between devices connected to the bus. Each device on the bus has its own unique address and can operate as a transmitter or receiver (depending on its particular function). Devices are further characterized as Masters or Slaves. A Master is defined as a device that initiates, controls (generates all framing and clock signals), and terminates a transfer. A Slave is defined as any device addressed by a Master. Multiple Masters are allowed by I2C bus specifications because a bus arbitration and clock synchronization scheme is defined so that messages are not corrupted when more than one device attempts to take Master control. This procedure is also facilitated because all connections to the bus are wired-AND connections. Both the SDA and SCL lines are bi-directional and are pulled up to the positive logic supply rail

(via pull-up resistors, both lines high for the bus idle state). When in the output mode, the SDA and SCL lines are open drain or open collector. Data is exchanged eight bits at a time and can be transferred at rates up to 100K bits/sec (STANDARD mode) or 400 kbits/sec (fast mode).

Data transfers on the I2C bus are controlled (and framed) via two unique bus states generated by the Bus Master. These bus states are the START and STOP bit conditions. A START condition is defined as a High-to-Low level transition on SDA while the SCL line is High. A STOP condition is defined as a Low-to-High level transition on SDA while the SCL line is High. Data must always be valid (stable) on the SDA line while SCL is high. The SDA line is only allowed to change during the SCL Low period. One bit of data is transmitted for each SCL period

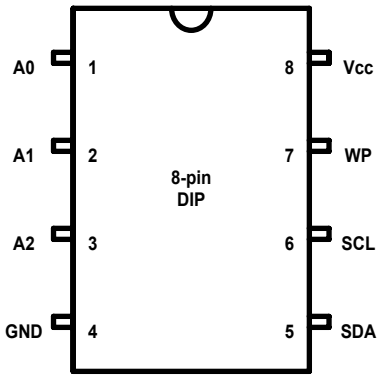
Following the START condition, the first 8-bit byte sent in a bus message is a 7-bit Slave address field along with a data direction or R/W bit. (This discussion is limited to the I2C 7-bit addressing mode.) The data direction bit (least significant bit) controls whether or not the Master transmits (0 = write) or receives (1 = read) data from the addressed Slave. For every 8-bit byte exchanged, the most significant bit is always transmitted first. All 8-bit byte bus transactions (whether address, data, and so on) are followed by an acknowledge bit. The acknowledge bit is a low-level signal placed on the SDA line by the receiving device (Master or Slave) during the Master-transmitted acknowledge clock pulse (ninth High SCL clock pulse of the byte transmission). If the receiver is unable to receive data (busy Slave receiver) or must signal the end of data condition (Master receiver), a *non-acknowledge* is sent (SDA High during the ninth High SCL clock pulse time). Following the START and Slave address transmission, data is exchanged between the Master and receiver as required. Upon exchange of the final byte and its acknowledge, the Master issues the STOP condition to end bus usage.

The previous general overview of I2C bus operation provides the foundation necessary to proceed with a serial EEPROM interface (a working knowledge of the I2C 7-bit addressing mode). See the I2C Bus Specifications published by Phillips Corporation for complete details of all the various modes this bus supports.

24Cxxx Serial EEPROM Overview

The 24Cxxx Family of two-wire serial EEPROM devices provide 256 to 2K (24C01A to 24C16) bytes of non-volatile storage and are I2C bus compatible. Figure 1 illustrates the standard 8-pin PDIP package pin-out.

Figure 1. 24Cxxx EEPROM Pin-Out



In addition to SDA and SCL, device control pins include:

- WP: Write Protect Input. When this pin is grounded, normal read/write EERPOM accesses are permitted. When connected to V_{CC}, the EEPROM is write-protected
- A2, A1, A0: Device Address Inputs. All Ax address pins are hard-wired to fix the device's programmable software address

Data is exchanged with the 24Cxxx as a standard I^2C bus Slave device. After the START command, the first byte transmitted is the EEPROM's address byte. The upper nibble identifies the device as memory (per the I^2C specification). The lower nibble selects which 24Cxxx is targeted for exchange (Ax, up to eight devices may be on the bus) and/or which internal 256-byte block in the EEPROM (Px).Table 1 contains address byte definitions.

Table 1. 24Cxx Device Name and Address Byte Definition

Device	Bytes	Max # Devices	Page Write	Device Address Byte
24C01A	128	8	8	1010 A2 A1 A0 R/W
24C02	256	8	8	1010 A2 A1 A0 R/W
24C04	512	4	16	1010 A2 A1 P0 R/W
24C08	1K	2	16	1010 A2 P1 P0 R/W
24C16	2K	1	16	1010 P2 P1 P0 R/W

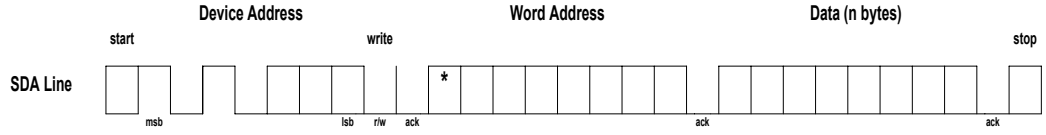
Note: R/W: 0 = Write, 1 = Read. Ax = Hard-wired device address. Px = Internal software address (selects the 256-byte block to address).

After the device address is transmitted, the desired 8-bit EEPROM word address is sent. Now, the desired device operation (read or write) is carried out.

