zilog
Embedded in Life
An IXYS Company

Application Note

Getting Started with ZNEO®-Based MCUs

AN031201-0211

Abstract

This application note describes how to use the peripheral elements of Zilog's Z16FMC Series of Flash Motor Control MCUs, which include an ADC, a UART, an I^2C, a PWM, an SPI and 16-bit timers.

Note: The source code files associated with this application note, AN0312-SC01, have been tested with ZDSII version 4.11.1.

Table of Contents

Features

The Z16FMC Series Development Board is a development and prototyping board for the Z16FMC Series MCU. The board allows you to evaluate the features of the Z16FMC Series MCU and to develop an application before building your hardware.

The Z16FMC Series of Flash microcontrollers are based on Zilog's advanced ZNEO 16-bit CPU core. This Z16FMC Series sets a new standard for performance and efficiency with a 24-bit address bus and a 16-bit data bus. The Z16FMC Series' External Interface allows seamless connection to external memory and peripherals. A 24-bit address bus and selectable 8-bit or 16-bit data bus allows parallel access up to 16MB.

The development board contains circuitry to support and present all of the key features of the Z16FMC Series, including:

- 20MHz ZNEO CPU Core
- Up to 128KB internal Flash program memory with 16-bit access and in-circuit programming capability
- Up to 4KB internal RAM with 16-bit access
- An external interface that allows a seamless connection to external data memory and peripherals with:
  - 6 chip selects with programmable wait states
  - 24-bit address bus which supports up to 16 MB
  - Selectable 8-bit or 16-bit data bus widths
  - Programmable Chip Select signal polarity
  - ISA-compatible mode
- Up to 12 channels of 10-bit ADC
- Operational Amplifier
- Analog Comparator
- 4-channel DMA controller which supports internal or external DMA requests
- Two full-duplex 9-bit Universal Asynchronous Receivers/Transmitters (UARTs) with support for Local Interconnect Network (LIN) and Infrared Data Association (IrDA) functionality
- Internal Precision Oscillator (IPO)

- I^2C Master-Slave controller
- Enhanced Serial Peripheral Interface (ESPI) controller
- 12-bit Pulse Width Modulator (PWM) module with three complementary pairs or six independent PWM outputs with deadband generation and fault trip input
- Three standard 16-bit timers with capture, compare and PWM capability
- Watchdog Timer (WDT) with internal RC oscillator
- Up to 76 I/O pins
- Up to 24 interrupts with programmable priority
- Single-pin on-chip debugger
- Power-On Reset (POR)
- Voltage Brownout Protection (VBO)
- 2.7V to 3.6V operating voltage with 5V tolerant inputs
- 0°C to +70°C standard temperature, –40°C to +105°C extended temperature, and –40°C to +125°C automotive operating ranges

For more information, refer to the Z16FMC Series Product Specification (PS0220), available for download at http://www.zilog.com.

Discussion

The Z16FMC Series devices feature a 12-channel successive-approximation analog-to-digital converter (ADC). This ADC converts an analog input signal to a 10-bit binary number. The features of the ADC include:

- 12 analog input sources multiplexed with GPIO ports
- Fast conversion time – less than 5µs
- Programmable timing controls
- Interrupt on conversion complete
- Internal voltage reference generator
- Internal reference voltage available externally
- Ability to supply external reference voltage
- Ability to do simultaneous or independent conversions

ADC Operation

The architecture of an Z16FMC Series MCU consists of a 12-input multiplexer, sample-and-hold amplifier and 10-bit successive approximation ADC. The ADC digitizes the signal on a selected channel and stores this digitized data in the ADC data registers. In an environment with high electrical noise, an external RC filter must be added at the input pins to reduce high frequency noise. See Figure 1.

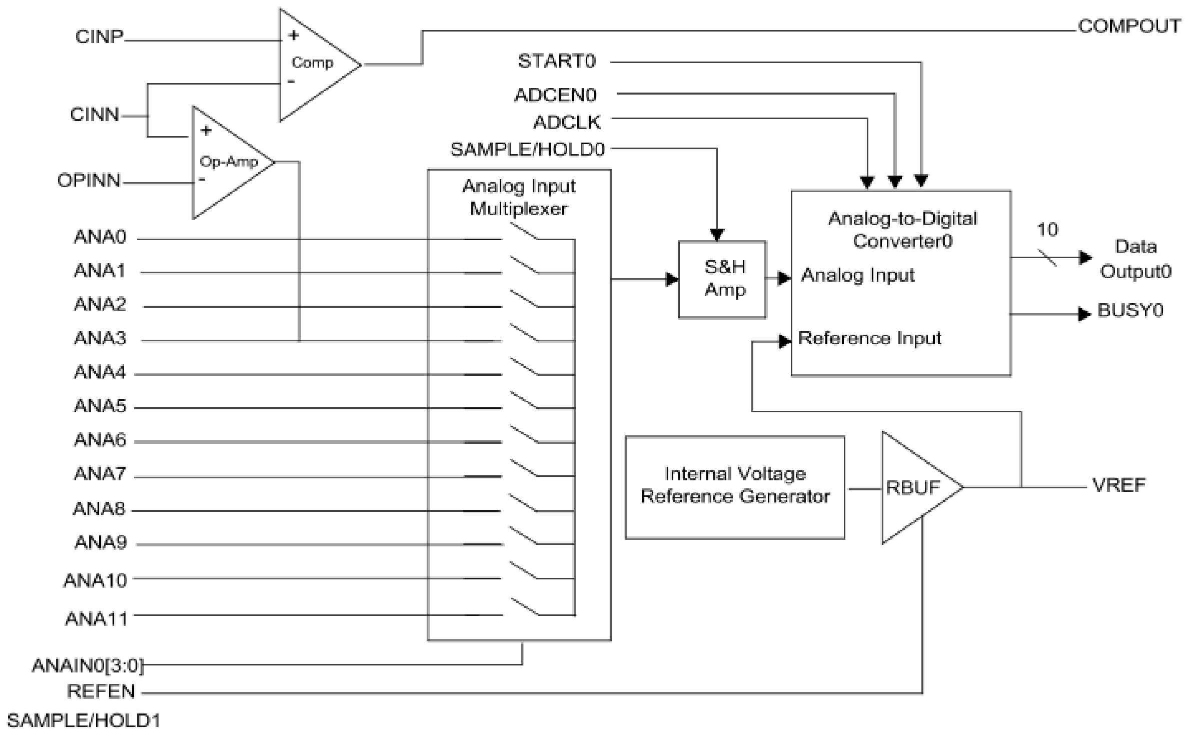


Figure 1. ADC Block Diagram

The ADC converts the analog input, ANAx, to a 10-bit digital representation. The equation used for calculating the digital value is represented by:

$$\text{ADC Output} = (2^{10} - 1) * (\text{ANAx}/\text{VREF})$$

Assuming zero gain and offset errors, any voltage outside the ADC input limits of AV_{SS} and V_{REF} returns all 0s or 1s, respectively.

A new conversion is initiated by either a software write to the ADC Control Register's START bit or by a PWM trigger. For this application note, the START bit is used for initiating ADC conversion.

Initiating a new conversion stops any conversion currently in progress and begins a new conversion. To avoid disrupting a conversion already in progress, the START bit is read to indicate ADC operation status, whether busy or available.

For purposes of this application, the Z16FMC Series Development Board, shown in Figure 2, is used to test the ADC. Potentiometer R10, circled in the figure, provides the analog input voltage for ADC Channel 5 (ANA5).

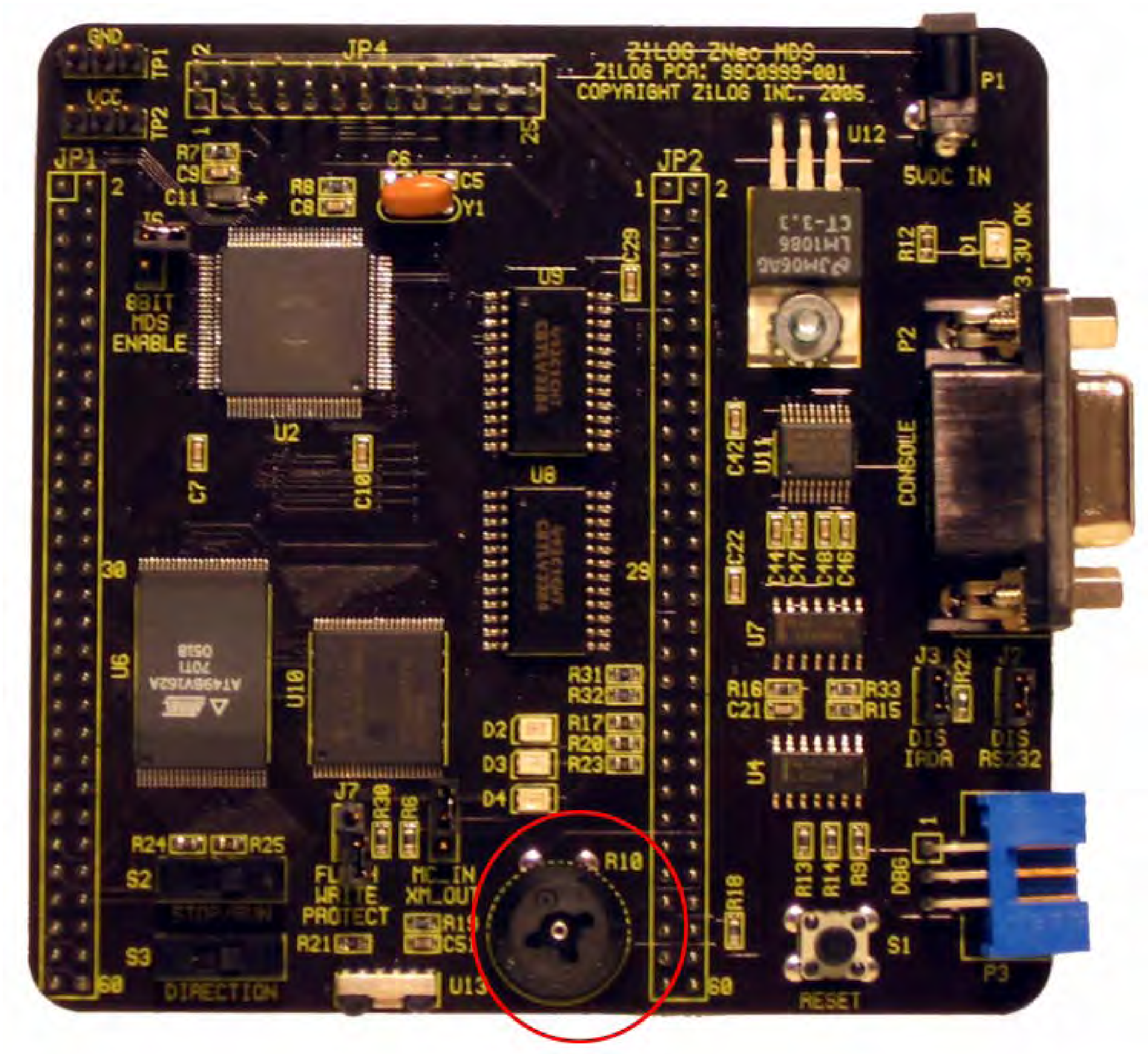


Figure 2. The Z16FMC Series Development Board

ADC Hardware Architecture

The input for the ADC is from the voltage divider circuit. It consists of potentiometer, a series resistor and a capacitor for filtering. Varying the resistance of the potentiometer varies the input voltage to the ADC. See Figure 3.

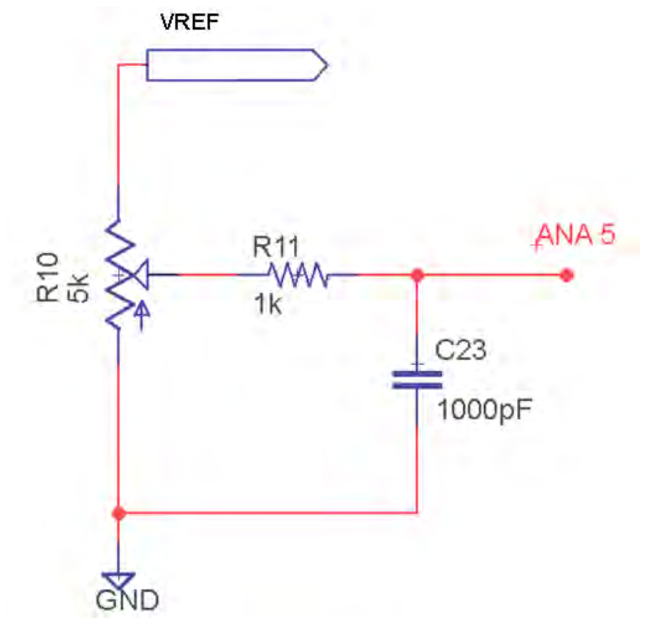


Figure 3. Analog Input Circuit

ADC Software Implementation

The software implementation is illustrated in Appendix B. Flowcharts (see Figure 37 on page 62). Flow starts with ADC, UART and clock initialization. After ADC is enabled, the START0 bit flag of the ADC Control Register is monitored until it becomes logic 0, indicating that conversion is completed. After conversion is completed, data from the ADC Data Byte Register is displayed to the Hyperterminal via UART. A new conversion is again initiated.

ADC Initialization

ADC initialization involves setting the port as analog input. Port B[7:0] and Port H[3:0] can be set as analog input through their alternate function. Since ANA5 is used, Port B5 is set as analog input through this code line:

```
PBAFL = PB_ANA5; //PB5 as analog input ANA5
```

Using that same line, other analog input on Port B can be selected. Table 1 shows the value for the PBAFL to set the Port B pins as analog input. Take note that Port B has ANA0 to ANA7 and Port H has ANA8 to ANA11. This means that setting Port H as analog input is through PHAF. As an example, if ANA8 is used as analog input, setting Port H0 as analog input is:

```
PHAF = PH_ANA8; //PH0 as analog input ANA8
```

Table 1. Analog Input Sources

Port	Analog Input Channel
PB_ANA0	ANA0
PB_ANA1	ANA1
PB_ANA2	ANA2
PB_ANA3	ANA3
PB_ANA4	ANA4
PB_ANA5	ANA5
PB_ANA6	ANA6
PB_ANA7	ANA7
PH_ANA8	ANA8
PH_ANA9	ANA9
PH_ANA10	ANA10
PH_ANA11	ANA11

Table 2 shows the functions of each bit on the ADC Control Register (ADC0CTL). This control register is used to initiates ADC conversion and provides ADC status information.

Table 2. ADC Control Register

Bits	7	6	5	4	3	2	1	0
Field	START0	CVTRD0	REFEN	ADC0EN	ANAIN0[3:0]			
RESET	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
ADDR	FF–E500H							

A START0 bit is used to monitor the ADC conversion status. If this bit is a logic 1, ADC conversion is in progress. If this bit becomes a logic 0, conversion is completed. Aside from monitoring ADC conversion status, this bit can also be used to start ADC conversion by writing a logic 1 on it.

CVTRD0 bit is used to operate ADC normally by writing logic 0 on it. Setting this bit to logic 1 means that ADC data from analog input ANA0 up to analog input defined on ADC0MAX register is read one at a time.

REFEN bit is used to set the source of the voltage reference. If it is logic 0, external voltage reference is used and if it is logic 1, internal voltage reference is enabled.

ADC0EN bit is used to enable ADC for normal operation by setting it to logic 1. If logic 0 is written on it, ADC is disabled for low power operation.

ANAIN0 bits are used to select the analog input source.

To set the ADC, this line of code is used:

```
ADC0CTL = ADC_ENABLE_NORMAL | ANA5;
```

This means that analog input ANA5 is selected, the use of external voltage reference is enabled, and ADC operates normally. In selecting the analog input, ANA0 to ANA11 can be used in place of ANA5. To enable internal voltage reference, INT_REF should be "ORed" like this:

```
ADC0CTL = ADC_ENABLE_NORMAL | ANA5 | INT_REF;
```

To start ADC conversion, or to set `START0` to `ADC0CTL`:

```
ADC0CTL |= START0;
```

Note: The variables ANA5, PB_ANA5 and I NT_REF are defined in `main.h`.

The Sample Settling Time register is used to set the length of time from the SAMPLE/HOLD signal to the START signal, when conversion begins; see Figure 4. The number of clock cycles required for settling varies from system to system depending on the system clock period used. This register must contain the number of clocks required to meet a 0.5µs minimum settling time.

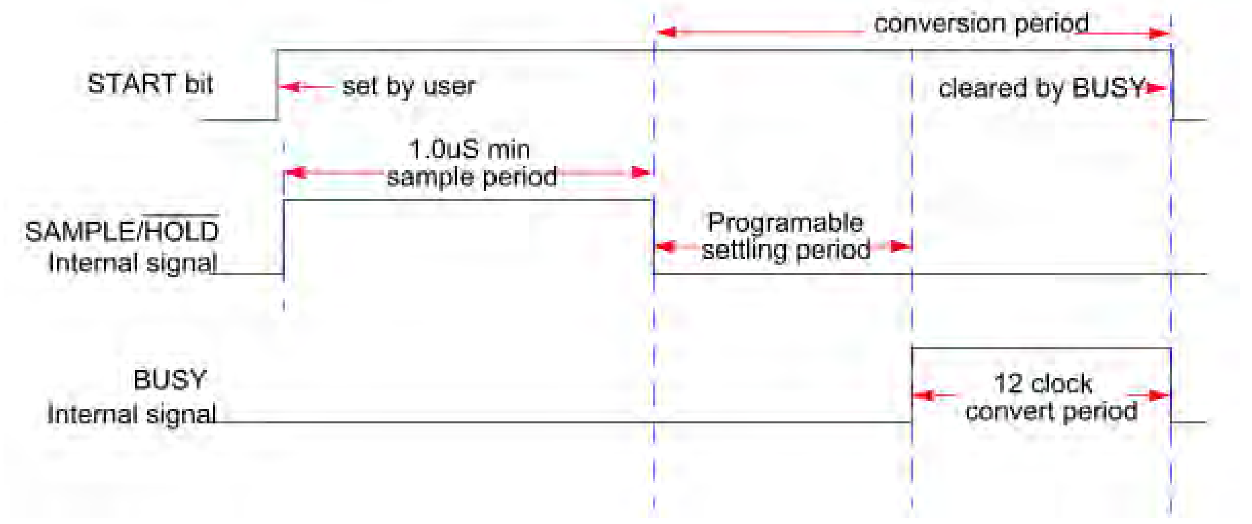


Figure 4. ADC Timing Diagram

Values for ADCSST register ranges from `0x00` up to `0x1F`. To determine the value for ADCSST, this condition must be satisfied:

$$0.5\ \mu s \leq ADCSST * (1 / \text{system clock frequency})$$

For the internal clock frequency of 5,529,600Hz, a minimum value of `0x03` for ADCSST can be used.

The sample time register is used to program the length of active time for the sample once a conversion begins by setting the START bit in the ADC control register or initiated by the PWM. The number of system clock cycles required for sample time varies from system to system depending on the clock period used. This register must contain the number of system clocks required to meet a 1 μs minimum sample time.

Values for ADCST register ranges from `0x00` up to `0x3F`. To determine the value for the ADCST, this condition must be satisfied.

$$1\ \mu s \leq ADCST * (1 / \text{system clock frequency})$$

For the internal clock frequency of 5,529,600 Hz, a minimum value of `0x06` for ADCST can be used.

The ADC Clock Prescale register is used to provide a divided system clock to the ADC. When this register is programmed with 0H or NODIV (as defined in main.h), the system clock is used for the ADC Clock.

