POLITECNICO DI TORINO

DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION ENGINEERING

THESIS DEGREE OF MASTERS

Functional Testing of Microprocessor Peripheral Drivers for Nano Satellites

Advisor:Prof. Leonardo Reyneri

Candidate: Muluneh Hailu Heyi

July,2014

ACKNOWLEDGEMENT

First of all I thank the almighty God for blessing and guiding me for making me able to accomplish my thesis work and in all my journey of life.
I would like to express my deepest gratitude to my thesis advisor Professor Leonardo Reyneri for his very useful advice and continuous guidance throughout the progress of this Thesis. I would like to express my heartfelt appreciation for his keen cooperation in all part of the project.
Also I would like to use this opportunity to thank POLITECNICO DI TORINO for giving me an opportunity to study with a variety of excellent people, highly qualified instructors, and an excellent environment to study.
Thank you my family for all the love and support you gave me since childhood, specially my three sisters for all the sacrifices they make to make me stand up on my own and my ante and her husband for being by my side always.

Muluneh Hailu Heyi

ABSTRACT

This paper explain about the work which I have done to test the peripheral drivers of MSP430 microcontroller families. The driver define all the operations and settings of MSP430F5xx, MSP430F4xx, MSP430FG41xx and MSP430F2xx families of microcontrollers. The drivers have been written in C++.
The test program used to test the driver unites by configuring and setting the microcontroller based on the operation and setting defined in the drivers. When the microcontroller peripheral behave differently than expected after they are configured with the existed drivers, I had to determine the fault code and modified the driver unit. For doing this I have to know about the structure and characteristics of MSP430 microcontroller.
Microcontroller based applications are usually debugged with the assistance of In-circuit emulators and logic analyzers. However, these traditional debug tools represent a huge investment for use. The development of a new low-cost debug tool that uses functional test to implement the basic functionality provided by an In-circuit emulator and a logic analyzer is a possible solution to overcome this economical problem.
A test developed on the basis of the functional information about the module under test aims at testing the functions rather than the faults (black box testing).
For developing the test the system use PC which used to run the software IAR Embedded Workbench Software and MSP-TS430PZ5x100 program development tool from Texas Instrument which allows programming and debugging of the microcontroller through JTAG interface. The system allows testing of the following peripheral drivers: Clock Generator, Timer, ADC, PWM, FLASH, CRC and USCI Modules (UART, SPI and I2C).
This paper first give a detailed introduction about the module and then it shows how to configure the module and how the functional test is performed. Finally it conclude the by analyzing the result found from the test.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

$ACLK$	Auxiliary Clock See Basic Clock Module
ADC	Analog to Digital Converter
BSL	Bootstrap Loader
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DCO	Digitally Controlled Oscillator
FLL	Frequency Locked Loop
Hz	Hertz
$I2C$	Inter-Integrated Circuit
ISR	Interrupt Service Routine
PWM	Pulse Width Modulation
$JTAG$	Joint Test Action Group
$LFSR$	Linear feedback shift register
$MCLK$	Master Clock
$MSP430$	mixed-signal microcontroller f
$SMCLK$	Sub-System Master Clock
SPI	Serial Peripheral Interface
TI	Texas Instruments
$UART$	Universal Asynchronous Receiver and Transmitter
$USCI$	Universal Serial Communication Interfaces
WDT	Watchdog Timer

Chapter 1

INTRODUCTION

1.1 Background

This paper explain about the work which I have done to test the peripheral drivers of MSP430 microcontroller families. The driver define all the operations and settings of MSP430F5xx, MSP430F4xx, MSP430FG41xx and MSP430F2xx families of microcontrollers. The drivers have been written in C++.
The test program used to test the driver unites by configuring and setting the microcontroller based on the operation and setting defined in the drivers. When the microcontroller peripheral behave differently than expected after they are configured with the existed drivers, I had to determine the fault code and modified the driver unit. For doing this I have to know about the structure and characteristics of MSP430 microcontroller.
The MSP430 is an Ultra-low power 16 bit RISC mixed-signal microprocessors family from Texas Instruments which is used for low power consumption embedded application. It consumes only 1/3 of current power consumption of previous technology implementations. As shown in fig 1 MSP430 provide several peripherals that allow designers to create applications quickly and easily. And these peripherals can operate without CPU intervention, reducing power consumption significantly. Different members of the family include different peripherals and different amounts of memory. They all include a JTAG interface. Product designers choose the particular family member that best suits their application, generally speaking, cost and power consumption increases with sophistication.

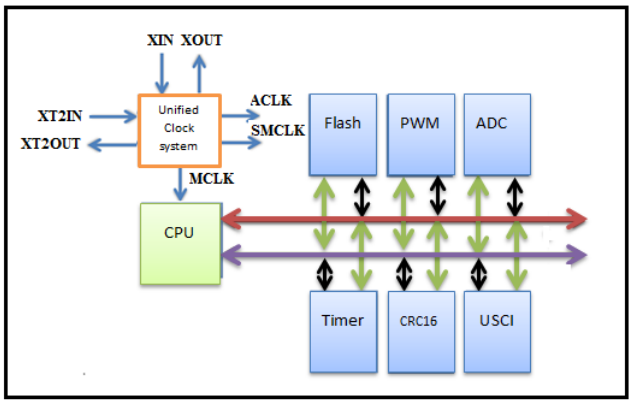

Figure 1.1: commone periperales of MSP430

Within this thesis I have tried to implement the functional test of the drivers of the following peripheral units of MSP430F5xx families:

- The Unified Clock System Module
- The Timer Module
- The PWM
- The ADC Module
- The CRC Module
- The FLASH Memory
- USCI Module
    - SPI
    - I2C
    - UART

In order to test these peripheral driver, the peripheral needs to be properly configured, enabled and should get a proper input. Since each of these peripherals are defined by a set of registers, we control peripherals by setting 0s and 1s bits of the registers found within the peripheral modules.
Functional test of Microcontroller drivers is based on uploading some test signal or data to the module of microcontroller under test and forcing the module testing the code then checking the produced results. For testing all the above peripheral drivers modules I have used the following hardware and software:

- PC for running the software IAR Embedded Workbench.
- IAR Embedded Workbench Software is used to compile the MSP430 application program and also it used for Programming and Debugging interface of the Microcontroller.
- JTAG allows a user to download and debug the code on the microcontroller. This interface is accessed by using a FET Programmer that connects the computer to the microcontroller.
- MSP-TS430PZ5x100 is the program development tool for the MSP430 ultralow-power microcontroller. It provides the following for the microcontroller:
    - Supply voltage consistent with Electrical Specifications (1.8V - 3.6V for most MSP430).
    - Decoupling capacitors to reduce noise on the supply voltage (no power supply is perfect).
    - 8MHz External Crystals which is used for the clock generation.
    - A Programmer/Debugger (JTAG) connector interface.
    - LEDs
- Oscilloscope which is used to check the output result of testing some peripheral modules such as clock, timer and PWM modules.
- Signal Generator which is used to give input to testing ADC peripherals modules
- One of MSP430 series microcontroller. For this thesis I have used only MSP430F5438A microcontroller.

The following chapters give the detail explanation about the test mechanism that I have used for each module drivers.

Figure 1.2: hardware and software used for the implementation

Chapter 2

TESTING UNIFIED CLOCK SYSTEM MODULE

2.1 Introduction

All the systems in microcontroller to work together the microcontroller need continues pulses of signal. This signal is called Clock. The clock system on the MSP430 is designed to be flexible and low power. Based upon their operating speeds the MSP430 classifies the peripherals into two categories slow and fast peripherals units. This categorization of peripherals is mainly to reduce the power consumption of CPU and the peripherals, i.e., high frequency operations require more power that compared with low frequency. So for fast system we use high frequency and for slow system we use low frequency clock. To implement these MSP430 provides different clock sources. By choosing the minimum clock speed necessary for a given module, power consumption is reduced and the particular synchronization needs of the module can be met.
Generally the MSP430 has five clock sources for generating three kinds of clocks. The source could be internal RC type oscillators or internal oscillator using external crystals.
The MSP430 can contain several internal oscillators. The internal oscillators are based on an RC network. The Digital Controlled Oscillator (DOC) and the VLO low frequency oscillator are based on internal oscillators. The DCO is digitally controlled because its frequency can be changed from several hundred kHz up to 25MHz.
The MSP430 also use external crystals with internal oscillator circuitry to generate both low frequency and high frequency (Up to 25MHz) clocks that are as accurate as the crystal used. MSP-TS430PZ5x100 development tool has 8MHZ crystal on the board. So for my testing system I used 8MHZ external clock.

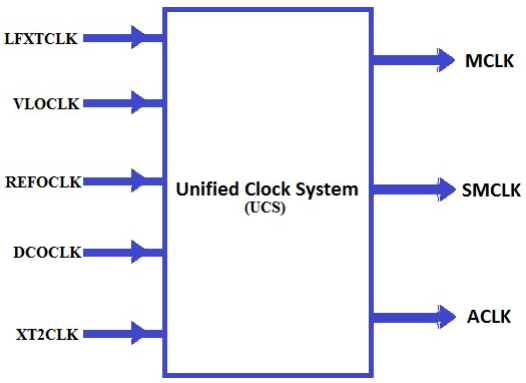

Figure 2.1: Input and output of the clock module

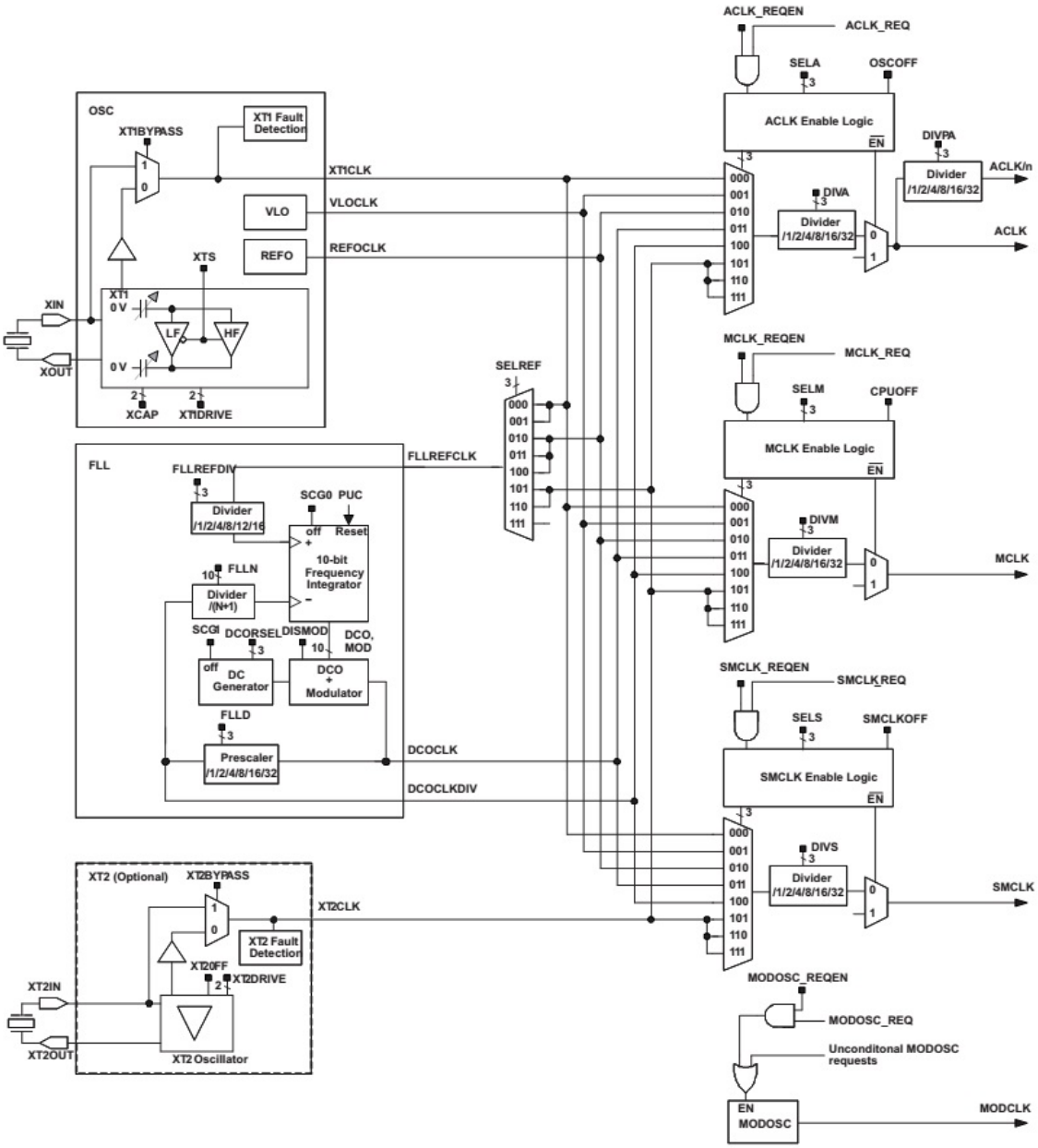

Figure 2.2: The clock module circuit taken form the datasheet

2.1.1 Clock Sources

MSP430 Clock sources include both crystals and internal oscillators. Not all MSP430 series have the same clock sources. For my system have used MSP430F5438A microcontoller which has five clock sources are:

- **LFXTCLK (Low Frequency/ High Frequency Oscillator Crystal Clock)** it mainly uses an external watch crystal which is connected to the pins XIN and XOUT of the microcontroller. In my case the source is connected to 8MHz external crystal. XT1CLK can be used as a clock reference into the FLL(Frequency Locked Loop).

- **XT2CLK (High Frequency Oscillator Crystal Clock)** this signal is the optional external clock source, and it is connected to the XT2IN and XT2OUT pins. In general, this signal is meant to be the high-speed clock source.
- **Digitally Controlled Oscillator Clock (DCOCLK)** is internally generates clock input, and it is the default clock source for the master clock upon reset.
- **VLOCLK**: Internal low frequency oscillator with 10-kHz nominal frequency. Its low frequency means very low power.
- **REFOCLK**:Internal, trimmed, low-frequency oscillator with 32768 Hz typical frequency, with the ability to be used as a clock reference into the FLL(Frequency Locked Loop .

The user of the MSP430 has flexibility to select the clock source. you can avoid all the external crystal and you can use only DCOCLK and VLOCLK or if you need more precision, use the external crystals at the expense of PCB space and some money. It is standard practice to use LFXT1 with a 32.768 kHz crystal, leaving XT2 to be used with a high frequency crystal.

2.1.2 Clock Outputs

The three clock outputs are:

- **Auxiliary Clock (ACLK)** -is use for slower subsystems to use in order to conserve power.it is generated by XT1CLK, REFOCLK, VLOCLK or DCOCLK clock sources.its clock can be divide with 1, 2, 4, 8, 16 and 32.it also software selctable by individual peripheral modules.
- **Master Clock (MCLK)** This clock is used as a clock source for the CPU and a few peripherals. This clock must be working properly for the processor to execute instructions.it is generated by XT1CLK, REFOCLK, VLOCLK, DCOCLK clock sources.MCLK can be divided by 1, 2, 4, 8, 16, or 32.
- **Sub master Clock (SMCLK)** - This clock is the source for most peripherals, and its source can XT1CLK, REFOCLK, VLOCLK or DCOCLK. SMCLK can be divided by 1, 2, 4, 8, 16, or 32.

Typically SMCLK runs at the same frequency as MCLK, both in the megahertz range. ACLK is often derived from a watch crystal and therefore runs at a much lower frequency. The CPU is always sourced by MCLK. Other peripherals are sourced by either SMCLK, MCLK, and ACLK. It is important to consider all the modules in the system and their frequency requirements to select their source.

2.1.3 Basic Clock Module Control Register

The system clock generator interacts with other parts using three module registers. These three module registers uses to control the basic clock generation. Each register is 8 bits in size. The user software handles the clock system requirements using these registers which are only addressable using byte instructions. The three registers are:

- Digitally Control Register (DCOCTL)
- Oscillator and Clock Control Register
  - Basic Clock System Control Register One (BCSCTL1)
  - Basic Clock System Control Register two (BCSCTL2)
- Special Function Register Bits (OFIE and OFIFG)

DCOCTL is eight bit register which used for DCO frequency selection and modulation selection. From bit 5 to bit 7 used as DCO frequency selection(DOCx) and form bit 4 to bit 0 used as modulation selection(MODx). DCO0, DCO1, DCO2 bits defines which one of the eight discrete frequencies selected. MOD0, MOD1, MOD2, MOD3, MOD4 bits define how often discrete frequency within the period of 32 DCOCLK cycle used.

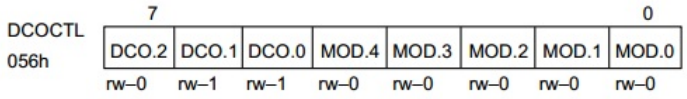

Figure 2.3: DCOCTL register bits

BCSCTL1 is eight bit register

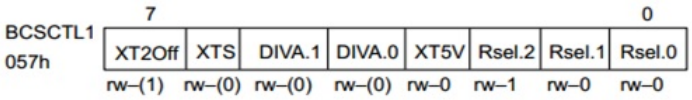

Figure 2.4: BCSCTL1 register bits

- Bit7 (XT2Off): is used to activate and deactivate XT2 Oscillator. If it is 1 it activate the oscillator and if it is 0 it will turn of the oscillator if it is not used for MCLK or SMCLK.
- Bit6 (XTS): is used to set the frequency mode of LFXT1. If it is 0 LFXT1the low-frequency oscillator is selected and if it is 1 the high-frequency oscillator is selected.
- Bit5-4(DIVAx): is used to set the divider for ACLK. It set the divider to 1,2,4,8 based the setting bits 00,01,10,11 respectively
- Bit3 (XT5V): used to set XT1 and XT2 to operate with 5v or with reduced voltage. If it is 0 the oscillator operates across the full supply voltage, but has an increased current consumption at 5 V. and if it is 1 the current consumption for the oscillator is reduced if the supply voltage is around 5 V.
- Bit2-0(Rselx): is used to select internal resistor. The value of the resistor defines the nominal frequency. The lowest nominal frequency is selected by setting RSELx=0.

BCSCTL2 is eight bit register

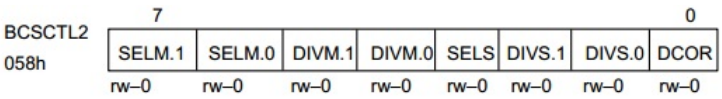

Figure 2.5: BCSCTL2 register bits

- Bit7-6(SELMx): is used to set the input clock source for generating MCLK
- Bit5-4(DIVMx): is used to set the divider for MCLK.
- Bit3(SELS): is used to set the input clock source for generating SMCLK
- Bit2-1(DIVSx): is used to set the divider for SMCLK.

- Bit0 (DCOR): is used to select the resistor for DCO.

The basic clock module uses two bits in the special function registers, OFIFG and OFIE. The oscillator fault interrupt enable (OFIE) bit is located in bit 1 of the interrupt enable register IE1. The oscillator fault interrupt flag (OFIFG) bit is located in bit 1 of the interrupt flag register IFG1.

2.2 Unified Clock System Driver

The figure below shows the class diagram of clock module driver which have operations(functions) and attributes which are used to set and configure the clock module of the microcontroller. This module uses for MSP430x43x,MSP430x24x and MSP430x54x families of Msp430 microcontrollers.

Figure 2.6: Clock Module Class diagram

The Attributes of this class used to declare and initialize the output of the clock sources and crystal input of the clock. Attributes of the Clock Module Driver class are:

- **private CLOCK_FREQ : ulong** : Crystal frequency, in Hz.
- **private ACLK_FREQ : ulong** : Frequency of ACLK (CPU auxiliary clock), in Hz.
- **private MCLK_FREQ** : ulong: Frequency of MCLK (CPU main clock), in Hz.
- **private SMCLK_FREQ : ulong**: Frequency of SMCLK (CPU secondary main clock), in Hz.
- **private CLOCK_SOURCE : t_ClockSource** : used to access the enumeration t_ClockSource which used to select the different clock sources.
- **private TIMEOUT : ushort** : Deactivates first oscillator (XT1).

The clock module driver also has operations which used to activate and deactivate the crystal clock sources and for setting and configuring the output of the clock.Functions(operations) of the Clock Module Driver class

- **public activateXT1 (freq : ulong)** : void - Activates first oscillator (XT1) and configures it to operate at frequency freq, in Hz.
- **public deactivateXT1 () : void** - Deactivates first oscillator (XT1).
- **public activateXT2 (freq : ulong) : void**- Activates second oscillator (XT2) and configures it to operate at frequency freq, in Hz.
- **public deactivateXT2 () : void** - Deactivates first oscillator (XT2).
- **public setMCLK (source : t_ClockSource, divide : ushort) : bool** - Selects source for MCLK (CPU main clock) from the source defined by argument source and sets clock division factor to divide. Clock frequency (in Hz) is stored into attribute MCLK_FREQ: ulong. Argument divide must be a power of 2; otherwise, the function returns false; the function returns false also for unsupported clock sources or in case of clock faults. In all other cases, it returns true.
- **public setACLK (source : t_ClockSource, divide : ushort) : bool** - Selects source for ACLK (CPU auxiliary clock) from the source defined by argument source and sets clock division factor to divide. Clock frequency (in Hz) is stored into attribute ACLK_FREQ: ulong. Argument divide must be a power of 2; otherwise, the function returns false; the function returns false also for unsupported clock sources or in case of clock faults. In all other cases, it returns true.
- **public setSMCLK (source : t_ClockSource, divide : ushort) : bool** - Selects source for SMCLK (CPU secondary main clock) from the source defined by argument source and sets clock division factor to divide. Clock frequency (in Hz) is stored into attribute SMCLK_FREQ: ulong. Argument divide must be a power of 2; otherwise, the function returns false; the function returns false also for unsupported clock sources or in case of clock faults. In all other cases, it returns true.
- **public setDCO (range : byte, freqFactor : ushort) : void** - Configures DCO to operate in frequency range given by parameter range, with frequency factor given by parameter freqFactor. For further details, read the processor manuals.
- **public setFLL (source : t_ClockSource, divide1 : ushort, divide2 : ushort, multiply : ushort) : bool** - Configures FLL (frequency locked loop) to lock to source given by source with FLL dividers given by divide1 and divide2 and the FLL multiplier given by multiply. For further details, read the processor manuals.
- **public get_MCLK_frequency () : unsigned long** - Returns frequency of MCLK, in Hz.
- **public get_ACLK_frequency () : unsigned long** - Returns frequency of ACLK, in Hz.
- **public get_SMCLK_frequency (): unsigned long** - Returns frequency of SMCLK, in Hz.
- **public resetFlags () : void** - reset fault flags and reset conditional requests for clocks
- **public activateSingleClock (clock : t_ClockSource) : void** - it activate clock source based on clock : t_ClockSource.
- **public activateSingleClock () : void** - it activate clock source by calling activateSingleClock (clock : t_ClockSource) function with clock=XT1CLK default value.

