Application Note

Using the Z8 Encore! XP® MCU as an I^{2}C Bus Master

AN012606-1207

Abstract

This Application Note describes the interfacing of the I^{2}C on-chip peripheral of Zilog's Z8 Encore! XP® microcontroller (MCU) to a two-wire I^{2}C EEPROM, the AT24C128 from Atmel Corporation. The I^{2}C on-chip peripheral functions as a Master and the AT24C128 EEPROM functions as a Slave in a single Master configuration. The software implementation for I^{2}C communication with the Slave EEPROM is performed in the polling, interrupt, and the DMA modes.

Additionally, this Application Note describes a set of ready-to-use Application Programming Interface (API) functions, written in C, that allow separate compilation of modules and also support easy modification.

These device-specific APIs offer services to perform a single-byte Read and a sequential Read from the I^{2}C EEPROM, and services to perform a single-byte Write and a page Write to the I^{2}C EEPROM.

A source code file, AN0126-SC01.zip, is associated with this Application Note and is available on www.zilog.com.

Z8 Encore! XP Flash Microcontrollers

Zilog's Z8 Encore! XP products are based on the new eZ8 CPU and introduce Flash memory to Zilog's extensive line of 8-bit MCUs. Flash memory in-circuit programming capability allows for faster development time and program changes in the field. The high-performance register-to-register based architecture of the eZ8 core maintains backward compatibility with Zilog's popular Z8® MCU.

Featuring Zilog's eZ8 CPU, the new Z8 Encore! XP microcontrollers combine a 20 MHz core with Flash memory, linear-register SRAM, and an extensive array of on-chip peripherals. These peripherals make the Z8 Encore! XP MCU suitable for a variety of applications including motor control, security systems, home appliances, personal electronic devices, and sensors.

Discussion

This section discusses I^{2}C basics and the I^{2}C EEPROM component (AT24C128) used for this application.

I^{2}C Basics

The I^{2}C bus uses a two-wire interface consisting of a Serial Data (SDA) line and a Serial Clock (SCL) line to exchange information between devices connected to the bus.

The two signals SCL and SDA are open collector input and output (the collector of the output transistor is not terminated). When idle, both SDA and SCL are High (positive supply voltage, logic 1). During normal data transfer mode, the SDA never changes its state when SCL is High. SDA toggles, if necessary, when SCL is Low.

START and STOP Condition

The I^{2}C bus protocol defines unique START (S) and STOP (P) conditions (see Figure 1).

A High to Low transition on the SDA line while SCL is High indicates a START condition. A Low to High transition on the SDA line while SCL is High defines a STOP condition. The I^{2}C Master always generates the START and STOP conditions.

The I^2C bus is considered to be busy after a START condition and is free again after a STOP condition.

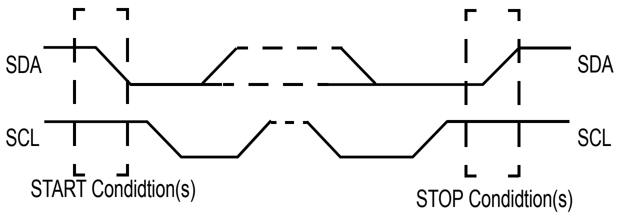


Figure 1. The START and STOP Condition of an I^2C Bus

Z8 Encore! XP MCU's built-in on-chip I^2C hardware makes it easy to the detect the START and STOP conditions of the I^2C bus. However, microcontrollers without such a capability must sample the SDA line at least twice per clock period to sense the START and STOP condition. This additional sampling adds to processor overhead.

Data Transfer

Data is always transferred in byte format. A typical data transfer specifies the address of the device to which data must be transferred (that is, there can be one or more slaves connected to the I^2C bus), and the operation to be performed (a Read or Write operation).

Byte Format— Every byte placed on the SDA line must be followed by an Acknowledge bit (ACK). Data is transferred with the most-significant bit (msb) first. The number of bytes that can be transmitted per transfer is not restricted.

Data Transfer Formats— Data transfers follow the formats displayed in Figures 3 and Figure 4.

After the START condition (S), a Slave address is transmitted. This address is 7 bits long, followed by an eighth bit that is a Read/Write data direction bit (see Figure 2).

- 0 indicates a Write
- 1 indicates a request for Read

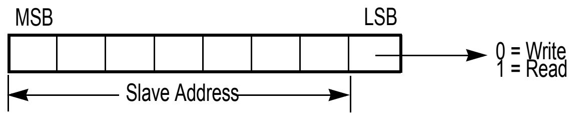


Figure 2. Address Format

A data transfer is always terminated by a STOP condition (P) generated by the Master. However, if a Master must still communicate on the I^2C bus, it can generate a repeated START condition (Sr) and address another Slave without generating a STOP condition. Various combinations of Read/Write formats are possible within such a transfer.

Acknowledge— Data transfer with an acknowledgement is obligatory. The Acknowledge-related clock pulse is generated by the Master. The transmitter releases the SDA line (High) during the Acknowledge clock pulse. The receiver must pull down the SDA line during the Acknowledge clock pulse so that it remains at a stable Low during the High period of this clock pulse.