ADCCP value divides the system clock frequency for ADC clock. The following are the values for ADCCP:

- DIV2 to divide system clock by 2
- DIV4 to divide system clock by 4
- DIV8 to divide system clock by 8
- DIV16 to divide system clock by 16

In a typical configuration, ADCCP has DIV4 when the system clock frequency is 20MHz. As example:

```
ADCCP = DIV4;
```

Testing ADC Operation

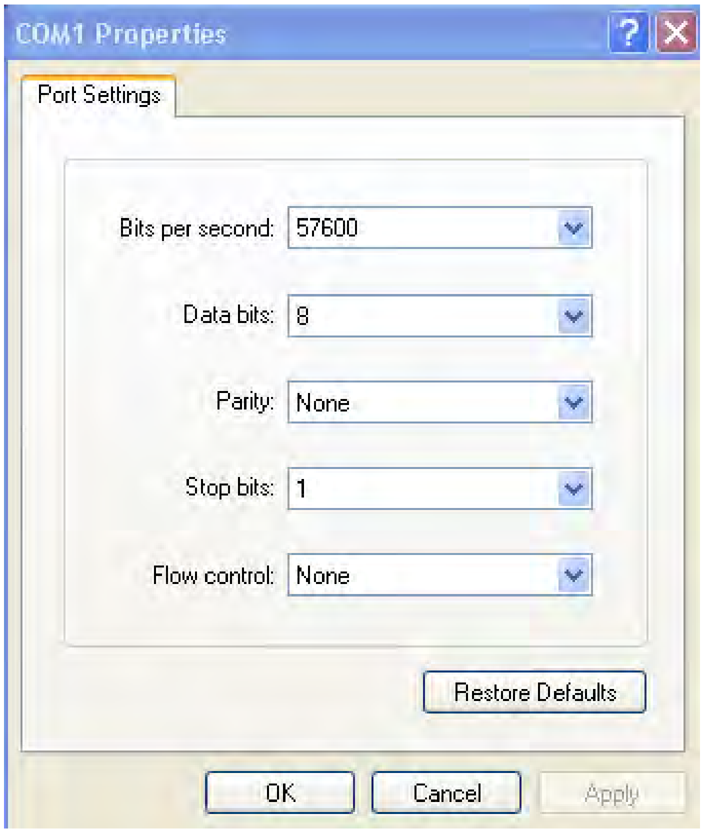
This section provides information about how to run the code and demonstrate this application note.

Equipment Used

- Z16FMC Series Development Kit
- Digital voltmeter to measure the input analog voltage on ANA5
- Power supply to provide external reference voltage
- PC to run the code and to display the ADC values in Hyperterminal

System Configuration

Configure Hyperterminal to reflect the settings shown in Figure 5.



Bits per second: 57600
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None

Figure 5. Hyperterminal Display Port Settings

Setup

Figure 6 illustrates the system setup to demonstrate the ADC peripheral. A potentiometer is used as an analog input for ANA5, a built-in feature of the Z16FMC Series Development Board. A digital voltmeter is used to measure the input voltage to ANA5. A PC running the Hyperterminal emulation program is used to display via the decimal and hexadecimal ADC values via UART0. A power supply for external reference voltage is connected to Pin 1 of the development board's JP4 jumper.

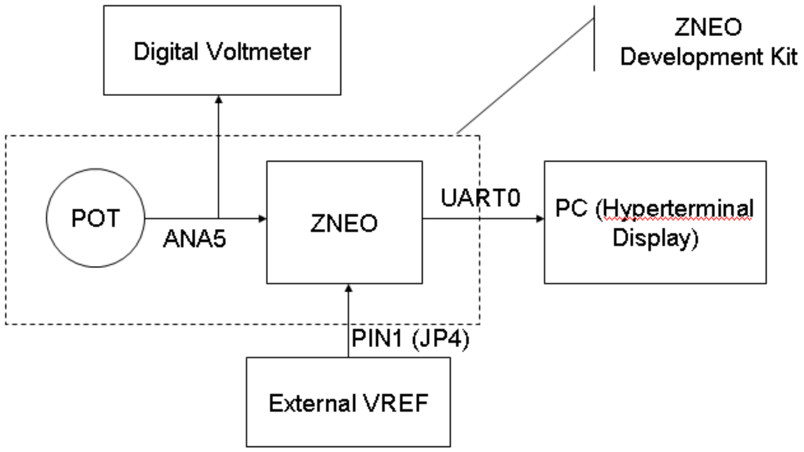
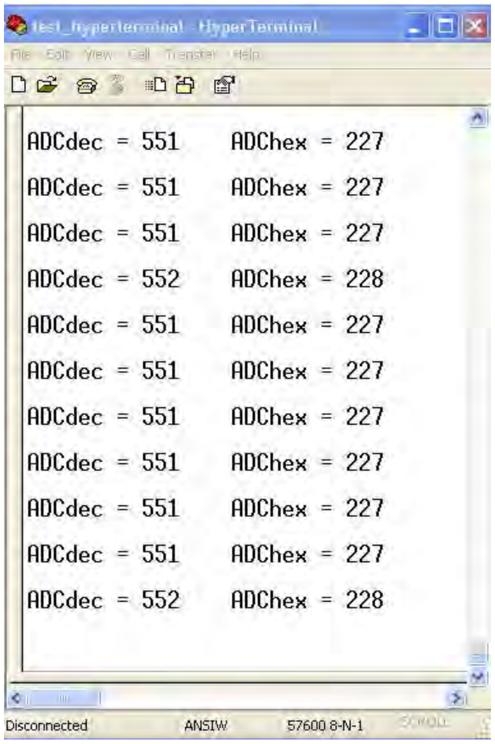


Figure 6. ADC Hardware Setup

Procedure

Observe the following brief instructions to test the operation of the ADC block of the Z16FMC MCU.

1. Open the AN0312-SC01 project in ZDSII for ZNEO.
2. On the code, EXT_CLOCK and EXT_VREF are defined on main.c. Comment out EXT_CLOCK if internal clock is intended to used. Also, if internal V_{REF} is used, comment out EXT_VREF.
3. Set up the Hyperterminal emulation program; refer to Figure 5 for the appropriate settings.
4. Follow the hardware set up in Figure 6.
5. Compile and run the code.
6. Sample view on Hyperterminal display is illustrated on Figure 7.


test_hyperterminal - HyperTerminal
File Edit View Call Transfer Help

```
ADCdec = 551    ADChex = 227
ADCdec = 551    ADChex = 227
ADCdec = 551    ADChex = 227
ADCdec = 552    ADChex = 228
ADCdec = 551    ADChex = 227
ADCdec = 551    ADChex = 227
ADCdec = 551    ADChex = 227
ADCdec = 551    ADChex = 227
ADCdec = 551    ADChex = 227
ADCdec = 551    ADChex = 227
ADCdec = 552    ADChex = 228
```

Disconnected
ANSIW
57600 8-N-1
SCROLL

Figure 7. Hyperterminal Display

Results

Figures 8 through 11 display the results of voltage input with respect to the following ideal and measured ADC values.

- Internal V_{REF} and Internal Clock
- Internal V_{REF} and External Clock
- External V_{REF} and Internal Clock
- External V_{REF} and External Clock

V_{IN}	ADC Value	
	Ideal	Measured
0	0	0
0.2	102	97
0.4	205	199
0.6	307	303
0.8	409	407
1	512	508
1.2	614	614
1.4	716	717
1.6	818	820
1.8	921	924
2	1023	1023

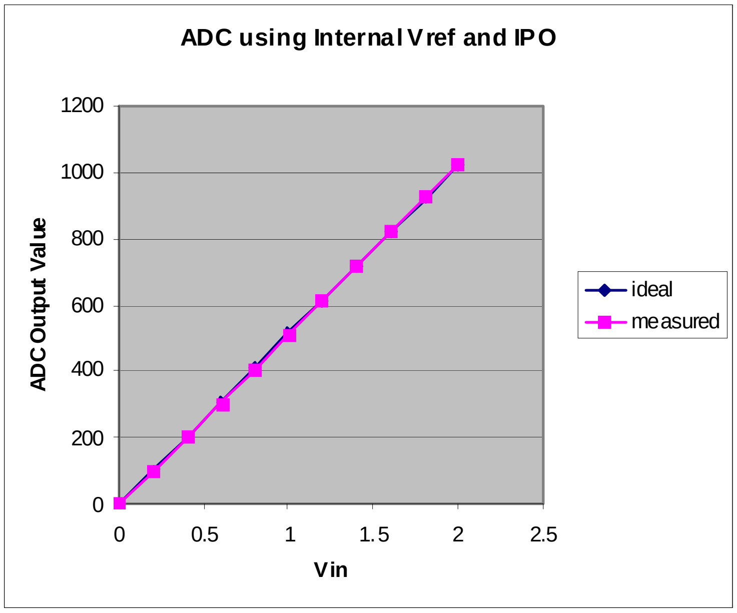


Figure 8. Internal ADC Reference Voltage and Internal Clock

V_{IN}	ADC Value	
	Ideal	Measured
0	0	1
0.2	102	97
0.4	205	200
0.6	307	302
0.8	409	405
1	512	507
1.2	614	610
1.4	716	713
1.6	818	816
1.8	921	917
2	1023	1022

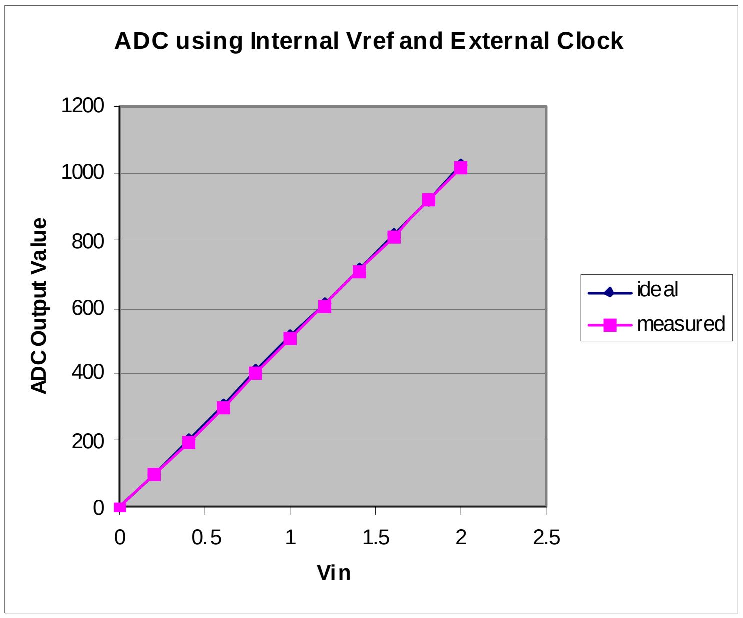


Figure 9. Internal ADC Reference Voltage and External Clock

V_{IN}	ADC Value	
	Ideal	Measured
0	0	1
0.2	102	97
0.4	205	200
0.6	307	302
0.8	409	405
1	512	507
1.2	614	610
1.4	716	713
1.6	818	816
1.8	921	917
2	1023	1022

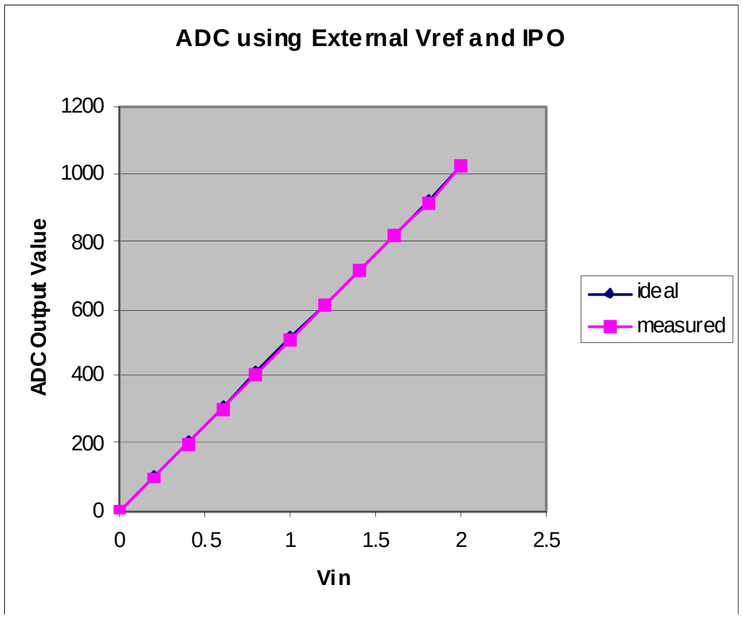


Figure 10. External ADC Reference Voltage and Internal Clock

V_{IN}	ADC Value	
	Ideal	Measured
0	0	1
0.2	102	95
0.4	205	198
0.6	307	302
0.8	409	403
1	512	506
1.2	614	613
1.4	716	711
1.6	818	814
1.8	921	918
2	1023	1021

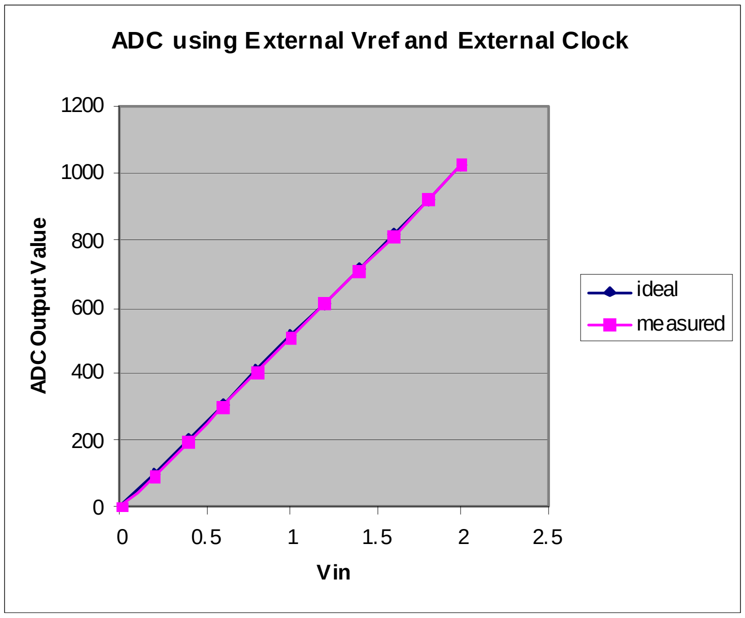


Figure 11. External ADC Reference Voltage and External Clock

UART Operation

Z16FMC MCUs provide two LIN-UART peripherals that are full-duplex communication channels capable of handling asynchronous data transfers. The LIN-UART includes the following features:

- 8-bit asynchronous data transfer
- Selectable even- or odd-parity generation and checking
- Option of one or two stop bits
- Separate transmit and receive interrupts or DMA requests
- Separate transmit and receive enables
- Framing, parity, overrun, and break detection
- 16-bit Baud Rate Generators (BRG)
- Selectable MULTIPROCESSOR (9-bit) mode with three configurable interrupt schemes
- Baud Rate Generator timer mode
- Driver enable output for external bus transceivers
- LIN protocol support for both MASTER and SLAVE modes:
  - Break generation and detection
  - Selectable slave autobaud
  - Check Tx versus Rx data when sending
- Configurable digital noise filter on receive data line

The LIN-UART consists of three primary functional blocks - transmitter, receiver, and BRG. The transmitter and the receiver function independently but use the same baud rate and data format. Figure 12 illustrates the LIN-UART architecture.

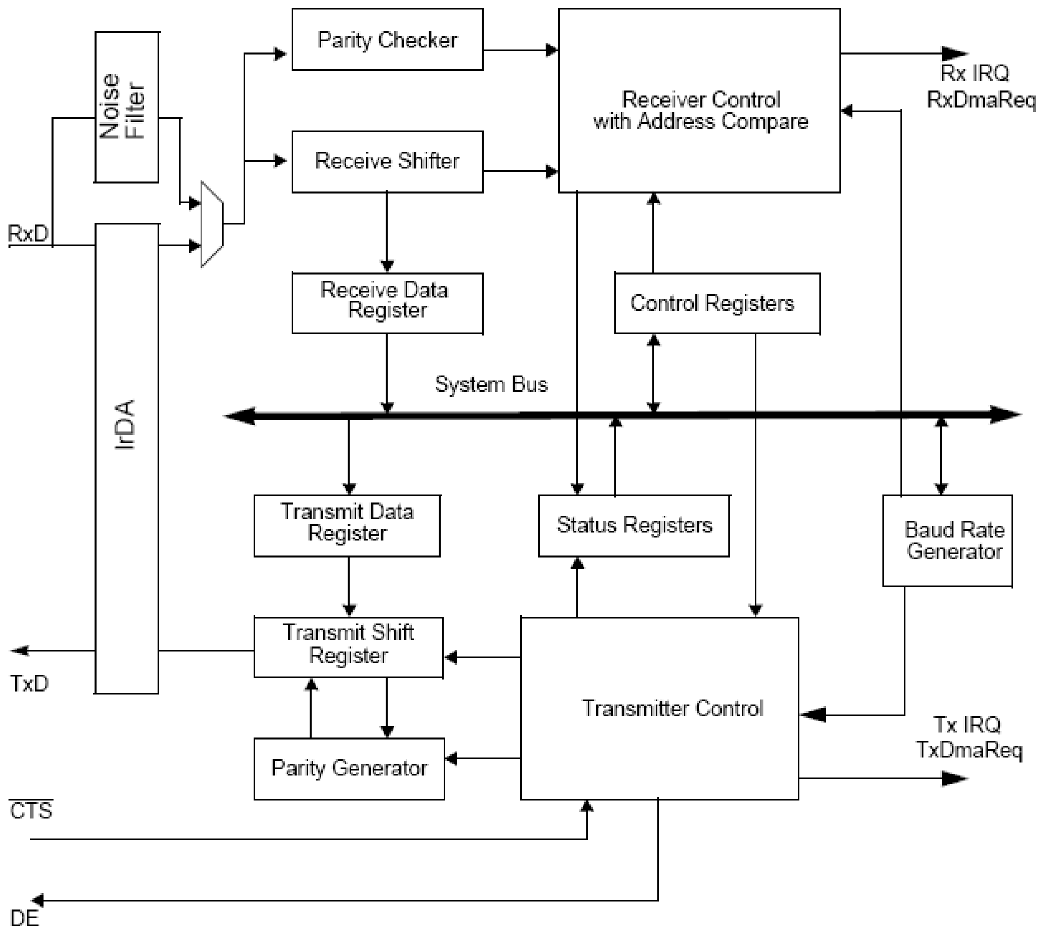


Figure 12. LIN-UART Block Diagram

LIN-UART Register Description

The ZNEO UART registers are briefly discussed in this section.

LIN-UART Control Register (UxCTL0, UxCTL1)

The LIN-UART Control 0 & 1 registers configure the basic properties of the LIN-UART's transmit and receive operations.

Table 3. LIN-UART Control Register 0

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Field	TEN	REN	CTSE	PEN	PSEL	SBRK	STOP	LBEN
RESET	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
ADDR	FF–E202H, FF–E212H							

Bit Position	**Description (Continued)**
7	**Transmit Enable (TEN)** Enables or disables the transmitter. Transmit enable can also be used in conjunction with CTS signal and CTSE bit.
6	**Receive Enable (REN)** Enables or disables the receiver.
5	**CTS Enable (CTSE)** Defines if CTS signal has no effect on the transmitter or if UART recognizes the CTS signal as an enable control for the transmitter.
4	**Parity Enable (PEN)** Enables or disables the parity bit.
3	**Parity Select (PSEL)** If parity is enabled, this bit specifies if odd- or even-parity is used.
2	**Send Break (SBRK)** Pauses or breaks data transmission.
1	**Stop Bit Select (STOP)** Defines the number of stop bits (1 or 2 stop bits) the transmitter should sent.
0	**Loop Back Enable (LBEN)** Determines if the transmitted data should be looped back to the receiver or not.

