Programming Notes MSP430F2618

# Core needed:

Clock

GIE

Timer

# Peripherals needed:

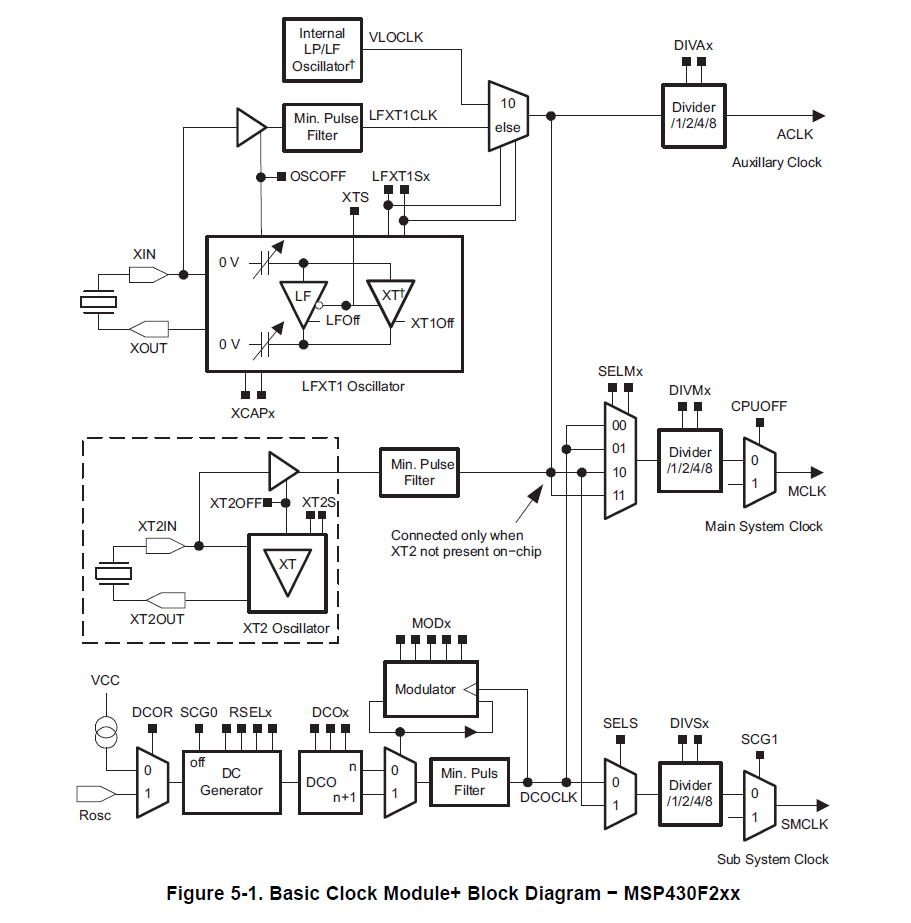
ADC

GPIO Interrupts

I2C

SPI

# Clock



2, 3 or 4 Clock sources available:

LFT1CLK: external low-/high-frequency oscillator. Can be used with 32768 crystals or standard crystals, resonators or external clock sources. Low frequency or 400kHz – 16Mhz.

XT2CLK: Optional. High frequency. Can be used with standard crystals, resonators or external clock sources. 400kHz – 16MHz.

DCOCLK: Internal Digitally Controlled Oscillator

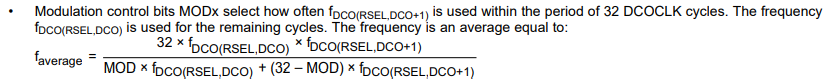
VLOCLK: Internal very low frequency oscillator. 12kHz

3 clock Signals available:

ACLK: Auxiliary clock. Software selectable as LFT1CLK, VLOCLK. Divided by 1, 2, 4 or 8