Usually, an addressed receiver must generate an ACK after each byte is received. However, when a Slave-receiver does not acknowledge the Slave address (for example, it was unable to receive the clocked data because it was performing some other real-time function), the data line must be allowed to remain High by the Slave. The Master can generate a STOP condition to abort the transfer.

If a Slave-receiver acknowledges the Slave address, but some time later during the transfer it cannot receive any more data bytes, then a No Acknowledge condition (NACK) is generated on the line because the Slave fails to send an Acknowledge. The Master can again abort the transfer. The Slave leaves the data line High and the Master generates the STOP condition.

If a Master-receiver is involved in a transfer, it must signal the end of data to the Slave-transmitter by generating a NACK on the final byte that is clocked out of the Slave. The Slave-transmitter must release the data line to allow the Master to generate a STOP or a repeated START condition.

Addressing

Every I^2C device is identified with an unique address. The I^2C Master first transmits the address of the required device to establish communication with a Slave device.

There are two different addressing modes: 7-bit addressing and 10-bit addressing. The Z8 Encore! XP MCU supports both addressing modes.

7-Bit Addressing Mode— In I^2C addressing, the first byte after the START condition determines which Slave is selected by the Master. The exception is the general call address that can address all devices. When this address is used, all devices respond with an ACK. However, devices can be configured to ignore this address. The second byte of the general call address defines the action to be taken.

Definition of Bits in the First Byte— When an address is transmitted, each device in a system compares the first seven bits after the START condition with its own address. If they match, the device considers itself addressed by the Master, as a Slave-receiver or Slave-transmitter, depending on the R/W bit. Figures 3 and Figure 4 display different data transfer sequences. The legend that follows supports both diagrams.

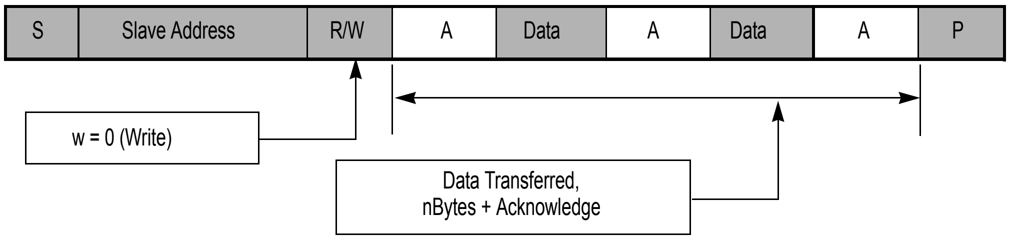


Figure 3. Master Transmitter Addressing a Slave Receiver for Write Operation

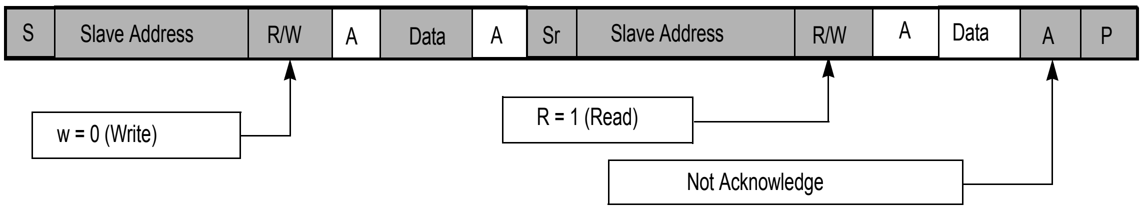


Figure 4. Format Illustrating Both Write and Read Operations

A = Acknowledge (SDA Low)	P = STOP Condition
$\overline{A}$ = Not Acknowledge (SDA High)	Shaded = Data transmitted from the Master
S = START Condition	Unshaded = Data / Acknowledge from Slave
Sr = Restart Condition	

10-Bit Addressing Mode— The 10-bit addressing mode does not change the format in the I²C-bus specification and does not affect existing 7-bit addressing. Devices with 7-bit and 10-bit addresses can be connected to the same I²C-bus. For more information about the I²C bus, refer to the data sheets from Philips Semiconductor Inc. (see References on page 9).

Description of External Components (AT24C128)

The AT24C128 EEPROM device provides 131,072 bits of serial, electrically-erasable, and programmable Read-only memory organized as 16,384 words of 8 bits each. The device's cascadable feature allows up to four devices to share a common 2-wire I²C bus.

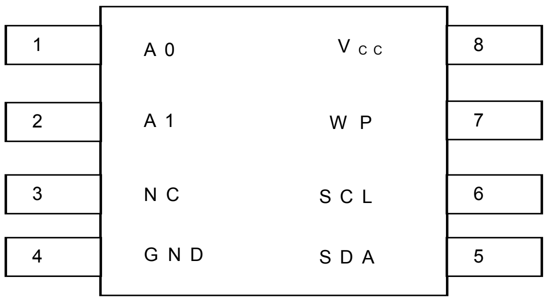


Figure 5. Pin Configuration

In addition to SCL and SDA, the device features the following control pins:

A0 and A1— These pins are device address programming inputs that are hardwired. As many as four of these devices can be addressed on a single bus system. When the pins are not hardwired, the default logic is zero.