Table 4. LIN-UART Control Register 1

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Field	MPMD(1)	MPEN	MPMD(0)	MPST	DEPOL	BRGCTL	RDAIRQ	IREN
RESET	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
ADDR	FF–E203H, FF–E213Hwith MSEL = 000b							

Bit Position	**Description (Continued)**
7:5	**Multiprocessor Mode (MPMD[1:0])** If multiprocessor mode (MPEN) is enabled, these bits selects the interrupt scheme to be used.
6	**Multiprocessor Enable (MPEN)** Enables or disables the multiprocessor (9-bit) mode.
4	**Multiprocessor Bit Transmit (MPBT)** If multiprocessor mode (MPEN) is enabled, this bit determines what data to send at the multiprocessor bit location (9th bit) of the data stream.
3	**Driver Enable Polarity (DEPOL)** Determines if DE signal is active low or active high.
2	**Baud Rate Control (BRGCTL)** This bit causes different UART functionality depending on whether UART receiver is enabled or disabled. Generally, this bit defines whether the BRG generates an interrupt or not.
1	**Receive Data Interrupt Enable (RDAIRQ)** Determines whether the receiver generates an interrupt on (1) data receive and/or receiver errors, or (2) receiver errors only.
0	**Infrared Encoder/Decoder Enable (IREN)** Enables or disables infrared encoder/decoder.

LIN-UART Status Register 0 (UxSTAT0)

The LIN-UART Status 0 Register identifies the current UART operating configuration and status.

Table 5. LIN-UART Status Register 0

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Field	RDA	PE	OE	FE	BRKD	TDRE	TXE	CTS
RESET	0	0	0	0	0	1	1	1
R/W	R	R	R	R	R	R	R	R
ADDR	FF–E201H, FF–E211H							

Bit Position	**Description (Continued)**
7	**Receive Data Available (RDA)** Indicates if new data is received. Reading the UART Receive Data Register clears this bit.
6	**Parity Error (PE)** Indicates that a parity error has occurred. Reading the UART Receive Data Register clears this bit.
5	**Overrun Error (OE)** Indicates that an overrun error has occurred. Reading the UART Receive Data Register clears this bit.
4	**Framing Error (FE)** Indicates that a framing error occurred (no stop bit following data reception was detected). Reading the UART Receive Data Register clears this bit.
3	**Break Detect (BRKD)** Indicates that a break occurred.
2	**Transmit Data Register Empty (TDRE)** Indicates that the UART Transmit Data Register is empty and is ready for additional data. Writing to the UART Transmit Data Register clears this bit.
1	**Transmitter Empty (TXE)** Indicates that the Transmit Shift Register is empty and that character transmission is finished.
0	**CTS Signal (CTS)** Reading this bit returns the level of the CTS signal.

LIN-UART Baud Rate High and Low Byte Registers (UxBRH, UxBRL)

The LIN-UART Baud Rate High and Low Byte registers combine to create a 16-bit baud rate divisor value, which sets the data transmission rate of the LIN-UART. The UART must be disabled when updating the baud rate registers because high and low registers must be written independently.

Table 6. LIN-UART Baud Rate High Byte Register

Bits	7	6	5	4	3	2	1	0
Field	COMP_ADDR							
RESET	00H							
R/W	R/W							
ADDR	FF–E205H, FF–E215H							

Table 7. LIN-UART Baud Rate Low Byte Register

Bits	7	6	5	4	3	2	1	0
Field	BRL							
RESET	1							
R/W	R/W							
ADDR	FF–E207H, FF–E217H							

The baud rate divisor for a given UART data rate can be calculated using the following equation:

$$\text{UART Baud Rate Divisor Value (BRG)} = Round\left(\frac{\text{System Clock Frequency (Hz)}}{16 \times \text{UART Data Rate (bits/s)}}\right)$$

The baud rate error relative to the appropriate baud rate is calculated using the following equation:

$$\text{UART Baud Rate Error (\%)} = 100 \times \left(\frac{\text{Actual Data Rate} - \text{Desired Data Rate}}{\text{Desired Data Rate}}\right)$$

LIN-UART Transmit Data Register (UxTXD)

Data bytes written to the LIN-UART Transmit Data register are shifted out on the TXD pin. This register is write-only, and shares a register file address with the read-only LIN-UART Receive Data register.

Table 8. LIN-UART Transmit Data Register

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Field	TxD							
RESET	X							
R/W	W							
ADDR	FF–E200H, FF–E210H							

LIN-UART Receive Data Register (UxRXD)

Data bytes received through the RXD pin are stored in the LIN-UART Receive Data register. This register is read-only, and shares a register file address with the write-only LIN-UART Transmit Data register.

Table 9. LIN-UART Receive Data Register

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Field	RxD							
RESET	X							
R/W	R							
ADDR	FF–E200H, FF–E210H							

UART Software Implementation

The ZNEO CPU has 2 independent UART peripherals, UART0 (at Port A) and UART1 (at Port D), capable of handling standard serial interfaces. The source code associated with this document initializes both UART0 and UART1 for serial communications at 8 data bits, 1 stop bit, no parity, 9600bps (default), with the ZNEO running using the 5.5296MHz internal clock source. It also comes with a selectable baud rate definition configured using the 5.5296MHz clock and a definition to select whether to run the UART in polled or interrupt mode.

```
/////////////////////////////////////////////////////////////////
#define INTERRUPT_MODE// uncomment this line to use
// UART in polled mode

/*****************************************************************
******************************************************************
**
**DEFINES & MACROS
**
```

```
*****************************************************************
****************************************************************/
// using internal clock source 5.5296MHz
#define BAUD_300((UINT16)1152)
#define BAUD_600((UINT16)576)
#define BAUD_1200((UINT16)288)
#define BAUD_2400((UINT16)144)
#define BAUD_4800((UINT16)72)
#define BAUD_9600((UINT16)36)
#define BAUD_19200((UINT16)18)
#define BAUD_38400((UINT16)9)
#define BAUD_57600((UINT16)6)
#define BAUD_115200((UINT16)3)

#define BAUDRATEBAUD_9600 // change, as required
/////////////////////////////////////////////////////////////////
```

In the main program, the user is prompted to input a string then press enter when done. As soon as the program detects a line feed (`0x0A`) in the user input, it will echo back the input to the user. This process will continue to loop until the user decides to quit. The program is designed to accept a maximum of 64 characters (MAX_CHARS) from the user. For polled mode, this size is defined in main.c; while in interrupt mode, MAX_CHARS depends on the buffer's size, which is defined in rbuf.h.

When running in interrupt mode, the application uses a buffer defined in rbuf.c to store incoming and outgoing messages. Note, however, that both UART0 and UART1 use the same input and output buffers. This application is for demo purposes only and it does not intend to use both UARTs at the same time. The user must create separate buffers for UART0 and UART1, if attempting to use both UARTs at the same time. The relevant routines for interrupt mode are:

```
/////////////////////////////////////////////////////////////////
void UARTx_Init(void);// Initialize UART
void UARTx_StartTx(void);// Start UART transmission
/////////////////////////////////////////////////////////////////
```

The interrupt routines are declared as follows:

```
/////////////////////////////////////////////////////////////////
void interrupt UARTx_TxIsr(void) _At UARTx_TX
{
// if TXD buffer is not empty, do nothing
if(!(UxSTAT0 & 0x04))
                return;

// If there is data to Tx, get Tx data from output buffer
if( RBUF_GetLengthOutBuffer() > 0 )
                UxTXD = RBUF_GetByteFromOutBuffer();
}
```

```
void interrupt UARTx_RxIsr(void) _At UARTx_RX
{
UINT8 temp = UxRXD;
if((UxSTAT0 & 0x78) == 0x78)// ERROR occurred!!!
                return;     // read data to clear this bit
RBUF_AddByteToInBuffer(temp);// add Rx data to input buffer
}
/////////////////////////////////////////////////////////////////
```

When running in polled mode, the application directly uses the buffers declared in `main()`, `namely – strInput[MAX_CHARS]` and `outBuff[MAX_CHARS]`. The relevant routines for polled mode are:

```
/////////////////////////////////////////////////////////////////
void UARTx_Init(void);// Initialize UART

void UARTx_SendByte(UINT8 data);
void UARTx_SendString(UINT8 *data, UINT8 len);

UINT8 UARTx_ReceiveByte(void);
void UARTx_ReceiveString(UINT8 len, UINT8 *outData);
/////////////////////////////////////////////////////////////////
```

Setup

The source code associated with this application note uses the Z16FMC Series Development Board connected to the Hyperterminal program via the PC's serial (RS-232) port; see Figure 13. The Hyperterminal settings should be 8-N-1 if using the default 9600 baud rate called out in the source code.

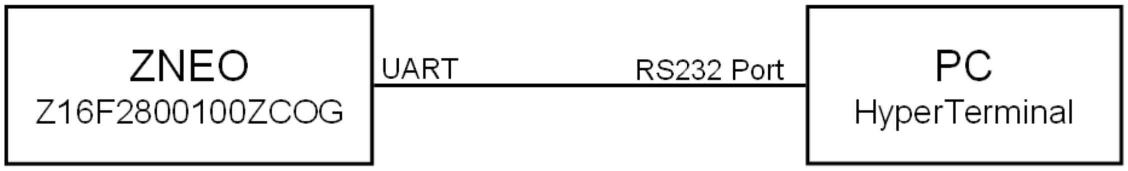


Figure 13. Z16FMC Series Development Kit to PC Connection via Serial (RS-232) Port

I^{2}C Operation

Features of the ZNEO I^{2}C controller include:

- Operates in Master/Slave or Slave Only modes
- Supports arbitration in a Multi-Master environment (Master/Slave mode)
- Supports data rates up to 400 Kbps
- 7-bit or 10-bit slave address recognition (interrupt only on address match)
- Optional general call address recognition
- Optional digital filter on receive SDA and SCL lines
- Optional interactive receive mode allows software interpretation of each received address and/or data byte before acknowledging
- Unrestricted number of data bytes per transfer
- A Baud Rate Generator can be used as a general-purpose timer with an interrupt if the I^{2}C controller is disabled

A block diagram of the I^{2}C Controller is shown in Figure 14.

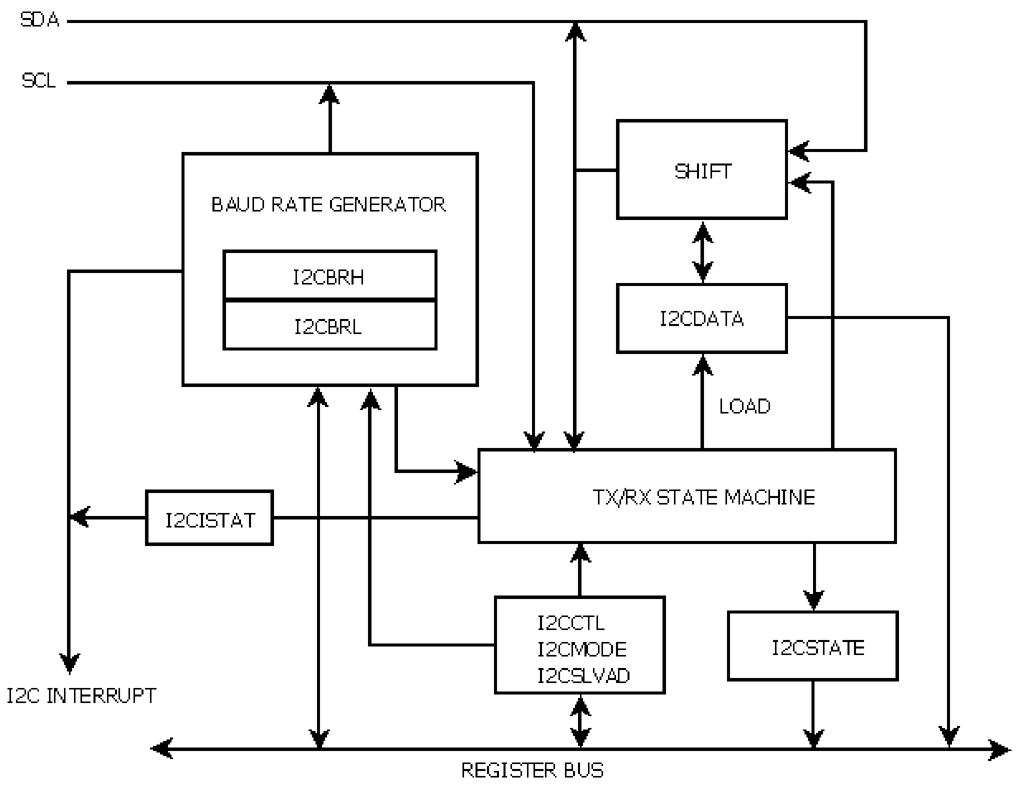


Figure 14. I^{2}C Controller Block Diagram

I^{2}C Hardware Architecture

The inter-integrated circuit (I^{2}C) protocol is demonstrated by interfacing the Z16FMC Series microcontroller with an I^{2}C EEPROM (in this application, we employ an AT24C128 from Atmel Corporation). The Z16FMC chip uses the I^{2}C controller in Master mode while the EEPROM acts as the slave. The software implementation for I^{2}C communication is performed in polling and interrupt modes.

Figure 15 shows a block diagram of an I^{2}C EEPROM device interfacing with a Z8 Encore! XP MCU. For more details about hardware connections, see Appendix A. Schematics on page 60.

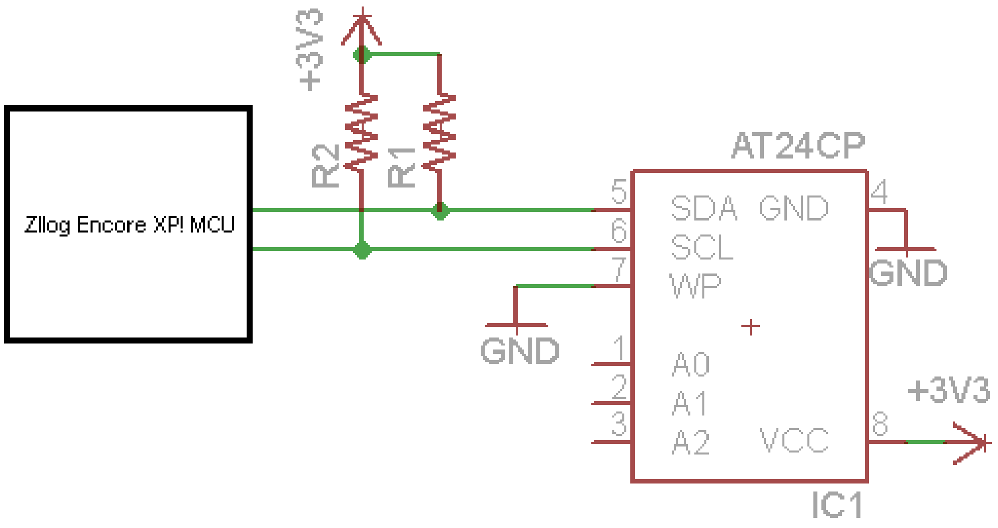


Figure 15. Product/Reference Design Block Diagram

I^{2}C Software Implementation

This reference design makes use of the Z16FMC MCU's on-chip I^{2}C controller. The software presented with this application initializes the peripheral and enables communication with the AT24C128 EEPROM chip via the I^{2}C protocol.

The following operations were performed using either a polling or an interrupt-driven method:

- Write a single byte to the EEPROM
- Read a single byte to the EEPROM
- Write a whole page to the EEPROM
- Read a whole page to the EEPROM

The software features two APIs for I^{2}C communications (found in `eeprom.c`), namely:

- Write API

- Read API

Figure 16 demonstrates a Master Write Transaction with a 7-bit address. Observe the following steps for a Master transmit operation to a 7-bit addressed I^2C EEPROM slave device.

1. Initialize the I^2C controller.
2. Check if the I^2C line is busy.
3. Issue a start condition.
4. Send a slave address (R/W bit = 0) to the I^2C device.
5. Send the Most Significant Byte of the address to be written to the EEPROM (this address is two bytes long).
6. Send the Least Significant Byte of the address to be written to the EEPROM (this address is two bytes long).
7. Send the data to be saved. For page writes, repeat this process if more bytes remain to be sent.
8. Issue a stop condition.

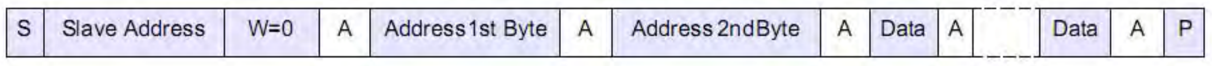


Figure 16. Master Write Transaction with a 7-Bit Address

Figure 17 demonstrates a Master Read Transaction with a 7-bit sequential address. Observe the following steps for a Master read operation to a 7-bit addressed I^2C EEPROM slave device.

1. Initialize the I^2C controller.
2. Check if the I^2C line is busy.
3. Issue a start condition.
4. Send a slave address (R/W bit = 0) to the I^2C device.
5. Send the Most Significant Byte of the address to be written to the EEPROM (this address is two bytes long).
6. Send the Least Significant Byte of the address to be written to the EEPROM (this address is two bytes long).
7. Reissue a start condition.
8. Send a slave address (R/W bit = 1) to the I^2C device.
9. Read the data sent by the slave device. The Master device should acknowledge (ACK) for all data that is read. However, if the data being read is the final byte, the master device issues a Not Acknowledge (NACK) instead.

10. Issue a stop condition.

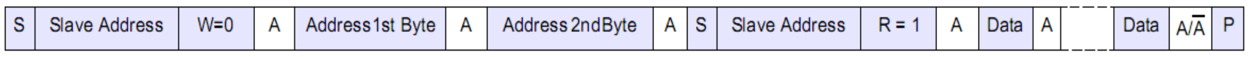


Figure 17. Master Read Transaction with a 7-Bit Address

Notes: When data is sent by the Master device, the slave device responds by an Acknowledgement/No Acknowledgement.

For page writes, a maximum of 32bytes can be written to the AT24C128.

Testing I^{2}C Operation

This section explains the test procedure to verify the APIs developed to demonstrate the I^{2}C capabilities of the Z16FMC series microcontrollers.

There are two modes to communicate with the I^{2}C EEPROM slave device, namely:

- Polling method
- Interrupt-driven method

The following definition in `main.h` is used to choose between the above two methods:

```
#define      I2C_POLLING_METHOD
```

Only one method can be used at a time. (Change the `#define` to `#undef` if testing for the interrupt-driven method.)

The following definitions and buffers are used to test the read/write capabilities:

```
#define TEST_BYTE1      0xAA
#define TEST_BYTE2      ~TEST_BYTE1
#define TEST_ADDRESS    0xF000
#define TEST_PAGE_NO    0x10

unsigned char message1[BYTES_PER_PAGE] =
{ "Testing I2C Protocol on ZNEO MCU" };
unsigned char message2[BYTES_PER_PAGE] =
{ "ZNEO I2C on-chip peripheral test" };
unsigned char message3[BYTES_PER_PAGE] =
{ "~~ZNEO I2C polling method test~~" };
unsigned char message4[BYTES_PER_PAGE] =
{ "~ZNEO I2C interrupt driven test~" };
unsigned char compare_buffer[BYTES_PER_PAGE];
```

These can be found in `main.h` and `main.c`, respectively.

The following definitions in `I2cRegisterDefinesF2811.h` can be used to change the I^{2}C parameters:

```
#define FREQUENCY   20000000
#define BAUD        100000
#define BRG         FREQUENCY / (4*BAUD)
```

Equipment Used

The following equipment is used for the setup.