- **public init () : void** - it initialize the clock system by activating the XT1CLK clock source and setting the three clock outputs.
- **public output_MCLK (on : bool) : void** -If on==true, MCLK is output on the corresponding pin. If on==false, MCLK is not output and the corresponding pin is available as a normal IO, configured as an input.
- **public output_ACLK (on : bool) : void** -If on==true, ACLK is output on the corresponding pin. If on==false, ACLK is not output and the corresponding pin is available as a normal IO, configured as an input.
- **public output_SMCLK (on : bool) : void** -If on==true, SMCLK is output on the corresponding pin. If on==false, SMCLK is not output and the corresponding pin is available as a normal IO, configured as an input.

2.3 Test program for Unified Clock System Driver

This diagram describe how the test program for clock driver is composed. As mentioned above the clock driver work for different MSP430 families but to implement the test I have used only MSP430F5438A microcontroller.

Figure 2.7: Test program for Clock Module driver

2.3.1 Class Testing Clock Generator

Testing_ClockGenerator is a class used to test the the Unified clock system driver functionally. This class has two attributes:
private source : t_ClockSource - let the user to set the different input for clock source: such as XT1CLK, XT2CLK, DCOCLK, VLOCLK and REFOCLK
private cpu : MSP_430F5438A - define which microcontroller series is under test. and it uses to access the clock driver.
This class also has one main function(operator). The source code for testing the main function is found in appendix A. The code let the user to select the input clock sources by entering letters from 'a' to 'e' and also the clock divider by letting the user to insert the numbers 1, 2, 4, 8, 16 and 32. then set the output clocks frequency ACLK_FREQ, SMCLK_FREQ and MCLK_FREQ.
The algorithm I followed to write the test code:

- First we need to disable the watch dog of the microcontroller to protect software rest every time. The code used to disable the watch dog is:

  cpu.wdt.disable();

- Set the pin for the clock outputs: ACLK, SMCLK and MCLK. For setting ACLK clock output you have to set P1.0 pin to output direction and use PxSEL to set the signal ACLK.

  cpu.clock.output_ACLK(true);

  For setting MCLK clock output you have to set P2.0 pin to output direction and use PxSEL to set the signal MCLK

  cpu.clock.output_MCLK(true);

  For setting SMCLK clock output you have to set P1.6 pin to output direction and use PxSEL to set the signal SMCLK.

  cpu.clock.output_SMCLK(true);

- Set the input for the clock source. Since there are five clock input source so we have to check all the input sources by setting each step by step. If the microcontroller needs external crystal we should have to activate the input and output pins for crystal connection to the microcontroller by setting the direction register and the function select register. The direction registers PxDIR are used to specify individual bits for input or output. For input, the bit it should be 0, and for output it should be 1. The function select registers PxSEL are used to choose a signal function. For setting LFXTCLK you have to connect external crystal to the microcontroller by setting P7.0 input terminal for crystal oscillator XT1 and P7.1 output terminal of crystal oscillator XT1 using the code:
  For setting XT2CLK you have to connect external crystal to the microcontroller by setting P5.2 input terminal for crystal oscillator XT2 and P5.3 output terminal of crystal oscillator XT2 using the code:
  The main function let the user to insert the letter to activate and deactivate different clock sources and let the user to insert numbers to select the divider for the clock.

- By using clock module control register configure the clock system

- To choose the input source for the output clock call the following function which used to activate the selected clock source based on the parameter that passed to **source**.

```
cpu.clock.activateSingleClock(source)
```

- To set the output frequency by using the setting the divider call the following functions which are used to set the frequency os ACLK, SMCLK and MCLK.

```
cpu.clock.setACLK(source ,divide);
cpu.clock.setMCLK(source ,divide);
cpu.clock.setSMCLK(source ,divide);
```

- To activate and deactivate the external crystals oscillator

If some one want to change this test program for other microcontroller series which are supported by clock driver,there is no need to change the whole program we only need to replace MSP_430F5438A class by other.

2.4 Testing Procedure of Unified Clock System Driver

Testing of the clock generator performed by setting the clock input source, configures the frequency and measures the output of the clock with the oscilloscope from the output pins. Follow the following step to configure the microcontroller for testing the unified clock system:-

- Connect the output pin to the oscilloscope. The user has to first connect the oscilloscope with the pins of ACLK, MCLK and SMCLK of the microcontroller. The pins are shown in the tables below :

Table 2.1: Clock Input pins for MSP430F5438A

Inputs Pins	Pins
X2IN	P5.2
X2OUT	P5.3
XTIN	P7.0
XTOUT	P7.1

Table 2.2: Clock Output pins for MSP430F5438A

Output Pins	Pins
ACLK	P1.0
MCLK	P2.0
SMCLK	P1.6

- The source code for testing the clock module is found in appendix A. open the IAR workbench and load the code to the microcontroller using the JTAG cable. The IAR has Input output panel

and it let the user to select each input source by using a letters and it also divide the input clock source by entering a numbers. To select the different clock sources the user insert a letters from a to e.

- Letter a selects XT1CLK.
- Letter b selects VLOCLK.
- Letter c selects REFOCLK.
- Letter d selects DCOCLK.
- Letter e selects DCOCLKDIV.

For selecting the divider the user insert the number 1, 2, 4, 8, 16 and 32.
so first press the correct letter from the keyboard then press enter then press the correct number from the keyboard then press then press enter.

- Check the result form the oscilloscope by selecting the possible input sources and divider values. If the output result is based on your set up it means the clock driver is working else not working.

2.5 Testing Result

The following tables shows the test result for each clock outputs. The column shows the frequency result for each divider value at one clock source.

Table 2.3: ACLK output result

Divider	1	2	4	8	16	32
frequency of ACLK for XT1CLK	8MHZ	4 MHZ	2MHZ	1MHZ	500KHZ	250KHZ
frequency of ACLK for VLOCLK	9.259KHZ	4.630KHZ	2.315KHz	1.157KHz	578.7HZ	289HZ
frequency of ACLK for REFOCLK	32.26KHZ	16.39KHZ	8.197KHZ	4.098KHZ	2.053KHZ	1.027KHz
frequency of ACLK for DCOCLK	2 .128MHz	1.042MHZ	529.1KHZ	264.7KHZ	131.9KHZ	65.79KHZ
frequency of ACLK for DCOCLKDIV	1.064MHz	526.3KHZ	264.6KHZ	131.9KHZ	65.96KHZ	33KHZ

Table 2.4: MCLK output result

Divider	1	2	4	8	16	32
frequency of MCLK for XT1CLK	8MHZ	4 MHZ	2MHZ	1MHZ	500KHZ	250KHZ
frequency of MCLK for VLOCLK	9.259KHZ	4.630KHZ	2.315KHz	1.157KHz	578.7HZ	289HZ
frequency of MCLK for REFOCLK	32.26KHZ	16.39KHZ	8.197KHZ	4.098KHZ	2.053KHZ	1.027KHz
frequency of MCLK for DCOCLK	2 .128MHz	1.042MHZ	529.1KHZ	264.7KHZ	131.9KHZ	65.79KHZ
frequency of MCLK for DCOCLKDIV	1.064MHz	526.3KHZ	264.6KHZ	131.9KHZ	65.96KHZ	33KHZ

Table 2.5: SMCLK output result

Divider	1	2	4	8	16	32
frequency of SMCLK for XT1CLK	8MHZ	4 MHZ	2MHZ	1MHZ	500KHZ	250KHZ
frequency of SMCLK for VLOCLK	9.259KHZ	4.630KHZ	2.315KHz	1.157KHz	578.7HZ	289HZ
frequency of SMCLK for REFOCLK	32.26KHZ	16.39KHZ	8.197KHZ	4.098KHZ	2.053KHZ	1.027KHz
frequency of SMCLK for DCOCLK	2 .128MHz	1.042MHZ	529.1KHZ	264.7KHZ	131.9KHZ	65.79KHZ
frequency of SMCLK for DCOCLKDIV	1.064MHz	526.3KHZ	264.6KHZ	131.9KHZ	65.96KHZ	33KHZ

2.5.1 Conclusion

To say the system is functionally working or not we have to know the value for each input clock source:

- since the board has 8MHZ external crystal, the value of XT1CLK frequency is 8MHZ.
- The internal clock source VLOCLK has typical 10 kHz frequency.
- REFOCLK has typical 32.768kHz.
- DCOCLK has typical 2 MHz.
- DCOCLKDIV has typical 1 MHz.

From the table we can see that the frequency of the output clock at divider 1 is almost equal with the input clock source and the output clock also divided correctly for other divider setups. So Based on the result that we get from the oscilloscope for each clock output sources and the input clock source that we provide we can say that the unified clock module driver is work very well.

Chapter 3

TESTING TIMER MODULE

3.1 Introduction

Timers are fundamentally counters driven by a clock signal, commonly incrementing or decrementing the counter on each clock tick. It is simply a register. Most MSP430 timers have the resolution of 16 bit. So16 bits timer is 16 bits wider and is capable of holding a number from 0 to 65535.
The timer starts counting based on the clock input till it reaches the maximum value. When the timer reaches to its predefined value the timer generates an interrupt.so the timer to function perfectly it need the following three components.

- A clock input that tick at a constant rate
- A counter that count UP or DOWN based on the mode setting
- An interrupt which is set when the counter reaches its limit.

Figure 3.1: general input output result of timer

The timer clocks can be sourced from both ACLK and SMCLK or from two special external sources INCLK or TBCLK(TAxCLK).
In this MSP430 family there are two different 16-bit timer modules which are represented by TimerA and TimerB. Timer B is larger and more versatile than TimerA

3.2 TimerA

There are two TimerA blocks in MSP430 family: TimerA0 and TimerA1. Both have the same function but have there own different block. TimerA is a 16-bit timer/counter with seven capture/compare registers.
TimerA can support multiple capture/compares, PWM outputs, and interval timing. TimerA also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers. It is Asynchronous 16-bit timer/counter with four operating modes: UP, DOWN, CONTINOUS and UP/DOWN modes. It has configurable and

selectable clock source. We can select ACLK clock or SMCLK clock based on the value sated for register. It has configurable output with PWM capability.

3.2.1 TimerA Registers

TimerA has its own 5 kinds of registers:

- TimerA Control Register (TAxCTL)
- TimerA Counter Register (TAxR)
- seven TimerA Capture/Compare Register (TAxCCRx)
- seven TimerA Capture/Compare Control Register (TAxCCTLx)
- TimerA Interupt Vector (TAIV)

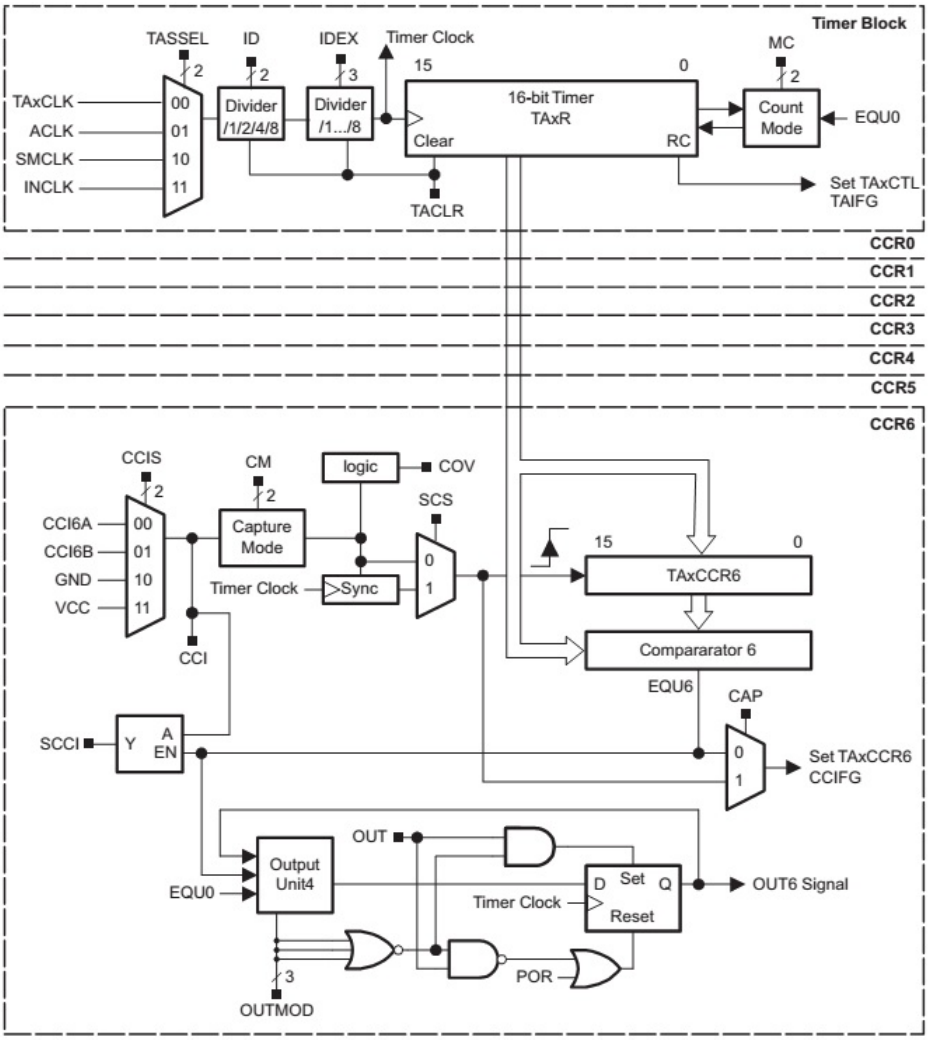

Figure 3.2: timerA internal block diagram taken from the data sheet

TimerA Control Register(TAxCTL)

TimerA Control Register (TAxCTL) is 16 bit register and each bit has their own function to select the clock source or to determine the frequency by using the divider or to set the mode of the operation for the timer.
Bits15-10 and Bit 3 are Unused Bits.
Bits 9-8 TASSELx Bits used to select the clock source.

- If TASSELx is 00 the clock source will be TACLK
- If TASSELx is 01 the clock source will be ACLK
- If TASSELx is 10 the clock source will be SMCLK
- If TASSELx is 11 the clock source will be INCLK

Bits7-6 IDx Bits is used to divide the selected input clock. It didvide the input clock by 1,2,4 and 8.

- If IDx is 00 the input clock will be divide 1
- If IDx 01 the input clock will be divide by 2
- If IDx 10 the input clock will be divide by 4
- If IDx 11 the input clock will be divide 8

Bits5-4 MCx Bits is used to set the mode of the counter.

- If MCx is 00 the timer will be in Stop mode. the timer is stop counting
- If MCx is 01 the timer will be in Up mode: the timer counts up to TACCR0 by incrementing one in each clock cycle. But first we have to set the value of TACCR0.
- If MCx is 10 the timer will be in Continuous mode. The timer counts up to the maximum value (65535).
- If MCx is 11 the timer will be in Up/down mode. The timer counts up to TACCR0 by incrementing then it countdown to 0 by decrementing.

Bit 2 TACLR is used to clear the timer. Setting this bit resets TAR register, the TACLK divider, and the count direction. The TACLR bit is automatically reset and is always read as zero. Bit 1 TAIE is used to enable the interrupt of timer. This bit enables the TAIFG interrupt request.

- If it is 0 the Interrupt will be disabled
- If it is 1 the Interrupt will be Interrupt enabled

Bit 0 TAIFG is a flag used interrupt state of timer

- When No interrupt pending it become 0.
- When Interrupt pending it become 1.

TimerA Counter Register (TAR)

TAR is a 16 bit register its value increment or decrement each rising edge of clock based on the selected mode of the timer. It counts up to 65536 or TACCR0. It can be read any time to see the current value of Timer A .

TimerA Capture/Compare Control Register (TACCTLx)

TACCTLx is a 16 bit register used to control the capture and compare register and process.
Bits 15-14 CMx Bit is used to select the capture mode.

- if CMx bits are 00 the capture mode will be No capture
- if CMx bits are 01 the capture mode will be Capture on rising edge
- if CMx bits are 10 the capture mode will be Capture on falling edge
- if CMx bits are 11 the capture mode will be Capture on both rising and falling edges

Bits 13-12 CCISx Bit is used to select the Capture/compare input. These bits select the TACCRx input signal. See the device-specific datasheet for specific signal connections.

- 00 CCIxA
- 01 CCIxB
- 10 GND
- 11 VCC

Bit 11 SCS Bit is used to Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock.

- 0 Asynchronous capture
- 1 Synchronous capture

Bit 10 SCCI Bit is used to Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit Bit 9 is unused bit. Read only. Always read as 0. Bit 8 CAP Bit is used to set Capture mode or compare mode.

- 0 Compare mode
- 1 Capture mode

Bit 7-5 OUTMODx Bits is used to set the Output mode. Modes 2, 3, 6, and 7 are not useful for TACCR0 because EQUx= EQU0.

- 000 OUT bit value
- 001 Set
- 010 Toggle/reset
- 011 Set/reset
- 100 Toggle
- 101 Reset
- 110 Toggle/set
- 111 Reset/set

Bit4 CCIE Bit is used to enable interrupt for Capture/compare. This bit enables the interrupt request of the corresponding CCIFG flag.

- 0 Interrupt disabled
- 1 Interrupt enabled

Bit3 CCI Bit is used to set Capture/compare input. The selected input signal can be read by this bit.
Bit2 OUT Bit is used indicates the state of the output. For output mode 0, this bit directly controls the state of the output.

- 0 Output low
- 1 Output high

Bit1 COV Bit is used to detect Capture overflow. This bit indicates a capture overflow occurred. COV must be reset with software.

- 0 No capture overflow occurred
- 1 Capture overflow occurred

Bit0 CCIFG Bit is a flag used to indicate Capture/compare interrupt status.

- 0 No interrupt pending
- 1 Interrupt pending

TimerA Capture/Compare Register (TAxCCRx)

TAxCCRx is a 16 which bit register holed the number from 0 to 65535. This value is used to capture purpose or comparing purpose based on Bit 8 (CAP) of TimerA Capture/Compare Control Register (TACCTLx) setup. CAP is one bit value which used to set Capture mode if it is 1 or Compare mode if it is 0.
Capture mode is used to record time events. It can be used for speed computations or time measurements.
The compare mode is used to generate PWM output signals or interrupts at specific time intervals. When TAxR counts to the value in a TAxCCRx, where n the specific capture/compare register. When the value of TAxR and TAxCCRx equal the interrupt flag CCIFG is set and the internal signal EQUx become 1. EQUn affects the output according to the mode of timer set up (UP or UP/Down or Continuous).

TimerA Interupt Vector (TAIV)

It Is 16 bit register which has two interrupts:

- TAxCCR0 interrupt vector for TAxCCR0 CCIFG
- TAxIV interrupt vector for all other CCIFG flags and TAIFG

3.3 TimerB

TimerB is a 16-bit timer/counter with up to seven capture/compare registers. It is identical with TimerA except the following exception:

- The length of TimerB is programmable to be 8, 10, 12, or 16 bits.
- All TimerB outputs can be put into a high-impedance state.

- The SCCI bit function is not implemented in TimerB.

TimerB is configurable to operate as an 8-, 10-, 12-, or 16-bit timer with the CNTLx bits. The maximum count value, TBR(max), for the selectable lengths is 0FFh, 03FFh, 0FFFh, and 0FFFFh, respectively. Data written to the TBR register in 8-, 10-, and 12-bit mode is right-justified with leading zeros.
Except the above differences TimerA and TimerB have the same property that means they have the same register and the way to set the register also the same except to identify the name of the block A is substituted by B in the register name.