Write Protect (WP)— The write protect input, when tied to GND, allows normal Write operations. When WP is tied High to V_{CC}, all Write operations to memory are inhibited. If WP remains unconnected, it is internally pulled down to GND. Switching WP to V_{CC} prior to a Write operation creates a software write protect function.

For complete specifications about the AT24C128 I²C EEPROM, refer to the ATMEL AT24C128 data sheet from Atmel Corporation (see References on page 9).

Developing the Z8 Encore! XP I²C Bus Master Application

This section begins with a brief description of the Z8 Encore! XP MCU's I²C Controller and its operating modes, followed by details of the hardware architecture and software implementation of the I²C bus Master application.

Z8 Encore! XP I²C Controller

The I²C Controller makes the Z8 Encore! XP bus compatible with the I²C protocol. The I²C Controller consists of two bidirectional bus lines—SDA and a SCL.

Features of the I²C Controller include:

- Transmit and receive operation in Master mode
- Maximum data rate of 400 kbps
- 7-bit and 10-bit addressing modes
- Unrestricted number of data bytes transmitted per transfer

I²C Controller's Operating Modes

The I²C Controller operates in Master mode to transmit and receive data. Only a single Master is supported. Arbitration between two masters must be accomplished in the software.

I²C supports the following data transfer operations:

- Master to a 7-bit Slave
- Master to a 10-bit Slave
- 7-bit Slave to Master
- 10-bit Slave to Master

Note: *The Z8 Encore! XP I²C Controller does not operate in Slave mode.*

I²C Interrupts— There are three interrupt sources for the I²C Controller:

- Transmit
- Receive
- Not Acknowledge Interrupts (NCKI)

For transmit interrupts to occur, the TXI bit must be set to 1 in the I²C Control register. Transmit interrupts occur under the following conditions when the transmit data register is empty:

- The I²C Controller is idle (not performing any operation)
- The Start bit is set and there is no valid data in the I²C Data register or in the I²C shift register to shift out
- The first bit of the address byte is shifted out and the RD bit of the I²C Status register is deasserted
- The first bit of a 10-bit address shifts out
- The first bit of the Write data is shifted out

Note: *Writing to the I²C Data register always clears a transmit interrupt.*

Receive interrupts occur when a byte of data is received by the I²C Master. Reading from the I²C Data register clears this interrupt. If no action is taken, the I²C Controller waits until this interrupt is cleared before performing any other action.

NCKI interrupts occur when a Not Acknowledge is received or transmitted, and neither the START nor the STOP bit are active. This interrupt can only be cleared by setting the START or STOP bit.

For details about the I²C Controller in Z8 Encore! XP devices, refer to the device-specific Product Specification documents listed in References on page 9.

Hardware Architecture

Figure 6 displays a block diagram to interface the I²C EEPROM device with a Z8 Encore! XP MCU.

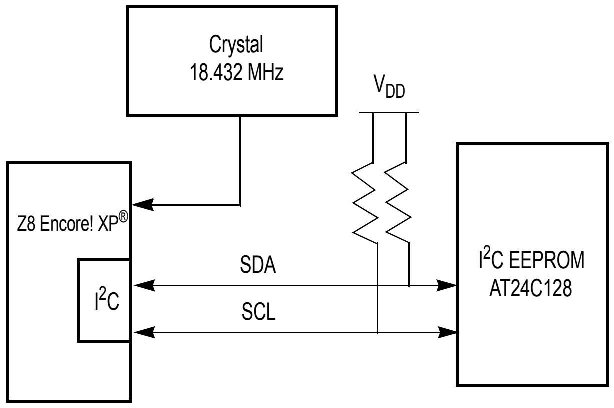


Figure 6. Interfacing I²C EEPROM Slave Device to the Z8 Encore! XP MCU

The AT24C128 device, an external I²C EEPROM, is interfaced with the Z8 Encore! XP MCU Development Board. For a detailed schematic, see Appendix B—Schematics on page 11. Figure 7 displays an instance of connecting multiple devices on the same I²C bus.

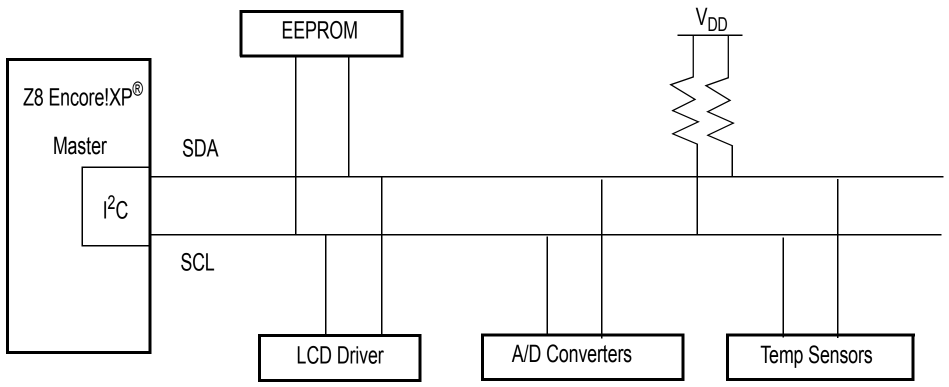


Figure 7. Connecting the Z8 Encore! XP MCU to Many I²C Peripherals

For a complete description of the Z8 Encore! XP I²C hardware and 10-bit addressing techniques of the I²C devices, refer to the Z8 Encore! XP MCU Product Specification documents listed in References on page 9.