- Z16FMC Series Development Kit
- Two 1K resistors
- AT24C128
- Connecting wires
- PC with installed ZDS II ZNEO 4.11.1 IDE and a serial port

Setup

For the setup, please refer to Figure 6 on page 12 and Appendix A. Schematics on page 60.

Procedure

Observe the following steps to test I^{2}C Bus Master operation.

1. Connect the external I^{2}C EEPROM (AT24C128) device to the Z16FMC Series MCU Development Board, as displayed on Appendix A. Schematics on page 60
2. Download the project file and open it in the ZDS II IDE (AN0312-SC01.zip).
3. Compile and build the project. Download the hex code to the Z16FMC Series Development Board.
4. Create a new Hyperterminal connection with the following properties: 57600 bps, 8 data bits, no parity with 1 stop bit and no flow control
5. Execute the program, a message prompt can be seen on the Hyperterminal on the status of the I^{2}C read and writes
6. Stop the program and change the test defines then repeat steps 3 to 5.

Results

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

- Polling method
- Interrupt-driven method

The operations were performed in different baud rates. The waveforms for the Read and Write operations were captured using a logic analyzer and displayed in Figures 18 and 19; sample Hyperterminal output is shown in Figure 20.

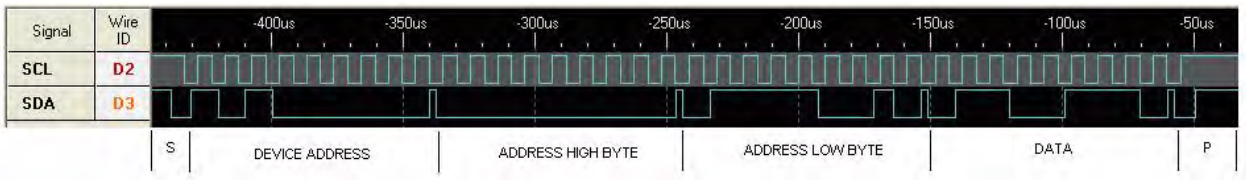


Figure 18. Master Write Transaction Waveform

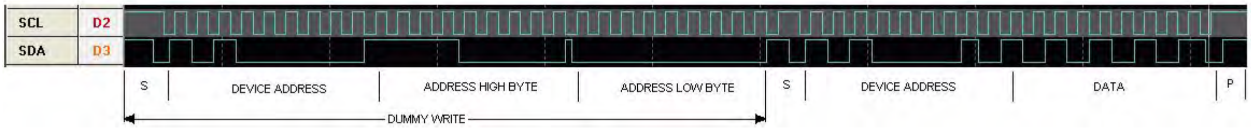
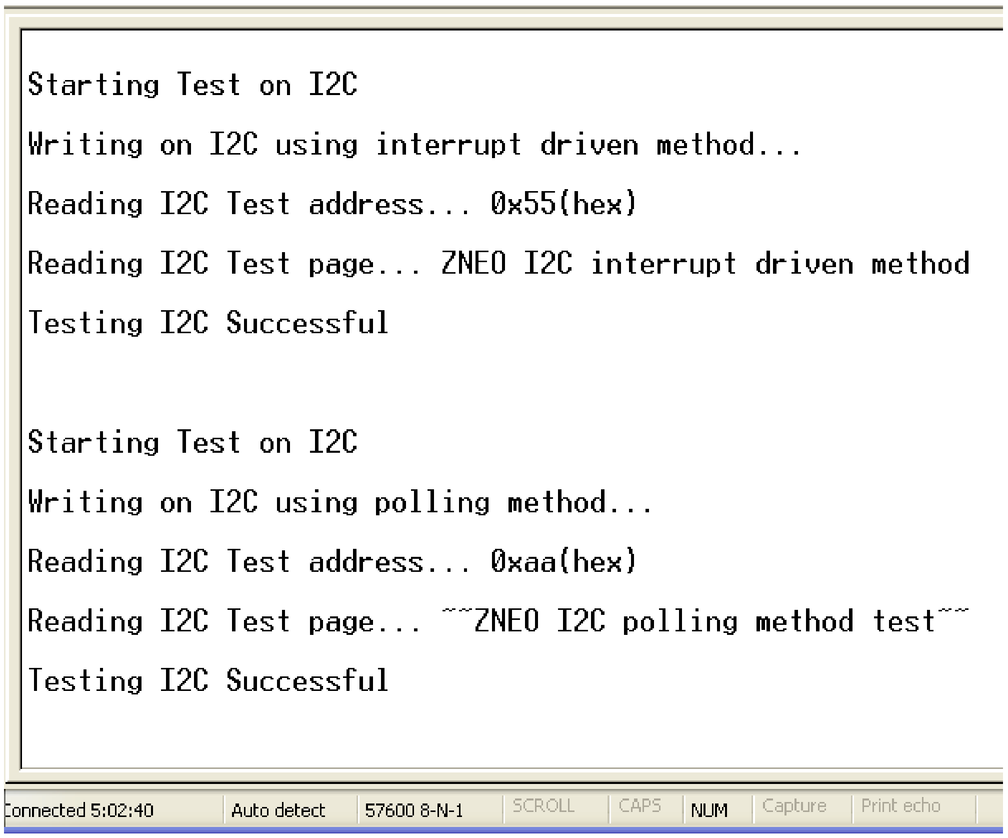


Figure 19. Master Read Transaction Waveform



```
Starting Test on I2C
Writing on I2C using interrupt driven method...
Reading I2C Test address... 0x55(hex)
Reading I2C Test page... ZNEO I2C interrupt driven method
Testing I2C Successful


Starting Test on I2C
Writing on I2C using polling method...
Reading I2C Test address... 0xaa(hex)
Reading I2C Test page... ~~ZNEO I2C polling method test~~
Testing I2C Successful
```

Figure 20. Hyperterminal Output

Multi-Channel PWM Module Operation

The Z16FMC MCUs feature a flexible PWM module with three complementary pairs or six independent PWM outputs supporting deadband operation and fault protection trip input. These features provide multiphase control capability for a variety of motor types and ensure safe operation of the motor by providing immediate shutdown of the PWM pins during Fault condition.

The Z16FMC Series includes a Multi-Channel PWM optimized for motor control applications. The PWM includes the following features:

- Six independent PWM outputs or three complementary PWM output pairs.
- Programmable deadband insertion for complementary output pairs.
- Edge-aligned or center-aligned PWM signal generation.
- PWM off-state is an option bit programmable.
- PWM outputs driven to off-state on System Reset.
- Asynchronous disabling of PWM outputs on system fault. Outputs are forced to off-state.
- Fault inputs generate pulse-by-pulse or hard shutdown.
- 12-bit reload counter with 1, 2, 4, or 8 programmable clock prescaler.
- High current source and sink on all PWM outputs.
- PWM pairs used as general purpose inputs when outputs are disabled.
- ADC synchronized with PWM period.
- Synchronization for current-sense sample and hold.
- Narrow pulse suppression with programmable threshold.

The AN0312-SC01.zip file associated with this application note demonstrates the usage of the Multichannel Timer feature. It initializes the Multichannel Timer to demonstrate three Complimentary PWM outputs & six Independent PWM outputs. See a block diagram of the PWM in Figure 21.

Note: The AN0312-SC01 source code has been tested with ZDSII – ZNEO version 4.11.1.

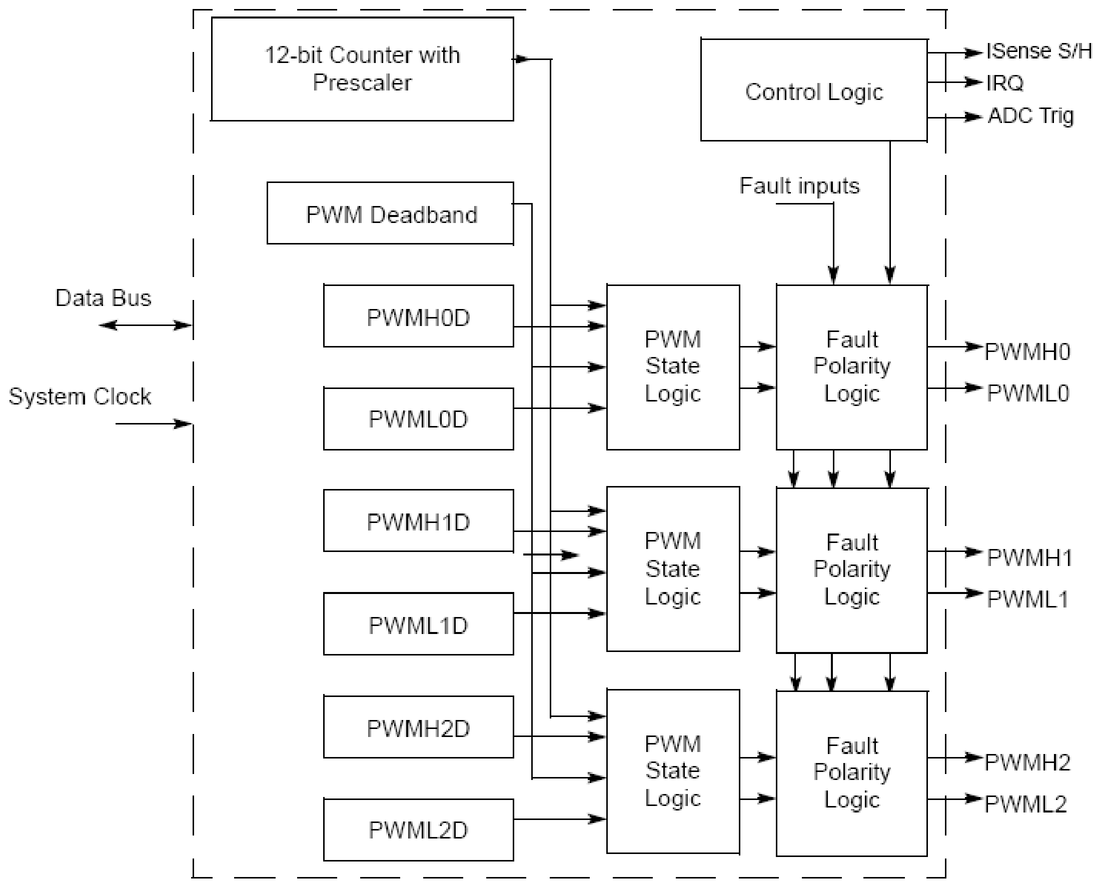


Figure 21. PWM Block Diagram

PWM Functions

Table 10 describes each of the seven PWM functions.

Table 10. Pulse-Width Modulator Functions

Function	Description
Init_PWM_GPIO	This function initializes the GPIO pins required for the multichannel timer operation.
Select_PWM_Alignment	This function selects the PWM alignment. Alignment passed can either be EDGE_ALIGN or CENTER_ALIGN.
Select_PWM_Polarity	This function selects the PWM output polarity. Polarity passes can either be NON_INVERT or INVERT.
Init_PWM_Registers	This function initializes all the Multichannel Timer PWM registers.
Initial_PWM_Duty_Cycle	This function initializes the PWM duty cycle registers with a start duty cycle value of 20% respectively.

Table 10. Pulse-Width Modulator Functions (Continued)

Function	Description
Independent_Mode_PWM_ Dutycycle	This function changes the duty cycle of the Pulse Width Modulator.
Complementary_Mode_PWM_ Dutycycle	This function changes the duty cycle of the Pulse Width Modulator.

PWM Software Implementation

The Flash_Option1 Register must be configured when implementing a Multi-Channel PWM Timer. See Table 11.

FLASH_OPTION1

```
FLASH_OPTION1 = 0xFD //for PWM Independent Mode
FLASH_OPTION1 = 0xFC //for PWM Complementary Mode
```

Table 11. Option Bits at Program Memory Address 0001H

Bits	7	6	5	4	3	2	1	0
Field	Reserved					MCEN	PWNHI	PWMLO
RESET	U	U	U	U	U	U	U	U
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
ADDR	Program Memory 0001H							
Note: U = Unchanged by Reset. R/W - Read/Write.								

Bit Position	Description
7:3	Reserved.
2	**Motor Control Enable (MCEN)** 0 = Motor control pins are enabled on reset. 1 = Normal pin operation.
1	**High side off initial value (PWMHI)** 0 = The high side off value is equal to zero. 1 = The high side off value is equal to one.
0	**Low side off initial value (PWMLO)** 0 = The low side off value is equal to zero. 1 = The low side off value is equal to one.

Init_PWM_GPIO

Enable alternate functions for the PWM port. See Tables 12 and 13.

```
PCAFH |= 0xC0;      // PC6,PC7 are 1 for Alternate Function 2 High register
```

```
PCAFL &= 0x3F;       // PC6,PC7 are 0 for Alternate Function 2 Low register
PDAFH &= 0x78;       // PD7,PD2,PD1,PD0 are 0 for Alternate Function 1 high
PDAFL |= 0x87;       // PD7,PD2,PD1,PD0 are 1 for Alternate Function 1 low
```

Table 12. Port A–K Alternate Function High Registers (PxAFH)

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Field	AFH[7]	AFH[6]	AFH[5]	AFH[4]	AFH[3]	AFH[2]	AFH[1]	AFH[0]
RESET	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
ADDR	FF_E104, FF_E124, FF_E134, FF_E174							

Table 13. Port A–K Alternate Function Low Registers (PxAFL)

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Field	AFH[7]	AFH[6]	AFH[5]	AFH[4]	AFH[3]	AFH[2]	AFH[1]	AFH[0]
RESET	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
ADDR	FF_E105, FF_E115, FF_E125, FF_E135, FF_E155, FF_E165, FF_E175, FF_E195							

Select_PWM_Alignment

This function selects the PWM alignment. The alignment can be either EDGE_ALIGN or CENTER_ALIGN.

```
PWMCTL0 &= ~(0x20);         // Enable the Edge Aligned mode
PWMCTL0 |= 0x20;            // Enable the Center Aligned mode
```

Table 14. PWM Control 0 Register (PWMCTL0)

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Field	PWMOFF	OUTCTL	ALIGN	Reserved	ADCTRIG	Reserved	READY	PWMEN
RESET	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
ADDR	FF_E380H							

Select_PWM_Polarity

This function selects the PWM output polarity. Polarity passes can be either NON_INVERT or INVERT.

```
PWMCTL1 &= 0xE3;            // POL45 = POL23 = POL10 = 0
PWMCTL1 |= 0x1C;            // POL45 = POL23 = POL10 = 1
```

Table 15. PWM Control 1 Register (PWMCTL1)

Bits	7	6	5	4	3	2	1	0
Field	RLFREQ[1:0]		INDEN	POL45	POL23	POL10	PRES[1:0]	
RESET	00		0	0	0	0	00	
R/W	R/W		R/W	R/W	R/W	R/W	R/W	
ADDR	FF_E381H							

Init_PWM_Registers

This function initializes all the Multichannel Timer PWM registers.

```
void Init_PWM_Registers( unsigned char deadband,
unsigned char min_pulse_filter );
```

Parameters

```
deadband            // This determines the number of system clock cycles
                    // inserted as dead time in complimentary mode of
                    // operation.
min_pulse_filter    // The minimum pulse width, either high or low, that can be
                    // generated by a PWM module.
```

Note: Make sure that the value of the Pulse Filter passed as an argument is a nonzero value; otherwise the PWM output will be distorted.

```
PWMDB = deadband;
            // Deadband is necessary in complementary mode of operation.
            // The minimum deadband value is 1.
PWMMPF = min_pulse_filter;
            // Pulse Filter register sets the minimum allowed output pulse
width // in PWM clock cycles. This register can only be written when PWEN // is
cleared.
PWMOUT = 0x00;
            // PWM Output Control Register enables modulator control of the
            // 6 PWM output signals.
PWMFM = 0x27;
            // PWM Fault Mask Register enables individual fault sources.
            // DBGMSK->0,F1MASK->1,Comparator Fault is disabled
            // FAULT0->disabled.
PWMFCTL = 0x44;
            // PWM Fault Control register determines,
            // how the PWM recovers from a fault condition.
            // DBGRST->1, CMP0RST->1.
```

```
PWMFSTAT = 0xFF;
            // PWM fault status register provides status of fault inputs
            // and timer reload.
            // The fault flags indicate which fault source is active.
PWMHL = 0x0000;
            // PWM counter initial value.
PWMR = s_PWMReloadValue;
            // PWM counter reload value.
```

Initial_PWM_Duty_Cycle

This function initializes the PWM duty cycle registers with a start duty cycle value of 20%. If the mode is Independent, load the remaining PWM Low duty cycle registers.

```
initial_duty_cycle = ( 20 * s_PWMReloadValue ) / 100;
s_PWMReloadValue = PWM_RELOAD_VALUE;
#define PWM_RELOAD_VALUE SYS_CLK_FREQ/(PWM_FREQUENCY*PWM_PRESCALAR)

// The initial values loaded into the PWM High duty cycle registers are
// applicable to both the Independent or Complementary mode of operations.
            PWMH0D  = initial_duty_cycle;// PWM High 0 duty cycle value.
            PWMH1D  = initial_duty_cycle; // PWM High 1 duty cycle value.
            PWMH2D  = initial_duty_cycle;// PWM High 2 duty cycle value.

// If mode is Independent, load the remaining PWM Low duty cycle registers.
// PWM Low side registers are loaded with initial duty cycle
            PWML0D  = initial_duty_cycle;// PWM Low 0 duty cycle value.
            PWML1D  = initial_duty_cycle;// PWM Low 1 duty cycle value.
            PWML2D  = initial_duty_cycle;// PWM Low 2 duty cycle value.

PWMCTL1 |= 0x20;                // INDEN bit is made 1 for INDEPENDENT mode
            PWMCTL1 &= 0xDF;    // INDEN bit is made 0 for COMPLEMENTARY mode
            PWMCTL0 |= 0x83;    // Enable PWM by enabling the PWEM bit.
                                // Ready bit->1 and PWMOFF->1.
```

Independent_Mode_PWM_Dutycycle

This function changes the duty cycle of the Pulse Width Modulator in INDEPENDENT mode and requires six parameters because all six PWM outputs will exhibit six independent duty cycles.

Note: Pass the required value, in percentage terms, to obtain the required PWM output. For example, passing 10 will result in a 10% duty cycle.

Parameters

```
  pwmh0_ind_dutycycle   //This indicates duty cycle in percentage for PWMH0D.
  pwml0_ind_dutycycle   //This indicates duty cycle in percentage for PWML0D.
```

```
pwmh1_ind_dutycycle   //This indicates duty cycle in percentage for PWMH1D.
pwml1_ind_dutycycle   //This indicates duty cycle in percentage for PWML1D.
pwmh2_ind_dutycycle   //This indicates duty cycle in percentage for PWMH2D.
pwml2_ind_dutycycle   //This indicates duty cycle in percentage for PWML2D.
```

Complementary_Mode_PWM_Dutycycle

This function changes the duty cycle of the Pulse Width Modulator in COMPLEMENTARY mode. This duty cycle requires only three parameters because the other three PWM outputs are the complementary pairs of the High-side PWMs.

Note: Pass the required value, in percentage terms, to obtain the required PWM output. For example, passing a value of 10 will result in a 10% duty cycle. The complementary PWM output will exhibit a 90% PWM duty cycle.