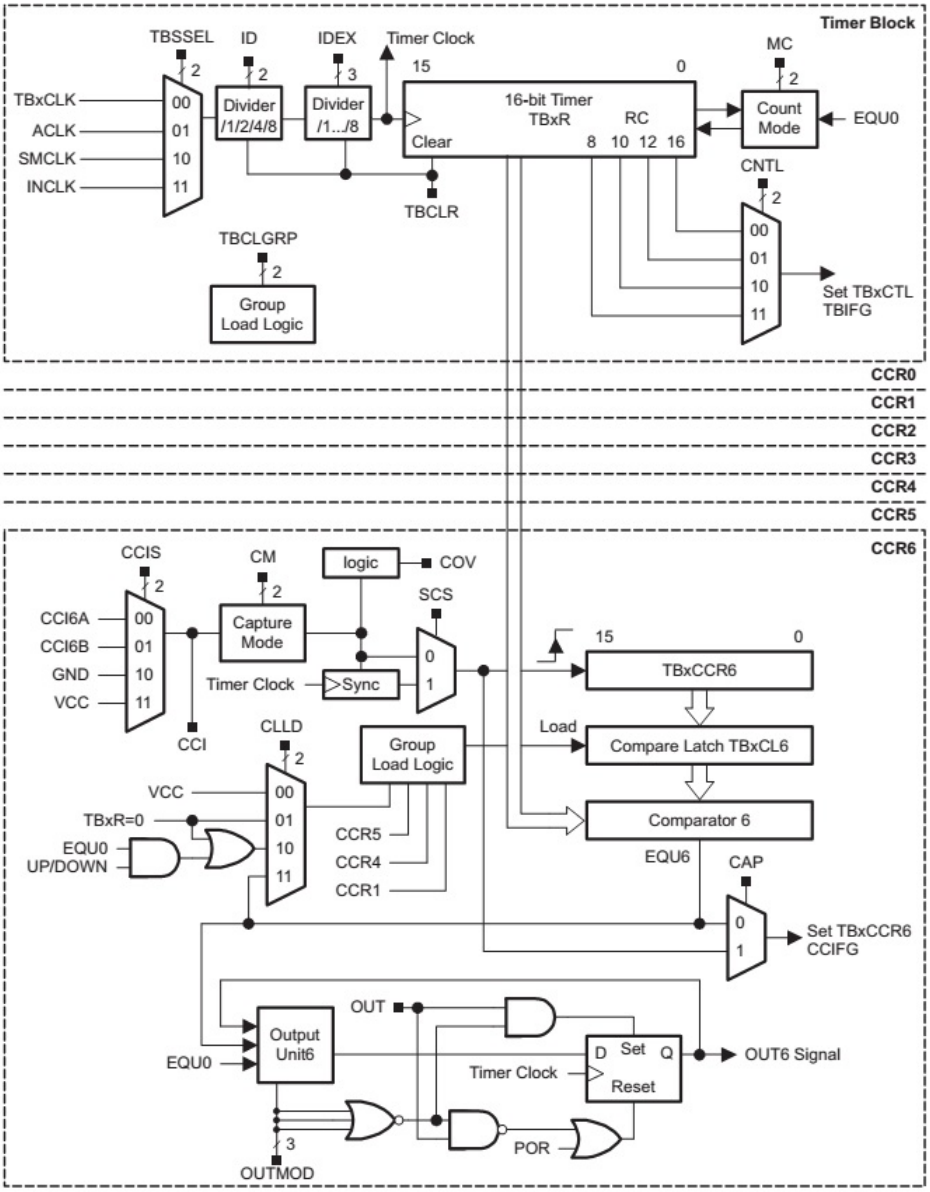

Figure 3.3: timerB internal block diagram taken from the data sheet

3.4 Timer Module Driver

The figure below shows the class diagram of timer modules which have operations and attributes which are used to set and configure the timer module of the microcontroller. This module used for MSP430x43x,MSP430x24x and MSP430x54x families of Msp430 microcontrollers. Since there are three timer module TimerA0, TimerA1 and TimerB0 we have three class:

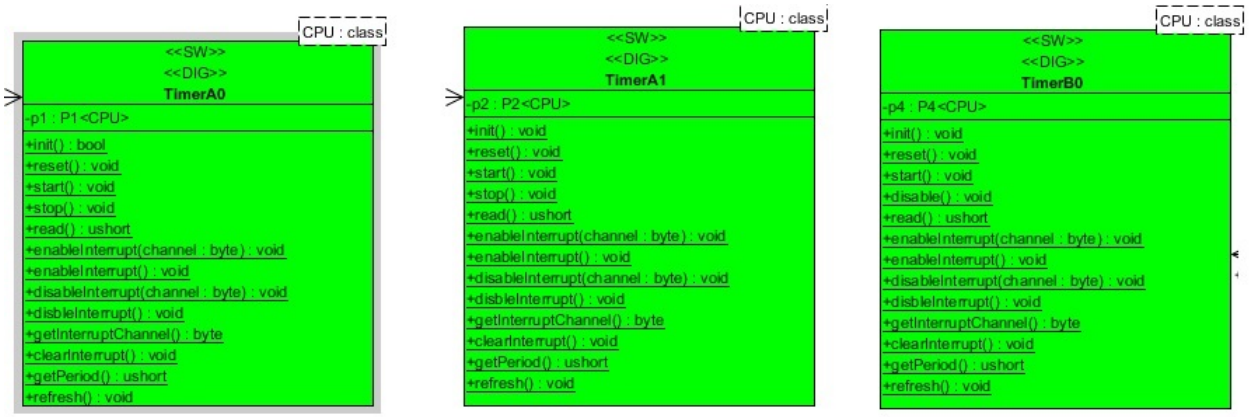

Figure 3.4: Timer Module Class diagrams

Attributes of the Timer Module Driver
private p1 : P1 - 8 bit port 1 pins.
Functions(operations)of the TimerA0 Module Driver

- **public init () : bool** - Initializes Timer to have clock from SMCLK (having frequency CPU::SMCLK_FREQ Hz), to go in stop mode, no capture, and to generate interrupt (when enabled) at rate 1e6/CPU::TIMERA0_PERIOD Hz; interrupt disabled.After init(timerFreq: ushort), the user shall call start() in order to activate the timer. For MSP_430_Family, $1e6/CPU :: TIMERA0\_PERIOD < (CPU :: SMCLK\_FREQ/2^16/8)$. It usually returns true, except when timer period is not achievable with the given clock frequency.
- **public reset () : void** - Resets Timer counting and all pending interrupt flags.It does not affect whether the timer is running or not, it only resets its count value. If timer is running (see start()), the first interrupt is generated at the end of counting.
- **public start () : void** - Starts (or restarts) Timer counting by setting it in up mode. Count value is not reset.
- **public stop () : void** - Stops Timer counting by setting it in stop mode. Count value is not affected.
- **public read () : ushort** - Reads Timer counting. Counting starts from 0, therefore an immediate call to read(): ushort right after reset() will return either 0 or a very small number.
- **public enableInterrupt (channel : byte) : void** -Enables the following interrupts of Timer:
  - capture/compare channel 0, for channel=0; this will trigger interrupt vector TIMER0_A0_VECTOR;
  - capture/compare channel 1/2/3/4/5/6, for channel=1/2/3/4/5/6, respectively; this will trigger interrupt vector TIMER0_A1_VECTOR;

- timer overflow, for channel=0x7; this will trigger interrupt vector TIMER0_A1_VECTOR;

If an interrupt is pending, this is immediately triggered. Routine getInterruptChannel(): byte can be used to discriminate among capture/compare channels 1 through 7 and timer overflow.

- **public enableInterrupt () : void** - Enables the interrupts of capture/compare channel 0, which will trigger interrupt vector TIMER0_A0_VECTOR If an interrupt is pending, this is immediately triggered. Routine getInterruptChannel(): byte can be used to discriminate among capture/compare channels 1 through 7 and timer overflow.

- **public disableInterrupt (channel : byte) : void** -Disables interrupts of Timer. See enableInterrupt(channel: byte): void for the meaning of parameter channel.If the interrupt is disabled within an interrupt service routine, the interrupt flag also has to be cleared (see clearInterrupt(): void).

- **public disbleInterrupt () : void** - Disables interrupts of Timer, capture/compare channel 0. If the interrupt is disabled within an interrupt service routine, the interrupt flag also has to be cleared (see clearInterrupt(): void).

- **public getInterruptChannel () : byte** - Returns the channel of the currently pending interrupt associated with vector TIMER0_A1_VECTOR:

  - 0 for no interrupt pending;
  - 1/2/3/4/5/6 for capture/compare channel 1/2/3/4/5/6, respectively. Note that capture/compare channel 0 is associated with a different interrupt vector (TIMER0_A0_VECTOR);
  - 7 for Timer overflow;

  This routine returns a correct value independently of the interrupt being enabled or not. If the interrupt is enabled, the corresponding interrupt service routine is also called. The latter shall explicitly clear interrupt flag (clearInterrupt(): void) before exiting, otherwise the same routine will be improperly called again upon exit.

- **public clearInterrupt () : void** - Clears interrupt flag. Must be called at the end of the interrupt service routine, otherwise interrupt service routine is called endlessly. If other interrupts are pending, another interrupt service routine is immediately triggered.

- **public getPeriod () : ushort** - Returns period of interrupt triggering, in us.

- **public refresh () : void** - Refreshes TMR variables against radiation-induced effects or other soft errors.

The functions for TimerA1 and TimerB0 classes are the same as TimerA0 with there own register setup

3.5 Test program for Timer Driver

This diagram describe how the test program for Timer driver is composed. as mentioned above the timer driver work for different MSP430 families but to implement the the test I have used only MSP430F5438A microcontroller.

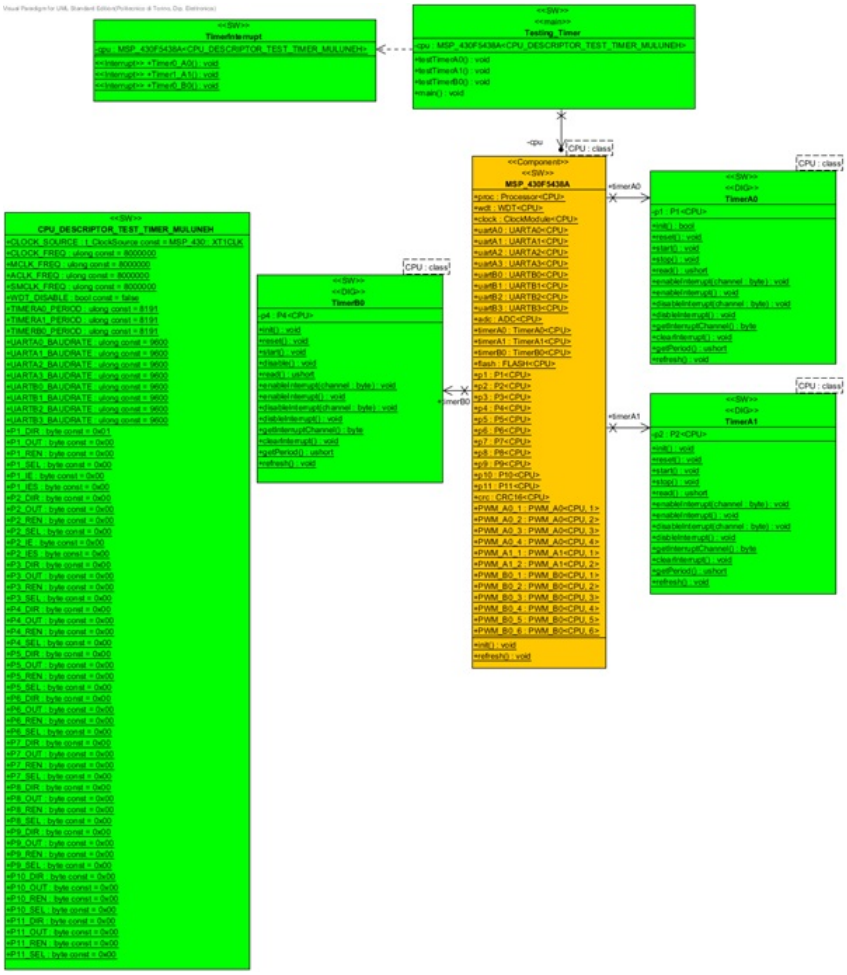

Figure 3.5: Test program for Timer Module driver

3.5.1 Class Testing Timer

Testing_Timer is a class used to test the timer system driver functionally. This class has an attributes: **private cpu : MSP_430F5438A** define which microcontroller series is under test. and it uses access the timer driver.
This class has the operations for setting and defining each timer modules and one main function.
public testTimerA0 () : void - it call three function sequentially from TimerA0 driver for initializing the TimerA0, enabling the interrupt and starting the counter.
public testTimerA1 () : void - it call three function sequentially from TimerA1 driver for initializing the TimerA1, enabling the interrupt and starting the counter.
public testTimerB0 () : void - it call three function sequentially from TimerB0 driver for initializing the TimerB0, enabling the interrupt and starting the counter.
public main () : void - The source code for the main function is found in appendix B. The code let the user to select the three timers by entering letters from 'a' to 'c'. The algorithm I followed to write the test code:

1. First we need to disable the watch dog of the microcontroller to protect software rest every time. The code used to disable the watch dog is:

   cpu.wdt.disable();

2. Set the pin 1.0 to output direction to measure the timer frequency with Oscilloscope.

```
cpu.p1.init();
P1DIR | = BIT0;
```

3. Initialize the timer by setting its TimerAx control register (TAxCTL) for
   - Selecting the clock input by setting the TASSELx bit. We have two possible choices ACLK or SMCLK
   - Vary the frequency of the clock by setting the divider value IDx according to the purpose
   - Rest all the TAR registers to start the counting from zero by setting TACLR bit

```
cpu.timerA0.init(); // for TimerA0
```

4. Enable the interrupt to setting the fourth bit of TimerA Capture/Compare Control Register (TACCTLx). This is used to turn on or off the LED when it the maximum value.

```
cpu.timerA0.enableInterrupt(0); //for TimerA0
```

5. Starts (or restarts) Timer counting by setting mode MCx value greater than zero.

```
cpu.timerA0.start(); // set up mode for TimerA0
```

3.5.2 Class TimerInterrupt

This class is used to handle the interrupt service routine for the timers. It has an attribute:
private cpu : MSP_430F5438A - define which microcontroller series is under test and it uses to access the timer driver.
It has three functions for each timers:
public Timer0_A0 () : void - TimerA0 interrupt service routine which is used to set and reset pin P1.0.
public Timer1_A1 () : void - TimerA1 interrupt service routine which is used to set and reset pin P1.0.
public Timer0_B0 () : void - TimerB0 interrupt service routine which is used to set and reset pin P1.0.

3.6 Test Procedure of Timer Module

Testing of the Timer performed by measures the output of the timer with the oscilloscope after we have configured the timer with the correct clock input and interrupt. Follow the following step to configure the microcontroller for testing the timer module:-
The source code for testing the Timer modules is found in appendix B. Open the IAR workbench and load the code to the microcontroller using the JTAG cable. The IAR has Input output panel and it let the user to select each the timer to be tested. since the MSP430F5438A has three timers TimerA0, TimerA1 and TimerB0. To select select TimerA insert 'a', TimerA1 insert'b' and TimerB0 insert 'c'. But before runnning the code first we have to connect the oscilloscope with pin 17 of the microcontroller or P1.0. The code used to set and rest the P1.0 when the count value reach the maximum.

3.7 Testing Result

The test result for TimerA0
the P1.0 set and rest with frequency 60.98 Hz.
The test result for TimerA1
the P1.0 set and rest with frequency 60.98 Hz.
The test result for TimerB0
the P1.0 set and rest with frequency 60.98 Hz.
The wave form result for all the timer that I got form the oscilloscope is the same:

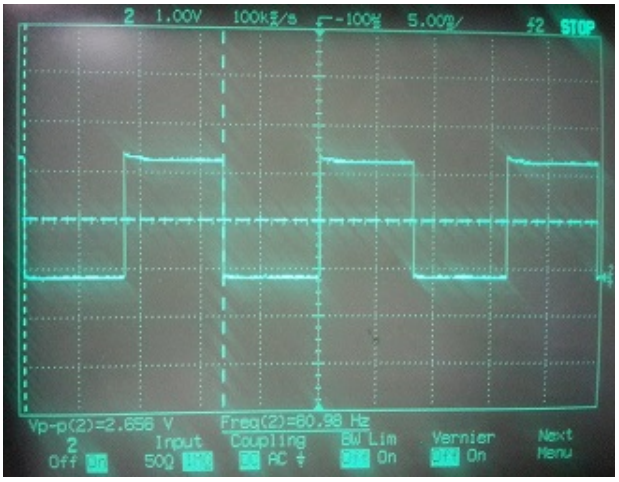

Figure 3.6: The wave form generated with Timer

3.7.1 Conclusion

To conclude the timer module driver functionally working or not first we have to know our input clock source frequency, the divider value that we set and the maximum value that we set for the counter(the period of the timer).
In this testing I have set SMCLK as a clock source to the timer which have 8MHZ frequency. the divider(IDx) for the input clock source for the timer is set to 8 and the mode of the counter is UP mode. the maximum value the timer count is set to 8191. with this setup the the counter register value increment up every 1us rising edge of the input clock and set to zero when the it reach the maximum counter value 8191 that means 8.191ms.the output pin P1.0 is 0 for the first 8.191ms and 1 for the next 8.191ms therefor its period is twice of 9.191ms and its frequency become:

$$P1.0frequency = \frac{1}{2 * 8.191ms} = 61.0425Hz \tag{3.1}$$

The value of the frequency that we read from the oscilloscope is almost the same with the expected frequency value so we can say that the timer driver is functionally working.

Chapter 4

TESTING PWM MODULE

4.1 Introduction

A Pulse width modulation (PWM) is a method of generating the analog signal by using a digital source. The behavior of PWM is defined by two components: a duty cycle and frequency. The duty cycle defines the percentage of time the signal goes high (on) or low (off) state with in a period. The frequency determines the total period of the signal that means the time of both on and off times.
MSP430 has no its own module for PWM. We use the Timer (Counter) module and a comparator module to generate PWM output.Configuring the PWM module is done by configuring the timer module of the microcontroller. The timer start counting when it triggered by the clock and the

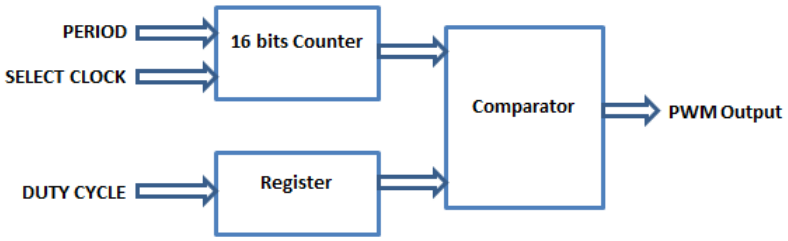

Figure 4.1: PWM IO block diagram

output of PWM set to high and it remains high till it rest after the counter value become equal to the set duty cycle value. But the initial set of the PWM output can be high or low based on our setting. to see the output of the PWM we can use oscilloscope.

4.2 PWM Module Driver

The figure below shows the class diagram of PWM modules which have operations which are used to set and configure the PWM module of the microcontroller. this module used for MSP430x43x,MSP430x24x and MSP430x54x families of Msp430 microcontrollers. since there are three PWM module we have three class:

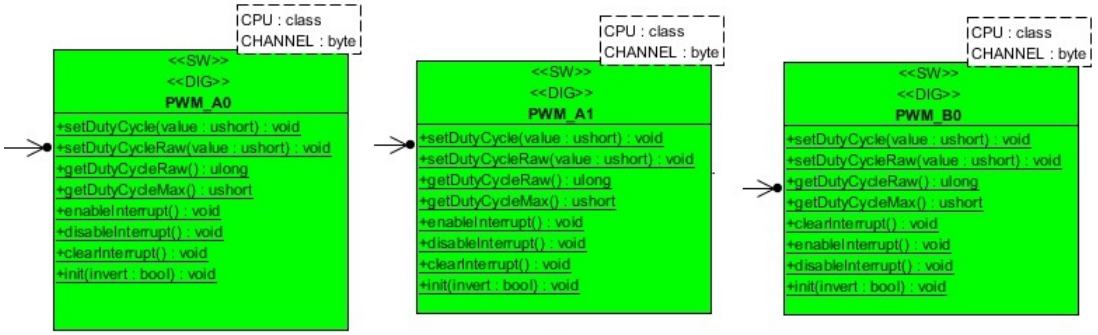

Figure 4.2: PWM Module Class diagrams

Functions or operations of the PWM Module Driver

- **public setDutyCycle (value : ushort) : void** - Sets duty cycle of output signal of channel number CHANNEL of PWM_A1 as close as possible to the value given by $value/2^16$. For instance, when:
  - value = 0, duty cycle = 0
  - value = 0x7FFF, duty cycle = 0.5
  - value = 0xFFFF, duty cycle = 1
- **public setDutyCycleRaw (value : ushort) : void** - Sets duty cycle of output signal of channel number CHANNEL of PWM_A0 in raw format, between 0 and the max count value of the associated Timer (which is available using getDutyCycleMax(): ushort function). For instance, when:
  - value = 0, duty cycle = 0;
  - value = 0.5 *getDutyCycleMax(): ushort, duty cycle = 0.5;
  - value = getDutyCycleMax(): ushort, duty cycle = 1.
- **public getDutyCycleRaw () : ulong** - Returns actual duty cycle (in raw format; see setDutyCycleRaw(value: ushort): void) of output signal of channel number CHANNEL of PWM_A0. For instance, when:
  - duty cycle == 0, returns 0;
  - duty cycle == 0.5, returns 0.5 *getDutyCycleMax(): ushort;
  - duty cycle == 1, returns getDutyCycleMax(): ushort,.
- **public getDutyCycleMax () : ushort** - Returns the max value for the setDutyCycleRaw(value: ushort): void function.
- **public enableInterrupt () : void** - Enables the interrupts of capture/compare channel 0, which will trigger interrupt vector TIMER0_A0_VECTOR If an interrupt is pending, this is immediately triggered.
- **public disableInterrupt () : void** -Disables interrupts of Timer, capture/compare channel 0. If the interrupt is disabled within an interrupt service routine, the interrupt flag also has to be cleared.

- **public clearInterrupt () : void** - Clears interrupt flag. Must be called at the end of the interrupt service routine, otherwise interrupt service routine is called endlessly. If other interrupts are pending, another interrupt service routine is immediately triggered.
- **public init (invert : bool) : void** - Initializes output pin for CHANNEL of PWM_A1. The associated Timer MUST be already initialized. When invert is false, the PWM output is not inverted, namely a $10 When invert is true, the PWM output is inverted, namely a 10$

4.3 Test program for PWM Driver

This diagram describe how the test program for PWM driver is composed. as mentioned above the PWM driver work for different MSP430 families but to implement the the test I have used only MSP430F5438A microcontroller.