Software Implementation

The software implementation is a set of APIs and API services developed for use by the Z8 Encore! XP MCU's I²C on-chip peripheral. The API services are listed below:

- Initialize the I²C ports and set up the baud rate for I²C data transfer
- Create the Start condition on the I²C bus
- Load the data byte (to be transmitted) to the I²C data register
- Check whether the data from the Master is completely transmitted
- Check whether an Acknowledge is transmitted or received by the Master when a byte of data is received or transmitted
- Send a Not Acknowledge (NACK) command to the Slave from the Master (occurs when the Master receives final byte of data from the Slave)
- Read the received data from the Slave
- Check whether reception of data from the Slave is complete
- Create the Stop condition on the I²C bus

Using the above API services, the device-specific (EEPROM) APIs were constructed to perform the following operations:

- Write one byte of data to the I²C EEPROM
- Read one byte of data from the I²C EEPROM
- Write a page to the I²C EEPROM
- Read a page of data from the I²C EEPROM (sequential Read)

These APIs appear in the source code file `eeprom.c`. A description of the I²C EEPROM APIs are available in Appendix D on page 14.

Figure 11 on page 12 displays the flow of the EEPROM Read sequence, and Figure 12 on page 13 displays the EEPROM Write sequence.

Testing

This section explains the test procedures to verify the APIs developed to demonstrate the Z8 Encore! XP MCU's I²C capabilities.

The `main.c` file contains code to communicate with the HyperTerminal. Data transfer is initiated in any one of the different modes at any given time by selecting the appropriate HyperTerminal menu option. When data reading is initiated from the I²C

EEPROM, the corresponding data is printed in the HyperTerminal display.

Setup

For the setup to test the I^{2}C bus Master implementation using the Z8 Encore! XP MCU, see Figure 6 on page 5.

Equipment Used

The following equipment are used for the setup:

- Z8 Encore! XP Development Kits featuring Z8F64xx Development Boards
- ZDS II–IDE with C Compiler/Simulator/ Online debugger
- A PC equipped with HyperTerminal configured to the following settings:
  - 9600 baud rate
  - 8 data bits
  - No parity
  - One stop bit
  - No flow control

Procedure

Follow the steps below to test the I^{2}C Bus Master application.

1. Connect the external I^{2}C EEPROM (AT24C128) device to the Z8 Encore! XP MCU Development Kit, as displayed in Appendix B—Schematics on page 11.
2. Download the project file (available in the `AN0126-SC01.zip` file) to the Z8 Encore! XP Development Board.
3. Execute the program. The user interface routine prints a list of I^{2}C data transfer modes in the HyperTerminal window.
4. Select a data transfer mode. Relevant APIs are called automatically and data transfer occurs between the external I^{2}C EEPROM device and the Z8 Encore! XP MCU.

Test Results

Data was successfully written to and read from the EEPROM device in the following modes:

- Byte Write
- Byte Read
- Page Write
- Sequential Read

All of these operations were performed at the following frequencies:

- 100 kHz
- 200 kHz
- 300 kHz
- 400 kHz

The waveforms for these byte Read and Write operations were captured on a logic analyzer and are displayed in Figure 8 and Figure 9 on page 8.