Parameters

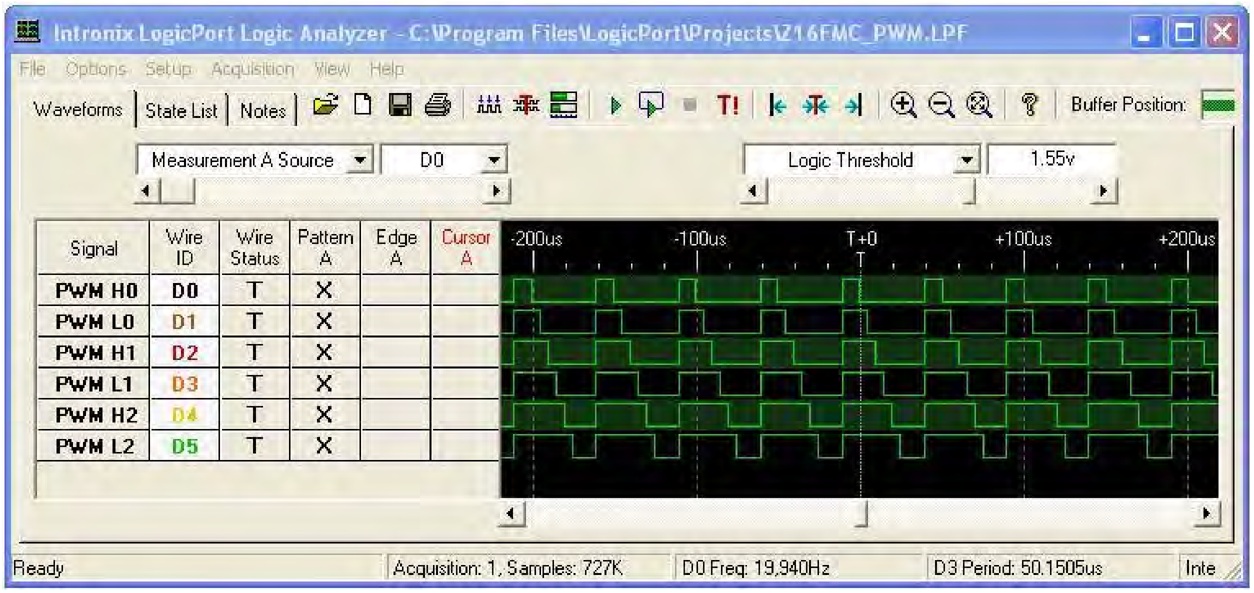
```
pwmh0_comp_dutycycle  //This indicates duty cycle in percentage for PWMH0DL.
pwmh1_comp_dutycycle  //This indicates duty cycle in percentage for PWMH1DL.
pwmh2_comp_dutycycle  //This indicates duty cycle in percentage for PWMH2DL.
```

Macro Definitions

```
#define PWM_PRESCALAR   4

// MACRO to change the Prescaler of the Pulse Width Modulator for Edge &
// Center Aligned modes..

#define PWM_FREQUENCY   20000

// MACRO to change the PWM output frequency..

#define COMPLEMENTARY   0
#define INDEPENDENT               1
#define MODE                      INDEPENDENT

// MACRO selects the mode of operation of the PWM
#define PWM_RELOAD_VALUE   SYS_CLK_FREQ/(PWM_FREQUENCY * PWM_PRESCALAR)
#define SET_PWM_PRESCALAR0x02
```

Results

Figures 22 and 23 show sample output from testing the PWM duty cycles in INDEPENDENT and COMPLEMENTARY modes, respectively.

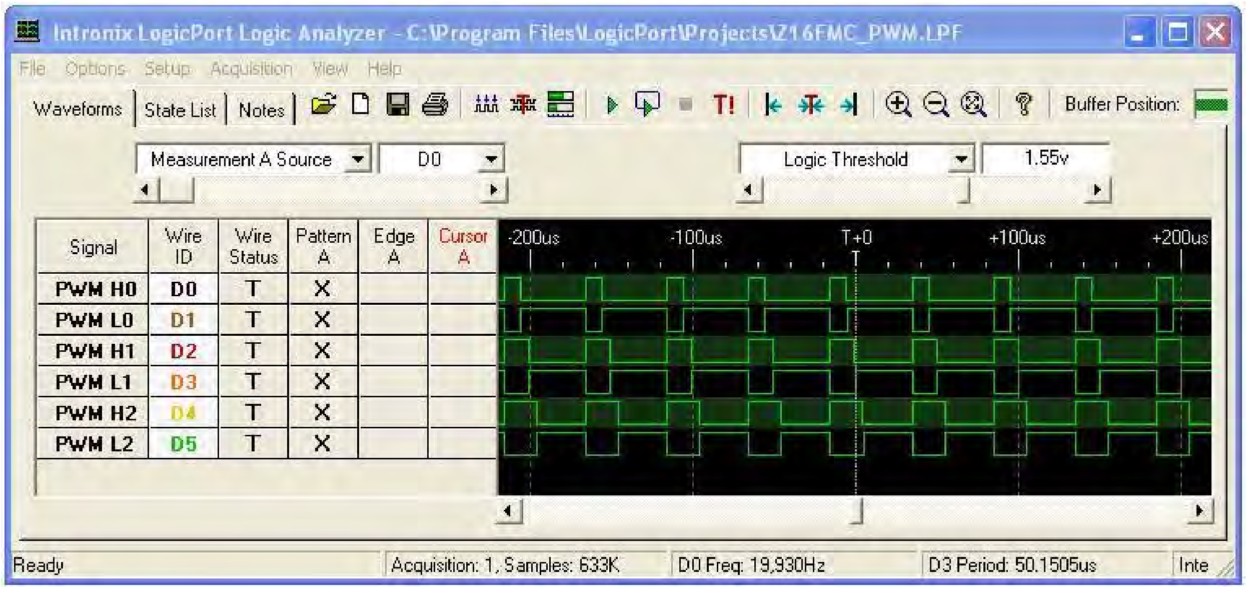

Intronix LogicPort Logic Analyzer - C:\Program Files\LogicPort\Projects\Z16FMC_PWM.LPF
File Options Setup Acquisition View Help
Waveforms State List Notes
Buffer Position:
-200us -100us T+0 +100us +200us

Measurement A Source: D0
Logic Threshold: 1.55v

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
PWM H0	D0	T	X		
PWM L0	D1	T	X		
PWM H1	D2	T	X		
PWM L1	D3	T	X		
PWM H2	D4	T	X		
PWM L2	D5	T	X		

Ready | Acquisition: 1, Samples: 727K | D0 Freq: 19,940Hz | D3 Period: 50.1505us | Inte

Figure 22. Sample Output Using Six PWMs in Independent Mode


Intronix LogicPort Logic Analyzer - C:\Program Files\LogicPort\Projects\Z16FMC_PWM.LPF
File Options Setup Acquisition View Help
Waveforms State List Notes
Buffer Position:
-200us -100us T+0 +100us +200us

Measurement A Source: D0
Logic Threshold: 1.55v

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
PWM H0	D0	T	X		
PWM L0	D1	T	X		
PWM H1	D2	T	X		
PWM L1	D3	T	X		
PWM H2	D4	T	X		
PWM L2	D5	T	X		

Ready | Acquisition: 1, Samples: 633K | D0 Freq: 19,930Hz | D3 Period: 50.1505us | Inte

Figure 23. Sample Output Using Three PWMs in Complementary Mode

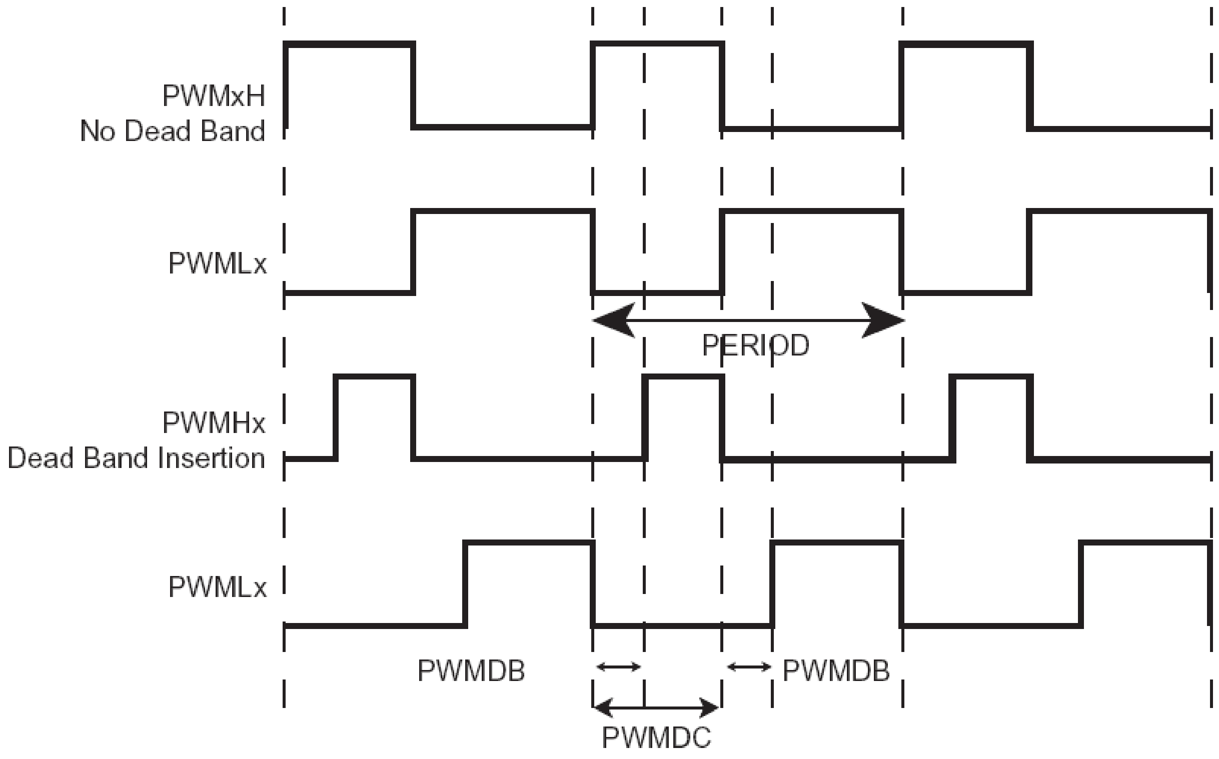


Figure 24. Edge-Aligned PWM Output

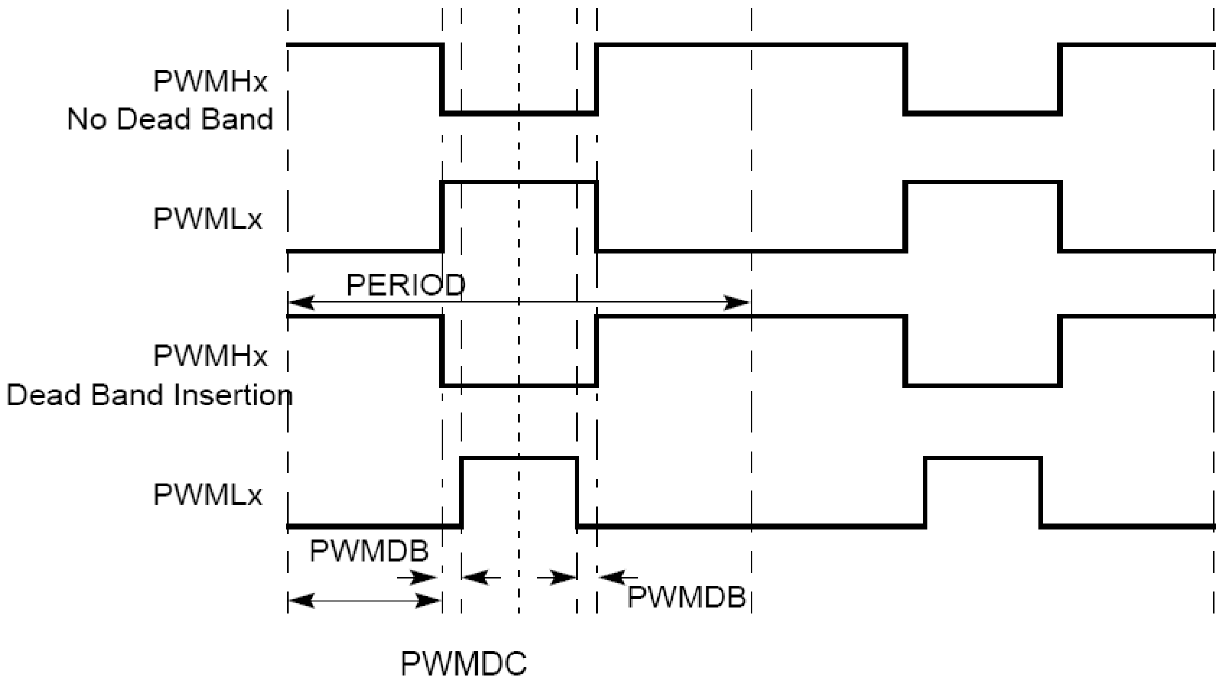


Figure 25. Center-Aligned PWM Output

Timer Operation

The Z16FMC Series MCUs contain three 16-bit reloadable timers used for timing, event counting or generation of pulse-width modulated (PWM) signals.

Timer Features

The timers include the following features:

- 16-bit reload counter
- Programmable prescaler with values ranging from 1 to 128
- PWM output generation (single or differential)
- Capture and compare capability
- An external input pin for event counting, clock gating or signal capture
- Complementary timer output pins
- A timer interrupt

A block diagram of the timer functions is shown in Figure 26.

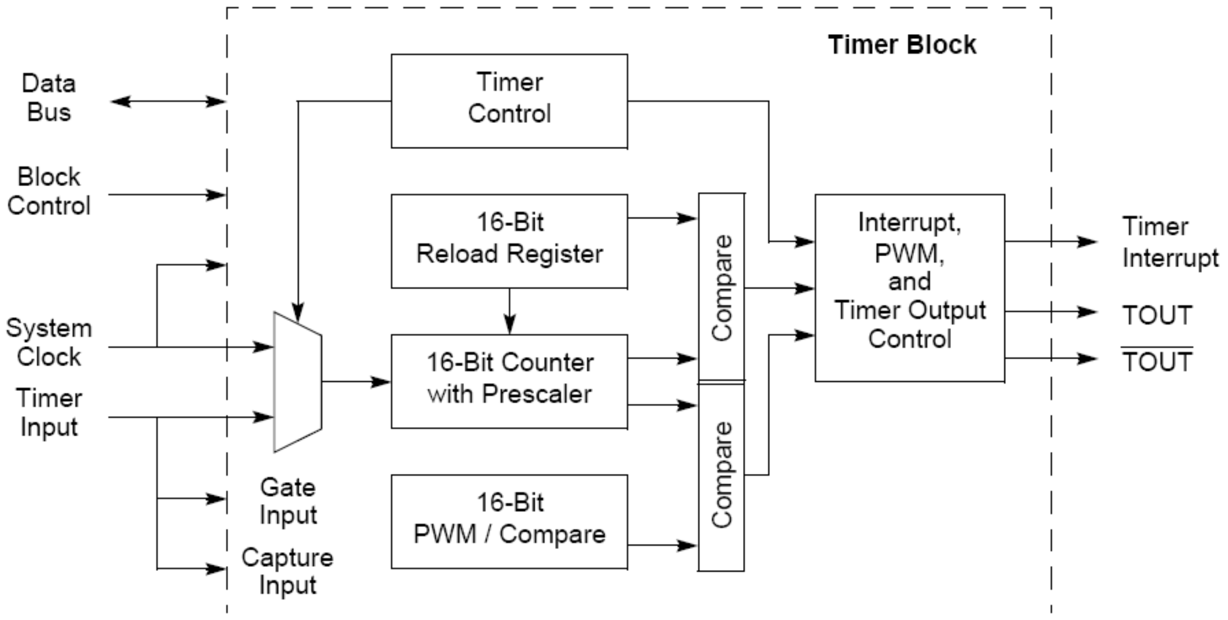


Figure 26. Timer Block Diagram

Timer Software Implementation

The general-purpose timer is a 16-bit upcounter. Under normal operation, the timer is initialized to `0001H`. When the timer is enabled, it counts up to the value contained in the reload high and low byte registers, then resets to `0001H`.

The counter either halts or continues depending on the mode. Minimum time-out delay (1 system clock) is set by loading the value `0001H` into the Timer Reload High and Low byte registers and setting the prescale value to 1. Maximum time-out delay (216 * 27 system clocks) is set by loading the value `0001H` into the Timer

In this demonstration, we reload the High and Low Byte Registers and set the prescale value to 128. When the timer reaches `FFFFH`, the timer rolls over to `0000H`. If the reload register is set to a value less than the current counter value, the counter continues counting until it reaches `FFFFH`, then resets to `0000H`. The timer next continues to count until it reaches the reload value and it resets to `0001H`.

The preprocessor directives used to enable or disable the timer, to switch the timer between different operating modes, and to change the prescaler are as follows:

```
#define RELOAD      (0x1389)// 0x1389 for 1msec timer @ 20MHz
#define PRESCALE    (PRESCALAR_4)
#define STARTCOUNT(0x0001)
#define COUNTER     (0x0005)// timer counter value

// Different Operating modes of the TIMER
#define ENABLE_TIMER(0x80)
```

```
#define DISABLE_TIMER(0x7F)
// Configure in Timer 0-2 Control 1 Register (TxCTL1)
// TxCTL1 = TMODE [2-0]
#define ONE_SHOT_MODE     (0x00)
#define CONTINUOUS_MODE   (0x01)
#define COUNTER_MODE      (0x02)
#define PWM_SINGLE_MODE   (0x03)
#define CAPTURE_MODE      (0x04)
#define COMPARE_MODE      (0x05)
#define GATED_MODE        (0x06)
#define CAPTURE_COMPARE_MODE (0x07)
// Configure in Timer 0-2 Control 1 Register (TxCTL1)
// and in Timer 0-2 Control 0 Register (TxCTL0)
// TxCTL1 [2-0] = TMODE [2-0]
// TxCTL0 [7] = TMODE [3]
#define PWM_DUAL_MODE     (0x08)
#define CAPTURE_RESTART_MODE(0x09)
#define COMPARATOR_COUNTER_MODE(0x0A)
#define TRIGGERED_ONE_SHOT_MODE(0x0B)

// Prescaler Values
#define PRESCALAR_1       (0x00)        // Divide by 1
#define PRESCALAR_2       (0x08)        // Divide by 2
#define PRESCALAR_4       (0x10)        // Divide by 4
#define PRESCALAR_8       (0x18)        // Divide by 8
#define PRESCALAR_16      (0x20)        // Divide by 16
#define PRESCALAR_32      (0x28)        // Divide by 32
#define PRESCALAR_64      (0x30)        // Divide by 64
#define PRESCALAR_128     (0x38)        // Divide by 128

// Directives for various Timer Registers
#define PWM_HIGH        (0x00)
#define PWM_LOW         (0x04)

#define POLARITY_HIGH   (0x40)
#define POLARITY_LOW    (0x00)


// Directives for Timer Interrupt Configurations
#define TICONFIG_RELOAD_COMPARE_INPUT(0x00)
#define TICONFIG_CAPTURE_DEASSERT(0x40)
#define TICONFIG_RELOAD_COMPARE(0x60)

void Enable_Timer2_OutputPin (void)
{
  PCAFH &= ~0x80;       // select port alternate function 1
  PCAFL |= 0x80;
}

void Enable_Timer2_InputPin (void)
{
  PCAFH &= ~0x40;       // select port alternate function 1
  PCAFL |= 0x40;
}

// To change the interrupt priority of the timer, use the routines illustrated
below:
void TMR2_Priority_Low (void)
{
  IRQ0ENH &= ~0x80;
  IRQ0ENL |= 0x80;
  IRQ0SET &= ~0x80;     // reset interrupt request
}

void TMR2_Priority_Nominal (void)
{
  IRQ0ENH |= 0x80;
  IRQ0ENL &= ~0x80;
  IRQ0SET &= ~0x80;     // reset interrupt request
}

void TMR2_Priority_High (void)
{
  IRQ0ENH |= 0x80;
  IRQ0ENL |= 0x80;
  IRQ0SET &= ~0x80;     // reset interrupt request
}
```

Note: Ensure that changing the Timer0 priority does not change the priorities of other interrupt resources.