Figure 4.3: Test program for PWM Module driver

4.3.1 Class Testing PWM

Testing_PWM is a class used to test the PWM system driver functionally. This class has an attributes: **private cpu : MSP_430F5438A** define which microcontroller series is under test. and it uses access the PWM driver.
This class has three function for handling interrupt service routine:
public Timer0_A0 () : void - TimerA0 interrupt service routine which is used to set and reset pin

P1.0.
public Timer0_A1 () : void - TimerA1 interrupt service routine which is used to set and reset pin P1.0.
public Timer0_B0 () : void - TimerB0 interrupt service routine which is used to set and reset pin P1.0.
public main () : void - The source code for the main function is found in appendix C. The code let the user to select the three PWM with by entering letters from 'a' to 'g'. The algorithm I followed to write the test code:

1. First we need to disable the watch dog of the microcontroller to protect software rest every time. The code used to disable the watch dog is:

   cpu.wdt.disable(); // Stop WDT

2. initialize,enable the interrupt and start the counter of each timer before implementing the PWM.

   cpu.timerA0.init();
   cpu.timerA0.enableInterrupt();
   cpu.timerA0.start();
   cpu.timerA1.init();
   cpu.timerA1.start();
   cpu.timerB0.init();
   cpu.timerB0.start();

3. Set the output pin of the timer modules to see the output of the PWM with oscilloscope and toggle output direction.
   Set the pin 1.5 to output direction to measure the timer frequency and select the TimerA0

   cpu.PWM_A0_x.init(false); //

   Set the pin 2.3 to output direction to measure the timer frequency and select the TimerA1

   cpu.PWM_A1_x.init(false);

   Set the pin 4.6 to output direction to measure the timer frequency and select the TimerB0

   cpu.PWM_B0_x.init(false);

4. Enable the interrupt to setting the fourth bit of TimerA Capture/Compare Control Register (TACCTLx). This is used to turn on or off the LED when it the maximum value.

   cpu.PWM_A0_1.enableInterrupt();

5. set the PWM duty cycle by setting TAxCCRx for timerA and TBxCCRx for timerB value.

   cpu.PWM_A0_1.setDutyCycle(DutyCycle);

6. check the duty cycle correctly set by checking register TAxCCR0.

   temp=((cpu.PWM_A0_1.getDutyCycleRaw()¡¡16)/cpu.PWM_A0_1.getDutyCycleMax())+1;

7. see the result through oscilloscope or input output panel of IAR workbench

4.4 Testing Procedure of PWM Module

Testing of the PWM performed by measures the output of the timer with the oscilloscope after we have configured the PWM timer with the correct clock input, mode and interrupt and after we have set the duty cycle. Follow the following step to configure the microcontroller for testing the PWM module:- The source code for testing the Timer modules is found in appendix C. open the IAR workbench and load the code to the microcontroller using the JTAG cable. The IAR has Input output panel and it let the user to select each PWM module to be tested. since the MSP430F5438A has three timers TimerA0, TimerA1 and TimerB0 each of them can have two or more PWM channels.so to select each PWM insert letter 'a' to 'g' and to change the wave form inverted insert 'A' to 'G'. but before runnning the code first we have to connect the oscilloscope with pins of the timers

- connect pin 1.5 to oscilloscope to measure the PWM frequency result of TimerA0.
- connect pin 2.3 to output direction to measure the PWM frequency result of TimerA1.
- connect pin 4.6 to output direction to measure the PWM frequency result of TimerB0.

Chapter 5

TESTING ADC MODULE

5.1 Introduction

An Analog to Digital Converter (ADC) is an electronic circuit that converts an analog signal to digital signal. The ADC system of MSP430 microcontrollers are based on 10 bits or 12 bits conversion. For this thesis I used a microcontroller with 12 bit conversion mode so it can represent the binary number from 0 to 212-1(4095). If a voltage being sampled has value Vin and the reference voltage for conversion is VREF, then the decimal number representing that input will be:

$$N = \frac{Vin}{VREF} * 4095 \tag{5.1}$$

For VREF+ (upper) and VREF− (lower) reference voltage the digital output N is equal to 4095 when the input signal is equal to or higher than VREF. The digital output N is equal to zero when the input signal is equal to or lower than VREF−.

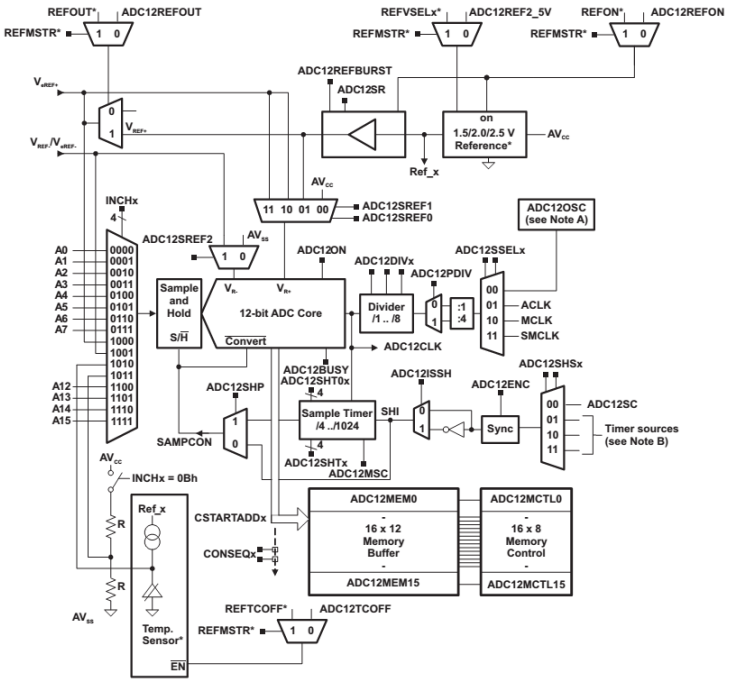

Figure 5.1: ADC circuit of the microcontroller taken form the datasheet

ADC12 has its own registers:

- ADC12 Control Register 0 (ADC12CTL0)
- ADC12 Control Register 1 (ADC12CTL1)
- ADC12 Control Register 2 (ADC12CTL2)
- ADC12 Conversion Memory Register (ADC12MEMx)
- ADC12 Conversion Memory Control Register (ADC12MCTLx)
- ADC12 Interrupt Enable Register (ADC12IE)
- ADC12 Interrupt Flag Register (ADC12IFG)
- ADC12 Interrupt Vector Register(ADC12IV)

5.2 ADC Module Driver

The figure below shows the class diagram of ADC modules which have operations and attributes which are used to set and configure the ADC module of the microcontroller. This module used for MSP430x43x, MSP430x24x and MSP430x54x families of Msp430 microcontrollers.

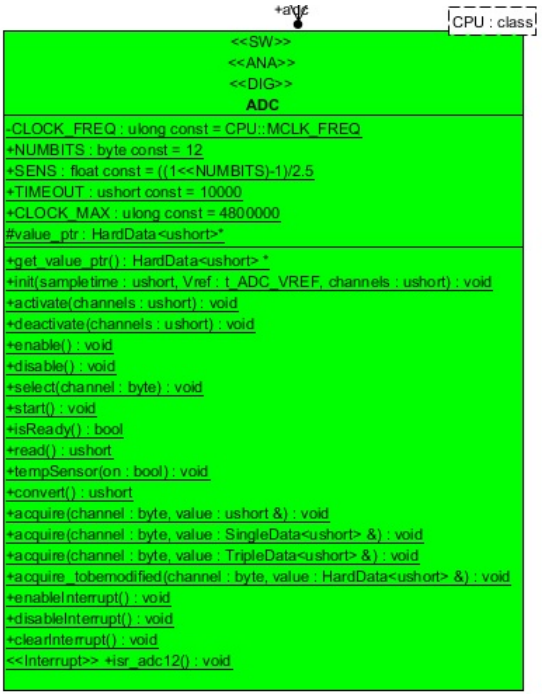

Figure 5.2: ADC Module Class diagrams

Attributes of the ADC Module Driver

- **private CLOCK_FREQ : ulong** - Frequency of MCLK (CPU main clock), in Hz.
- **public NUMBITS : byte** - Number of bits of ADC output. Value is right-aligned.

- **public SENS : float** - Sensitivity of ADC in units(LSB)/ V. it has $\frac{2^{NUMBITS}-1}{2.5}$ initial value
- **public TIMEOUT : ushort** - Timeout for all blocking operations. The value is in arbitrary units, which depend on several HW factors.it has 1000 initial value.
- **public CLOCK_MAX : ulong** - Max clock frequency of ADC, in Hz.it has 4800000 initial value
- **protected value_ptr : HardData** - First pointer to location where to store the value after acquisition (from interrupt service routine isr_adc12()). It should be identical to value_ptr2, for redundancy. If different, value is not stored anywhere.

Functions or operations of the ADC Module Driver

- **public init (sampletime : ushort, Vref : t_ADC_VREF, channels : ushort) : void** - Initializes ADC for all following conversions.Arguments are:
  - sampletime: indicates the desired sample time of the sample and hold circuit, in microseconds. Only a limited number of values are permitted, depending on the processor and its clock frequency.
  - Vref: the source of reference voltage channels: a word to specify which ADC channels are used: each bit is associated with a different channel; bit 0 with channel 0, up to bit 15 which is associated with channel 15. Each bit shall be set/reset to activate/deactivate the corresponding channel, respectively.
- **public activate (channels : ushort) : void** - Activates one or more ADC channels for all following conversions.The ADC must already be initialized (see init(sampletime: ushort, Vref: t_ADC_VREF, channels: ushort): void) before calling this activate(channels: ushort): void.Arguments are:
  channels: a word to specify which ADC channels are to be activated: each bit is associated with a different channel; bit 0 with channel 0, up to bit 15 which is associated with channel 15. Each bit shall be set to activate the corresponding channel. All other channels are not touched.
  To deactivate one or more channels, use deactivate(channels: ushort): void).
- **public deactivate (channels : ushort) : void** - Deactivates one or more ADC channels for all following conversions. The ADC must already be initialized (see init(sampletime: ushort, Vref: t_ADC_VREF, channels: ushort): void) before calling this activate(channels: ushort): void.Arguments are:
  channels: a word to specify which ADC channels are to be deactivated: each bit is associated with a different channel; bit 0 with channel 0, up to bit 15 which is associated with channel 15. Each bit shall be set to activate the corresponding channel. All other channels are not touched.
  To activate one or more channels, use activate(channels: ushort): void).
- **public enable () : void** - Turns on ADC and its reference. Conversion is NOT started. It shall be started with either start(): void or acquire(channel: byte, value: SingleData&): void or equivalent functions.
- **public disable () : void** - Turns off ADC and its reference and disables conversion.
- **public select (channel : byte) : void** - Selects the input channel to the ADC, for all following conversions. The input channel is identified by argument channel (from 0 to a Processor-dependent value; often either 0-7 or 0-15).
  NOTE: the use of select(channel: byte): void + start(): void + read(): ushort operations is NOT compatible with the use of acquire(channel: byte, value: SingleData&): void operation.

- **public start () : void** - Holds input voltage from the last chosen channel, starts conversion and exits immediately.The channel to convert must be previously selected by means of the select(channel: byte): void operation. The user shall then wait until the isReady(): bool operation returns true before calling read(): ushort, otherwise unpredictable results may occur.
  NOTE: the use of select(channel: byte): void + start(): void + read(): ushort operations is NOT compatible with the use of acquire(channel: byte, value: SingleData&): void operations.

- **public isReady () : bool** - Returns true when the ADC has terminated conversion; false otherwise. Reading converted value (via read(): ushort operation) does not reset the returned conversion status.

- **public read () : ushort** - Returns converted data from ADC, in the range 0 to $2^{NUMBITS}-1$. 0V converts to 0, while full scale converts to $2^{NUMBITS}-1$. Full scale depends on the Vref parameter in init(sampletime: ushort, Vref: t_ADC_VREF, channels: ushort): void Data conversion should have been started by means of the start(): void operation. The user shall then wait until the isReady(): bool operation returns true before using this operation, otherwise unpredictable results may occur.
  NOTE: the use of select(channel: byte): void + start(): void + read(): ushort operations is NOT compatible with the use of acquire(channel: byte, value: SingleData&): void operations.

- **public tempSensor (on : bool) : void** - it enable or disable the temperature sensor of the micro controller.

- **public convert () : ushort** - Holds input voltage from the last chosen channel, starts conversion, waits until end of conversion, then returns converted value. Waiting is interrupted after TIMEOUT: ushort internal units (actual delay is not predictable). For range of converted values, see read(): ushort operation.
  NOTE: the use of convert(): ushort operation is NOT compatible with the use of acquire(channel: byte, value: SingleData&): void operations.

- **public acquire (channel : byte, value : ushort) : void** - Starts hold and conversion of input channel defined by the channel parameter. It enables interrupt for ADC. At the end of conversion, ADC automatically calls interrupt service routine isr_adc12(): void to transfer converted result into the location defined by the parameter value.

- **public acquire (channel : byte, value : SingleData) : void** - Starts hold and conversion of input channel defined by the channel parameter. It enables interrupt for ADC. At the end of conversion, ADC automatically calls interrupt service routine isr_adc12(): void to transfer converted result into the location defined by the parameter value.

- public acquire (channel : byte, value : TripleData) : void - Starts hold and conversion of input channel defined by the channel parameter. It enables interrupt for ADC. At the end of conversion, ADC automatically calls interrupt service routine isr_adc12(): void to transfer converted result into the location defined by the parameter value.

- **public acquire_tobemodified (channel : byte, value : HardData) : void** - Starts hold and conversion of input channel defined by the channel parameter. It enables interrupt for ADC. At the end of conversion, ADC automatically calls interrupt service routine isr_adc12(): void to transfer converted result into the location defined by the parameter value.

- **public enableInterrupt () : void** - Enables interrupt at the end of ADC conversion.

- **public disableInterrupt () : void** - Disables interrupt at the end of ADC conversion and clears the corresponding interrupt flags.

- **public clearInterrupt () : void** - Clears interrupt flag. Must be called at the end of the interrupt service routine, otherwise interrupt service routine is called endlessly. If other interrupts are pending, another interrupt service routine is immediately triggered.
- **public isr_adc12 () : void** - Interrupt service routine which is called at the end of ADC conversion, if the corresponding interrupt is enabled (see enableInterrupt(): void). It stores the converted value into location pointed by value_ptr: HardData*. If the two pointers do not match, value is not stored anywhere and memory is not affected. At the end, this routine disables interrupt on ADC conversion. This routine is internally used by the acquire(channel: byte, value: SingleData&): void operations which enable ADC interrupt.

5.3 Test program for ADC Driver

This diagram describe how the test program for ADC driver is composed. as mentioned above the ADC driver work for different MSP430 families but to implement the the test I have used only MSP430F5438A microcontroller.

Figure 5.3: Test program for ADC Module driver

5.3.1 Class Testing ADC

Testing_is ADC class used to test ADC system driver functionally. This class has an attributes:
private cpu : MSP_430F5438A define which microcontroller series is under test. and it uses access the timer driver.
private Vref : t_ADC_VREF let the user to set the different reference voltage: such as VREF_EXT,

VREF_1_5, VREF_2_0, VREF_2_5 and VREF_VDD
This class also has one main function(operator).The source code for testing the main function is found in appendix E. The code let the user to select the reference voltages by entering letters from 'a' to 'd' and also the input channel letting the user to insert byte. The algorithm I followed to write the test code:

1. First we need to disable the watch dog of the microcontroller to protect software rest every time. The code used to disable the watch dog is:

   cpu.wdt.disable(); // Stop WDT

2. Select the reference voltage. All ADCs need to have a voltage reference which the input voltage can be compared to. There not only is an upper voltage level which the signal is reference to, but also lower level voltage. For the ADC12, you can use the following as references.

3. select the input channels

4. Initialize the ADC module by setting with selected reference voltage and Clock source for the conversion operation.Select the sample-and-hold time for the conversion.

   cpu.adc.init(100,Vref,0x200);

5. enable the ADC.

   cpu.adc.enable();

6. Initialize the ADC module by setting Port pins that will be used as analog input channels.

   cpu.adc.select(channel);

7. Start conversion and read the conversion result from ADC12MEM0 memory.

   value=cpu.adc.convert();

8. compare the result with the expected result.

5.4 Test Procedure of ADC Module

Testing of the ADC performed by setting the reference voltage, the input channel and read the conversion result from ADC12MEM0 memory. Follow the following step to configure the microcontroller for testing ADC driver system:-

- The source code for testing the ADC module is found in appendix D. open the IAR workbench and load the code to the microcontroller using the JTAG cable. The IAR has Input output panel and it let the user to select each reference voltages by using a letters and it also set the input channel by letting the user to insert a four bits number. To select the different reference voltage the user insert a letters from a to d.

- Letter a selects VREF_EXT.
- Letter b selects VREF_1_5.
- Letter c selects VREF_2_5.
- Letter d selects VREF_VDD.

For selecting the input channels the user insert four bit number.

Table 5.1: Input Channels of ADC

Input channels	pin	four bits number
A0	P6.0	0000
A1	P6.1	0001
A2	P6.2	0010
A3	P6.3	0011
A4	P6.4	0100
A5	P6.5	0101
A6	P6.6	0110
A7	P6.7	0111
VREF+	P5.0	1000
VREF+/VREF-	P5.1	1001
Temperature sensor		1010
AVcc/AVss		1011
A12	P7.4	1100
A13	P7.5	1101
A14	P7.6	1110
A15	P7.7	1111

So first press the correct letter from the keyboard then press enter then press the correct number from the keyboard then press then press enter. before this we have to give analog input voltage form signal generator if the input channels is different form 1000, 1001, 1010 and 1011.

- Check the result form the ADC12MEM0 memory. If the output result is based on your set up it means the ADC driver is working else not working.

Chapter 6

TESTING FLASH MEMORY MODULE

6.1 Introduction

Flash memory is an electronic non-volatile storage medium that can be electrically erased and re-programmed. MSP430 family of microcontrollers have integrated flash memory for nonvolatile data storage which is used to store the software code and data in microcontroller.
The Flash memory in the MSP430 is usually divided into two sections, the main ash and the Information ash:

- Main Flash - Represents the bulk of the MSP430s ash and used for program code and constant data. Main Flash itself divided into several ash banks, with each bank further divided into segments. When a microcontroller is set to have64kB of Flash, it is referring to the Main ash.
- Information Flash - Several extra segments of ash separate from the Main ash. These are primarily intended for information such as calibration constants, but are much smaller than the main banks.

The location and amount of each of these sections depends on the specic MSP430 youre using, and the details are contained in the Memory Map included in the datasheet of the particular MSP430 youre using. Here is an example of the memory organization of MSP430F5438A devices, taken from their datasheets.

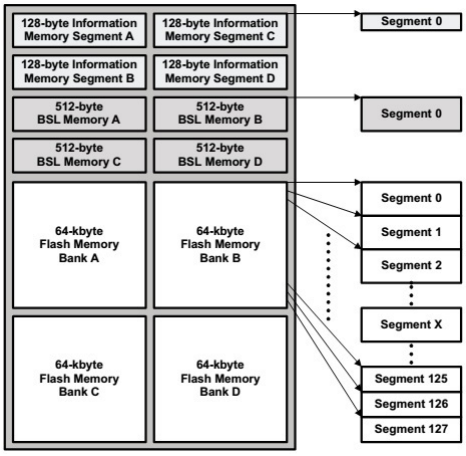

Figure 6.1: 256-KB Flash Memory Organization taken from the data sheet

6.2 Flash Module Driver

The figure below shows the class diagram of Flash modules driver class which have operations and attributes which are used to set and configure Flash module of the microcontroller. This module used for MSP430x43x,MSP430x24x and MSP430x54x families of Msp430 microcontrollers.

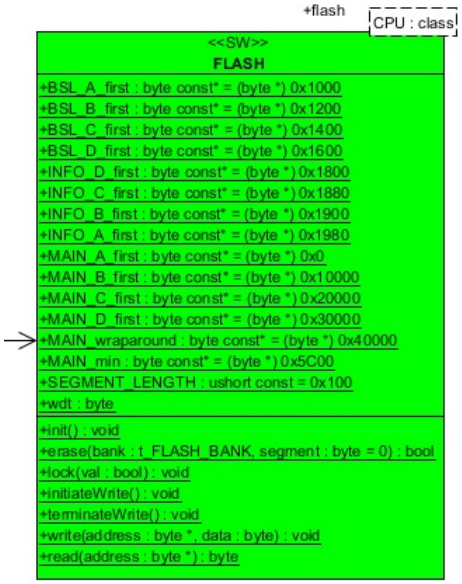

Figure 6.2: Flash Class diagrams

Attributes of the Flash Module Driver

- **public BSL_A_first : byte** - initialize the initial address of BSL_A_to 0x1000.
- **public BSL_B_first : byte** - initialize the initial address of BSL_B_to 0x1200.
- **public BSL_C_first : byte** - initialize the initial address of BSL_C_to 0x1400.
- **public BSL_D_first : byte** - initialize the initial address of BSL_D_to 0x1600.
- **public INFO_D_first : byte** - initialize the initial address of INFO_D_to 0x1800.
- **public INFO_C_first : byte** - initialize the initial address of INFO_C_to 0x1880.
- **public INFO_B_first : byte** - initialize the initial address of INFO_B_to 0x1900.
- **public INFO_A_first : byte** - initialize the initial address of INFO_A_to 0x1980.
- **public MAIN_A_first : byte** - initialize the initial address of MAIN_A_to 0x0.
- **public MAIN_B_first : byte** - initialize the initial address of MAIN_B_to 0x10000.
- **public MAIN_C_first : byte** - initialize the initial address of MAIN_C_to 0x20000.
- **public MAIN_D_first : byte** - initialize initial address of MAIN_D_to 0x30000.
- **public MAIN_wraparound : byte** -
- **public MAIN_min : byte** - minimum address of the main back

- **public SEGMENT_LENGTH : ushort**- initialize the segment length to 0x100
- **public wdt : byte**

Functions of the Flash Module Driver

- **public erase (bank : t_FLASH_BANK, segment : byte) : bool** -Erases an FLASH segment or bank. If segment is:
  - MASS, all FLASH is mass-erased
  - INFO, all information segments are erased
  - INFO_A through INFO_D, the corresponding segment of the information FLASH is erased
  - BSL, all BSL segments are erased
  - BSL_A through BSL_D, the corresponding segment of the BSL FLASH is erased
  - MAIN, all main banks are erased
  - MAIN_A through MAIN_D, all segments of the the corresponding bank of the main FLASH are erased
  - MAIN_A_seg through MAIN_D_seg, only the segment indicated by argument segment is erased in the main FLASH
- **public lock (val : bool) : void** - used to lock and unlock the memory. if Val is true it lock the memory and if Val is false it unlock the memory.
- **public initiateWrite () : void** - unlock the memory to start write.
- **public terminateWrite () : void** - lock the memory to stop write.
- **public write (address : byte, data : byte) : void** - write the value of data:byte to the address location address:byte.
- **public read (address : byte) : byte** - return the data stored at address location address:byte.