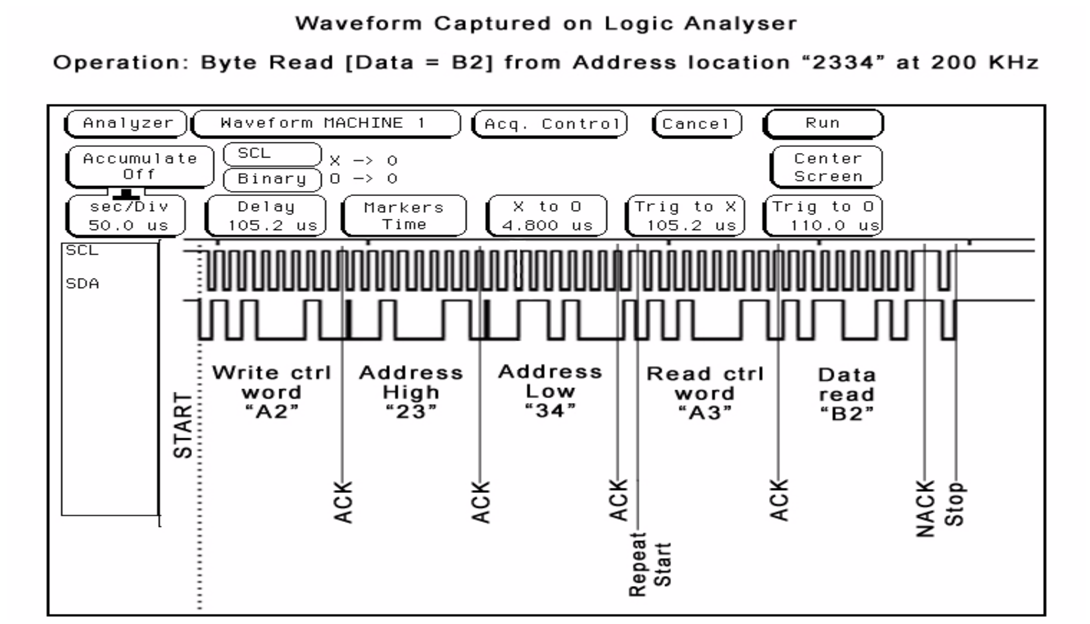


Figure 8. Waveform of a Byte Read Operation

Note: *Read and Write control words are specific to the AT24C128 I^{2}C EEPROM.*

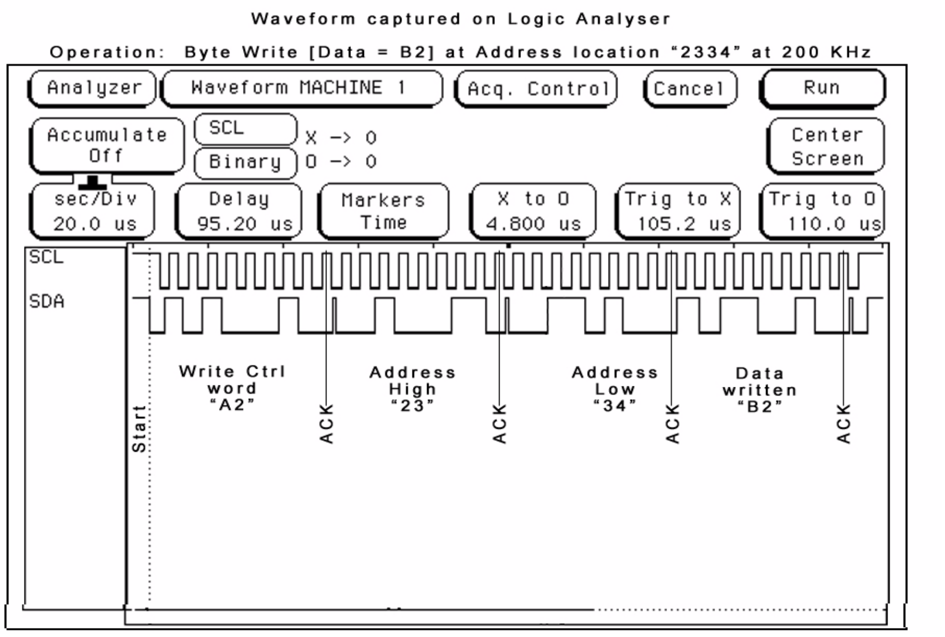


Figure 9. Waveform of a Byte Write Operation

Note: *The Write control word is specific to the AT24C128 I^{2}C EEPROM.*

Summary

This Application Note provides an overview of the I^2C bus along with details describing the interface between the Z8 Encore! XP I^2C on-chip peripheral and Atmel's AT24C128 I^2C EEPROM device.

The presence of I^2C hardware on the Z8 Encore! XP MCU makes it very easy to interface with any I^2C device. Implementing an I^2C function to facilitate communicate between these devices causes minimum overhead on the processor as compared to a similar I^2C implementation on a microcontroller that lacks on-chip I^2C hardware.

The APIs presented in the software section can be used directly, with little or no modification, to interface with any other I^2C device.

References

The documents associated with Z8 Encore! XP MCU and ZDS II are provided below. These documents are available for download at zilog.com.

- Z8 Encore! XP® 64K Series Flash Microcontrollers Product Specification (PS0199)
- Zilog Developer Studio II–Z8 Encore!® User Manual (UM0130)
- eZ8™ CPU Core User Manual (UM0128)
- AT24C128 data sheet: www.atmel.com (ATMEL AT24C128 I2C EEPROM)
- Philips Semiconductor data sheet: www.philips.com (Philips I2C bus)