Timer Modes

The Z16FMC Series MCU can operate in a n umber of different timing modes depending upon application requirements. These twelve modes, listed below, are each described in this section where indicated.

- ONE SHOT Mode on page 44
- TRIGGERED ONE SHOT Mode on page 44
- CONTINUOUS Mode on page 45
- COUNTER Mode on page 46
- COMPARATOR COUNTER Mode on page 47
- PWM SINGLE OUTPUT Mode on page 47
- PWM DUAL OUTPUT Mode on page 48
- CAPTURE Mode on page 49

- CAPTURE RESTART Mode on page 50
- CAPTURE/COMPARE Mode on page 50
- COMPARE Mode on page 51
- GATED Mode on page 51

ONE SHOT Mode

In ONE-SHOT mode, the timer counts up to the 16-bit reload value stored in the Timer Reload High and Low Byte registers. The timer input is the system clock. When the timer reaches the reload value, it generates an interrupt and the count value in the Timer High and Low Byte registers is reset to `0001H`. The timer is automatically disabled and stops counting.

The basic equation to satisfy the time-out period in ONE SHOT mode is shown below.

$$\text{One-Shot Mode Time-Out Period (s)} = \frac{(\text{Reload Value} - \text{Start Value} + 1) \times \text{Prescale}}{\text{System Clock Frequency (Hz)}}$$

For example, the reload value for a time-out period of 1 msec is calculated as follows:

Reload value = ((1 msec * system clock frequency) / prescaler) + start value

Reload value = ((0.001 * 20000000) / 4) + 1 = 0x1389

The Timer0 Reload High Register (T0RH) must be loaded with a value of `13h` and the Timer0 Reload Low Register (T0RL) must be loaded with a value of `89h`.

The code listed below illustrates how to configure the timer in ONE-SHOT mode.

```
T2CTL1 = ONE_SHOT_MODE | PRESCALE;// Disable timer, ONE-SHOT mode,
                                  // prescale = 4
T2CTL0 = 0x00;
T2H = (STARTCOUNT >> 8);          // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (RELOAD >> 8);             // Set reload value = 1381h (1msec)
T2RL = (RELOAD & 0x00FF);
TMR2_Priority_Nominal ();         // timer interrupt priority = nominal
Enable_Timer2_OutputPin ();       // Configure Port C for alternate
// function operation
T2CTL1 |= ENABLE_TIMER;           // Enable timer
```

TRIGGERED ONE SHOT Mode

In TRIGGERED ONE-SHOT mode, the timer is non-active until a trigger is received. The timer trigger is taken from the timer input pin. The TPOL bit in the Timer Control 1 Register selects whether the trigger occurs on the rising edge or the falling edge of the timer input signal.

Following the trigger event, the timer counts system clocks up to the 16-bit reload value stored in the Timer Reload High and Low Byte registers.

After reaching the reload value, the timer outputs a pulse on the timer output pin, generates an interrupt, and resets the count value in the Timer High and Low Byte registers to `0001H`. The duration of the output pulse is a single system clock. The TPOL bit also sets the polarity of the output pulse.

The timer now idles until the next trigger event. Trigger events, which occur while the timer is responding to a previous trigger, are ignored.

The timer period is calculated by the following equation, in which the start value = 1):

$$\text{Triggered One-Shot Mode Time-Out Period (s)} = \frac{(\text{Reload Value} - \text{Start Value} + 1) \times \text{Prescale}}{\text{System Clock Frequency (Hz)}}$$

For example, the reload value for a time-out period of 1 msec is calculated as follows:

Reload value = ((1 msec * system clock frequency) / prescaler) + start value

Reload value = ((0.001 * 20000000) / 4) + 1 = 0x1389

The Timer0 Reload High Register (T0RH) must be loaded with a value of `13h` and the Timer0 Reload Low Register (T0RL) must be loaded with a value of `89h`.

The code listed below indicates how to configure the timer in TRIGGERED ONE-SHOT mode:

```
T2CTL1 = (TRIGGERED_ONE_SHOT_MODE & 0x07) | PRESCALE;
T2CTL0 = (TRIGGERED_ONE_SHOT_MODE << 4) & 0x80;// Disable timer,
// PWM Dual Output mode
T2H    = (STARTCOUNT >> 8);          // Set starting count value = 0001h
T2L    = (STARTCOUNT & 0x00FF);
T2RH   = (COUNTER >> 8);             // Set reload value = 0005h (5 counts)
T2RL   = (COUNTER & 0x00FF);
T2CTL0 = TICONFIG_RELOAD_COMPARE;// Set Timer to interrupt on
                                     // reload/compare events only
TMR2_Priority_High();                // Enable timer interrupt priority
Enable_Timer2_OutputPin ();          // Configure Port C for alternate
       // function operation
Enable_Timer2_InputPin ();           // Enable timer input pin
T0CTL1 |= ENABLE_TIMER;              // Enable timer
```

CONTINUOUS Mode

In CONTINUOUS mode, the timer counts up to the 16-bit reload value stored in the Timer Reload High and Low Byte registers. After reaching the reload value, the timer generates an interrupt, the count value in the Timer High and Low Byte registers is reset to `0001H`, and counting resumes. If the timer output alternate function is enabled, the timer output pin changes state (from Low to High or from High to Low) after timer reload.

The timer period is determined by the following equation:

$$\text{Continuous Mode Time-Out Period (s)} = \frac{\text{Reload Value} \times \text{Prescale}}{\text{System Clock Frequency (Hz)}}$$

If an initial starting value other than `0001H` is loaded into the Timer High and Low Byte registers, use the ONE SHOT Mode equation to determine the first time-out period.

For example, the reload value for a time-out period of 1msec is calculated as follows:

Reload value = ((1 msec * system clock frequency) / prescaler) + start value

Reload value = ((0.001 * 20000000) / 4) + 1 = 0x1389

The Timer0 Reload High Register (T0RH) must be loaded with a value of `13h` and the Timer0 Reload Low Register (T0RL) must be loaded with a value of `89h`.

The code listed below indicates how to configure the timer in CONTINUOUS mode:

```
T2CTL1 = CONTINUOUS_MODE | PRESCALE;// Disable timer,
                                    // ONE-SHOT mode, prescale = 4
T2CTL0 = 0x00;
T2H = (STARTCOUNT >> 8);            // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (RELOAD >> 8);               // Set reload value = 1381h (1msec)
T2RL = (RELOAD & 0x00FF);
TMR2_Priority_High ();              // timer interrupt priority = nominal
Enable_Timer2_OutputPin ();         // Configure Port C for alternate
                                    // function operation
T2CTL1 |= ENABLE_TIMER;             // Enable timer
```

COUNTER Mode

In COUNTER mode, the timer counts input transitions from a GPIO port pin. The timer input is taken from the associated GPIO port pin. The TPOL bit in the Timer Control 1 Register selects whether the count occurs on the rising edge or the falling edge of the timer input signal. In COUNTER mode, the prescaler is disabled.

The code below indicates how to configure the timer in COUNTER mode:

```
T2CTL1 = COUNTER_MODE;        // Disable timer, COUNTER mode
T2H = (STARTCOUNT >> 8);      // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (COUNTER >> 8);        // Set reload value = 0005h (5 counts)
T2RL = (COUNTER & 0x00FF);
T2CTL0 = TICONFIG_RELOAD_COMPARE;// Set Timer to interrupt on
                              // reload/compare events only
TMR2_Priority_High();         // Enable timer interrupt priority
Enable_Timer2_OutputPin ();   // Configure Port C for alternate
                              // function operation
Enable_Timer2_InputPin ();    // Enable timer input pin
```

```
T0CTL1 |= ENABLE_TIMER;       // Enable timer
```

COMPARATOR COUNTER Mode

In COMPARATOR COUNTER mode, the timer counts output transitions from an analog comparator output. The timer takes its input from the output of the comparator. The TPOL bit in the Timer Control 1 Register selects whether the count occurs on the rising edge or the falling edge of the comparator output signal. The prescaler is disabled in COMPARATOR COUNTER mode.

The code below indicates how to configure the timer in COMPARATOR COUNTER mode:

```
CMPOPC |= 0x01;               // Enable Comparator
PCAFH |= 0x01;                // Initialize Comparator0
PCAFL |= 0x01;

T2CTL0 = (COMPARATOR_COUNTER_MODE << 4) & 0x80;
T2CTL1 = COMPARATOR_COUNTER_MODE & 0x07;// Disable timer,
                              // COMPARATOR COUNTER mode
T2H = (STARTCOUNT >> 8);      // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (COUNTER >> 8);        // Set reload value = 0005h (5 counts)
T2RL = (COUNTER & 0x00FF);
TMR2_Priority_High();         // Enable timer interrupt priority
Enable_Timer2_OutputPin;      // Configure Port C for alternate
                              // function operation
T2CTL1 |= ENABLE_TIMER;       // Enable timer
```

PWM SINGLE OUTPUT Mode

In PWM SINGLE OUTPUT mode, the timer outputs a PWM output signal through a GPIO port pin. The timer first counts up to the 16-bit PWM match value stored in the Timer PWM High and Low Byte registers. When the timer count value matches the PWM value, the timer output toggles. The timer continues counting until it reaches the reload value stored in the Timer Reload High and Low Byte registers. When it reaches the reload value, the timer generates an interrupt. The count value in the Timer High and Low Byte registers is reset to `0001H`, and counting resumes.

The timer output signal begins with a value = `TPOL` and then transits to `TPOL`, when the timer value matches the PWM value. The timer output signal returns to `TPOL` after the timer reaches the reload value and is reset to `0001H`.

The PWM period is determined by the following equation:

$$\text{PWM Period (s)} = \frac{\text{Reload Value} \times \text{Prescale}}{\text{System Clock Frequency (Hz)}}$$

If an initial starting value other than `0001H` is loaded into the Timer High and Low Byte registers, use the ONE SHOT Mode equation to determine the first PWM time-out period.

If `TPOL` is set to 0, the ratio of the PWM output High time to the total period is determined by the following equation:

$$\text{PWM Output High Time Ratio (\%)} = \frac{\text{Reload Value} - \text{PWM Value}}{\text{Reload Value}} \times 100$$

If `TPOL` is set to 1, the ratio of the PWM output High time to the total period is determined by:

$$\text{PWM Output High Time Ratio (\%)} = \frac{\text{PWM Value}}{\text{Reload Value}} \times 100$$

The code below indicates how to configure the timer in PWM SINGLE OUTPUT mode:.

```
T2CTL1 = PWM_SINGLE_MODE | PRESCALE;// Disable timer, PWM mode
T2H = (STARTCOUNT >> 8);      // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (RELOAD >> 8);         // Set reload value = 1381h (1msec)
T2RL = (RELOAD & 0x00FF);
T2PWMH = (RELOAD / 2) >> 8;   // Set PWM value (50% duty cycle)
T2PWML = (RELOAD / 2) & 0x00FF;
TMR2_Priority_High ();        // Enable timer interrupt priority
Enable_Timer2_OutputPin ();   // Configure Port C for alternate
                              // function operation
T0CTL1 |= ENABLE_TIMER;       // Enable timer
```

PWM DUAL OUTPUT Mode

In PWM DUAL OUTPUT mode, the timer outputs a PWM output signal and also its complement through two GPIO port pins, the timer also generates a second PWM output signal, timer output complement (TOUT). A programmable deadband is configured (PWMD field) to delay (0 to 128 system clock cycles) the Low to a High (inactive to active) output transitions on these two pins. This configuration ensures a time gap between the deassertion of one PWM output to the assertion of its complement.

The PWM period is determined by the following equation:

$$\text{PWM Period (s)} = \frac{\text{Reload Value} \times \text{Prescale}}{\text{System Clock Frequency (Hz)}}$$

If `TPOL` is set to 0, the ratio of the PWM output High time to the total period is determined by the following equation.

$$\text{PWM Output High Time Ratio (\%)} = \frac{\text{Reload Value} - \text{PWM Value}}{\text{Reload Value}} \times 100$$

If TPOL is set to 1, the ratio of the PWM output High time to the total period is determined by the following equation.

$$\text{PWM Output High Time Ratio (\%)} = \frac{\text{PWM Value}}{\text{Reload Value}} \times 100$$

The code below indicates how to configure the timer in PWM DUAL OUTPUT mode.

```
T2CTL0 = (PWM_DUAL_MODE << 4) & 0x80; // Disable timer, PWM Dual Output mode
T2CTL1 = (PWM_DUAL_MODE & 0x07) | PRESCALE;
T2H = (STARTCOUNT >> 8);              // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (RELOAD >> 8);                 // Set reload value = 1381h (1msec)
T2RL = (RELOAD & 0x00FF);
T2PWMH = (RELOAD / 2) >> 8;           // Set PWM value (50% duty cycle)
T2PWML                                = (RELOAD / 2) & 0x00FF;
TMR2_Priority_High ();                // Enable timer interrupt priority
Enable_Timer2_OutputPin ();           // Configure Port C for alternate
                                      // function operation
T0CTL1 |= ENABLE_TIMER;               // Enable timer
```

CAPTURE Mode

When the timer is enabled in CAPTURE mode, it counts continuously and resets to 0000H from FFFFH. When a capture event occurs, the timer counter value is captured and stored in the PWM High and Low Byte registers, an interrupt is generated, and the timer continues counting up to the 16-bit reload value stored in the Timer Reload High and Low Byte registers. Upon reaching the reload value, the timer generates an interrupt and continues counting.

In CAPTURE mode, the elapsed time from timer start to the capture event is calculated using the following equation, in which the start value = 1.

$$\text{Capture Elapsed Time (s)} = \frac{(\text{Capture Value} - \text{Start Value} + 1) \times \text{Prescale}}{\text{System Clock Frequency (Hz)}}$$

The code below indicates how to configure the timer in CAPTURE mode.

```
T2CTL1 = CAPTURE_MODE | PRESCALE; // Disable timer, CAPTURE mode
T2H = (STARTCOUNT >> 8);          // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (0xFFFF >> 8);             // Set reload value
T2RL = (0xFFFF & 0x00FF);
T2PWMH = 0x00;                    // Clear the PWM high & low byte registers
T2PWML = 0x00;
TMR2_Priority_High();             // Enable timer interrupt priority
Enable_Timer2_InputPin ();        // Enable timer input pin
T0CTL1 |= ENABLE_TIMER;           // Enable timer
```

CAPTURE RESTART Mode

When the timer is enabled in CAPTURE RESTART mode, it counts continuously until a capture event occurs or the timer count reaches the 16-bit compare value stored in the Timer Reload High and Low Byte registers. If the capture event occurs first, the timer counter value is captured and stored in the PWM High and Low Byte registers, an interrupt is generated, the count value in the Timer High and Low Byte registers is reset to `0001H`, and counting resumes. If no capture event occurs upon reaching the reload value, the timer generates an interrupt, the count value in the Timer High and Low Byte registers is reset to `0001H`, and counting resumes.

The code below indicates how to configure the timer in CAPTURE RESTART mode.

```
T2CTL1 = (CAPTURE_RESTART_MODE & 0x07) | PRESCALE;
T2CTL0 = (CAPTURE_RESTART_MODE << 4) & 0x80;// Disable timer,
                                  // CAPTURE RESTART mode, prescale = 4
T2H = (STARTCOUNT >> 8);          // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (0xFFFF >> 8);             // Set reload value
T2RL = (0xFFFF & 0x00FF);
TMR2_Priority_High();             // Enable timer interrupt priority
Enable_Timer2_InputPin ();        // Enable timer input pin
T2CTL1 |= ENABLE_TIMER;           // Enable timer
```

CAPTURE/COMPARE Mode

CAPTURE/COMPARE mode is identical to CAPTURE RESTART mode except that counting does not start until the first appropriate external Timer Reload High and Low Byte input transition occurs. Every subsequent appropriate transition (after the first) of the Timer Reload High and Low Byte input signal captures the current count value. When the capture event occurs, an interrupt is generated, the count value in the Timer Reload High and Low Byte registers is reset to `0001H`, and counting resumes. If no capture event occurs, then upon reaching the compare value, the timer generates an interrupt, the count value in the Timer High and Low Byte registers is reset to `0001H`, and counting resumes.

The code below illustrates how to configure the timer in CAPTURE/COMPARE mode.

```
T2CTL1 = CAPTURE_COMPARE_MODE | PRESCALE;// Disable timer,
                                      // CAPTURE COMPARE mode, prescale = 4
T2H = (STARTCOUNT >> 8);              // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (RELOAD >> 8);                 // Set compare value = 1381h (1msec)
T2RL = (RELOAD & 0x00FF);
TMR2_Priority_High();                 // Enable timer interrupt priority
Enable_Timer2_InputPin ();            // Enable timer input pin
T0CTL1 |= ENABLE_TIMER;               // Enable timer
```

COMPARE Mode

In COMPARE mode, the timer counts up to the 16-bit compare value stored in the Timer Reload High and Low Byte input registers. After reaching the compare value, the timer generates an interrupt and counting continues (i.e., the timer value is not reset to 0001H). If the timer output alternate function is enabled, the timer output pin changes state (from Low to High or from High to Low).

If the timer reaches FFFFH, the timer rolls over to 0000H and continues counting.

The code below indicates how to configure the timer in COMPARE mode.

```
T0CTL1 = COMPARE_MODE | PRESCALE;     // Disable timer,
                                      // COMPARE mode, prescale = 4
T0H = (STARTCOUNT >> 8);              // Set starting count value = 0001h
T0L = (STARTCOUNT & 0x00FF);
T0RH = (RELOAD >> 8);                 // Set compare value = 1381h (1msec)
T0RL = (RELOAD & 0x00FF);
T0CTL0 = TICONFIG_RELOAD_COMPARE;     // Set Timer to interrupt on
                                      // reload/compare events only
TMR2_Priority_High();                 // Enable timer interrupt priority
Enable_Timer2_OutputPin();            // Configure Port C for alternate
                                      // function operation
T0CTL1 |= ENABLE_TIMER;               // Enable timer
```

GATED Mode

In GATED mode, the timer counts only when the timer input signal is in its active state as determined by the TPOL bit in the Timer Control 1 Register. When the timer input signal is active, counting begins. A timer interrupt is generated when the timer input signal transits from active to inactive state or a timer reload occurs. To determine if a timer input signal deassertion generated the interrupt, read the associated GPIO input value and compare to the value stored in the TPOL bit.

The timer counts up to the 16-bit reload value stored in the Timer Reload High and Low Byte input registers. Upon reaching the reload value, the timer generates an interrupt, the count value in the Timer High and Low Byte input registers is reset to 0001H, and counting continues as long as the timer input signal is active. If the timer output alternate function is enabled, the timer output pin changes state (from Low to High or from High to Low) at timer reload.

The code below illustrates how to configure the timer in GATED mode.

```
T2CTL1 = GATED_MODE | PRESCALE;       // Disable timer,
                                      // GATED mode, prescale = 4
T2H = (STARTCOUNT >> 8);              // Set starting count value = 0001h
T2L = (STARTCOUNT & 0x00FF);
T2RH = (RELOAD >> 8);                 // Set reload value = 1381h (1msec)
T2RL = (RELOAD & 0x00FF);
TMR2_Priority_High();                 // Enable timer interrupt priority
Enable_Timer2_OutputPin;              // Configure Port C for alternate
```

```
                                              // function operation
  T0CTL1 |= ENABLE_TIMER;                     // Enable timer
```

ESPI Operation

On Z16FMC MCUs, the ESPI is a full-duplex, synchronous, character-oriented channel that supports a four-wire interface (serial clock, transmit and receive data and Slave select). The ESPI block consists of a shift register, transmit and receive data buffer registers, a baud rate (clock) generator, control/status registers and a control state machine. Transmit and receive transfers are synchronous due to a single shift register for both transmit and receive data.