Two types of write operations (byte and page) are supported (Figure 2). Byte operations allow a random EEPROM address to be written. Byte write operations require transmission of the following:

- START condition (Master)
- EEPROM Device Address with R/W Bit = 0 (Master)
- Acknowledge Bit (EEPROM)
- Target EEPROM Word Address to be Written (Master)
- Acknowledge Bit (EEPROM)
- Data Byte to be Written (Master)
- Acknowledge bit (EEPROM)
- STOP Condition (Master)

Figure 2. Write Operations (Byte/Page)



Upon detecting the STOP bit, the EEPROM enters an internally timed write cycle to non-volatile memory. During this write cycle, all EEPROM inputs are disabled and the device responds to further bus activity until the write is complete. The Master device features two ways to determine when the EEPROM internal write cycle is completed: 1) timeout the maximum required write time (usually much longer than the device actually requires) or 2) perform an acknowledge poll, which is the more time-efficient way. The poll involves sending a start condition followed by the target EEPROM address. If the EEPROM acknowledges its address, the device write cycle is complete and the Master's desired operation may continue (otherwise, polling must be continued).

Page write operations are identical to the byte write operations described previously, except that instead of writing one data byte, as many as 8 or 16 bytes (up to the maximum page write size the EEPROM supports) may be sent between the EEPROM address and STOP bit transmission. See Table 1 and Figure 2. During these page writes, the EEPROM automatically increments its internal address pointer between bytes.

Note: If the device's maximum page number address is exceeded, the data writes wrap around.

When the STOP condition is received, the updated bytes are written to the non-volatile array. The great advantage of the page write feature is that only one write cycle timing is consumed to write multiple bytes of data.

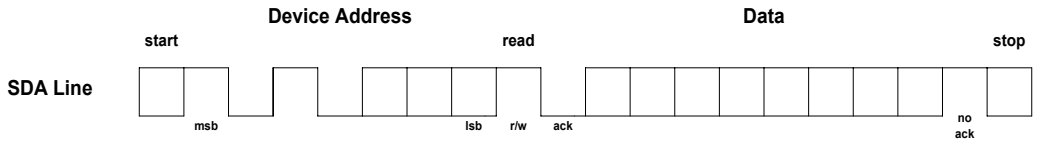
Three types of read operations are supported: Current Address, Random, and Sequential Read. Read operations begin just like write operations, except that the R/W bit is set to 1 for the device address byte.

For the CURRENT ADDRESS READ mode, no EEPROM byte address is written as the data transmitted by the addressed Slave to the Master is read from the location of the most recent access (incremented by one). This read transmission type sequence appears as:

- START condition (Master)
- EEPROM device address with R/W bit = 1 (Master)
- Acknowledge bit (EEPROM)
- Data byte to be read (EEPROM bytes sent from the addressed Slave's most recent pointed-to memory location incremented by 1)
- Non-acknowledge bit (Master)
- STOP condition (Master)

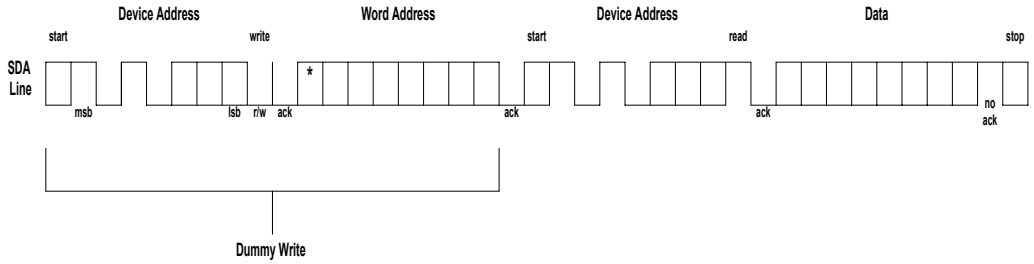
Note: The Master signals a non-acknowledge and STOP condition to terminate data exchange with the Slave EEPROM, as illustrated in Figure 3.

Figure 3. Current Address Read Operation



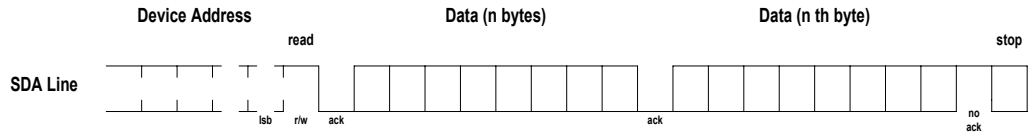
The RANDOM READ mode is begun with a dummy byte write cycle (Master sends a START condition followed by the device address and target word address) followed by a CURRENT ADDRESS READ mode cycle as described previously (Figure 4).

Figure 4. Random Read Operation



The SEQUENTIAL READ mode is initiated with either a Current Address Read or Random Read. Instead of the Master terminating the read after a single byte exchange (with a non-acknowledge), the Master responds with a valid acknowledge after each received data byte. This acknowledge instructs the Slave EEPROM to continue the read operation and transmit out the next data byte. Sequential reads continue until terminated by the Master via issuance of a non-acknowledge on the most recent byte read followed by the STOP condition (Figure 5).