Appendix A—Glossary

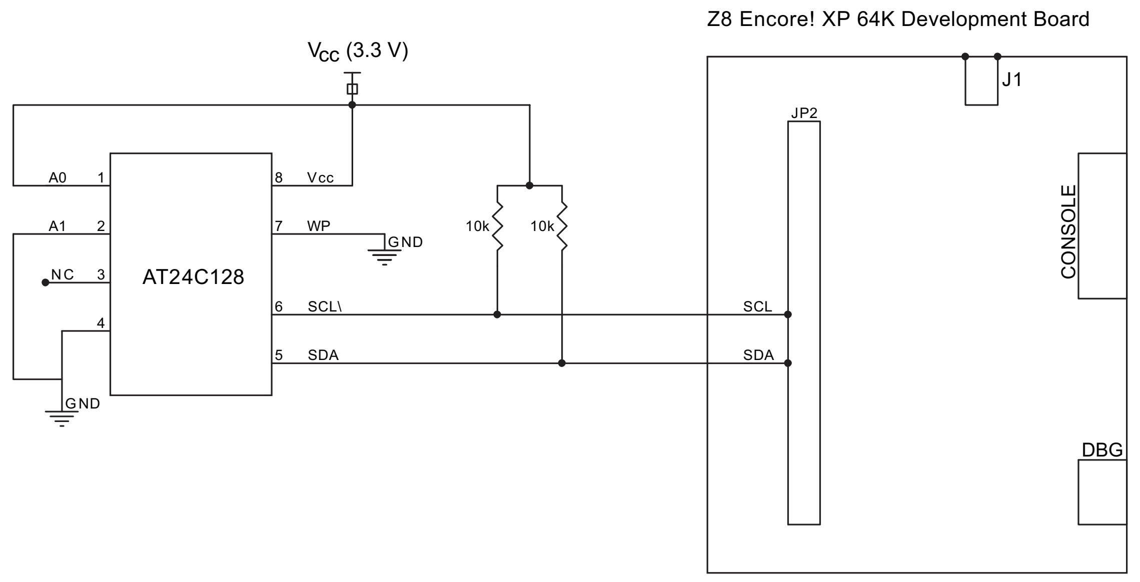
Definitions for terms and abbreviations used in this Application Note that are not commonly used are listed in Table 1.

Table 1. Glossary

Term/Abbreviation	Definition
Arbitration	Procedure to ensure that if more than one Master simultaneously tries to control the bus, only one Master is allowed to control the bus and the message is not corrupted.
Master	The device that initiates a transfer, generates clock signals, and terminates a transfer.
Multimaster	More than one Master controlling the I^2C bus at the same time without corrupting the message.
Receiver	The device that receives the data from the bus.
Slave	The device addressed by a Master in this application.
Transmitter	The device that transmits data to the bus.

Appendix B—Schematics

Figure 10 displays a schematic diagram that implements an I²C Bus Master using Zilog's Z8 Encore! XP® 64K Series Development Board and Atmel's AT24C128 EEPROM device.



J1: Power Supply Input to Development Board
J2: Connector on the Development Board

CONSOLE: Connection to PC serial port
DBG: Connection PC serial port

A0 and A1 are the address lines for AT24C128

Connect V_{CC} and GND to AT24C128 from the Z8 Encore! XP Development Board

Figure 10. Interfacing the AT24C128 I²C EEPROM to the Z8 Encore! XP MCU Development Board

Appendix C—Flowcharts

This appendix displays the flowcharts of EEPROM Read cycle and EEPROM write cycle (see Figure 11 and Figure 12). Figure 11 displays the flow of the EEPROM Read cycle.