The features of the ESPI block include:

- Full duplex, synchronous, character-oriented communication
- Four-wire interface (SS, SCK, MOSI, MISO)
- Transmit and receive buffer registers to enable high throughput
- Transfer rates up to a maximum of one-fourth the system clock frequency (SLAVE mode)
- Error detection
- Optional digital filter on receive SDA and SCL lines
- Dedicated programmable baud rate generator
- Data transfer control through polling, interrupt, or DMA

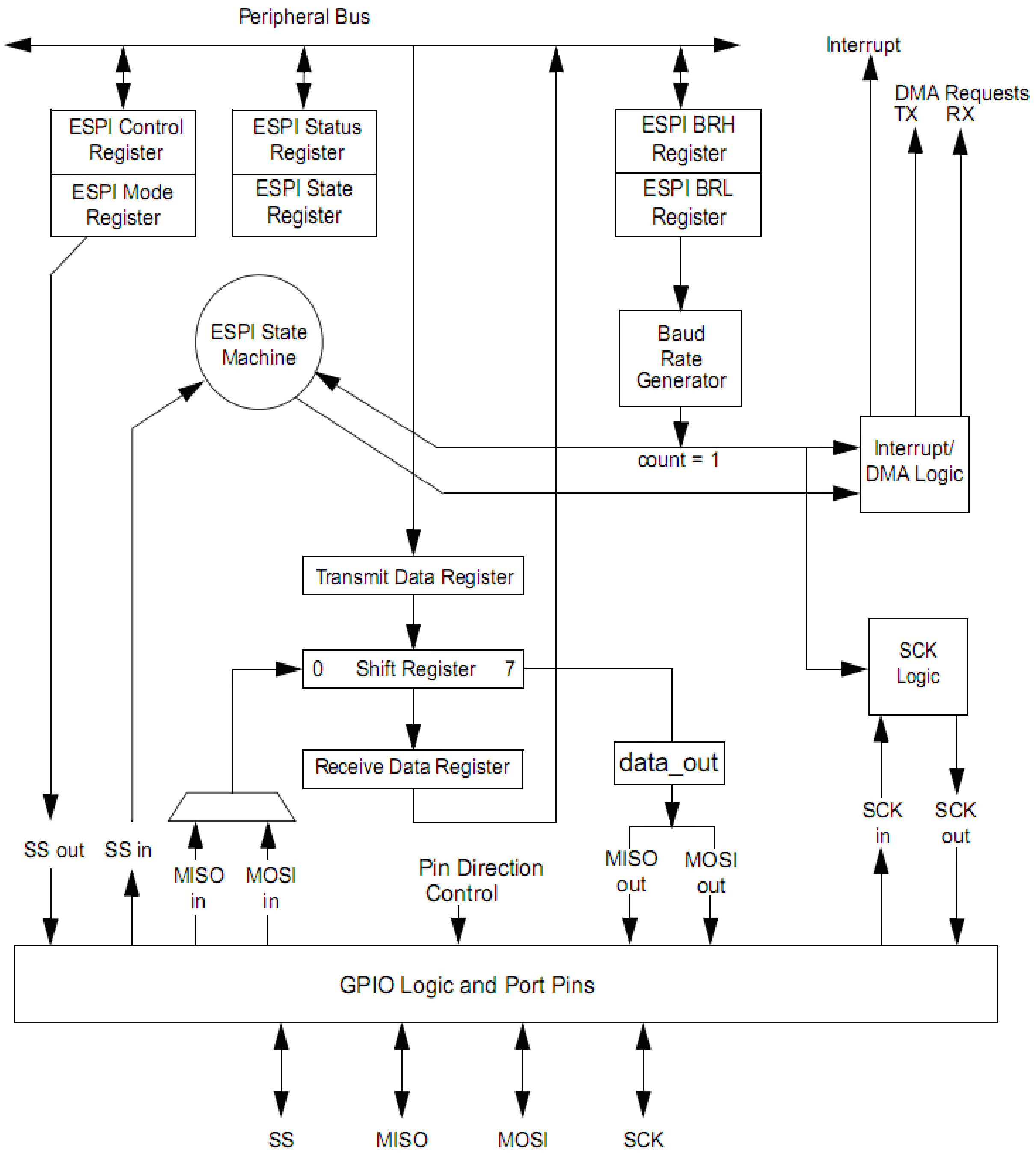


Figure 27. ESPI Block Diagram

Overview of the SPI Protocol

The SPI bus uses four logic lines:

- SCLK – serial clock (output from master)
- MOSI – master output, slave input (output from master)
- MISO – master input, slave output (output from slave)
- SS – slave select (active low, output from master)

The SPI bus can operate with a single master, plus one or more slave devices.

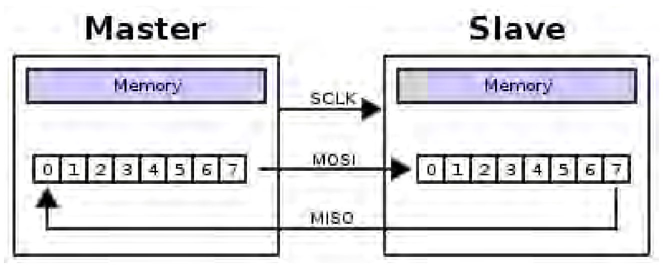


Figure 28. Sample SPI Configuration

SPI Data Transfer

The master first configures the clock using a frequency less than or equal to the maximum frequency that the slave device supports. The master then pulls the SS pin Low from the appropriate slave device. During the SPI clock cycle, a full duplex data transfer occurs (refer to Figure 28), the following actions occur.

- The master sends a bit on the MOSI line and the slave it reads it on the same line
- The slave sends a bit on the MISO line and the master reads it on the same line

Transmissions can involve any number of clock cycles; the most common are 8-bit and 16-bit word transfers. When there is no more data to transfer, the master stops toggling the clock and deselects the slave by pulling the SS pin High. The master can select only one slave at a time. Every slave on the bus that has not been selected must disregard the input clock and MOSI signals.

In addition to the clock frequency, the master must also configure the clock polarity (CPOL) and clock phase (CPHA).

If CPOL=0 (i.e., the clock base value is Low), the following actions occur:

- For CPHA=0, data is captured on the SCK’s rising edge (low to high transition)
- For CPHA=1, data is captured on the SCK’s falling edge (high to low transition)

If CPOL=1 (i.e., the clock base value is High), then the following actions occur:

- For CPHA=0, data is captured on the SCK's falling edge (high to low transition)
- For CPHA=1, data is captured on the SCK's rising edge (low to high transition)

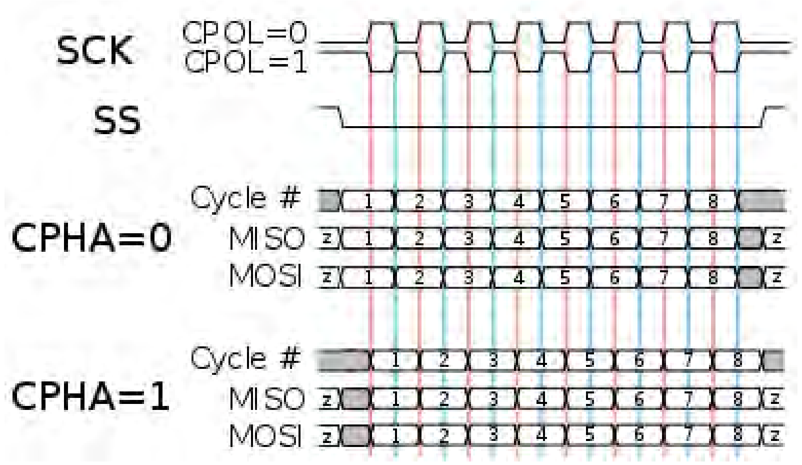


Figure 29. Data Transfer Timing Diagram

A CPHA value of 0 refers to a sampling on the leading (first) clock edge, while a CPHA of 1 means a sampling on the trailing (second) clock edge, regardless of whether that clock edge is rising or falling. For all CPOL and CPHA modes, the initial clock value must be stable before the chip select line goes active.

Table 16. SPI Modes

Mode	SCK Polarity	SCK Phase	SCK Transmit Edge	SCK Receive Edge	SCK Idle State
0	0	0	Falling	Rising	Low
1	0	1	Rising	Falling	Low
2	1	0	Rising	Falling	High
3	1	1	Falling	Rising	High

SPI Hardware Architecture

Figure 30 displays the block diagram for this application note. In this diagram, one Z16F2811 part acts as the SPI Master while another Z16F2811 part acts as the slave.

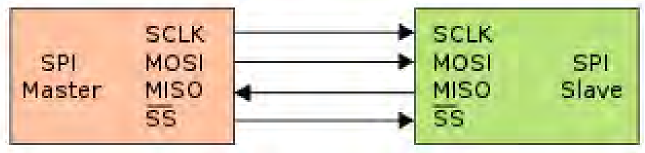


Figure 30. Design Block Diagram

SPI Software Implementation

This reference design makes use of the ZNEO CPU's enhanced serial peripheral interface. There are two sets of firmware presented in this reference design: one for the master and one for the slave; each is briefly described below. These firmware sets initialize the SPI peripheral and enable communication via an SPI protocol.

Master Mode. The master mode features the following two APIs:

- A Write API, in which the master writes to a certain address in a cyclic buffer of the slave
- A Read API, in which the master reads to a certain address in a cyclic buffer of the slave

Slave Mode. The slave saves the data to an appropriate address whenever a Write command is sent by the master. It also responds to the master by sending data being pointed to by the Read command.

Testing SPI Operation

The slave features a buffer array which stores data sent by the master. The master sends either a Write command/or a Read command to the slave. The slave responds to the master based on the command structure given by the master.

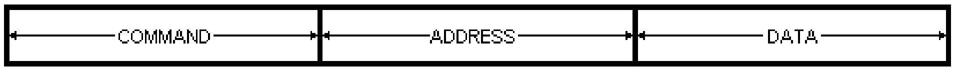


Figure 31. Master Command Structure

There are two types of commands:

- A Write command, which carries a command byte of `0xA0`
- A Read command, which carries a command byte of `0x50`

The slave determines which part of the buffer array is to be accessed by examining the address byte; the buffer array is 10 bytes long.

The data field provides the data to be written and is read by the slave when a Write command is sent. When a Read command is sent, the master reads the data sent by the slave.

Equipment Used

The following equipment is used to set up a test of the SPI peripheral:

- Two Z16FMC Series Development Kits
- Connecting wires
- PC with installed ZDS II ZNEO 4.11.1 IDE and a serial port

Setup

For the setup, please refer to Figure 28 on page 54 and Appendix A. Schematics on page 60.

Procedure

Observe the following steps to test the SPI functionality of the Z16FMC MCU.

1. Connect the two Z16FMC Series MCU development boards as indicated in Appendix A. Schematics on page 60.
2. Remove Jumper J1 on both boards.
3. Download the project file for the master and open it in the ZDS II IDE (AN0312-SC01.zip).
4. Compile and build the project. Download the hex code to one Z16FMC Series Development Board.
5. Download the project file for the slave and open it in the ZDS II IDE (AN0312-SC02.zip).
6. Compile and build the project. Download the hex code to the other Z16FMC Series Development Board.
7. Create a new Hyperterminal connection with the following properties: 57600 bps, 8 data bits, no parity with 1 stop bit and no flow control.
8. In the Hyperterminal program, navigate via the **Properties** menu to **Settings** → **ASCII Setup** → **Uncheck Echo typed characters locally**.
9. Execute the program for the master, a message prompt can be seen on the Hyperterminal.
10. Follow the message prompt.

Results

Data was successfully transmitted between the two Z16FMC Series Development Boards using the SPI protocol. The waveforms for the Read and Write operations were captured using a logic analyzer and are displayed in Figures 32 and 33.

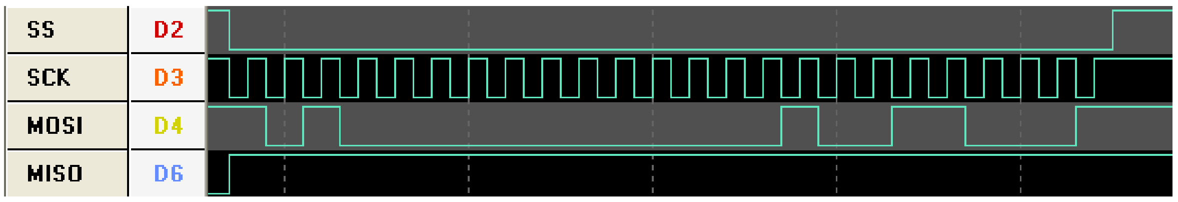


Figure 32. Master Write Transaction Waveform

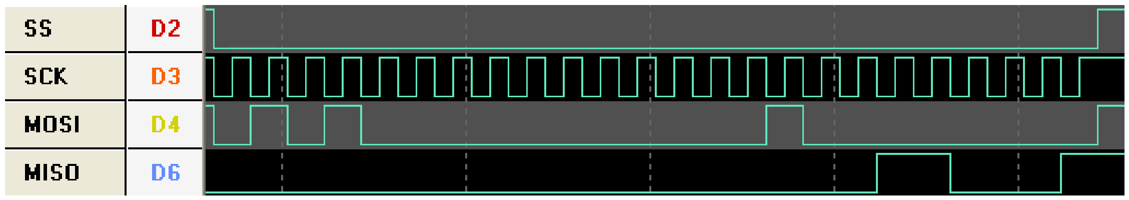
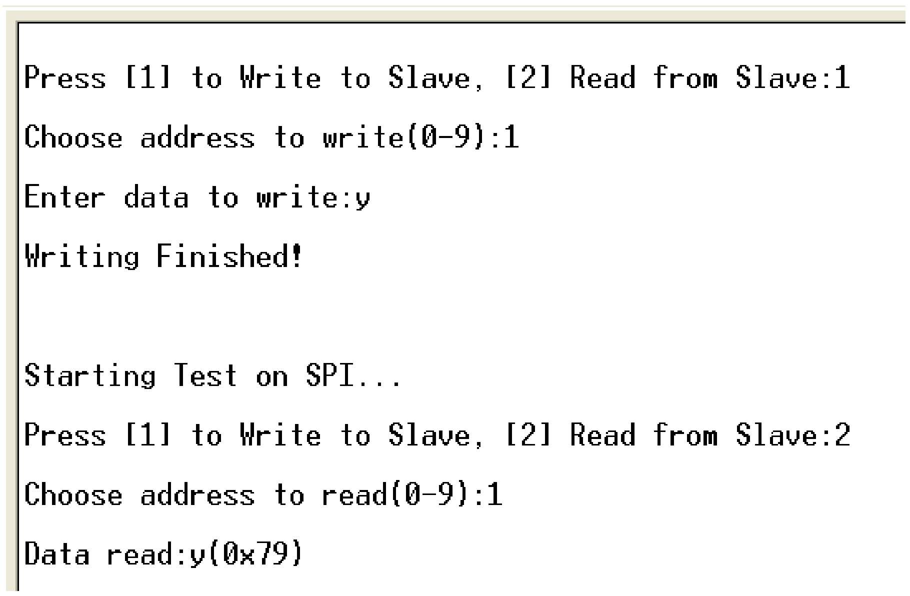


Figure 33. Master Read Transaction Waveform



```
Press [1] to Write to Slave, [2] Read from Slave:1
Choose address to write(0-9):1
Enter data to write:y
Writing Finished!


Starting Test on SPI...
Press [1] to Write to Slave, [2] Read from Slave:2
Choose address to read(0-9):1
Data read:y(0x79)
```

Figure 34. Hyperterminal Output

Summary

This Application Note discusses how to use the ADC, UART, I^{2}C, PWM, SPI and Timer peripherals of the ZNEO CPU, plus provides source code files to enable further examination and understanding of the Z16FMC Series of Flash Motor Control MCUs.

References

The documents associated with Zilog's Flash Motor Control MCUs can be found just one click away on www.zilog.com.

- ZNEO Z16F Series of Microcontrollers Development Kit User Manual (UM0202)
- ZNEO Z16F Series Product Specification (PS0220)
- Z16FMC Motor Control MCU Product Specification (PS0287)

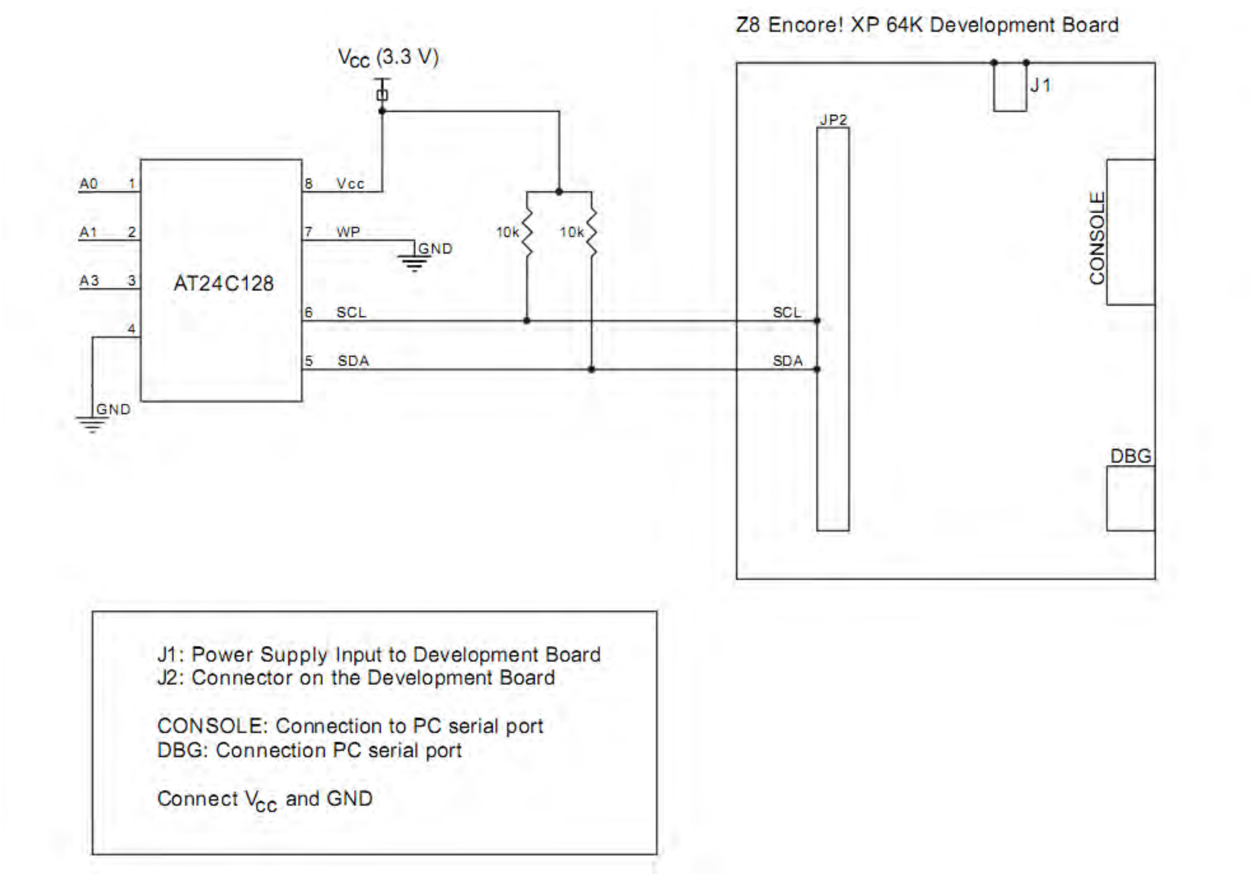
- Z16FMC Series Motor Control Development Kit User Manual (UM0234)
- Zilog Developer Studio II – ZNEO User Manual (UM0171)
- ZNEO CPU Core User Manual (UM0188)

Additional references:

- 2-wire serial EEPROM (AT24C128; www.atmel.com)
- I^{2}C protocol http://en.wikipedia.org/wiki/I%C2%B2C

Appendix A. Schematics

This appendix presents the schematic diagrams for the I^{2}C interface.