6.3 Test program for Flash Driver

This diagram describe how the test program for Flash driver is composed. As mentioned above the Flash driver work for different MSP430 families but to implement the the test I have used only MSP430F5438A microcontroller.

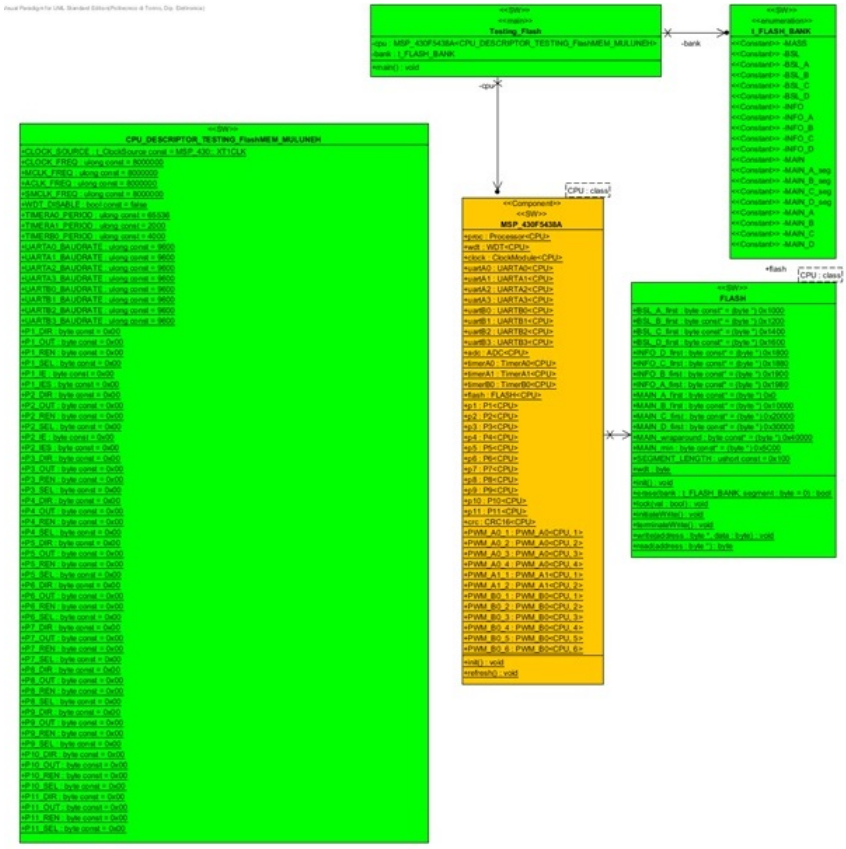

Figure 6.3: Test program for Flash Module driver

6.3.1 Class Testing Flash

Testing_is Flash class used to test Flash memory driver functionally. This class has an attributes:
private cpu : MSP_430F5438A define which microcontroller series is under test. and it uses access the Flash driver.
private bank : t_FLASH_BANK let the user to select different bank of the flash memory: such as MASS, BSL, BSL_A, BSL_B, BSL_C, BSL_D, INFO, INFO_A, INFO_B, INFO_C, INFO_D, MAIN, MAIN_A, MAIN_B, MAIN_C, MAIN_D, MAIN_A_seg, MAIN_B_seg, MAIN_C_seg and MAIN_D_seg
This class also has one main function(operator).The source code for testing the main function is found in appendix F. The code let the user to select the Banks of Flash memory by entering letters from 'a' to 'j'. The algorithm I followed to write the test code:

1. First we need to disable the watch dog of the microcontroller to protect software rest every time. The code used to disable the watch dog is:

   cpu.wdt.disable(); // Stop WDT

2. Select one bank and address inside the bank by using the letters from 'a' to 'j'.

3. Erase the selected bank and read the data at address. print the result to screen.

```
cpu.flash.erase(bank,segment);
Rdata = cpu.flash.read(address);
printf(%d\n,(int)Rdata);
```

4. Unlock the memory then write the data to the address and lock the memory after write and read the data at the selected address. print the reading result to screen.

```
cpu.flash.initiateWrite();
cpu.flash.write( address, Wdata);
cpu.flash.terminateWrite();
Rdata = cpu.flash.read(address);
printf(%d\n,(int)Rdata);
```

5. Increment the address and repeat the above.
6. Erase the selected bank and try to write without unlocking then read the data at the address and print to the screen the reading result.
7. Repeat the above step to all bank and segments of the flash memory.

6.4 Testing Procedure of Flash Module

Testing of the Flash memory performed by erasing, unlocking, writing and reading of an address in a bank or a segment. The user first select the bank or segment. Follow the following step to configure the microcontroller for testing Flash driver system:-

- The source code for testing the Flash module is found in appendix E. open the IAR workbench and load the code to the microcontroller using the JTAG cable. The IAR has Input output panel and it let the user to select each bank or segments of flash memory by using a letters. To select the different banks or segments of flash memory the user insert a letters from a to g.
  - Letter a selects INFO_A.
  - Letter b selects INFO_B.
  - Letter c selects INFO_C.
  - Letter d selects INFO_D.
  - Letter e selects MAIN_B.
  - Letter f selects MAIN_C.
  - Letter g selects MAIN_D.
  - Letter h selects MAIN_B_seg.
  - Letter i selects MAIN_C_seg.
  - Letter j selects MAIN_D_seg.
- Check the result displayed in the screen weather it is the same as expected or not.

Chapter 7

TESTING CRC MODULE

7.1 Introduction

Data corruption might occur when a data is transmitted or stored in the microcontroller. To overcome this problem we need a mechanism to detect the error and correct it. The CRC (cyclic redundancy check) calculation was the result of this. Cyclic Redundancy Code (CRC) is commonly used to determine the correctness of a data transmission or storage. CRC is based on a polynomial division. The CRC is calculated by dividing the data by a generator polynomial and recording the reminder after division.
The most common three polynomial generators used are:

$$CRC - 16 = x^{16} + x^{15} + x^2 + 1 \tag{7.1}$$

$$CRC - CCITT = x^{16} + x^{12} + x^5 + 1 \tag{7.2}$$

$$CRC - 32 = x^{32} + x^{26} + x^{23} + x^{22} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1 \tag{7.3}$$

CRC can be implemented both in hardware or software. Some MSP430 devices have a built-in hardware CRC calculator. The CRC signature is based on the polynomial given in the CRC-CCITT-BR polynomial

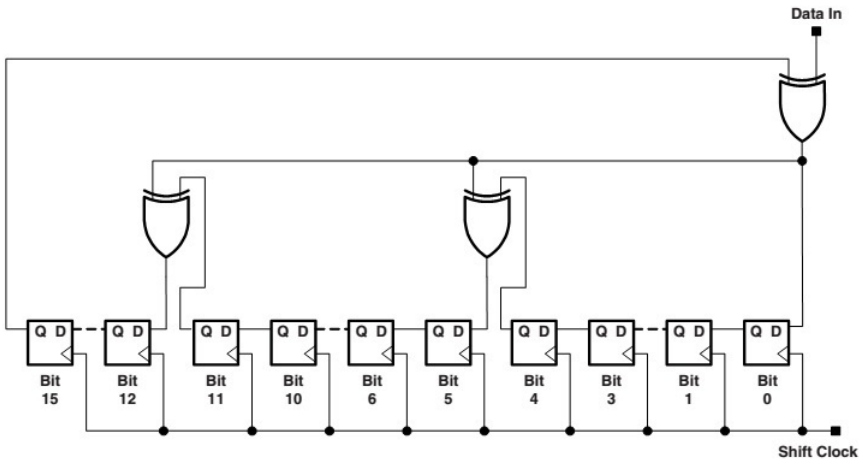

Figure 7.1: LFSR Implementation of CRC-CCITT Standard, Bit 0 is the MSB of the Result[1]

7.1.1 CRC Registers

The CRC modules has four its own 16 bit registers:

- CRC Data In Register(CRCDI)it hold the input data in to the CRC module
- CRC Data In Reverse Register(CRCDIRB)it holds the input data in to the CRC module in revers order.
- CRC Initialization and Result Register(CRCINIRES): it holds the current CRC result in normal order
- CRC Reverse Result Register(CRCRESR):it holds the current CRC module result bit in reverse order.

If we write the input data to CRCDI register the in two clock cycle of MCLK the check sum result is calculated and stored in CRCINIRES register. if we wish also to read the inverted check sum result of CRCRESR.

7.2 CRC Module Driver

The figure below shows the class diagram of CRC modules which have operations or function and attributes which are used to set and configure the CRC module of the microcontroller. this module used for MSP430x43x,MSP430x24x and MSP430x54x families of Msp430 microcontrollers.

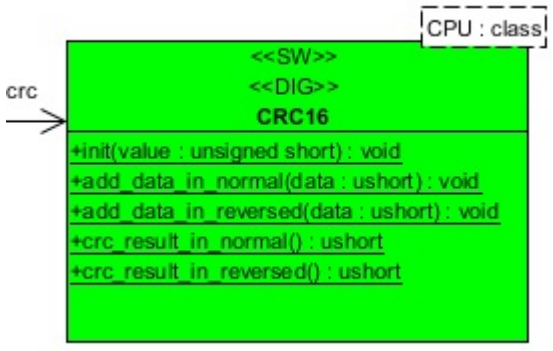

Figure 7.2: CRC Module Class diagrams

Functions or operations of the CRC Module Driver

- **public init (value : unsigned short) : void** - Initialize the CRC by setting CRC Initialization and Result Register(CRCINIRES) value. value:unsigned short - is any 16 bit number that you want to use as initial value for calculation start most of the time it is 0XFFFF.
- **public add_data_in_normal (data : ushort) : void** - write input data to CRC Data In Register(CRCDI).
- **public add_data_in_reversed (data : ushort) : void** - write input data to CRC Data In Reverse Register(CRCDIRB).
- **public crc_result_in_normal () : ushort** - return the CRC check result in normal order.
- **public crc_result_in_reversed () : ushort** - return the CRC check result in revers order.

7.3 Test program for CRC Driver

This diagram describe how the test program for CRC driver is composed. as mentioned above the CRC driver work for different MSP430 families but to implement the the test I have used only MSP430F5438A microcontroller.

Figure 7.3: Test program for CRC Module driver

7.3.1 Class Testing CRC

Testing_CRC class is used to test CRC system driver functionally.
To check the CRC is functionally working or not, we need to check the result that we get from the module with the expected result. Calculating the expected result with hand may take a time so I prepared a software code which calculate the expected result based on your input value.Then we cross check the software result with the hardware result(from the CRC module)if they are equal the result the module work very well else its not.
This class has an attributes:
private cpu : MSP_430F5438A define which microcontroller series is under test and it uses to access the CRC driver.
This class has two function:
public get_crc16_SW (in : ushort, int_value : ushort) : ushort - this function used to calculate the expected CRC check sum using the CRC16 algorithm without using the hardware the microcontroller. The function that used to calculate the expected CRC check sum is shown below. It get the

input data and calculate the check sum based on the CRC16 algorithm that microcontroller CRC use.

```
ushort get_crc16_SW(ushort in, ushort int_value) {
        int CRC[16];
        int data[16];
        int DoInvert;
        ushort result=0x0000;       // CRC Result

        for (int i=0; i<16; i++)    // Init CRC=int_value
        {
        if(int_value & 0x8000)
        {
        CRC[i] = 1;
        }
        else
        {
        CRC[i] = 0;
        }
        int_value <<= 1;
        }

        for (int i=15; i>=0; i--)
        {
        if(in & 0x8000)
        {
        data[i] = 1;
        }
        else
        {
        data[i] = 0;
        }
        in <<= 1;
        }

        for (int i=0; i<16; ++i)
        {
        DoInvert = data[i] ^ CRC[15];          // XOR required?

        CRC[15] = CRC[14];
        CRC[14] = CRC[13];
        CRC[13] = CRC[12];
        CRC[12] = CRC[11] ^ DoInvert;
        CRC[11] = CRC[10];
        CRC[10] = CRC[9];
        CRC[9] = CRC[8];
        CRC[8] = CRC[7];
        CRC[7] = CRC[6];
        CRC[6] = CRC[5];
        CRC[5] = CRC[4] ^ DoInvert;
        CRC[4] = CRC[3];
        CRC[3] = CRC[2];
        CRC[2] = CRC[1];
        CRC[1] = CRC[0];
        CRC[0] = DoInvert;
        }
        for (int i=15; i>=0; i--)
        {
        if(CRC[i] == 1)
        {
        result <<= 1;
        result = result |(0x0001);
        }
```

```
        else
        {
        result <<= 1;
        }
        }

        return(result);
}
```

public main () : void - The source code for the main function is found in appendix E. The code let the user to select the normal or revers order CRC calculation by entering letters from 'a' or 'b'. The algorithm I followed to write the test code:

1. First we need to disable the watch dog of the microcontroller to protect software rest every time. The code used to disable the watch dog is:

   cpu.wdt.disable();

2. Initialize the CRC by setting CRC Initialization and Result Register(CRCINIRES) value. If you don't initialize the this register it always start calculation using this value and you may get wrong result from that you expect.

   cpu.crc.init(0xFFFF);

3. Calculate the expected the CRC check sum result using the software.

   sw_result=get_crc16_SW(rdata,hw_result);

4. write the input data to CRC Data In Register(CRCDI) or CRC Data In Reverse Register(CRCDIRB). If we write the input data in CRCDI we get the output in normal order but if we put the result in CRCDIRB we get the output in reverse order.

   cpu.crc.add_data_in_normal(data); //16 bits input data
   or
   cpu.crc.add_data_in_reversed(data); //16 bits input data in reverse order

5. Read the data based on your wishes. If you need to read with normal order read from CRCINIRES register or if you need the result to be in reverse order read from CRCRESR register.

   hw_result=cpu.crc.crc_result_in_normal(); //checksum result
   or
   hw_result=cpu.crc.crc_result_in_reversed(); //checksum result in reverse order

7.4 Testing Procedure of CRC Module

Testing of the CRC is very easy. It only need to write a data in the input register and read the data from the output then compare the result with the software generated result.
The source code for testing the CRC module is found in appendix E. open the IAR workbench and load the code to the microcontroller using the JTAG cable. The IAR has Input output panel and it let the user to select the normal operation from reserved operation by using a letters a and b.

- Letter a selects normal data input.
- Letter b selects reversed data input.

By using the IRA workbench we can see the registers of CRC module. so before you start loading the code select View Register from the view menu and you will get a panel and select the CRC register. from the view menu select also IO display.
Follow the following step to configure the microcontroller for testing CRC driver system:-

- enter a.
- get the normal check sum result.
- enter b.
- get the the reverse check sum result.
- Check the result displayed in the screen and compare the software result with hardware result.

7.5 Testing Result

After I run my test code for MSP430F5438A then result that I found is shown in the figure below.

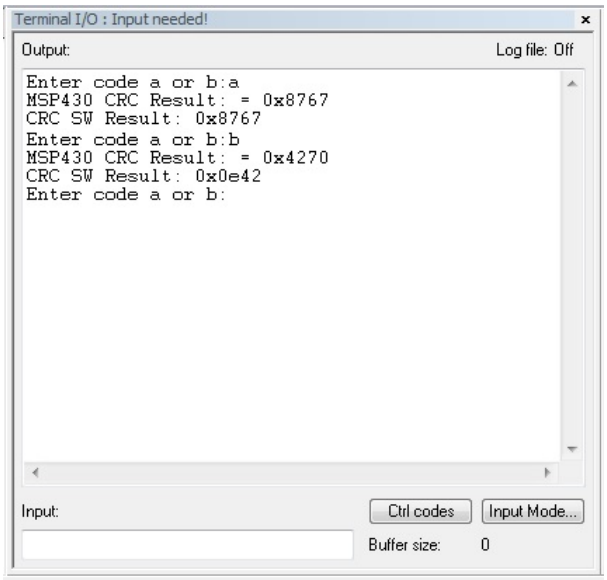

Figure 7.4: Test result of CRC

7.5.1 Conclusion

As we can see from the figure above when the user press 'a' the result of both software and hardware result is the same this mean CRC is working with normal mode. When the user press b the result of hardware is in revers order than the software result this mean the CRC also work in revers mode.

Chapter 8

TESTING USCI MODULE

8.1 Introduction

The universal serial communication interface (USCI) supports multiple serial communication modes with a single hardware module. USCI supports three kind serial communication protocols which are UART, SPI and I2C. USCI is not found in all series of MSP430. MSP430 series which have USCI are: MSP430F5xx, MSP430F4xx, MSP430FG41xx and MSP430F2xx.
MSP430 has two different individual USCI blocks which are named with a different letter: USCIA and USCIB. If more than one identical USCI module is implemented on one device, those modules are named with incrementing numbers Different USCI modules support different modes.
The USCIAx supports: **UART** mode, **IrDA** mode and **SPI** mode.
The USCIAx supports: **SPI** mode and **I2C** mode.

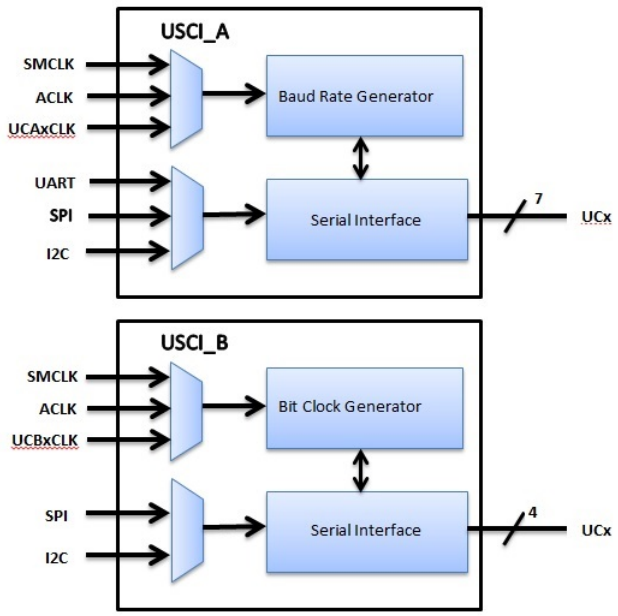

Figure 8.1: USCI block diagram

8.1.1 Testing SPI

Introduction

The SPI (Serial Peripheral Interface) bus is a synchronous serial communication interface which operates in full duplex mode.
SPI has the following four signal lines:

- Serial Clock (SCKL) is a clock provided to synchronize the communications between the master and slave.
- Chip Enable or Select (CS) or (STE) is necessary to select the slave with which we want to communicate.
- Serial Data Input (SDI) or (MOSI/SIMO)
- Serial Data Output (SDO) or (MISO/SOMI)

SPI has two mode of operation: Master and Slave mode. The communication always initiated by the Master SPI. The SPI master drives SCKL and CS. The SPI slave devices get their clock and chip select input from the Master SPI. Whenever an SPI slave device is not chip selected, its SDO output line is tri-stated (high impedance state).SPI interface hardware contains shift registers. One shift register is used to send out data and another shift register is used to receive data. The clocks are all synchronous and they use SCKL. The number of serial data bits can vary depending on the

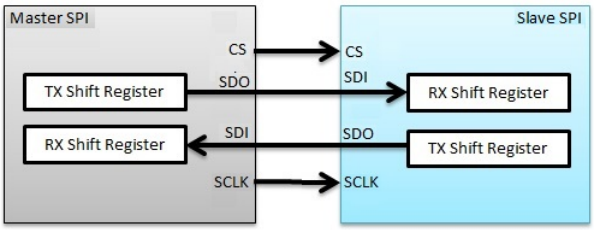

Figure 8.2: USCI block diagram

device. In SPI mode the USCI connect the MSP430 to an external system via three or four pins: UCxSIMO,UCxSOMI,UCxCLK and UCxSTE. SPI mode is selected when the UCSYNC bit is set and SPI mode (3-pin or 4-pin) is selected with the UCMODEx bits.

8.1.2 Testing UART

Introduction

The Universal Asynchronous Receiver/Transmitter (UART) module is asynchronous serial communication interface which have two lines for transmit (TX) and receive (RX) signals. Asynchronous transmission allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters (Baud Rate) prior transmission and special bits are added to each word to synchronize the sending and receiving units. In asynchronous transmission, the sender sends.

- a Start bit
- 5 to 8 data bits (LSB first),
- an optional Parity bit, and then

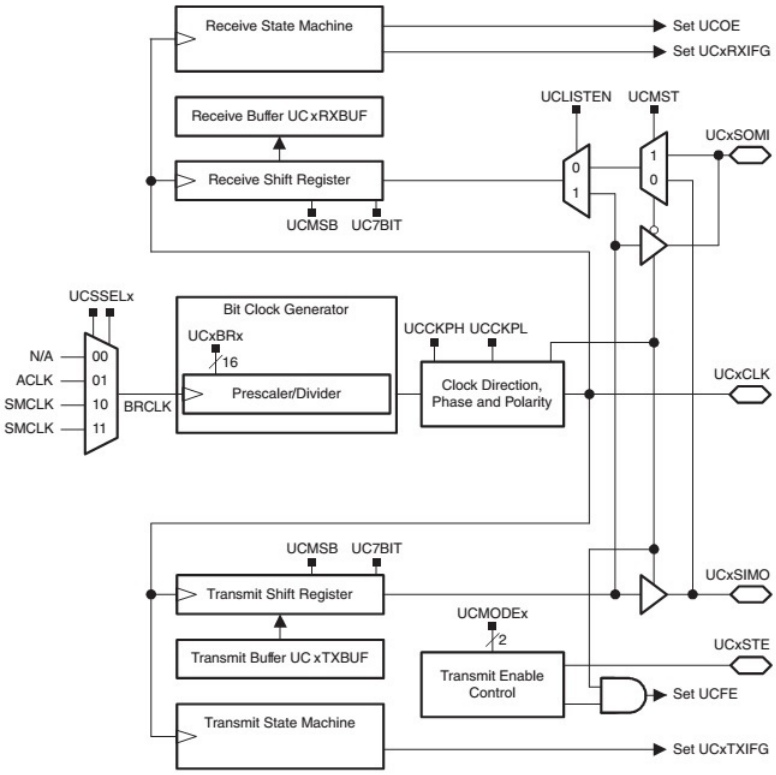

Figure 8.3: SPI

- 1, 1.5 or 2 Stop bits.