Figure 5. Sequential Read Operation

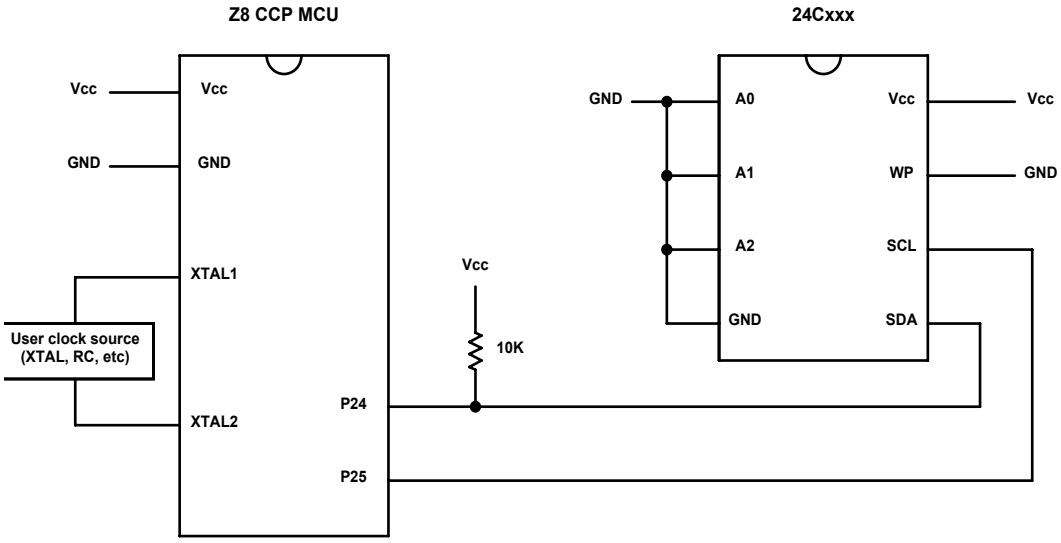


Z8 MCU Interface to 24Cxxx EEPROM

Figure 6 illustrates the hardware interface between the Z8 MCU and a 24Cxxx EEPROM. The Write Protect (WP) input is tied to logic ground so that the MCU may write the EEPROM. Address line A0, A1, and A2 are also tied to logic ground.

Note: An external pull-up is required on the SDA line (open-drain output).

Figure 6. Generic Z8 MCU 24Cxxx Interface



The program listed in the Technical Support section allows the writing/reading of an EEPROM in the stand-alone mode illustrated in Figure 6. The actual EEPROM write/read modules are easily exported for use in any user application. The Z8 MCU was clocked at 8 MHz for this sample code. The user is advised to ensure that the selected EEPROM set-up and hold times are met at the application's Z8 clock frequency.

Summary

Many MCU applications require specialty peripheral functions, such as multi-channel high precision Analog-to-Digital or Digital- to-Analog converters, scratch-pad EEPROM memory, or display drivers. Often these system peripheral requirements can be met more cost effectively by using components external to the MCU. These external peripherals often use a simple two-wire communication bus—the I2C bus. This application note provided an overview of the I2C bus and transportable I2C bus read and write modules for the Z8 MCU (MASTER mode). Use of these Z8 I2C bus code modules was demonstrated with an example interfacing the Z8 MCU (in MASTER mode) with a 24Cxxx family EEPROM (in Slave mode).