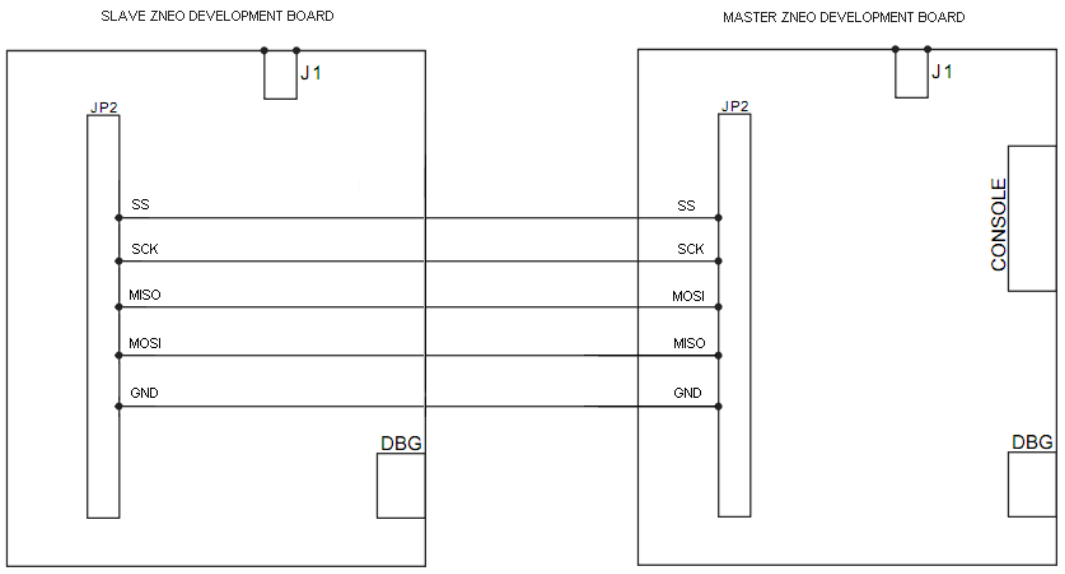


J1: Power Supply Input to Development Board
J2: Connector on the Development Board

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

Connect V_{CC} and GND

Figure 35. Setup for I^{2}C Interfacing with AT24C128



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

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

Figure 36. Setup for SPI Master-Slave Interface

Appendix B. Flowcharts

ADC Flowchart

This appendix displays the flowcharts of ZNEO Microcontroller ADC Code.

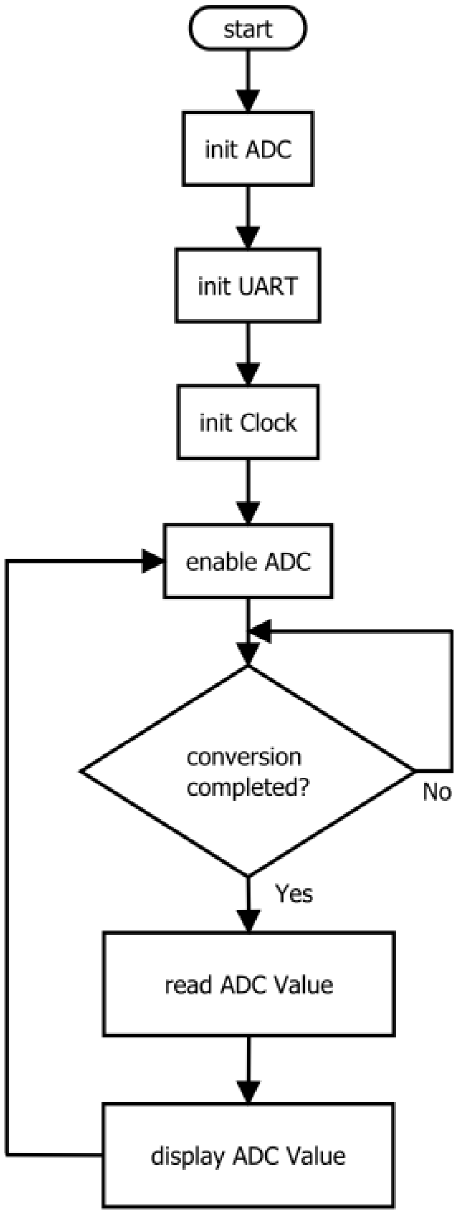


Figure 37. ADC Code Flowchart

I^{2}C Flowcharts

Figures 38 through 40 display the software flow of the I^{2}C main function as well as the Read and Write APIs.

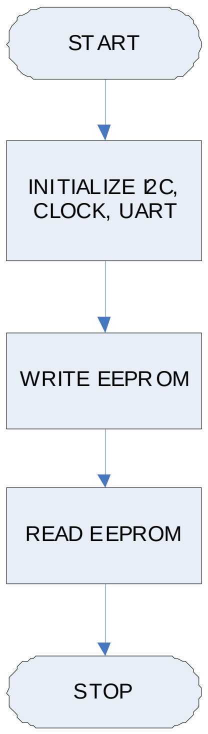


Figure 38. I^{2}C Main Function

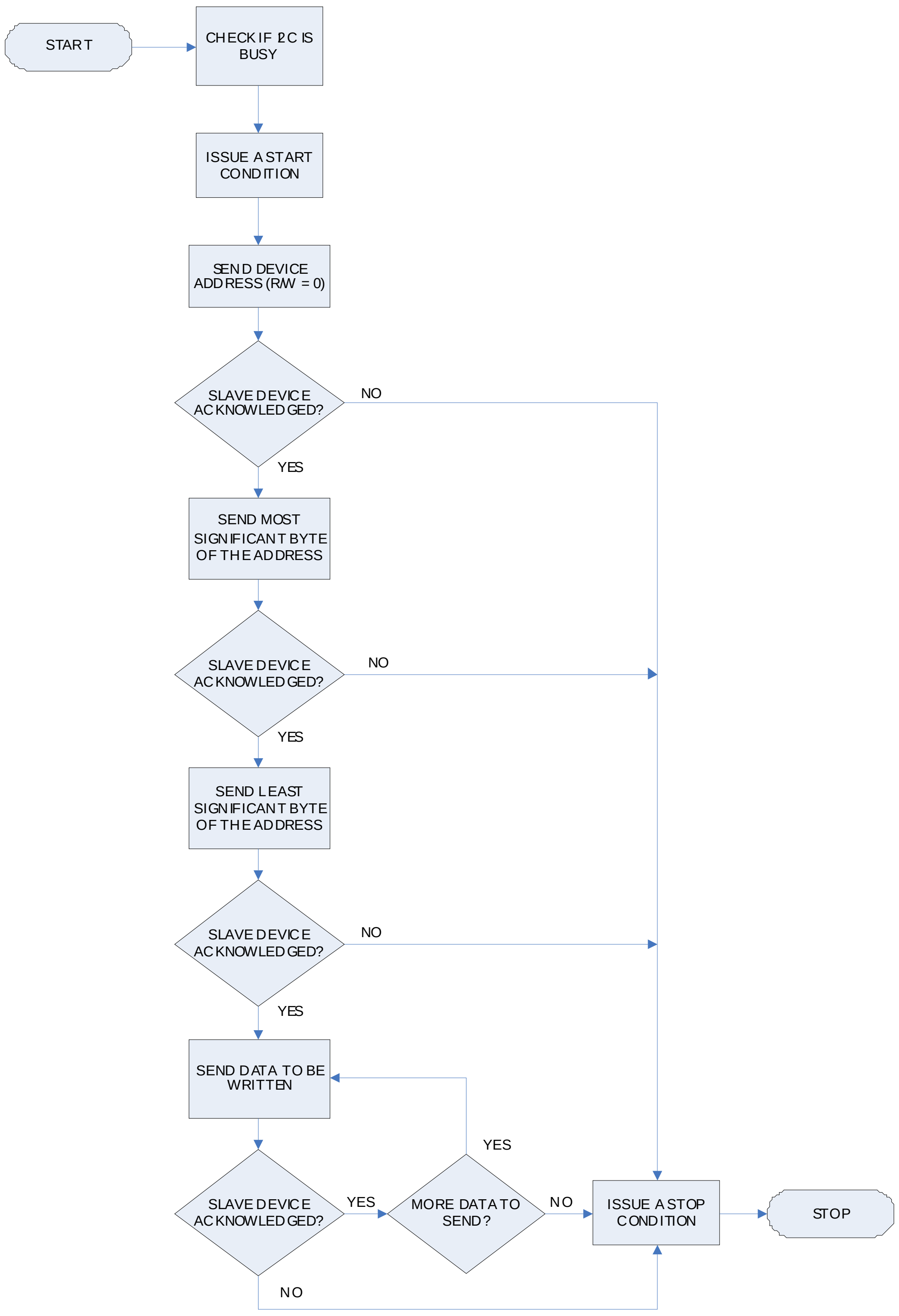

START
CHECK IF I2C IS BUSY
ISSUE A START CONDITION
SEND DEVICE ADDRESS (R/W = 0)
SLAVE DEVICE ACKNOWLEDGED?
NO
YES
SEND MOST SIGNIFICANT BYTE OF THE ADDRESS
SLAVE DEVICE ACKNOWLEDGED?
NO
YES
SEND LEAST SIGNIFICANT BYTE OF THE ADDRESS
SLAVE DEVICE ACKNOWLEDGED?
NO
YES
SEND DATA TO BE WRITTEN
YES
SLAVE DEVICE ACKNOWLEDGED?
YES
MORE DATA TO SEND?
NO
ISSUE A STOP CONDITION
STOP
NO

Figure 39. Master Write Transaction Flowchart

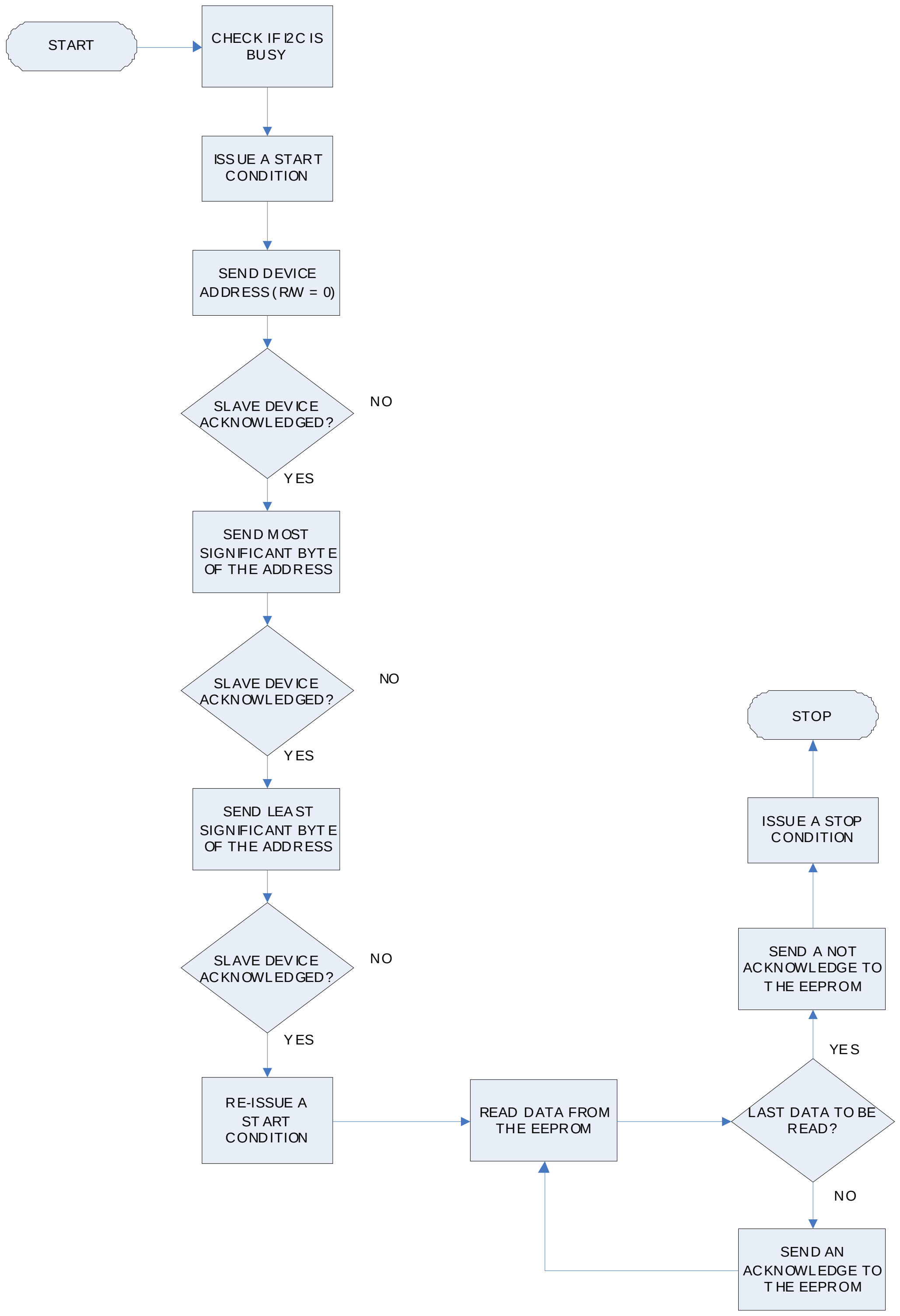

START
CHECK IF I2C IS BUSY
ISSUE A START CONDITION
SEND DEVICE ADDRESS (R/W = 0)
SLAVE DEVICE ACKNOWLEDGED?
NO
YES
SEND MOST SIGNIFICANT BYTE OF THE ADDRESS
SLAVE DEVICE ACKNOWLEDGED?
NO
YES
SEND LEAST SIGNIFICANT BYTE OF THE ADDRESS
SLAVE DEVICE ACKNOWLEDGED?
NO
YES
RE-ISSUE A START CONDITION
READ DATA FROM THE EEPROM
LAST DATA TO BE READ?
YES
NO
SEND A NOT ACKNOWLEDGE TO THE EEPROM
SEND AN ACKNOWLEDGE TO THE EEPROM
ISSUE A STOP CONDITION
STOP

Figure 40. Master Read Transaction Flowchart

SPI Flowcharts

Figures 41 and 42 display the software flow of the SPI main function in Master and Slave modes, respectively.

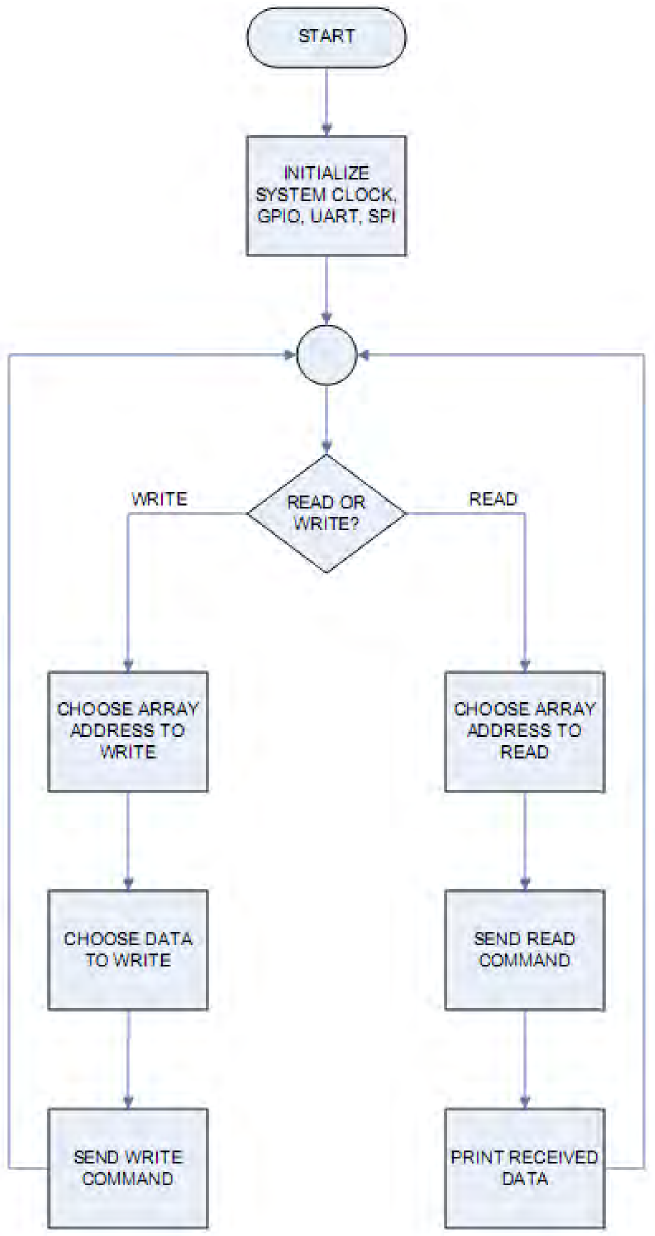


Figure 41. SPI Main Function (Master Mode)

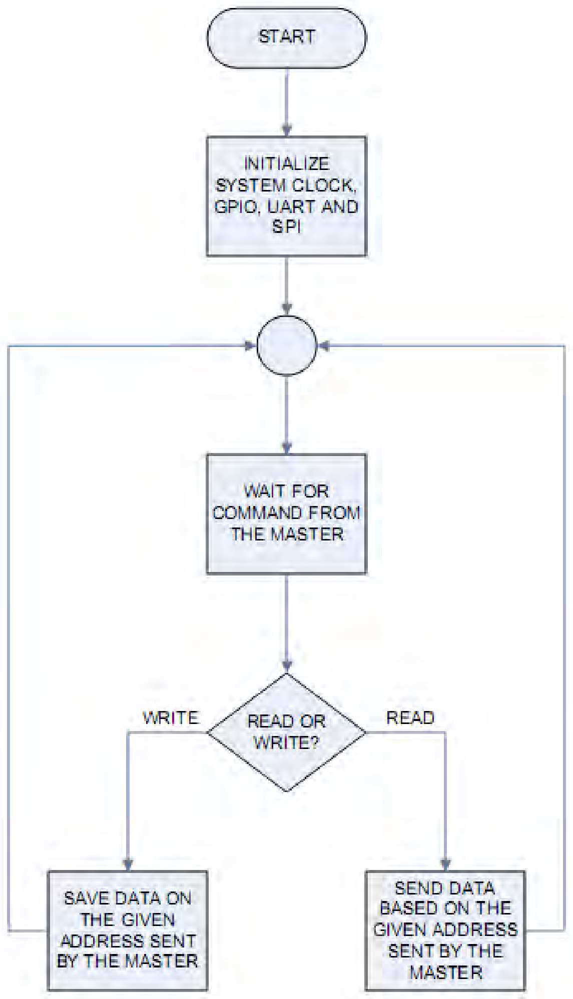

START
INITIALIZE SYSTEM CLOCK, GPIO, UART AND SPI
WAIT FOR COMMAND FROM THE MASTER
READ OR WRITE?
WRITE
READ
SAVE DATA ON THE GIVEN ADDRESS SENT BY THE MASTER
SEND DATA BASED ON THE GIVEN ADDRESS SENT BY THE MASTER

Figure 42. SPI Main Function (Slave Mode)

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.