When a word is passed to the UART for asynchronous transmissions, the Start bit is added at beginning of the word. The Start bit is used to inform the receiver that a word of data is about to be send, thereby forcing the clock in the receiver to be in sync with the clock in the transmitter. After the Start bit, the individual bits of the word of data are sent, beginning with the Least Significant Bit (LSB). When data is fully transmitted, an optional parity bit is sent to the transmitter. This bit is usually used by receiver to perform simple error checking. Lastly, Stop bit will be sent to indicate the end of transmission.
When the receiver has received all of the bits in the data word, it may check for the Parity Bits (both sender and receiver must agree on whether a Parity Bit is to be used), and then the receiver searches for a Stop Bit. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be garbled and will report a Framing Error to the host processor when the data word is read. The receiver detects the Start bit by detecting the voltage transition from logic 1 to logic 0 on the

Figure 8.4: UART Frame Structure

transmission line.
Transmitting and receiving UARTs must be set at the same baud rate, character length, parity, and stop bits for proper operation.
When two UART devices connected: the TX of one device is connected to RX of other device and RX connected to TX of the second device.

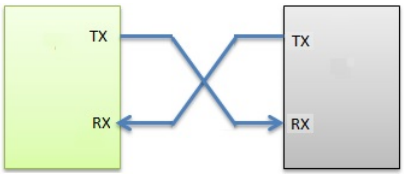

Figure 8.5: UART

8.1.3 Testing I2C

Introduction

IC is a multi-master protocol that uses only two signal lines, SDA (serial data) and SCL (serial clock). SCL functions as a clock line and SDA can function as a 1-bit serial data line or as a 1-bit serial address line. In I2C connection can have more than one master device in the two signal line. The Master device is the device that generates clock, starts communication, sends I2C commands and stops communication. The slave device is the device that listens to the bus and is addressed by the master. The communication between master and slave is based on using a protocol that defines:

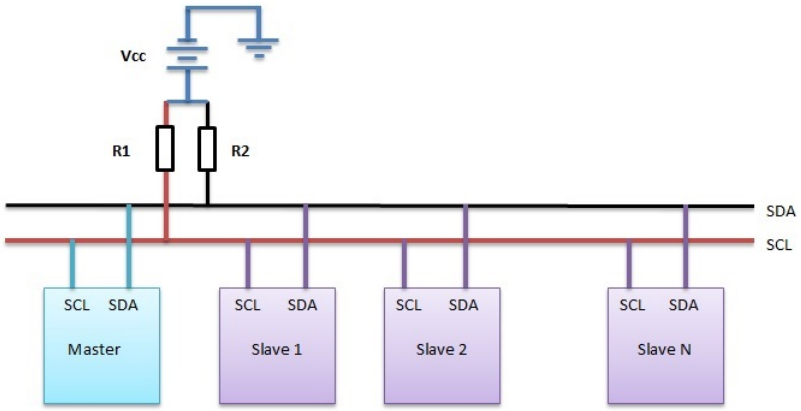

Figure 8.6: I2C

- The master sends the START bit by changing the SDA line from high to low.
- The master sends the slave address which is unique 7 bit or 10 bit I2C slave addresses.
- The master sends the Read/Write bit to determine the direction of data flow(master transmit-slave receive or master receive-slave transmit)
- Wait an acknowledge bit which the slave will only generate if its internal address matches the value sent by the master.
- Send/Receive data divided into 8-bit bytes
- Expect/Send acknowledge bit
- The master Send the STOP bit

8.2 USCI Module Driver

The figure below shows the class diagram of USCI modules drivers which have operations and attributes which are used to set and configure the USCI module of the microcontroller. This module used for MSP430F5xx, MSP430F4xx, MSP430FG41xx and MSP430F2xx families of Msp430 microcontrollers. There are four USCIA classes and four USCIB classes which are differentiated by a number 0 to 3.

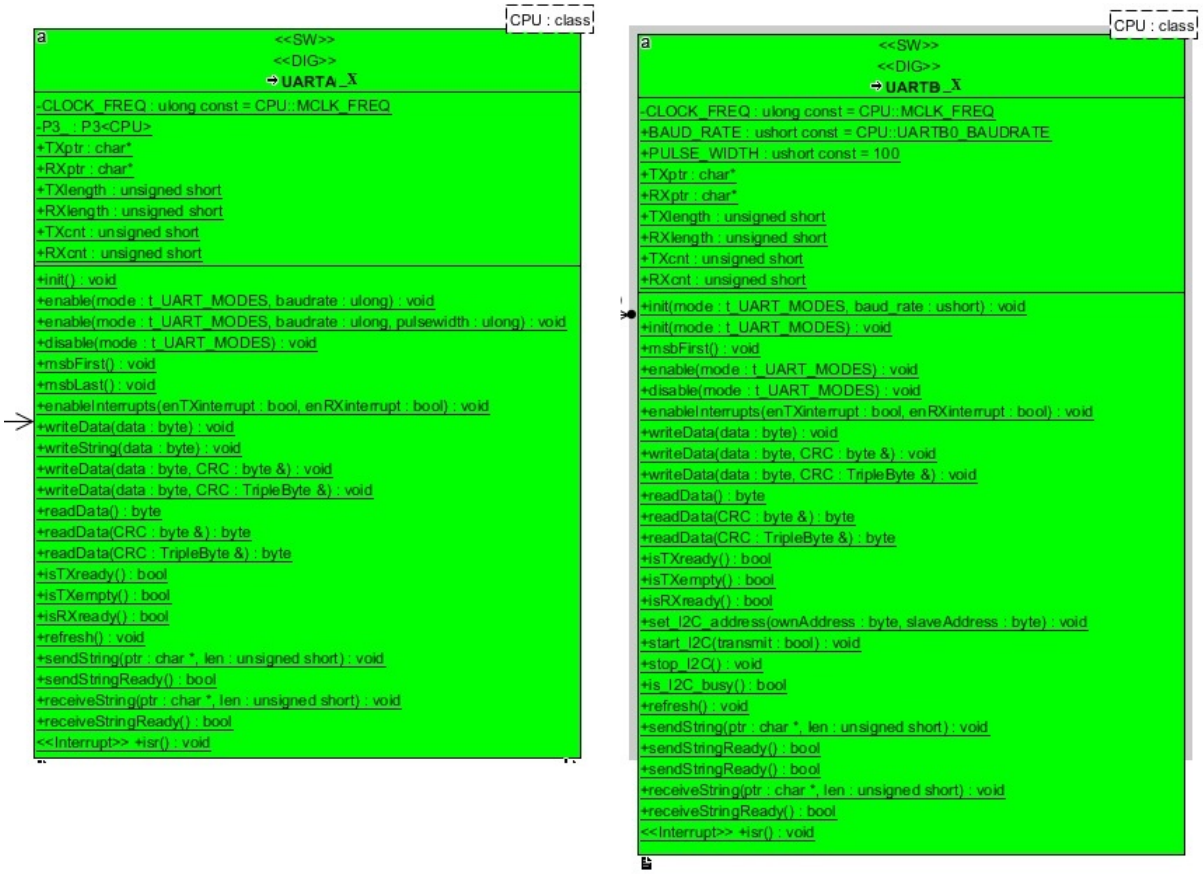

Figure 8.7: USCI Module Class diagrams

Attributes of the USCI Module Driver

- **private CLOCK_FREQ : ulong** - used to set the frequency of clock input for USCI module.
- **public TXptr : char** - is a pointer used to store the transmit data.
- **public RXptr : char** - is a pointer used to store the received data.
- **public TXlength : unsigned short** - is the number of data to be transmitted.
- **public RXlength : unsigned short** - is the number of data to be received.
- **public TXcnt : unsigned short** - is a counter which is used to count transmitted data.
- **public RXcnt : unsigned short** - is a counter which is used to count received data.

Functions or operations of the USCI Module Driver

- **public init () : void** - Initializes UARTA0. Clears all registers. Stops any ongoing transmission. Disables interrupts.
- **public enable (mode : t_UART_MODES, baudrate : ulong) : void** -Activates communication using the protocol defined by mode, with baudrate defined by baudrate and 8-bit data width. It also configures I/O pins accordingly. No action starts until data is either sent or received.
- **public enable (mode : t_UART_MODES, baudrate : ulong, pulsewidth : ulong) : void** - Activates communication using the protocol defined by mode, with baudrate defined by baudrate and 8-bit data width. It also configures IrDA pulse width to pulsewidth. It also configures I/O pins accordingly. No action starts until data is either sent or received.
- **public disable (mode : t_UART_MODES) : void** - Deactivates and aborts any ongoing data exchange. It also configures I/O pins to their normal digital I/O function, in particular as inputs. The mode must match the value used when enabling.
- **public msbFirst () : void** - Configures the UART to transit MSB first.
- **public msbLast () : void** - Configures the UART to transit MSB lasst.
- **public enableInterrupts (enTXinterrupt : bool, enRXinterrupt : bool) : void** - Enables (if true) or disables (if false):
  - UART TX interrupt (enTXinterrupt)
  - UART RX interrupt (enRXinterrupt)
- **public writeData (data : byte) : void** - It writes the value of data into the UART0 data buffer. Data is automatically sent when next data exchange starts, that is:
  - for the master, as soon as the previous transmission has terminated;
  - for the SPI slave, as soon as the master sends its first clock bit. If no data is written in time, the UART sends an unpredictable value.

  If a new data is written to the buffer before the previous has been moved to transmission register, the previous data is lost. The user shall therefore first verify that transmitter is ready by means of isTXready(): bool which must return true. If not, the user shall wait. The UART has to be enabled to allow transmission (first call to init(): void, then to enable(mode: t_UART_MODES, baudrate: ulong): void)
- **public writeString (data : byte) : void** - It writes the value of data into the UART0/UARTA0 data register.Data is automatically sent when next data exchange starts, that is:
  - for the master, as soon as the previous transmission has terminated;
  - for the SPI slave, as soon as the master sends its first clock bit. If no data is written in time, the UART sends an unpredictable value.

  The UART has to be enabled to allow transmission (first call to init(): void, then to enable(mode: t_UART_MODES, baudrate: ulong): void)
- **public writeData (data : byte, CRC : byte) : void** - Same as writeData(data: byte): void. Then it updates the CRC argument according the the chosen algorithm (bit-wise EXOR).

- **public writeData (data : byte, CRC : TripleByte) : void** - Same as writeData(data: byte, CRC: byte&): void but the parameter CRC is a radiation-tolerant TripleByte.
- **public readData () : byte** - It reads and returns received data from UART data register. The UART has to be enabled to allow reception (first call to init(): void, then to enable(mode: t_UART_MODES, baudrate: ulong): void). The caller shall first check that a byte has been received and not yet read, by means of the isRXready(): bool operation, which must return true. If not, the user shall wait. If no data has been received yet, it returns an unpredictable value. The user shall therefore first verify that transmitter is ready by means of isTXready(): bool which must return true. If not, the user shall wait.
- **public readData (CRC : byte) : byte** - Same as readData(): byte. Then it updates the CRC argument according the the chosen algorithm (bit-wise EXOR).
- **public readData (CRC : TripleByte) : byte** - Same as readData(CRC: byte&): byte but the parameter CRC is a radiation-tolerant TripleByte.
- **public isTXready () : bool** - Returns true if transmitter buffer is ready to receive a new byte for transmission; false otherwise.
- **public isTXempty () : bool** - Returns true when all data in the TX buffer and in the TX shift register have been sent; false otherwise. The caller shall wait until this routine returns true before calling disable(mode: t_UART_MODES): void, otherwise data transmission gets interrupted before completion.
- **public isRXready () : bool** - Returns true if receiver buffer is full, that is, a byte has been received but not yet read; false otherwise.
- **public refresh () : void** - Refreshes TMR variables against radiation-induced effects or other soft errors.
- **public sendString (ptr : char, len : unsigned short) : void** - Sends len bytes of the string of data pointed by ptr (independently of string termination), one byte at a time using the UART, which must be properly initialized and enabled (first call to init(): void then to enable(mode: t_UART_MODES, baudrate: ulong): void). This routine configures transmission, then it immediately exits. Transmission continues by means of the interrupt service routine isr(): void.
- **public sendStringReady () : bool** - Returns true when the transmission initiated by sendString(ptr: char*, len: unsigned short): void is finished, that is, exactly len bytes have been transmitted.
- **public receiveString (ptr : char, len : unsigned short) : void** - Receives len bytes of data and stores them into the string pointed by ptr (independently of string termination), one byte at a time using the UART, which must be properly initialized and enabled (first call to init(): void then to enable(mode: t_UART_MODES, baudrate: ulong): void). This routine configures reception, then it immediately exits. Reception continues by means of the interrupt service routine isr(): void.
- **public receiveStringReady () : bool** - Returns true when the reception initiated by receiveString(ptr: char*, len: unsigned short): void is finished, that is, exactly len bytes have been received.
- **public isr () : void** - interrupt service routine for USCI.

8.3 Test program for USCI Driver

This diagram describe how the test program for USCI driver is composed. as mentioned above the USCI driver work for different MSP430 families but to implement the the test I have used only MSP430F5438A microcontroller.

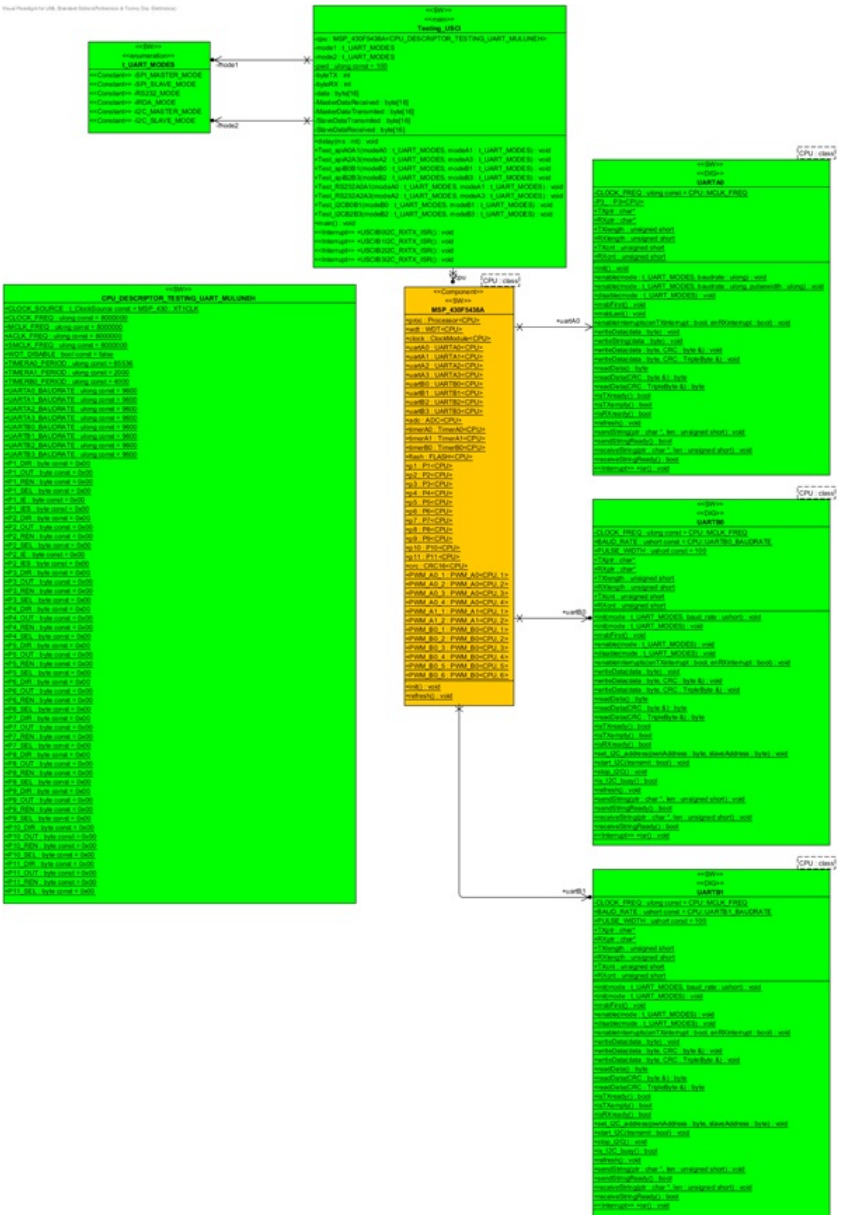

Figure 8.8: Test program for USCI Module driver

8.3.1 Class Testing USCI

Testing_is USCI class used to test USCI driver functionally. This class has an attributes:
private cpu : MSP_430F5438A define which microcontroller series is under test. and it uses access the USCI driver.
private mode1 : t_UART_MODES let the user to select different modes of the USCI module: such

as SPI_MASTER_MODE, SPI_SLAVE_MODE, RS232_MODE, IRDA_MODE, I2C_MASTER_MODE and I2C_SLAVE_MODE.
private mode2 : t_UART_MODES let the user to select different modes of the USCI module: such as SPI_MASTER_MODE, SPI_SLAVE_MODE, RS232_MODE, IRDA_MODE, I2C_MASTER_MODE and I2C_SLAVE_MODE.
private pwd : ulong
private byteTX : int - a counter used to count the transmit data.
private byteRX : int - a counter used to count the receive data.
private data : byte[16] - used to store a received data.
private MasterDataReceived : byte[16] is used to store data received at the master side.
private MasterDataTransmited : byte[16]is used to store data transmitted from the master side.
private SlaveDataTransmited : byte[16]is used to store data transmitted from the slave side.
private SlaveDataReceived : byte[16] is used to store data received at the slave side.
This class has following function:
public delay (ms : int) : void
public Test_spiA0A1 (modeA0 : t_UART_MODES, modeA1 : t_UART_MODES) : void - create a communication between USCIA0 and USCIA1 with SPI protocol. based on the parameter that is Passed to modeA0 and modeA1,both ports can be slave or master.
public Test_spiA2A3 (modeA2 : t_UART_MODES, modeA3 : t_UART_MODES) : void - create a communication between USCIA2 and USCIA3 with SPI protocol. based on the parameter that is Passed to modeA2 and modeA3,both ports can be slave or master.
public Test_spiB0B1 (modeB0 : t_UART_MODES, modeB1 : t_UART_MODES) : void- create a communication between USCIB0 and USCIB1 with SPI protocol. based on the parameter that is Passed to modeB0 and modeB1,both ports can be slave or master.
public Test_spiB2B3 (modeB2 : t_UART_MODES, modeB3 : t_UART_MODES) : void- create a communication between USCIB2 and USCIB3 with SPI protocol. based on the parameter that is Passed to modeB2 and modeB3,both ports can be slave or master.
public Test_RS232A0A1 (modeA0 : t_UART_MODES, modeA1 : t_UART_MODES) : void - create a communication between USCIA0 and USCIA1 with RS232 protocol.
public Test_RS232A2A3 (modeA2 : t_UART_MODES, modeA3 : t_UART_MODES) : void - create a communication between USCIA2 and USCIA3 with RS232 protocol.
public Test_I2CB0B1 (modeB0 : t_UART_MODES, modeB1 : t_UART_MODES) : void - create a communication between USCIB0 and USCIB1 with I2C protocol. based on the parameter that is Passed to modeB0 and modeB1,both ports can be slave or master.
public Test_I2CB2B3 (modeB2 : t_UART_MODES, modeB3 : t_UART_MODES) : void - create a communication between USCIB2 and USCIB3 with I2C protocol. based on the parameter that is Passed to modeB2 and modeB3,both ports can be slave or master.
This class has four function for handling interrupt service routine:
public USCIB0I2C_RXTX_ISR () : void - handles transmit and receive interrupt service routine for USCIB0.
public USCIB1I2C_RXTX_ISR () : void - handles transmit and receive interrupt service routine for USCIB1.
public USCIB2I2C_RXTX_ISR () : void - handles transmit and receive interrupt service routine for USCIB2.
public USCIB3I2C_RXTX_ISR () : void - handles transmit and receive interrupt service routine for USCIB3.
public main () : void - The source code for the main function is found in appendix G. The code let the user to select two pair of USCI port for each six modes of USCI by entering a number from '1' to '10'.

8.4 Testing Procedure of USCI Module

Testing of the USCI is little bit complicated. Before we are ruining the code we have to set up hardware connection using wire between two USCI based on the protocol. The table below shows the pin connection between two modules for each USCI mode.