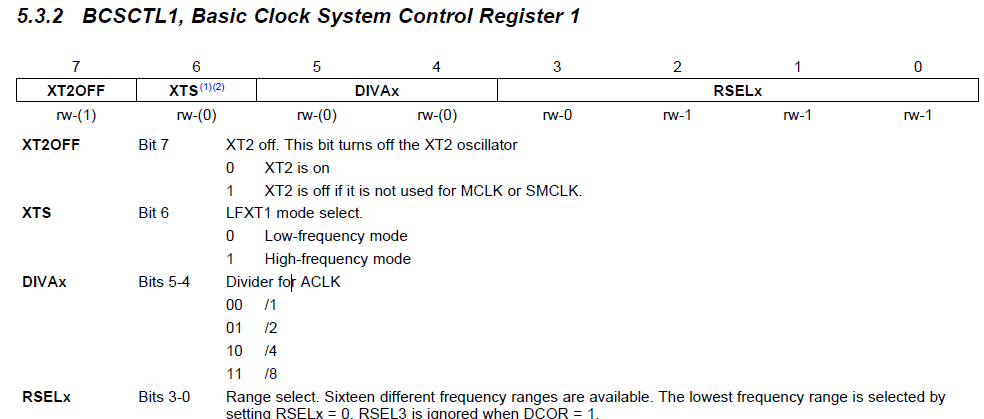
MCLK: Master clock. Software selectable as any. Divided by 1 ,2 ,4 or 8

SMCLK: Sub main clock. Software selectable as any. Divided by 1, 2, 4 or 8. Software selectable for individual peripheral modules.

MODx is used to fine adjust the frequency produced by the DCO. 

Set Main clock and Submain clock to DCO disabling LFTX1CLK and XT2CLK because no external Crystals or Resonators are available:

Basic Clock System Control Register 1 (BCSCTL1):

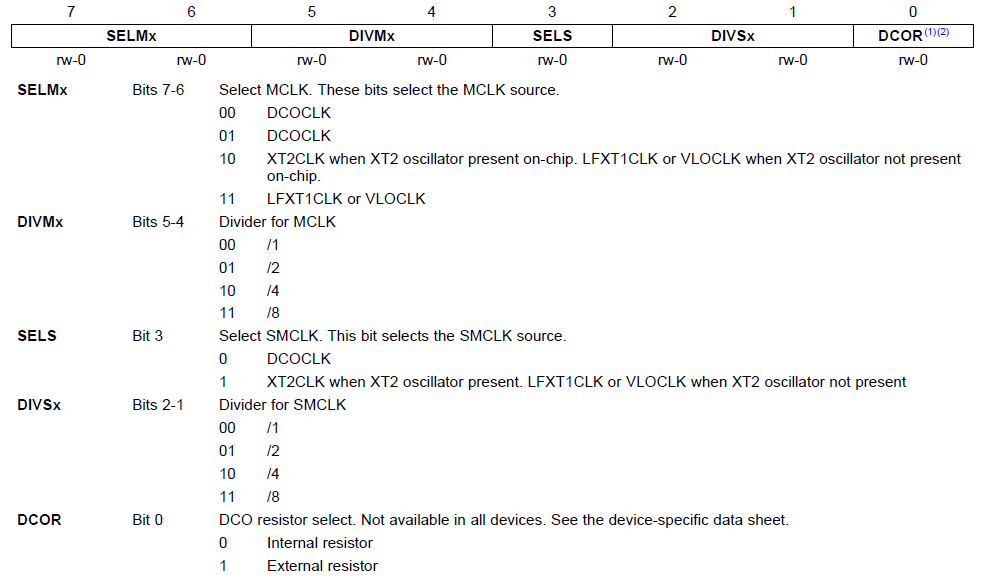


XT2OFF = 1

XTS = 0

RSELx -> See DCO configure later.

Basic Clock System Control Register 2 (BCSCTL2):