Technical Support

Source Code

```
;*****************************************************************************
;*     Module Name:   I2C.asm
;*     Version:       V1.0
;*     Copyright:     ZiLOG (c)1999
;*     Date:          August 23, 1999
;*     Created by:    Jon Veres - ZiLOG Ohio
;*     Compiler:      ZDS V2.11
;*     Description:   A stand-alone program that demonstrates the
;*                    generic Z8  MCU to 24Cxxx interface modules. The
;*                    Z8 supports EEPROM page write/page read operations
;*                    (page size of 1-16 bytes). In this example, the data
;*                    stored in register file bank 40h (the designated I2C
;*                    transmit buffer) is written to the first 16 bytes of
;*                    the EEPROM in a single 16-byte page write operation.
;*                    This EEPROM data is then read back 8 bytes at a time,
;*                    and stored in register file bank 50h (designated I2C
;*                    receive buffer). The data is read using two EEPROM
;*                    read techniques. The first 8 bytes are read using the
;*                    RANDOM ADDRESS SEQUENTIAL READ mode, while the next 8
;*                    bytes are read using the SEQUENTIAL CURRENT ADDRESS
;*                    READ mode.
;*****************************************************************************
;
;     tab size = 6
;
; Z86xxx MCU I/O definition:
;
; -  P2.4 = 24Cxxx serial data (SDA) I/O pin
; -  P2.3 = 24Cxxx serial clock (SCL) pin
;
; For this sample program, EEPROM set-up/hold time between SCLK and SDA
; changes is a minimum of 12 Z8 sclks. User is cautioned to check timing
; of the actual EEPROM used for exact requirements relative to the actual
; Z8 clock speed selected.
;
; This program uses a Z8 external clock frequency of 8 MHz (standard mode
; oscillator) and a 24CXXX device with maximum SCLK frequency of 100KHz @
; 2.5v.
;
;
     globals on
;
;*****************************************************************************
;*     EEPROM interface mask equates
;*****************************************************************************
;
EE_START    equ   00h         ; EEPROM start address (00h)
XRAM_START  equ   40h         ; bank 4 = I2C xmit buffer
RRAM_START  equ   50h         ; bank 5 = I2C rec buffer
RRAM_START1 equ   58h         ; bank 5 = I2C rec buffer (MID)
SDA_HI      equ   10h         ; EEPROM data bit hi - (P24)
SDA_LO      equ   0EFh        ; EEPROM data bit lo - (P24)
SCLK_HI     equ   08h         ; EEPROM clk bit hi - (P23)
```

```
SCLK_LO     equ   0F7h        ; EEPROM clk bit lo - (P23)
NUMBER_8    equ   08h         ; #bytes to page rd (8 byte page)
NUMBER_16   equ   10h         ; #bytes to page wr (16 byte page)
WR_CNTRL0   equ   0A0h        ; address/page 0 EEPROM wr control
;
;
; NOTE:  The SDA_xx AND SCLK_xx equates reflect the use of Z8 I/O p24 and
;        p23. If different I/O pins are used, the equates must be changed
;      to reflect the new I/O port bit selections.
;
;
      define bank6_data, space=rfile, org=60h         ; mapped to 60-6F
;
      segment bank6_data
;
; register bank 6 used for I2C control registers (direct address mode)
;
DATA        ds    1           ; serial I/O data holding register
VALUE       ds    1           ; data out/compare register
TEMP        ds    1           ; general purpose temp reg
BIT_CNT     ds    1           ; serial transfer bit loop counter
ACK_CNT     ds    1           ; EEPROM wr ack polling counter
EE_PTR      ds    1           ; byte address pointer to EEPROM
RAM_PTR     ds    1           ; address pointer to z8 register file
DEV_ADDR    ds    1           ; 24C01 Ax/Px device address/control byte
OLD_P2M     ds    1           ; p2m image register
BYTE_CNT    ds    1           ; byte count for EEPROM page transfers
;
;
;*****************************************************************************
*     Interrupt Vectors
*****************************************************************************

      vector      reset = begin
      vector      irq0  = IRQ0
      vector      irq1  = IRQ1
      vector      irq2  = IRQ2
      vector      irq3  = IRQ3
      vector      irq4  = IRQ4
      vector      irq5  = IRQ5
;
      segment code
;
; start of program
;
;*****************************************************************************
*     system initialization
*****************************************************************************
;
begin:      di                              ; program start location
            ld    p01m, #04h                ; set for int stack, p0=outputs
            ld    p3m, #01h                 ; p2 outputs are push-pull
            ld    p2, #0FFh                 ; init p2 outputs hi xxx1 1xxx)
            ld    OLD_P2M, #00h             ; p2m image reg (xxx0 0xxx)
            ld    p2m, OLD_P2M              ; p2 = outputs (xxx0 0xxx)
            ld    spl, #80h                 ; initialize top of stack
;
; the minimum system initialization code for this application is complete.
```

```
;
*************************************************************************
*     ee_wr:  The following instructions perform the EEPROM write
*             operation. Three pointers and a transfer count must be set-
*            up by the user prior to calling the pg_wr subroutine:
*
*        - DEV_ADDR = 24Cxx Ax/Px address with the R/W bit reset to 0,
*        - EE_PTR   = Target EEPROM address ptr to start data storage at,
*        - RAM_PTR  = Z8 reg file buffer ptr to start data storage from,
*        - BYTE_CNT = # bytes to xfer (1 to max page buffer number).
*
*        In this example, the data stored in register file locations 40h-
*        4Fh (the defined I2C transmit buffer) is read out and written to
*        EEPROM locations 00h-0Fh of bank 0 (a 16 byte page write).
*************************************************************************
;
ee_wr:      ld    DEV_ADDR, #WR_CNTRL0    ; init EEPROM addr/page/wr bits
            ld    EE_PTR, #EE_START       ; init EEPROM byte address ptr
            ld    RAM_PTR, #XRAM_START    ; init write buffer pointer
            ld    BYTE_CNT, #NUMBER_16    ; init EEPROM page xfer count
            call  pg_wr                   ; write "NUMBER" of bytes to
                                          ; EEPROM
;
; EEPROM write is complete and the non-volatile array is being updated -
; completion of this EEPROM storage cycle is validated by the wr_poll
; command sequence prior to the next read or write operation.
;
*************************************************************************
*     ee_rd:  The following instructions perform the EEPROM random address
*             sequential read operation. Three pointers and a transfer
*             count must be set-up by the user prior to calling the pg_rd
*             subroutine:
*
*        - DEV_ADDR = 24Cxx Ax/Px address with the R/W bit reset to 0,
*        - EE_PTR   = Target EEPROM address ptr to start data storage at,
*        - RAM_PTR  = Z8 reg file buffer ptr to start data storage from,
*        - BYTE_CNT = # bytes to xfer (1 to max page buffer number).
*
*        The above four bytes are required for a random read (BYTE_CNT =1)
*        and a random sequential read (BYTE_CNT >1). The EEPROM read is
*        accomplished by calling the pg_rd routine.
*
*        In this example, EEPROM addresses 00h-07h are read out and stored
*        to register file locations (50h-57h -- the first half of the I2C
*        receive buffer).
*
*        For a current address read or a current address sequential read,
*        the EE_PTR register is not used. This type of EEPROM read is
*        executed by calling the pg_rd1 routine - see ee_rd1.
*************************************************************************
;
ee_rd:      ld    DEV_ADDR, #WR_CNTRL0    ; init EEPROM addr/page/wr bits
            ld    EE_PTR, #EE_START       ; init EEPROM byte address ptr
            ld    RAM_PTR, #RRAM_START    ; init read buffer pointer
            ld    BYTE_CNT, #NUMBER_8     ; init EEPROM seq xfer byte cntr
            call  pg_rd                   ; read "NUMBER" of bytes from
                                          ; EEPROM
;
```

```
;
*****************************************************************************
*     ee_rd1: The following instructions perform the current address
*             sequential read operation. Two pointers and a transfer count
*             must be set-up by the user prior to calling the pg_rd1
*             subroutine:
*
*        - DEV_ADDR = 24Cxx Ax/Px address with the R/W bit reset to 0,
*        - RAM_PTR  = Z8 reg file buffer ptr to start data storage from,
*        - BYTE_CNT = # bytes to xfer (1 to max page buffer number).
*
*        For a current address read or a current address sequential read,
*        the EE_PTR register is not used (the internal EEPROM address
*        pointer is used). This type of EEPROM read is executed by calling
*        the pg_rd1 routine.
*
*        In this example, EEPROM addresses 08h-0Fh are read out and stored
*        to register file locations (58h-5Fh -- the second half of the I2C
*        receive buffer).
*****************************************************************************
;
ee_rd1:     ld    DEV_ADDR, #WR_CNTRL0    ; init EEPROM addr/page/wr bits
            ld    RAM_PTR, #RRAM_START1   ; init read buffer pointer
            ld    BYTE_CNT, #NUMBER_8     ; init EEPROM seq xfer byte cntr
            call  pg_rd1                  ; read "NUMBER" of bytes from
                                          ; EEPROM
;
; test that the EEPROM data written equals the data read back
;
verify:     ld    RAM_PTR, #XRAM_START    ; ptr to xmit buffer start
            ld    EE_PTR, #RRAM_START     ; ptr to receive buffer start
            ld    BYTE_CNT, #NUMBER_16
ver_1:      ld    DATA, @RAM_PTR          ; get xmit data byte
            cp    DATA, @EE_PTR           ; compare to EEPROM data read
            jp    ne, ee_err
            inc   RAM_PTR           ; advance ptrs for next byte
            inc   EE_PTR
            dec   BYTE_CNT          ; test for all bytes checked
            jr    nz, ver_1
;
done:       jr    done                    ; EEPROM write/read complete
;
;
*****************************************************************************
* The following routines perform the actual EEPROM byte write, page write,
* random read, current address read, and sequential read once the above
* described parameters are passed.
*****************************************************************************
;
*****************************************************************************
*        PG_WR:  This routine writes a page of data (length controlled by
*        reg BYTE_CNT) using the EEPROMs page write feature.  A byte write
*        accomplished by setting BYTE_CNT = 1.
*****************************************************************************
;
pg_wr:      call  wr_poll                 ; verify EEPROM ready,
                                          ; send start, wr command
            ld    DATA, EE_PTR            ; EEPROM wr byte address
```

```
            call  outbyt                  ; send write address byte
            call  rec_ack                 ; search for EEPROM ack command
            jp    c, ee_err               ; CF=1, no EEPROM ack received
pg_wr1:     ld    DATA, @RAM_PTR          ; get data to wr to EEPROM
            call  outbyt                  ; SEND WRITE DATA
            call  rec_ack                 ; search for EEPROM ack command
            jp    c, ee_err               ; CF=1, no EEPROM ack received
            inc   RAM_PTR                 ; inc for next reg file byte
            dec   BYTE_CNT                ; test for all bytes written
            jr    nz, pg_wr1
            call  stop                    ; page write complete,send stop
            ret                           ; to initiate EEPROM store cycle
;
*************************************************************************
*     pg_rd:  This routine reads a block of data (length controlled by
*             reg BYTE_CNT) by using the EEPROMs sequential read feature.
*            A single byte read is accomplished by setting BYTE_CNT = 1.
*************************************************************************
;
pg_rd:      call  wr_poll                 ; verify EEPROM ready,
                                          ; send start, wr command
            ld    DATA, EE_PTR            ; EEPROM W/R start address
            call  outbyt                  ; send write data
            call  rec_ack                 ; search for EEPROM ack command
            jp    c, ee_err               ; CF=1, no EEPROM ack received
;
; the "dummy" write is complete - proceed with read address parameters
;
; NOTE: This is also the start point for a current address read operation.
; User must insure that a EEPROM write cycle is not pending when a current
; read operation is initiated directly.
;
pg_rd1:     call  start                   ; send the start command
            or    DEV_ADDR, #01h          ; set R/W bit = 1, read command
            ld    DATA, DEV_ADDR          ; setup EEPROM read control
            call  outbyt                  ; send data
            call  rec_ack                 ; search for EEPROM ack command
            jp    c, ee_err               ; CF = 1, no EEPROM ack received
pg_rd2:     call  inbyt                   ; read byte from EEPROM
            ld    @RAM_PTR, DATA          ; store EEPROM data to reg file
            dec   BYTE_CNT                ; dec cnt, is page read complete
            jr    z, pg_rd3