Table 8.1: Connection between USCIA0 and USCIA1 for SPI

SPI	Pin for USCIA0	Pin for USCIA1
SOMI	P3.5	P5.7
Clock	P3.0	P3.6
Enable	P3.3	P5.5
SIMO	P3.4	P5.6

Table 8.2: Connection between USCIA2 and USCIA3 for SPI

SPI	Pin for USCIA2	Pin for USCIA3
SOMI	P9.5	P10.5
Clock	P9.0	P10.0
Enable	P9.3	P10.3
SIMO	P9.4	P10.4

Table 8.3: Connection between USCIB0 and USCIB1 for SPI

SPI	Pin for USCIB0	Pin for USCIB1
SOMI	P3.2	P5.4
Clock	P3.3	P5.5
Enable	P3.0	P3.6
SIMO	P3.1	P3.7

Table 8.4: Connection between USCIB2 and USCIB3 for SPI

SPI	Pin for USCIB2	Pin for USCIB3
SOMI	P9.2	P10.2
Clock	P9.3	P10.3
Enable	P9.0	P10.0
SIMO	P9.1	P10.1

Table 8.5: Connection between USCIA0 and USCIA1 for RS232 and IrDA

RS232	Pin for USCIA0	Pin for USCIA1
TDX $->$ RDX	P3.4	P5.7
RDX $<-$ TDX	P3.5	P5.6

Table 8.6: Connection between USCIA2 and USCIA3 for RS232 and IrDA

RS232	Pin for USCIA2	Pin for USCIA3
TDX $->$ RDX	P9.4	P10.5
RDX $<-$ TDX	P9.5	P10.4

Table 8.7: Connection between USCIB0 and USCIB1 for I2C

I2C	Pin for USCIB0	Pin for USCIB1
Data	P3.1	P3.7
Clock	P3.2	P5.4

Table 8.8: Connection between USCIB2 and USCIB3 for I2C

I2C	Pin for USCIB2	Pin for USCIB3
Data	P9.1	P10.1
Clock	P9.2	P10.2

In addition to the above pin connection when the mode is I2C we have to connect the data and clock line to Vcc with pull up resistor of 10 kohm.
The source code for testing the USCI module is found in appendix G. open the IAR workbench and load the code to the microcontroller using the JTAG cable. The IAR has Input output panel and it let the user to select different modules of USCI and different modes USCI operations by using a number from 1 to 10.
By using the IRA workbench we can see the registers of USCI module. so before you start loading the code select View Register from the view menu and you will get a panel and select the one of USCI mopdule register. From the view menu select also IO display.
Follow the following step to configure the microcontroller for testing USCI driver system:-

- enter any number between 1 to 10 to select the mode and the module to be tested.
- check the transmitted data one module is equal with the received data of the other module for the IO display panel.
- repeat the above for all modules or classes of USCI.

8.5 Testing Result

All operations of the drives has been tested but to reduce the space I have put only the few of results. When the user enter 1 the microcontroller USCIA0 is configured as SPI master and USCIA1 is configured as slave. For this test, USCIA0 send a number from 0 to 16 and read the data that is received by USCIA1.
When the user enter 2 the microcontroller USCIA0 is configured as SPI slave and USCIA1 is configured as master. For this test, USCIA1 transmit a number from 0 to 16 and read and display the data that is received by USCIA0. The figure below shows the test result of the SPI test of USCIA0 and USCIA1.

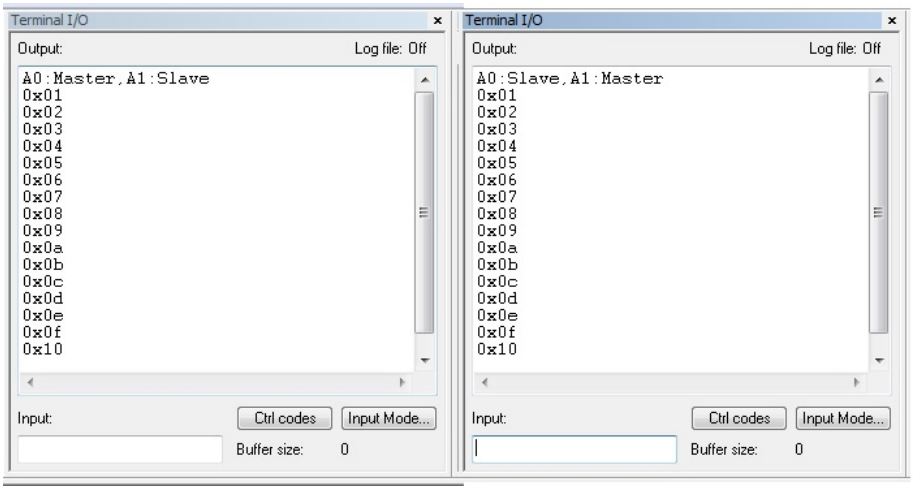

Figure 8.9: Test result of SPI connection of USCIA0 as master USCIA1 as slave

When the user enter 5 the microcontroller USCIA0 is configured as UART(RS232)and USCIA1 is configured also UART(RS232).For this test, USCIA0 a transmit a number from 0 to 16 and read and display the data that is received by USCIA1. After USCIA0 finish transmitting USCIA1 start

transmitting and USCIA0 start receiving. The figure below show the test result of the RS232 communication between USCIA0 and USCIA1.

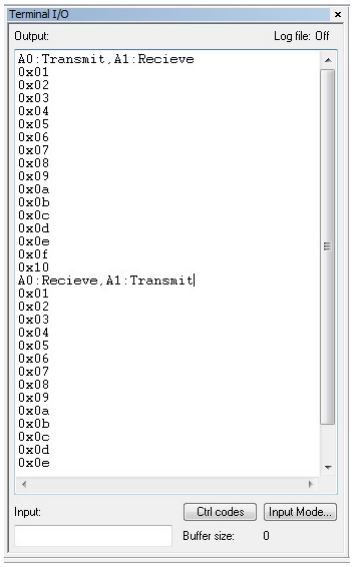

Figure 8.10: Test result of RS232 connection of USCIA0 to USCIA1

When the user enter 7 the microcontroller USCIA0 is configured as UART(IrAD)and USCIA1 is configured also UART(IrAD).For this test, USCIA0 a transmit a number from 0 to 16 and read and display the data that is received by USCIA1. After USCIA0 finish transmitting USCIA1 start transmitting and USCIA0 start receiving. The figure below show the test result of the IrAD communication between USCIA0 and USCIA1.

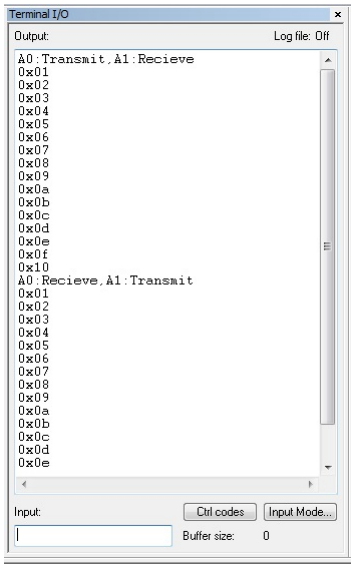

Figure 8.11: Test result of IrAD connection of USCIA0 to USCIA1

When the user enter 9 the microcontroller USCIB0 is configured as master I2C and USCIB1 is configured also slave. For this test,USCIB1 transmit a number from 16 to 31 in slave mode the read and display the data that is received by USCIB0. After USCIB1 finish transmitting USCIB0 start transmitting and USCIB1 start receiving. USCIB0 transmit a number from 0 to 15 and read and display the data that is received by USCIB1. The figure below show the test result of the I2C communication between USCIB0 and USCIB1.

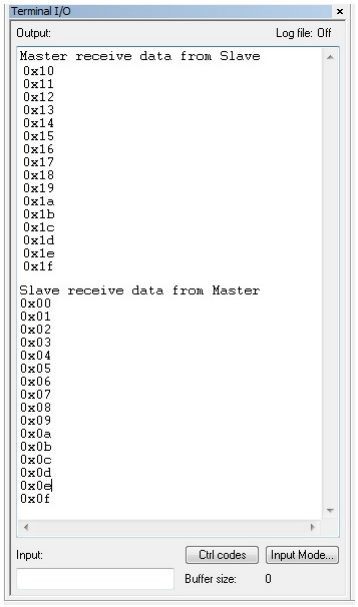

Figure 8.12: Test result of I2C connection of USCIB0 as master to USCIB1 as slave

Chapter 9

CONCLUSION AND FUTURE WORK

9.1 Conclusion

My work able to test all the drivers of the peripheral successfully. In some drivers I have found error codes and I have replaced them with the right code. With my test program we can successfully test all the peripheral drivers of MSP430F5438A microcomputer but if we want to test for the other families of the MSP430 we have to replace MSP430F5438A with the microcontroller that we want before we generate the code form Visual Paradigm for UML.

9.2 Future work

This thesis work can extend to a broader scope by considering different case of Functional Testing. In this thesis the functional testing done using limited resource. If we use TTL converter to USB to connect the pins of microcontroller with the PC , we can do some applilcation software which run on the pc and we will test all drivers of the microcontroller form the pc. for example, for testing the serial communication interface drivers I have used two module of the microcontroller but instead we can use the connector with the PC to test one module only.

BIBLIOGRAPHY

[1] SLAU208M, *MSP430x5xx and MSP430x6xx Family User Guide*, TEXAS INSTRUMENT 2013.

[2] SLAU278P, *MSP430 Hardware Tools User Guide*, TEXAS INSTRUMENT 2013.

[3] SLAS655D, *MSP430F5438A Data Sheet*, TEXAS INSTRUMENT 2013.

[4] James O. Hamblen, *INTRODUCTION TO EMBEDDED SYSTEMS USING WINDOWS EMBEDDED CE*, School ofElectrical and Computer Engineering GeorgiaInstitute of Technology January 2007.

[5] Gustavo Litovsky, *Beginning Microcontrollers with the MSP430 Tutorial*, Version 0.4.

Appendices

Appendix A

THE PROGRAM USED TO CONFIGURE THE CLOCK MODULE

```
#include "Testing_ClockGeneratorE.h"

#include "CPU_DESCRIPTOR_TESTING_MULUNEH.h"
#include "MSP_430F5438A.h"
#include "t_ClockSource.h"


MSP_430::MSP_430F5438A<CPU_DESCRIPTOR_TESTING_MULUNEH> cpu;
MSP_430::t_ClockSource source;


void main() {
cpu.wdt.disable();
cpu.clock.output_MCLK(true);
cpu.clock.output_SMCLK(true);
cpu.clock.output_ACLK(true);
ushort divide;
char c;
do
{
  char x;
  printf ("Enter code\n");
  x=getchar();
  switch(x)
  {
  case 'a' :
    source = MSP_430::XT1CLK;
    break;
  case 'b' :
    source = MSP_430::VLOCLK;
    break;
  case 'c' :
    source = MSP_430::REFOCLK;
    break;
  case 'd' :
    source = MSP_430::DCOCLK;
    break;
  case 'e' :
    source = MSP_430::DCOCLKDIV;
    break;
  case 'f' :
```

```
    source = MSP_430::XT2CLK;
    break;
  case 'g' :
    cpu.clock.output_MCLK(false);
    break;
  case 'h' :
    cpu.clock.output_MCLK(true);
    break;
  case 'i' :
    cpu.clock.output_SMCLK(false);
    break;
  case 'j' :
    cpu.clock.output_SMCLK(true);
    break;
  case 'k' :
    cpu.clock.output_ACLK(false);
    break;
  case 'l' :
    cpu.clock.output_ACLK(true);
    break;
  default:
    source = MSP_430::XT1CLK;
    break;
  }
  //accept the divisor from the user
  int y;
  scanf("%d", &y);
  if((y == 1) || (y ==2) || (y == 4) || (y ==8) || (y ==16) || (y ==32))
    divide=y;
  else
    divide=1;
  //set ACLK
  cpu.clock.activateSingleClock(source);
  cpu.clock.setACLK(source ,divide);
  //set MCLK
  cpu.clock.setMCLK(source ,divide);
  //set MCLK
  cpu.clock.setSMCLK(source ,divide);
} while ((c = getchar())!= 'x');
cpu.clock.resetFlags();
}
```

Appendix B

THE PROGRAM USED TO CONFIGURE TIMER MODULES

```
#include "Testing_Timer.h"

#include "MSP_430F5438A.h"
#include "CPU_DESCRIPTOR_TEST_TIMER_MULUNEH.h"


MSP_430::MSP_430F5438A<Testing_Timer_Muluneh::CPU_DESCRIPTOR_TEST_TIMER_MULUNEH> cpu;


void testTimerA0() {
        cpu.timerA0.init();
        cpu.timerA0.enableInterrupt(0);
        cpu.timerA0.start();
}

void testTimerA1() {
        cpu.timerA1.init();
        cpu.timerA1.enableInterrupt(0);
        cpu.timerA1.start();
}

void testTimerB0() {
        cpu.timerB0.init();
        cpu.timerB0.enableInterrupt(0);
        cpu.timerB0.start();
}

void main() {
        cpu.wdt.disable();
        cpu.init();
        cpu.p1.init();
        __enable_interrupt();
        char c;
        char x='a';
        do
        {
          x=getchar();
          switch(x)
          {
          case 'a' :
            cpu.timerA1.disableInterrupt(0);
```

```
            cpu.timerB0.disableInterrupt(0);
            testTimerA0();
            break;
          case 'b' :
            cpu.timerA1.disableInterrupt(0);
            cpu.timerB0.disableInterrupt(0);
            testTimerA1();
            break;
          case 'c' :
            cpu.timerA1.disableInterrupt(0);
            cpu.timerB0.disableInterrupt(0);
            testTimerB0();
            break;
          case 'd' :
            cpu.timerA0.disableInterrupt(0);
            cpu.timerA1.disableInterrupt(0);
            cpu.timerB0.disableInterrupt(0);
            break;
          }
          for (volatile long i=0;i<1000000;i++);
        } while (true);
}
```

Appendix C

THE PROGRAM USED TO CONFIGURE PWM MODULES

```
#include "Testing_PWM.h"

#include "MSP_430F5438A.h"
#include "CPU_DESCRIPTOR_TESTING_PWM_MULUNEH.h"


MSP_430::MSP_430F5438A<Testing_PWM_Muluneh::CPU_DESCRIPTOR_TESTING_PWM_MULUNEH> Testing_PWM_Muluneh::Testin

 void Testing_PWM_Muluneh::Testing_PWM::main() {
                cpu.wdt.disable();
                cpu.proc.init();
                cpu.clock.init();
                P1DIR |= BIT0;                                  // Set P1.0 to output direction
                P1OUT &= ~BIT0;                                 // Set the red LED on
                ushort DutyCycle=0xffff;
                ulong temp;
                __enable_interrupt();
                ushort x='a';
                do
                {
                  scanf("%c",&x);
                  scanf("%d",&DutyCycle);

                  cpu.timerA0.init();
                  cpu.timerA0.enableInterrupt();
                  cpu.timerA0.start();
                  cpu.timerA1.init();
                  cpu.timerA1.start();
                  cpu.timerB0.init();
                  cpu.timerB0.start();
                  switch(x)
                  {
                  case 'a':
                    cpu.PWM_A0_1.init(false);
                    cpu.PWM_A0_1.enableInterrupt();
                    cpu.PWM_A0_1.setDutyCycle(DutyCycle);                   // PWM duty cycle, time cycle on
                    temp=((cpu.PWM_A0_1.getDutyCycleRaw()<<16)/cpu.PWM_A0_1.getDutyCycleMax())+1;
                  case 'A':
                    cpu.PWM_A0_1.init(true);
                    cpu.PWM_A0_1.enableInterrupt();
                    cpu.PWM_A0_1.setDutyCycle(DutyCycle);                   // PWM duty cycle, time cycle on
```

```
      temp=((cpu.PWM_A0_1.getDutyCycleRaw()<<16)/cpu.PWM_A0_1.getDutyCycleMax())+1;



      break;
    case 'b' :
      cpu.PWM_A0_2.init(true);
      cpu.PWM_A0_2.setDutyCycle(DutyCycle);                    // PWM duty cycle, time cycle
      temp=((cpu.PWM_A0_2.getDutyCycleRaw()<<16)/cpu.PWM_A0_2.getDutyCycleMax())+1;


      break;
    case 'c' :
      cpu.PWM_A1_1.disableInterrupt();
      cpu.PWM_B0_1.disableInterrupt();
      cpu.PWM_A0_3.init(true);
      cpu.PWM_A0_3.setDutyCycle(DutyCycle); // PWM duty cycle, time cycle on vs. off
      temp=((cpu.PWM_A0_3.getDutyCycleRaw()<<16)/cpu.PWM_A0_3.getDutyCycleMax())+1;

      break;
    case 'd' :
      cpu.PWM_A0_4.setDutyCycle(DutyCycle); // PWM duty cycle, time cycle on vs. off
      temp=((cpu.PWM_A0_4.getDutyCycleRaw()<<16)/cpu.PWM_A0_4.getDutyCycleMax())+1;
      break;

    case 'e':
      cpu.PWM_A1_1.init(false);
      cpu.PWM_A1_1.setDutyCycle(DutyCycle);
      temp=((cpu.PWM_A1_1.getDutyCycleRaw()<<16)/cpu.PWM_A1_1.getDutyCycleMax())+1;


      break;
    case 'f':
      cpu.PWM_A1_2.init(true);
      cpu.PWM_A1_2.setDutyCycle(DutyCycle);
      temp=((cpu.PWM_A1_2.getDutyCycleRaw()<<16)/cpu.PWM_A1_2.getDutyCycleMax())+1;


      break;
    case 'g':
      cpu.PWM_B0_1.init(false);
      cpu.PWM_B0_1.setDutyCycle(DutyCycle);
      temp=((cpu.PWM_B0_1.getDutyCycleRaw()<<16)/cpu.PWM_B0_1.getDutyCycleMax())+1;



    }
      if(DutyCycle==temp) {
        printf("DC_OK!\n");
      } else {
        volatile int x = 0;
        printf("DC_not_OK!\n");
      }

    for (volatile long j=0;j<1000000;j++);



  } while (true);
}
```

Appendix D

THE PROGRAM USED TO CONFIGURE ADC MODULE

```
#include "Testing_ADC.h"

#include "CPU_DESCRIPTOR_TESTING_ADC_MULUNEH.h"
#include "MSP_430F5438A.h"
#include "t_ADC_VREF.h"


MSP_430::MSP_430F5438A<CPU_DESCRIPTOR_TESTING_ADC_MULUNEH> cpu;
MSP_430::t_ADC_VREF Vref;


main() {
        cpu.wdt.disable();
        cpu.init();
        do {
          char x;
        // printf ("Enter code\n");
          x=getchar();
          switch(x)
          {
          case 'a' :
              Vref = MSP_430::VREF_EXT;
              break;
          case 'b' :
              Vref = MSP_430::VREF_1_5;
              break;
          case 'c' :
              Vref = MSP_430::VREF_2_5;
              break;
          case 'd' :
              Vref = MSP_430::VREF_VDD;
              break;
         default:
              Vref = MSP_430::VREF_VDD;
              break;
        }
          //select the channel
          int y;
          byte channel;
          scanf("%d", &y);
          if(y>=0 && y < 16)
```

```
            channel=(byte)y;
          else
        {
            channel=0x0A;
        }


        cpu.adc.init(100,Vref,0x200);

        cpu.adc.enable();
        if( channel==0x0A)
        {
        cpu.adc.tempSensor(true);
        }
        cpu.adc.select(channel);
        ushort i=100;
        volatile ushort value;
        while(i--)
        {
          value=cpu.adc.convert();
          value=value;
        }
         cpu.adc.disable();
         cpu.adc.deactivate(channel);
         for (volatile long j=0;j<1000000;j++);

        }while(true);

}
```

Appendix E

THE PROGRAM USED TO CONFIGURE CRC MODULE

```
#include "Testing_CRC.h"

#include "MSP_430F5438A.h"
#include "CPU_DESCRIPTOR_TESTING_CRC_MULUNEH.h"


MSP_430::MSP_430F5438A<Testing_CRC_Muluneh::CPU_DESCRIPTOR_TESTING_CRC_MULUNEH> cpu;


ushort get_crc16_SW(ushort in, ushort int_value) {
        int CRC[16];
        int data[16];
        int DoInvert;
        ushort result=0x0000;       // CRC Result

        for (int i=0; i<16; i++)    // Init CRC=int_value
        {
        if(int_value & 0x8000)
        {
        CRC[i] = 1;
        }
        else
        {
        CRC[i] = 0;
        }
        int_value <<= 1;
        }

        for (int i=15; i>=0; i--)
        {
        if(in & 0x8000)
        {
        data[i] = 1;
        }
        else
        {
        data[i] = 0;
        }
        in <<= 1;
        }
```

```
        for (int i=0; i<16; ++i)
        {
        DoInvert = data[i] ^ CRC[15];          // XOR required?

        CRC[15] = CRC[14];
        CRC[14] = CRC[13];
        CRC[13] = CRC[12];
        CRC[12] = CRC[11] ^ DoInvert;
        CRC[11] = CRC[10];
        CRC[10] = CRC[9];
        CRC[9] = CRC[8];
        CRC[8] = CRC[7];
        CRC[7] = CRC[6];
        CRC[6] = CRC[5];
        CRC[5] = CRC[4] ^ DoInvert;
        CRC[4] = CRC[3];
        CRC[3] = CRC[2];
        CRC[2] = CRC[1];
        CRC[1] = CRC[0];
        CRC[0] = DoInvert;
        }
        for (int i=15; i>=0; i--)
        {
        if(CRC[i] == 1)
        {
        result <<= 1;
        result = result |(0x0001);
        }
        else
        {
        result <<= 1;
        }
        }

        return(result);
}

main() {
        cpu.wdt.disable();
        cpu.crc.init(0xFFFF);

        ushort hw_result=0xFFFF;
        ushort sw_result;
        do
        {
        ushort data = 0x001F;
        ushort rdata=0x00F8;
        char x;
        printf ("Enter code\n");
        scanf("%s", &x);
        switch(x)
        {
        case 'a' :
        sw_result=get_crc16_SW(data,hw_result);
        cpu.crc.add_data_in_normal(data);// Put '1' into the data in register
        hw_result=cpu.crc.crc_result_in_normal();

        break;
        case 'b' :
        sw_result=get_crc16_SW(rdata,hw_result);
        cpu.crc.add_data_in_reversed(data);// Put '1' into the data in register
        hw_result=cpu.crc.crc_result_in_reversed();
```

```
            break;
            default:
            printf("Enter a or b  \n");
            break;
            }
            printf("MSP430 CRC Result = 0x%04x\n",hw_result);
            printf("CRC SW Result: 0x%04x\n",sw_result);
            } while (1);
}
```