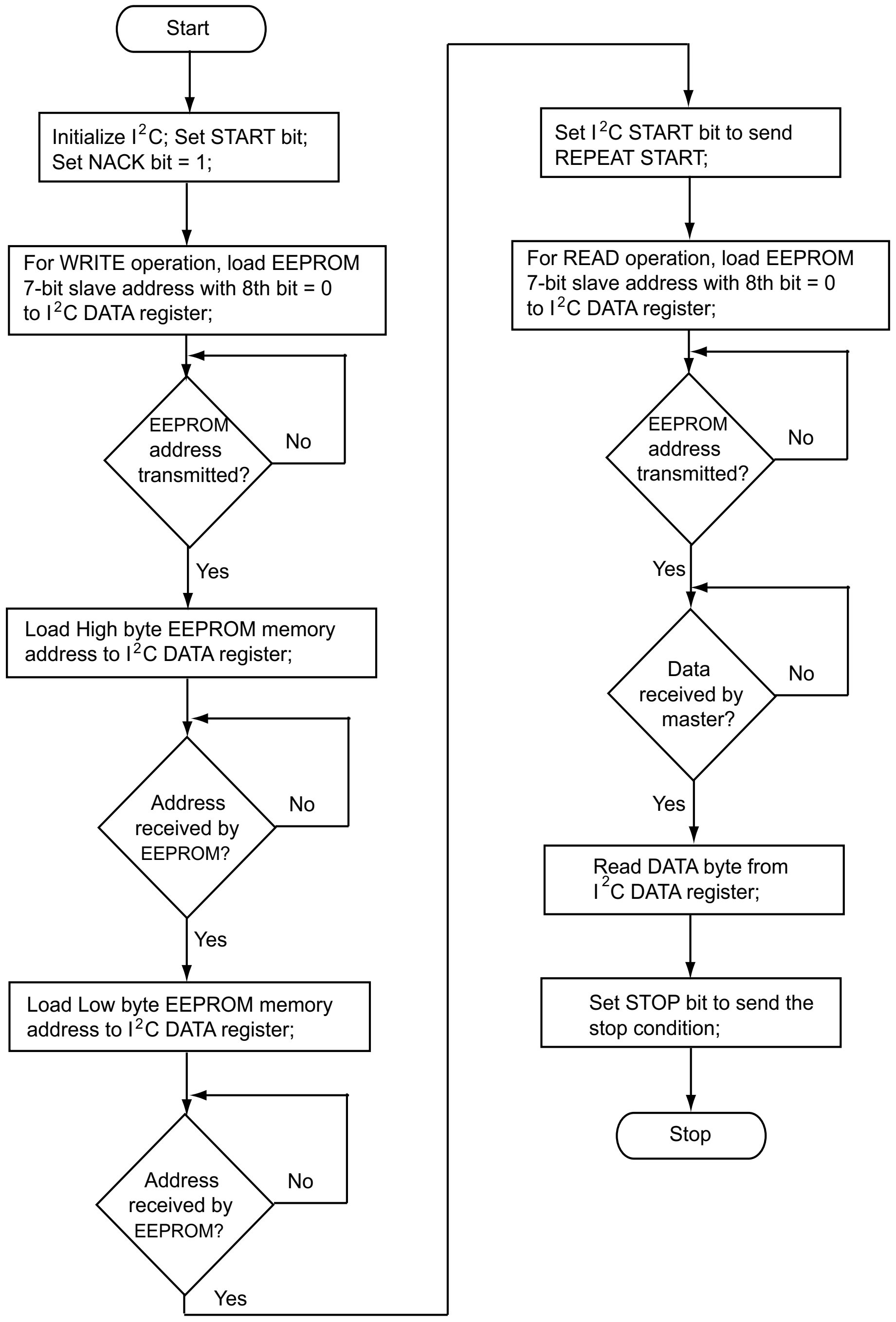

Start
Initialize I²C; Set START bit; Set NACK bit = 1;
For WRITE operation, load EEPROM 7-bit slave address with 8th bit = 0 to I²C DATA register;
EEPROM address transmitted?
No
Yes
Load High byte EEPROM memory address to I²C DATA register;
Address received by EEPROM?
No
Yes
Load Low byte EEPROM memory address to I²C DATA register;
Address received by EEPROM?
No
Yes
Set I²C START bit to send REPEAT START;
For READ operation, load EEPROM 7-bit slave address with 8th bit = 0 to I²C DATA register;
EEPROM address transmitted?
No
Yes
Data received by master?
No
Yes
Read DATA byte from I²C DATA register;
Set STOP bit to send the stop condition;
Stop

Figure 11. EEPROM Read Cycle Flowchart

Figure 12 displays the flow of the EEPROM Write cycle.