            call  ack                     ; send the page read byte ack
            inc   RAM_PTR
            jr    pg_rd2
;
pg_rd3:     call  nack                    ; page read done, issue nack,
            call  stop                    ; issue stop to finish exchange
            ret
;
*************************************************************************
*     inbyt: Shift in 8 bits from the EEPROM to the data register.
*************************************************************************
;
inbyt:      or    OLD_P2M, #SDA_HI        ; change SDA to an input
            ld    p2m, OLD_P2M            ; update p2m
            ld    BIT_CNT, #08h           ; load up for 8 shift times
in_1:       call  clock                   ; clock the data
```

```
            rlc   DATA                    ; build serial data from EEPROM
            dec   BIT_CNT                 ; loop for 8 bit times
            jr    nz, in_1
            and   OLD_P2M, #SDA_LO        ; change SDA to an output
            ld    p2m, OLD_P2M            ; update P2M
            ret
;
;*****************************************************************************
*     outbyt:  Shift out 8 bits from the data register to the EEPROM.
*****************************************************************************
;
outbyt:     ld    BIT_CNT, #08H           ; load up for 8 shift times
out_1:      rlc   DATA                    ; rotate to xmit into CF
            jr    c, out_2                ; send carry state to SDA
            and   p2, #SDA_LO             ; write 0 to SDA
            jr    out_3
out_2:      or    p2, #SDA_HI             ; write 1 to SDA
out_3:      call  clock                   ; clock out this data bit
            dec   BIT_CNT                 ; loop till all 8 bits sent
            jr    nz, out_1
            ret
;
;*****************************************************************************
*     rec_ack:  Test for the EEPROM receive acknowledge (result in CF).
*****************************************************************************
;
rec_ack:    or    OLD_P2M, #SDA_HI        ; change SDA to an input
            ld    p2m, OLD_P2M            ; update p2m
            or    p2, #SDA_HI
            call  clock                   ; generate a clock pulse
            and   OLD_P2M, #SDA_LO        ; change SDA to an ouptut
            ld    p2m, OLD_P2M            ; update p2m
            ret
;
;*****************************************************************************
*     wr_poll:  Poll the EEPROM to determine when the actual NV write
*               cycle is complete.  The routine tests 256 times for
*               a valid EEPROM ack which is indicated by a reset CF upon
*               return.
*****************************************************************************
;
wr_poll:    ld    ACK_CNT, #00h           ; max # of times to poll = 256
wr_poll1:   dec   ACK_CNT
            jr    z, wr_poll2
            call  start                   ; set-up to access EEPROM
            ld    DATA, DEV_ADDR          ; set-up EEPROM address/page
            call  outbyt                  ; send write data
            call  rec_ack                 ; search for EEPROM ack command
            jr    C, wr_poll1             ; re-test if no ack received
            ret
;
wr_poll2:   call  start                   ; issue a start command
            call  stop                    ; issue a stop/low pwr command
            jp    ee_err                  ; no EEPROM ack rec'd
;
;*****************************************************************************
*     start:  Issue a start command
*****************************************************************************
```

```
;
start:      or    p2, #SDA_HI             ; send start - SDA set to 1
            nop                           ; hold time
            nop
            or    p2, #SCLK_HI            ; set clock output high
            nop                           ; give set-up time
            nop
            and   p2, #SDA_LO             ; reset SDA to 0
            nop                           ; hold time
            nop
            and   P2, #SCLK_LO            ; set clock output high
            ret
;
*****************************************************************************
*     stop:  Issue a stop command
*****************************************************************************
;
stop:       and   p2, #SDA_LO             ; write SDA to 0,stop condition
            nop                           ; hold time
            nop
            or    p2, #SCLK_HI            ; set clock output hi
            nop                           ; give set-up time
            nop
            or    p2, #SDA_HI             ; write SDA to 1
            ret
;
*****************************************************************************
*     ack:  Issue a ack - continue a page read
*****************************************************************************
;
ack:        and   p2, #SDA_LO             ; reset SDA to 0
            call  clock                   ; generate a clock pulse
            ret
;
*****************************************************************************
*     nack:  Issue a nack - terminate a byte read.
*****************************************************************************
;
nack:       or    p2, #SDA_HI             ; set SDA to 1
            call  clock                   ; generate a clock pulse
            ret
;
*****************************************************************************
*     clock:  Issue a clock pulse - SDA level stored in carry flag.
*****************************************************************************
;
clock:      or    p2, #SCLK_HI            ; set the clock line hi
            scf                           ; init CF = no EEPROM ack, with
            nop                                 ;set-up time before bit read
            tm    p2, #SDA_HI             ; test SDA input for a logic 0
            jr    nz, clock_1             ; jp if SDA hi,no EEPROM ack
            rcf                           ; EEPROM ack recv'd, reset CF
clock_1:    and   p2, #SCLK_LO            ; reset clock line lo
            ret
;
*****************************************************************************
*     ee_err:  This routine performs the system shutdown procedure when
*              the EEPROM does not respond to a write/read request.
```

```
;*******************************************************************************
;
ee_err:     jp    ee_err                ; put your error handler here
;
;*******************************************************************************
;*     Interrupt Service
;*******************************************************************************
;
      IRQ0:
      IRQ1:
      IRQ2:
      IRQ3:
      IRQ4:
      IRQ5:
;
            jp    begin
;
            .END
```