Appendix F

THE PROGRAM USED TO CONFIGURE FLASH MEMORY

```
#include "Testing_Flash.h"
#include "stdio.h"

#include "MSP_430F5438A.h"
#include "t_FLASH_BANK.h"
#include "CPU_DESCRIPTOR_TESTING_FlashMEM_MULUNEH.h"


MSP_430::MSP_430F5438A<Testing_FlashMEM_Muluneh::CPU_DESCRIPTOR_TESTING_FlashMEM_MULUNEH> cpu;
MSP_430::t_FLASH_BANK bank;


void main() {

byte * address;
byte segment=0;
byte Wdata = 0x5;
byte Rdata;
while(true)
{
 printf("Enter code:\n");

char x=getchar();
switch(x)
{
case 'a':
bank = MSP_430::INFO_A;
address = (byte *)0x1980;
segment=0;
break;
case 'b':
bank = MSP_430::INFO_B;
address = (byte *)0x1900;
segment=0;
break;
case 'c':
bank = MSP_430::INFO_C;
address = (byte *)0x1880;
segment=0;
break;
case 'd':
```

```
bank = MSP_430::INFO_D;
address = (byte *)0x1800;
segment=0;
break;
case 'e':
bank = MSP_430::MAIN_B;
address = (byte *)0x10000;
segment=0;
break;
case 'f':
bank = MSP_430::MAIN_C;
address = (byte *)0x20000;
segment=0;
break;
case 'g':
bank = MSP_430::MAIN_D;
address = (byte *)0x30000;
segment=0;
break;
case 'h':
bank = MSP_430::MAIN_B_seg;
address = (byte *)0x10000;
segment=1;
break;
case 'i':
bank = MSP_430::MAIN_C_seg;
address = (byte *)0x20000;
segment=1;
break;
case 'j':
bank = MSP_430::MAIN_D_seg;
address = (byte *)0x30000;
segment=1;
break;
}

cpu.flash.erase(bank,segment);
Rdata = cpu.flash.read(address);
printf ("%d\n", (int) Rdata);

cpu.flash.initiateWrite();
cpu.flash.write( address, Wdata);
cpu.flash.terminateWrite();
Rdata = cpu.flash.read(address);
printf ("%d\n", (int) Rdata);

Wdata++;
cpu.flash.initiateWrite();
cpu.flash.write( address, Wdata);
cpu.flash.write( address+1, Wdata);
cpu.flash.terminateWrite();
Rdata = cpu.flash.read(address);
printf ("%d\n", (int) Rdata);

cpu.flash.erase(bank,segment);
cpu.flash.initiateWrite();
cpu.flash.write( address, Wdata);
cpu.flash.write( address+1, Wdata);
cpu.flash.terminateWrite();
Rdata = cpu.flash.read(address);
printf ("%d\n", (int) Rdata);
```

```
cpu.flash.erase(bank,segment);
cpu.flash.write( address, Wdata);
cpu.flash.write( address+1, Wdata);
Rdata = cpu.flash.read(address);
printf ("%d\n", (int) Rdata);
}
}
```

Appendix G

THE PROGRAM USED TO CONFIGURE USCI MODULE

```
#include "Testing_USCI.h"

#include "CPU_DESCRIPTOR_TESTING_UART_MULUNEH.h"
#include "MSP_430F5438A.h"
#include "t_UART_MODES.h"



MSP_430::MSP_430F5438A<CPU_DESCRIPTOR_TESTING_UART_MULUNEH> cpu;
MSP_430::t_UART_MODES mode1;
MSP_430::t_UART_MODES mode2;
ulong const pwd= 100;

int byteTX;
int byteRX;
byte data[16];
byte MasterDataReceived[16];
byte MasterDataTransmited[16];
byte SlaveDataTransmited[16];
byte SlaveDataReceived[16];


void delay(int ms) {
         int i, j;
        for (i = 0; i <= ms; i++)
        {
        for (j = 0; j<=255; j++);
        }
}

void Test_spiA0A1(MSP_430::t_UART_MODES modeA0, MSP_430::t_UART_MODES modeA1) {
        cpu.uartA0.init();
        cpu.uartA1.init();

        cpu.uartA0.enable(modeA0,9600);
        cpu.uartA1.enable(modeA1,9600);

        cpu.uartA0.msbFirst();
        cpu.uartA1.msbFirst();


        if(modeA0==MSP_430::SPI_MASTER_MODE && modeA1==MSP_430::SPI_SLAVE_MODE)
```

```
        {
        printf("A0:Master,A1:Slave\n");
        int j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartA0.isTXready());
        cpu.uartA0.writeData(k);
        while (!cpu.uartA0.isTXempty());
        while(!cpu.uartA1.isRXready());
        data[j]=cpu.uartA1.readData();
        j++;
        }
        }

        else if(modeA0==MSP_430::SPI_SLAVE_MODE && modeA1==MSP_430::SPI_MASTER_MODE)
        {
        printf("A0:Slave,A1:Master\n");
        int j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartA1.isTXready());
        cpu.uartA1.writeData(k);
        while (!cpu.uartA1.isTXempty());
        while(!cpu.uartA0.isRXready());
        data[j]=cpu.uartA0.readData();
        j++;
        }
        }
        cpu.uartA0.disable(modeA0);
        cpu.uartA1.disable(modeA1);
}

void Test_spiA2A3(MSP_430::t_UART_MODES modeA2, MSP_430::t_UART_MODES modeA3) {
        cpu.uartA2.init();
        cpu.uartA3.init();

        cpu.uartA2.enable(modeA2,9600);
        cpu.uartA3.enable(modeA3,9600);

        cpu.uartA2.msbFirst();
        cpu.uartA3.msbFirst();


        if(modeA2==MSP_430::SPI_MASTER_MODE && modeA3==MSP_430::SPI_SLAVE_MODE)
        {
        printf("A2:Master,A3:Slave\n");
        int j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartA2.isTXready());
        cpu.uartA2.writeData(k);
        while (!cpu.uartA2.isTXempty());
        while(!cpu.uartA3.isRXready());
        data[j]=cpu.uartA3.readData();
        j++;
        }
        }

        else if(modeA2==MSP_430::SPI_SLAVE_MODE && modeA3==MSP_430::SPI_MASTER_MODE)
        {
        printf("A2:Slave,A3:Master\n");
        int j=0;
```

```
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartA3.isTXready());
        cpu.uartA3.writeData(k);
        while (!cpu.uartA3.isTXempty());
        while(!cpu.uartA2.isRXready());
        data[j]=cpu.uartA2.readData();
        j++;
        }
        }

        cpu.uartA2.disable(modeA2);
        cpu.uartA3.disable(modeA3);
}

void Test_spiB0B1(MSP_430::t_UART_MODES modeB0, MSP_430::t_UART_MODES modeB1) {
        cpu.uartB0.init(modeB0);
        cpu.uartB1.init(modeB1);



        cpu.uartB0.msbFirst();
        cpu.uartB1.msbFirst();


        if(modeB0==MSP_430::SPI_MASTER_MODE && modeB1==MSP_430::SPI_SLAVE_MODE)
        {
        printf("B0:Master,B1:Slave\n");
        int j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartB0.isTXready());
        cpu.uartB0.writeData(k);
        while (!cpu.uartB0.isTXempty());
        while(!cpu.uartB1.isRXready());
        data[j]=cpu.uartB1.readData();
        j++;
        }
        }

        else if(modeB0==MSP_430::SPI_SLAVE_MODE && modeB1==MSP_430::SPI_MASTER_MODE)
        {
        printf("B0:Slave,B1:Master\n");
        int j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartB1.isTXready());
        cpu.uartB1.writeData(k);
        while (!cpu.uartB1.isTXempty());
        while(!cpu.uartB0.isRXready());
        data[j]=cpu.uartB0.readData();
        j++;
        }
        }

        cpu.uartB0.disable(modeB0);
        cpu.uartB1.disable(modeB1);
}

void Test_spiB2B3(MSP_430::t_UART_MODES modeB2, MSP_430::t_UART_MODES modeB3) {
        cpu.uartB2.init(modeB2);
        cpu.uartB3.init(modeB3);
```

```
        cpu.uartB2.msbFirst();
        cpu.uartB3.msbFirst();


        if(modeB2==MSP_430::SPI_MASTER_MODE && modeB3==MSP_430::SPI_SLAVE_MODE)
        {
        printf("B2:Master,B3:Slave\n");
        int j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartB2.isTXready());
        cpu.uartB2.writeData(k);
        while (!cpu.uartB2.isTXempty());
        while(!cpu.uartB3.isRXready());
        data[j]=cpu.uartB3.readData();
        j++;
        }
        }

        else if(modeB2==MSP_430::SPI_SLAVE_MODE && modeB3==MSP_430::SPI_MASTER_MODE)
        {
        printf("B2:Slave,B2:Master\n");
        int j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartB3.isTXready());
        cpu.uartB3.writeData(k);
        while (!cpu.uartB3.isTXempty());
        while(!cpu.uartB2.isRXready());
        data[j]=cpu.uartB2.readData();
        j++;
        }
        }

        cpu.uartB2.disable(modeB2);
        cpu.uartB3.disable(modeB3);
}

void Test_RS232A0A1(MSP_430::t_UART_MODES modeA0, MSP_430::t_UART_MODES modeA1) {
        cpu.uartA0.init();
        cpu.uartA1.init();

        if(modeA0==MSP_430::RS232_MODE)
        {
        cpu.uartA0.enable(modeA0,9600);
        }
        else if(modeA0==MSP_430::IRDA_MODE)
        {
          cpu.uartA0.enable(modeA0,9600,pwd);
        }
        if(modeA1==MSP_430::RS232_MODE)
        {
        cpu.uartA1.enable(modeA1,9600);
        }
        else if(modeA1==MSP_430::IRDA_MODE)
        {
         cpu.uartA1.enable(modeA1,9600,pwd);
        }
```

```
        cpu.uartA0.msbLast();
        cpu.uartA1.msbLast();

        printf("A0:Transmit,A1:Recieve\n");
        int j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartA0.isTXready());
        cpu.uartA0.writeData(k);
        while (!cpu.uartA0.isTXempty());
        while(!cpu.uartA1.isRXready());
        data[j]=cpu.uartA1.readData();
        j++;
        }
        for (int i =0 ; i< 16 ; i++)
        {
        printf("0x%02x\n",data[i]);
        }


        for (int n =0 ; n< 16 ; n++)
        {
        data[n]=0x00;
        }
        printf("A0:Recieve,A1:Transmit\n");
        j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartA1.isTXready());
        cpu.uartA1.writeData(k);
        while (!cpu.uartA1.isTXempty());
        while(!cpu.uartA0.isRXready());
        data[j]=cpu.uartA0.readData();
        j++;
        }
        cpu.uartA0.disable(modeA0);
        cpu.uartA1.disable(modeA1);
}

void Test_RS232A2A3(MSP_430::t_UART_MODES modeA2, MSP_430::t_UART_MODES modeA3) {
        cpu.uartA2.init();
        cpu.uartA3.init();

        if(modeA2==MSP_430::RS232_MODE)
        {
        cpu.uartA2.enable(modeA2,9600);
        }
        else if(modeA2==MSP_430::IRDA_MODE)
        {
          cpu.uartA2.enable(modeA2,9600,pwd);
        }
        if(modeA3==MSP_430::RS232_MODE)
        {
        cpu.uartA3.enable(modeA3,9600);
        }
        else if(modeA3==MSP_430::IRDA_MODE)
        {
         cpu.uartA3.enable(modeA3,9600,pwd);
        }

        cpu.uartA2.msbLast();
        cpu.uartA3.msbLast();
```

```
        printf("A2:Transmit,A3:Recieve\n");
        int j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartA2.isTXready());
        cpu.uartA2.writeData(k);
        while (!cpu.uartA2.isTXempty());
        while(!cpu.uartA3.isRXready());
        data[j]=cpu.uartA3.readData();
        j++;
        }
         for (int i =0 ; i< 16 ; i++)
        {
        printf("0x%02x\n",data[i]);
        }


        for (int n =0 ; n< 16 ; n++)
        {
        data[n]=0x00;
        }
        printf("A2:Recieve,A3:Transmit\n");
        j=0;
        for (byte k = 0x01; k <= 0x10 ; k++)
        {
        while (!cpu.uartA3.isTXready());
        cpu.uartA3.writeData(k);
        while (!cpu.uartA3.isTXempty());
        while(!cpu.uartA2.isRXready());
        data[j]=cpu.uartA2.readData();
        j++;
        }

        cpu.uartA2.disable(modeA2);
        cpu.uartA3.disable(modeA3);
}

void Test_I2CB0B1(MSP_430::t_UART_MODES modeB0, MSP_430::t_UART_MODES modeB1) {
        for (byte k = 0x00; k < 0x10 ; k++)
        {
        MasterDataTransmited[k]=k;
        SlaveDataTransmited[k]= k + 0x10;
        }

        cpu.uartB1.init(modeB1);
        cpu.uartB1.enable(modeB1);
        cpu.uartB1.set_I2C_address(0x60,0x51);

        cpu.uartB0.init(modeB0);
        cpu.uartB0.enable(modeB0);
        cpu.uartB0.set_I2C_address(0x51,0x60);
        __enable_interrupt();
        /*************************************************************/
        while (cpu.uartB0.is_I2C_busy());          // wait for bus to be free
        //  0: I2C bus is idle,
        //  1: communication is in progress
        /*************************************************************/
        // start transmitting
        byteTX = 15 ;
        byteRX = 15 ;
```

```
        cpu.uartB0.enableInterrupts(false,true);
        cpu.uartB1.enableInterrupts(true,false);
        cpu.uartB1.start_I2C(true);           // I2C start condition
        cpu.uartB0.start_I2C(false);           // I2C start condition
        /*************************************************************/
        while (cpu.uartB0.is_I2C_busy());          // wait for bus to be free

        /*************************************************************/
        // start transmitting
        byteTX = 15 ;
        byteRX = 15 ;

        cpu.uartB0.enableInterrupts(true,false);
        cpu.uartB1.enableInterrupts(false,true);
        cpu.uartB0.start_I2C(true);
        /*************************************************************/
        while (cpu.uartB0.is_I2C_busy());          // wait for bus to be free

        __disable_interrupt();
        cpu.uartB0.disable(modeB0);
        cpu.uartB1.disable(modeB1);
}

void Test_I2CB2B3(MSP_430::t_UART_MODES modeB2, MSP_430::t_UART_MODES modeB3) {
        for (byte k = 0x00; k < 0x10 ; k++)
        {
        MasterDataTransmited[k]=k;
        SlaveDataTransmited[k]= k + 0x10;
        }
        cpu.uartB3.init(modeB3);
        cpu.uartB3.enable(modeB3);
        cpu.uartB3.set_I2C_address(0x50,0x51);

        cpu.uartB2.init(modeB2);
        cpu.uartB2.enable(modeB2);
        cpu.uartB2.set_I2C_address(0x51,0x50);
        /*************************************************************/
        __enable_interrupt();
        /*************************************************************/
        while (cpu.uartB2.is_I2C_busy());          // wait for bus to be free
                              //  0: I2C bus is idle,
                             //  1: communication is in progress
        /*************************************************************/
        // start transmitting
        byteTX = 15 ;
        byteRX = 15 ;
        cpu.uartB2.enableInterrupts(false,true);
        cpu.uartB3.enableInterrupts(true,false);
        cpu.uartB2.start_I2C(false);           // I2C start condition
        cpu.uartB3.start_I2C(true);           // I2C start condition
        /*************************************************************/
        while (cpu.uartB2.is_I2C_busy());          // wait for bus to be free

        /*************************************************************/
        // start transmitting
        byteTX = 15 ;
        byteRX = 15 ;

        cpu.uartB2.enableInterrupts(true,false);
        cpu.uartB3.enableInterrupts(false,true);
        cpu.uartB2.start_I2C(true);
        /*************************************************************/
```

```
        while (cpu.uartB2.is_I2C_busy());          // wait for bus to be free

        __disable_interrupt();
        cpu.uartB2.disable(modeB2);
        cpu.uartB3.disable(modeB3);
}

void main() {
        cpu.wdt.disable();

        for (int n =0 ; n< 16 ; n++)
        {
        data[n]=0x00;
        }

        int x=9;
        //scanf("%d",&x);
        switch(x)
        {
        case 1:
        mode1=MSP_430::SPI_MASTER_MODE;
        mode2=MSP_430::SPI_SLAVE_MODE;
        Test_spiA0A1(mode1,mode2);
        break;
        case 2 :
        mode1=MSP_430::SPI_SLAVE_MODE;
        mode2=MSP_430::SPI_MASTER_MODE;
        Test_spiA0A1(mode1,mode2);
        break;
        case 3:
        mode1=MSP_430::SPI_MASTER_MODE;
        mode2=MSP_430::SPI_SLAVE_MODE;
        Test_spiA2A3(mode1,mode2);
        break;
        case 4:
        mode1=MSP_430::SPI_SLAVE_MODE;
        mode2=MSP_430::SPI_MASTER_MODE;
        Test_spiA2A3(mode1,mode2);
        break;
        case 5 :
        mode1=MSP_430::RS232_MODE;
        mode2=MSP_430::RS232_MODE;
        Test_RS232A0A1(mode1,mode2);
        break;
        case 6 :
        mode1=MSP_430::RS232_MODE;
        mode2=MSP_430::RS232_MODE;
        Test_RS232A2A3(mode1,mode2);
        break;
        case 7:
        mode1=MSP_430::IRDA_MODE;
        mode2=MSP_430::IRDA_MODE;
        Test_RS232A0A1(mode1,mode2);
        break;
        case 8:
        mode1=MSP_430::IRDA_MODE;
        mode2=MSP_430::IRDA_MODE;
        Test_RS232A2A3(mode1,mode2);
        break;
        case 9:
        mode1=MSP_430::I2C_MASTER_MODE;
        mode2=MSP_430::I2C_SLAVE_MODE;
```

```
        Test_I2CB0B1(mode1,mode2);
        break;
        case 10:
        mode1=MSP_430::I2C_SLAVE_MODE;
        mode2=MSP_430::I2C_MASTER_MODE;
        Test_I2CB0B1(mode1,mode2);
        break;
        defaut:
        break;
        }
        if(mode1==MSP_430::I2C_SLAVE_MODE || mode2==MSP_430::I2C_SLAVE_MODE || mode1==MSP_430::I2C_MASTER_M
        {
        printf("Master receive data from Slave \n");
        for(int i=0; i<16;i++)
        {
        printf("0x%02x\n",MasterDataReceived[i]);
        }
        printf("\n");
        printf("Slave receive data from Master\n");
        for(int i=0; i<16;i++)
        {
        printf("0x%02x\n",SlaveDataReceived[i]);
        }
        }
        else
        {
        for (int i =0 ; i< 16 ; i++)
        {
        printf("0x%02x\n",data[i]);
        }
        }
}


#pragma vector = USCI_B0_VECTOR
__interrupt void USCIB0I2C_RXTX_ISR() {
        if (UCB0IFG & UCRXIFG)
        {
        if ( byteRX == 0 )
        {
        cpu.uartB0.stop_I2C();
        MasterDataReceived[byteRX] = cpu.uartB0.readData();
        }
        else
         {
        MasterDataReceived[byteRX] = cpu.uartB0.readData();
        byteRX--;
        }
        }
        else if (UCB0IFG & UCTXIFG)
        {
        if (byteTX == 0){
        cpu.uartB0.stop_I2C();
        UCB0IFG &= ~UCTXIFG;                        // Clear USCI_B0 TX int flag
        }
        else
        {
        cpu.uartB0.writeData(MasterDataTransmited[byteTX]);
        byteTX--;
        }
        }
}
```

```
#pragma vector = USCI_B1_VECTOR
__interrupt void USCIB1I2C_RXTX_ISR() {
        if (UCB1IFG & UCRXIFG)
        {
        if ( byteRX == 0 )
        {
        SlaveDataReceived[byteRX] = cpu.uartB1.readData();
        }
        else
        {
        SlaveDataReceived[byteRX]= cpu.uartB1.readData();
        byteRX--;
        }
        }
        else if (UCB1IFG & UCTXIFG)
        {
        if (byteTX == 0)
        {
        cpu.uartB1.writeData(SlaveDataTransmited[byteTX]);
        }
        else
        {
        cpu.uartB1.writeData(SlaveDataTransmited[byteTX]);
        byteTX--;
        }
        }
}


#pragma vector = USCI_B2_VECTOR
__interrupt void USCIB2I2C_RXTX_ISR() {
        if (UCB2IFG & UCRXIFG)
        {
        if ( byteRX == 0 )
        {
        cpu.uartB2.stop_I2C();
        MasterDataReceived[byteRX] = cpu.uartB2.readData();
        }
        else
         {
        MasterDataReceived[byteRX] = cpu.uartB2.readData();
        byteRX--;
        }
        }
        else if (UCB2IFG & UCTXIFG)
        {
        if (byteTX == 0){
        cpu.uartB2.stop_I2C();
        UCB2IFG &= ~UCTXIFG;                        // Clear USCI_B2 TX int flag
        }
        else
        {
        cpu.uartB2.writeData(MasterDataTransmited[byteTX]);
        byteTX--;
        }
        }
}


#pragma vector = USCI_B3_VECTOR
```

```
__interrupt void USCIB3I2C_RXTX_ISR() {
        if (UCB3IFG & UCRXIFG)
        {
        if ( byteRX == 0 )
        {
        SlaveDataReceived[byteRX] = cpu.uartB3.readData();
        }
        else
        {
        SlaveDataReceived[byteRX]= cpu.uartB3.readData();
        byteRX--;
        }
        }
        else if (UCB3IFG & UCTXIFG)
        {
        if (byteTX == 0)
        {
        cpu.uartB3.writeData(SlaveDataTransmited[byteTX]);
        }
        else
        {
        cpu.uartB3.writeData(SlaveDataTransmited[byteTX]);
        byteTX--;
        }
        }
}
```