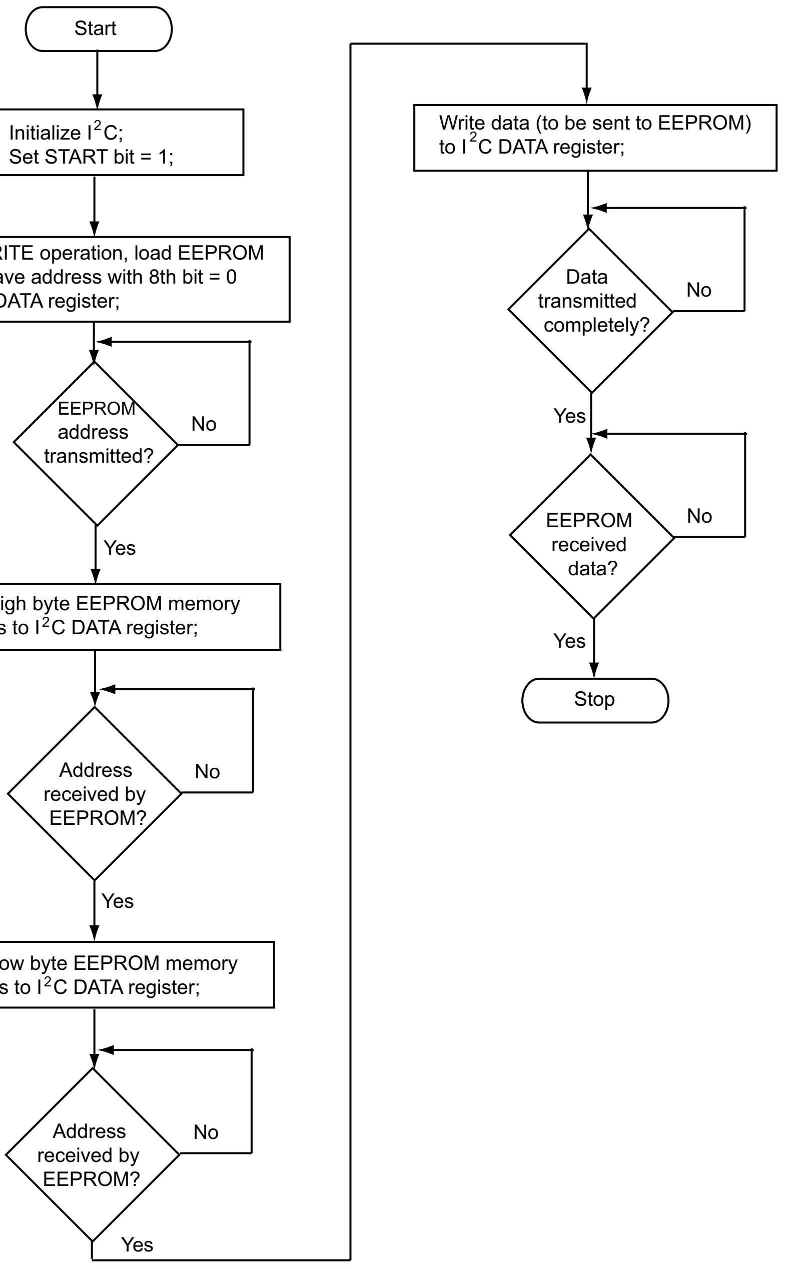

Start
Initialize I2C; Set START bit = 1;
For WRITE operation, load EEPROM 7-bit slave address with 8th bit = 0 to I2C DATA register;
EEPROM address transmitted?
No
Yes
Load High byte EEPROM memory address to I2C DATA register;
Address received by EEPROM?
No
Yes
Load Low byte EEPROM memory address to I2C DATA register;
Address received by EEPROM?
No
Yes
Write data (to be sent to EEPROM) to I2C DATA register;
Data transmitted completely?
No
Yes
EEPROM received data?
No
Yes
Stop

Figure 12. EEPROM Write Cycle Flowchart

Appendix D—API Description

This section provides detailed explanation of the APIs that initiate data transfer with the Atmel I^{2}C EEPROM.Table 2 lists the APIs that initiate data transfer with the Atmel I^{2}C EEPROM.

Table 2. I^{2}C EEPROM APIs

`Read_EEPROM( )`	Reads byte of data to I^{2}C EEPROM.
`Write_EEPROM( )`	Writes byte of data from I^{2}C EEPROM.
`page_Read_EEPROM( )`	Performs a Page Read/Sequential Read operation.
`page_write_EEPROM( )`	Performs a Page Write operation.

Descriptions for the I^{2}C EEPROM APIs begin on the next page. The include files for these APIs are:

- `i2c.h`
- `define.h`
- `ez8.h`

For more details about the APIs, refer to `AN0126-SC01.zip` file available on www.zilog.com.

Read_EEPROM( )

Prototype

```
unsigned char Read_EEPROM
(
unsigned char addrH,
unsigned char addrL,
char *status
)
```

Description

The `Read_EEPROM()` API reads a byte of data from an I^{2}C EEPROM at a specified address.

Argument(s)

`addrH`	High byte memory address
`addrL`	Low byte memory address
`*status`	Holds the success or failure value

Note: *The High and Low byte memory address together constitute the complete address from where the data is to be read.*

Return Value(s)

Returns the byte of data that was read.

Example

```
char status;
Read_EEPROM (0x12,0x34,&status);
```

The API reads the data from memory location `1234h` of the I^{2}C EEPROM and returns the data that is read. Be careful not to transmit invalid addresses (beyond the addressable range for the device) as arguments.

Write_EEPROM( )

Prototype

```
void Write_EEPROM
(
unsigned char addrH,
unsigned char addrL,
unsigned char data,
char *status
)
```

Description

The `Write_EEPROM()` API writes a byte of data to the I^{2}C EEPROM at a specified address.

Argument(s)

`addrH`	High byte memory address
`addrL`	Low byte memory address
`data`	The data to be written to the specified address
`*status`	Holds the success or failure value

Note: *The High and Low byte memory address together constitute the complete address to where the data is to be written.*

Return Value(s)

None.

Example

```
char status;
Write_EEPROM (0x12, 0x34, 0xAB, &status);
```

This API writes data `AB` to memory location `1234h` of the I^{2}C EEPROM. Be careful not to send invalid addresses (beyond the addressable range for the device) as arguments.

page_Read_EEPROM( )

Prototype

```
void page_Read_EEPROM
(
unsigned char addrH,
unsigned char addrL,
unsigned char *temp,
unsigned char no_of_bytes,
char *status
)
```

Description

The `page_Read_EEPROM()` API performs Page Read/Sequential Read operations, where one page is 64 bytes. The data is read into an array called `temp` which is used for additional computation.

Argument(s)

`addrH`	High byte memory address
`addrL`	Low byte memory address
`*temp`	pointer to an array that holds the data that was read
`no_of_bytes`	Number of bytes to be read (subject to maximum of 64 bytes)
`*status`	Holds the success or failure value

Return Value(s)

None.

Example

```
char status;
Page_Read_EEPROM (0x12,0x34,temp,5,&status);
```

The API reads 5 bytes of data from the I^{2}C EEPROM, starting from location `1234h`. The data read is written to the `temp` array.

page_write_EEPROM( )

Prototype

```
void page_write_EEPROM
(
unsigned char addrH,
unsigned char addrL,
unsigned char *temp,
char *status
)
```

Description

The `page_write_EEPROM` API performs the Page Write operation, where one page for the EEPROM device is 64 bytes. When setting page numbers, the programmer must ensure that each page number is directed to its appropriate page.

Argument(s)

`addrH`	High byte memory address
`addrL`	Low byte memory address
`*data`	pointer to an array which holds the data to be written to EEPROM page
`*status`	Holds the success or failure value

Return Value(s)

None.

Example

```
char status;
page_write_eeprom (0x12, 0x34, data,&status);
```

The API writes data from an array called `temp` to the I^{2}C EEPROM, starting from location `1234h`. The data must be written within the page boundaries only.