Flowcharts

Figure 7. Z8 MCU 24Cxxx Interface: WR_POLL Subroutine

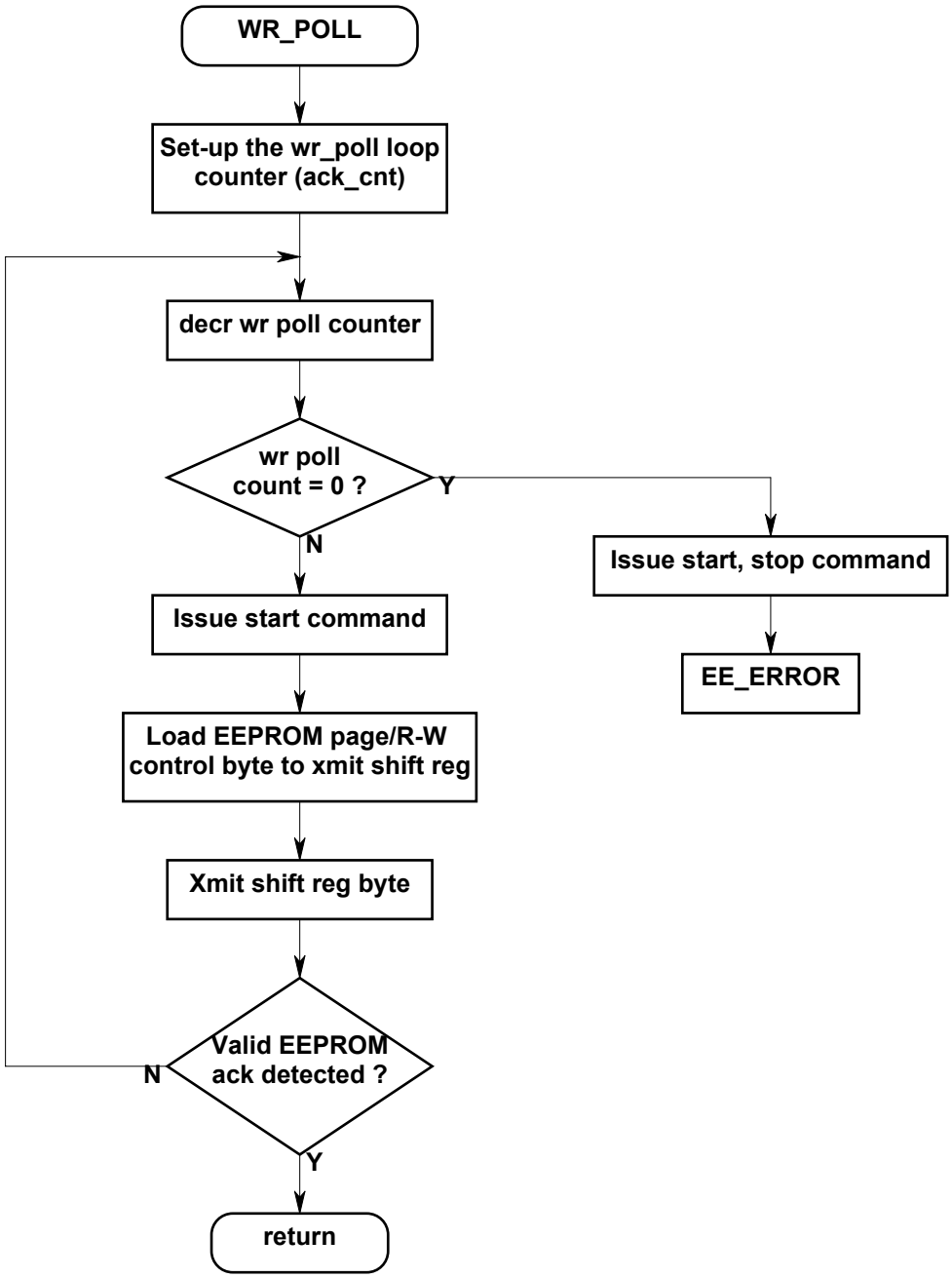


Figure 8. Z8 MCU 24Cxxx Interface: OUTBYT Subroutine

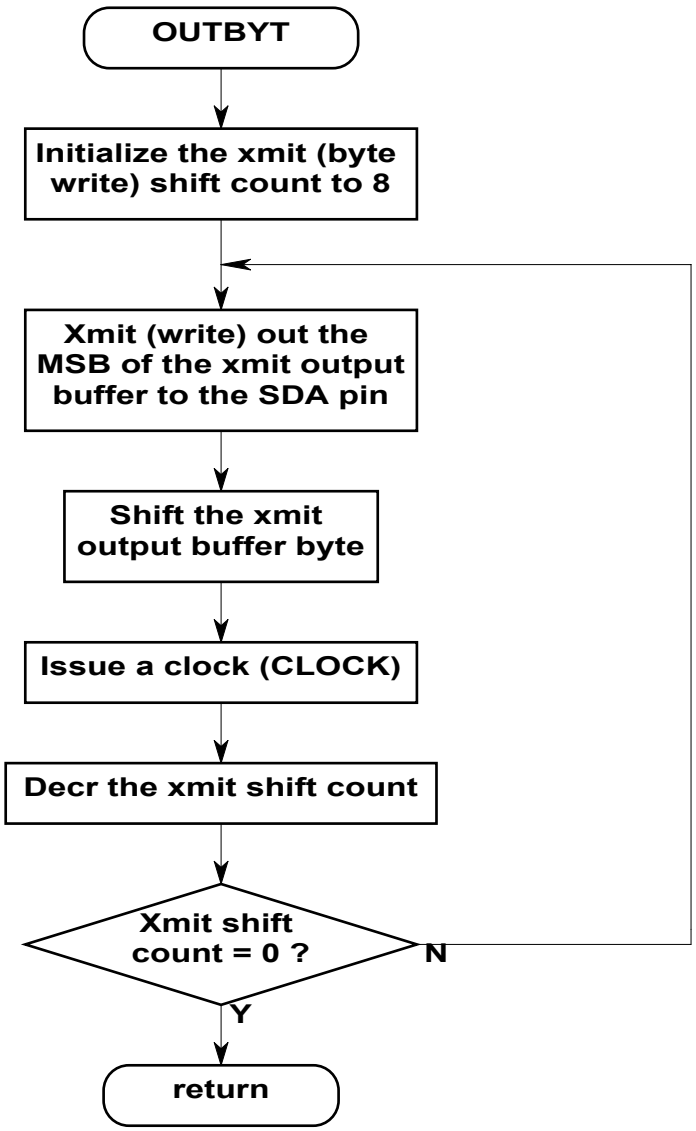


Figure 9. Z8 MCU 24Cxxx Iterface: INBYT Subroutine

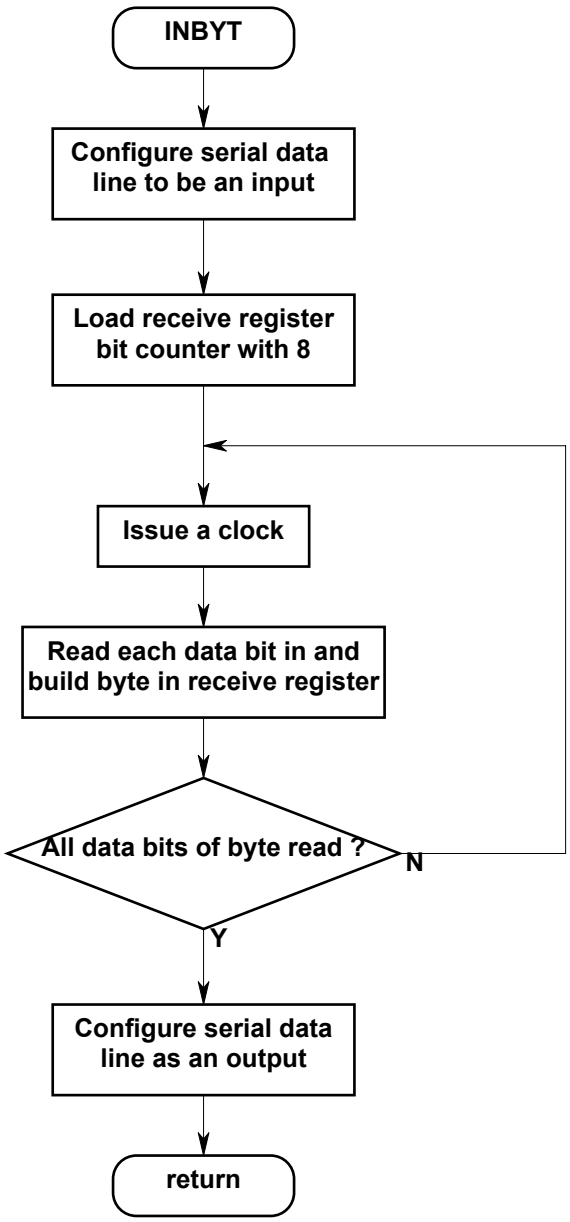
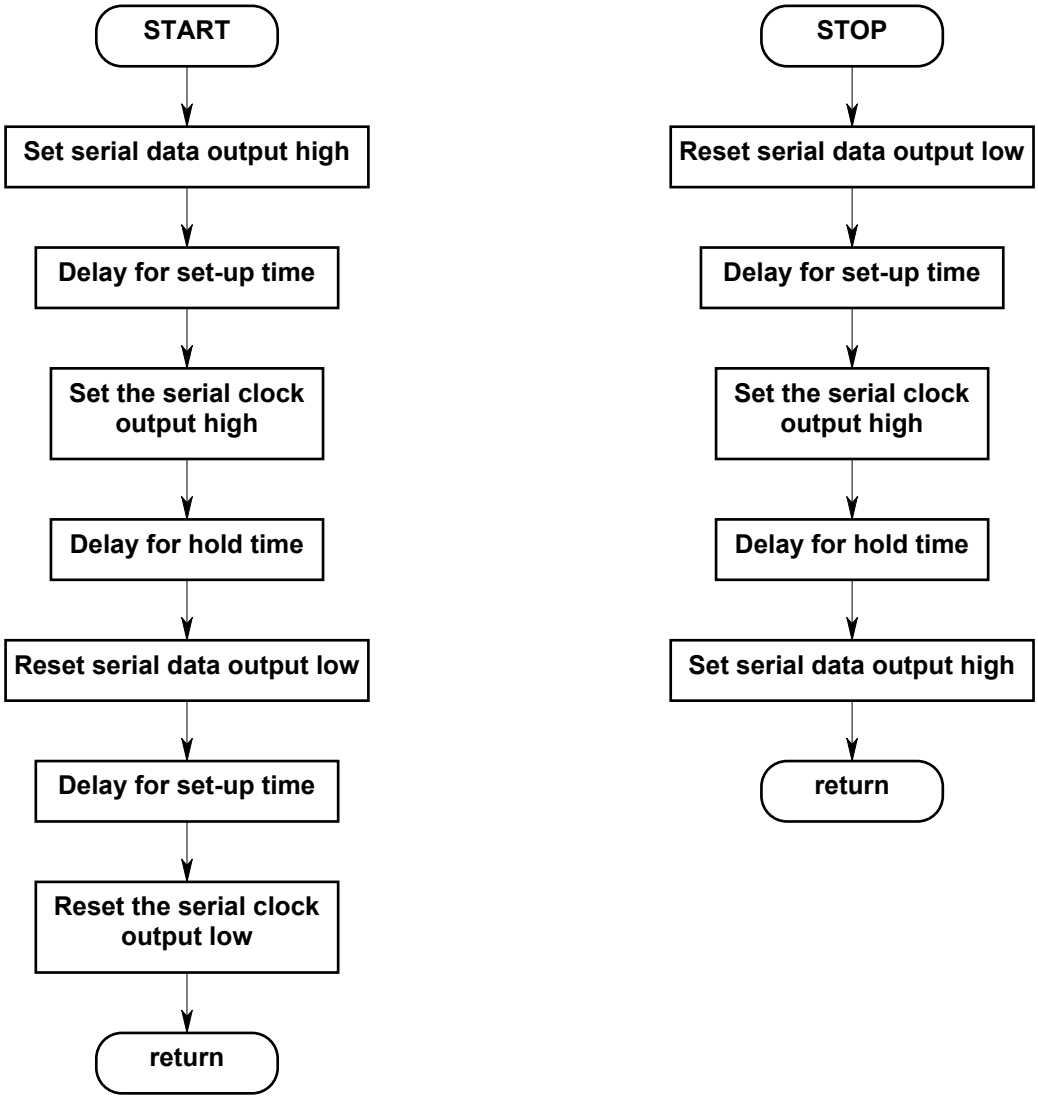


Figure 10. Z8 MCU 24Cxxx Interface: START/STOP Subroutines



Test Procedure

Equipment Used

Testing the Z8 I2C Bus Master demonstration program requires the following:

- Windows 95/98/NT-based PC with ZDS 2.11 installed
- Z8601ZEM
- Z8600ZAC
- 8V power supply
- Tektronix Scopemeter (only nrequired to view serial data stream if desired)
- Target test board with Z8 connections to target EEPROM (24LC04B)

General Test Setup and Execution

Testing was performed with ZDS 2.1, using the Z86C04 as the target chip. The 18-pin emulator target cable is connected from the emulator to the target EEPROM board (pin 1 of emulator cable to pin1, row F of the target board). The target EEPROM board is wired as illustrated in Figure 6. Specific Z8 MCU pin numbers are not provided because this code works on any Z8.

ZDS 2.11 is used to assemble the source program (`I2C.asm`) and monitor the Z86C04 register file memory windows. The Z8 write buffer is at locations `40h-4Fh` while the read buffer is at locations `50h-5Fh`. Data to be stored to the EEPROM is entered into the write buffer area via the Z8 register file watch window. The data read out of the EEPROM is stored in the read buffer and monitored via the Z8 register file watch window. In the demonstration program, the first 16 bytes of the EEPROM are exercised.

To run and verify the demo program, enter the data to be written to EEPROM (as detailed above) and set a breakpoint at the `done` and `ee_err` program labels. Then issue the RESET+GO command. The program breaks at the `done` label if working correctly.

Different areas of the EEPROM can be written by varying the `EE_START` value (starting address in the selected memory page) and `WR_CNTRL0` (memory page selected).

To read the EEPROM without first writing (EEPROM data retention checking), start program instruction execution at the `ee_rd` label. To read after an emulator power-up or RESET command, execute the system initialization instructions prior to starting execution at the `ee_rd` label. One way to execute these initialization instructions is to single step from the begin label to the `ee_wr` label (another way is to set a breakpoint at label `ee_wr` and then perform a RESET+GO command).

When at the `ee_wr` label, place the cursor at the `ee_rd` label and execute the JUMP TO CURSOR command. The read now can be performed by executing from `ee_rd`.

Test Results

The program works as specified and allows the I2C Master Z8 MCU to write/read all pages/locations in the Slave EEPROM. device.

References

- *Atmel Nonvolatile Memory Data Book*, Atmel Corporation, 1996.
- *I2C Bus Specifications*, Philips Semiconductors Microcontroller Data Handbook, 1996.