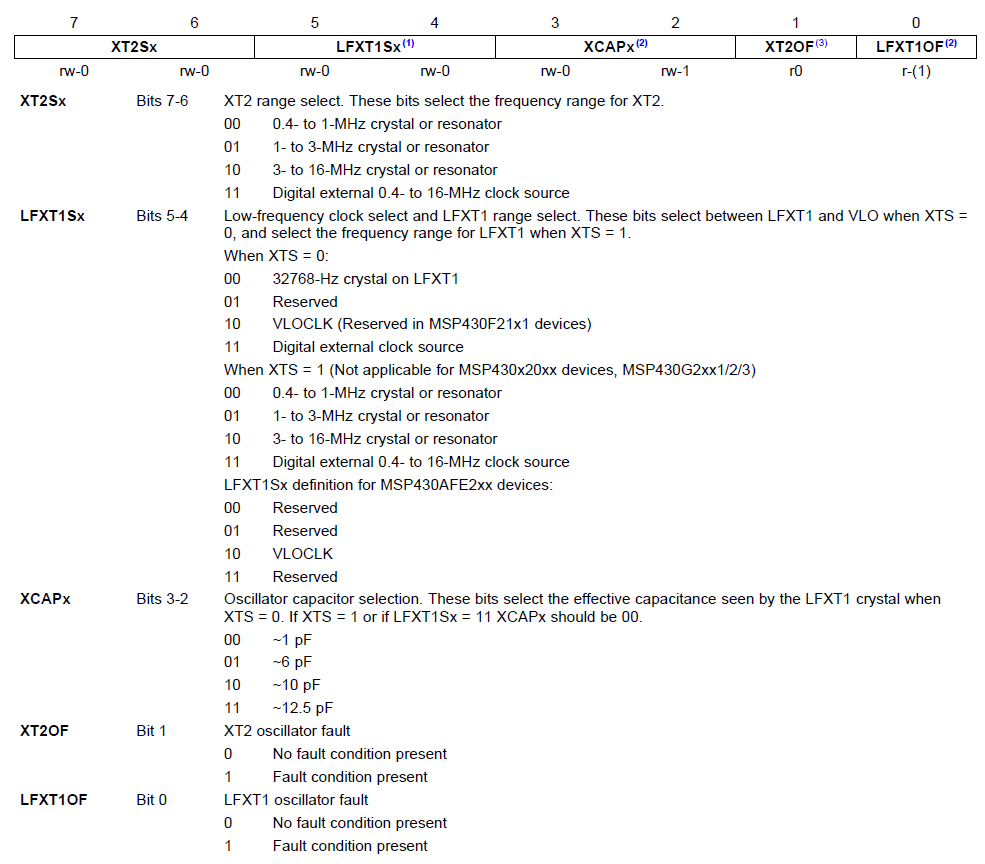


SELMx = 00

SELS = 0.

DCOR = 0

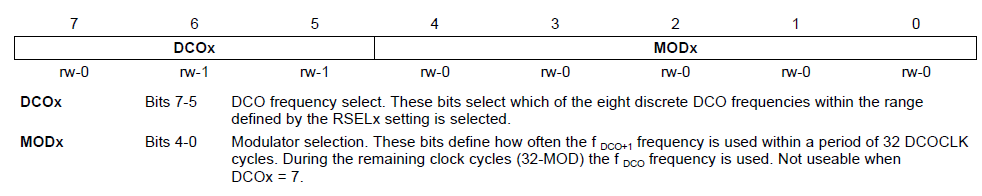
Basic Clock System Control Register 2 (BCSCTL2):



LFXT1Sx = 10

Programming the clock speed for the DCO

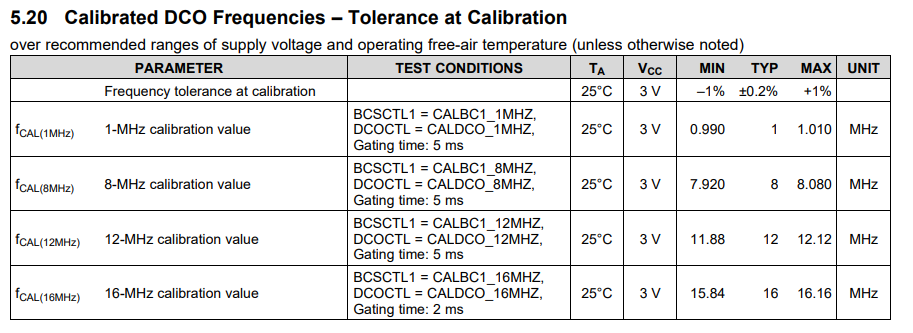
DCO Control Register (DCOCTRL):



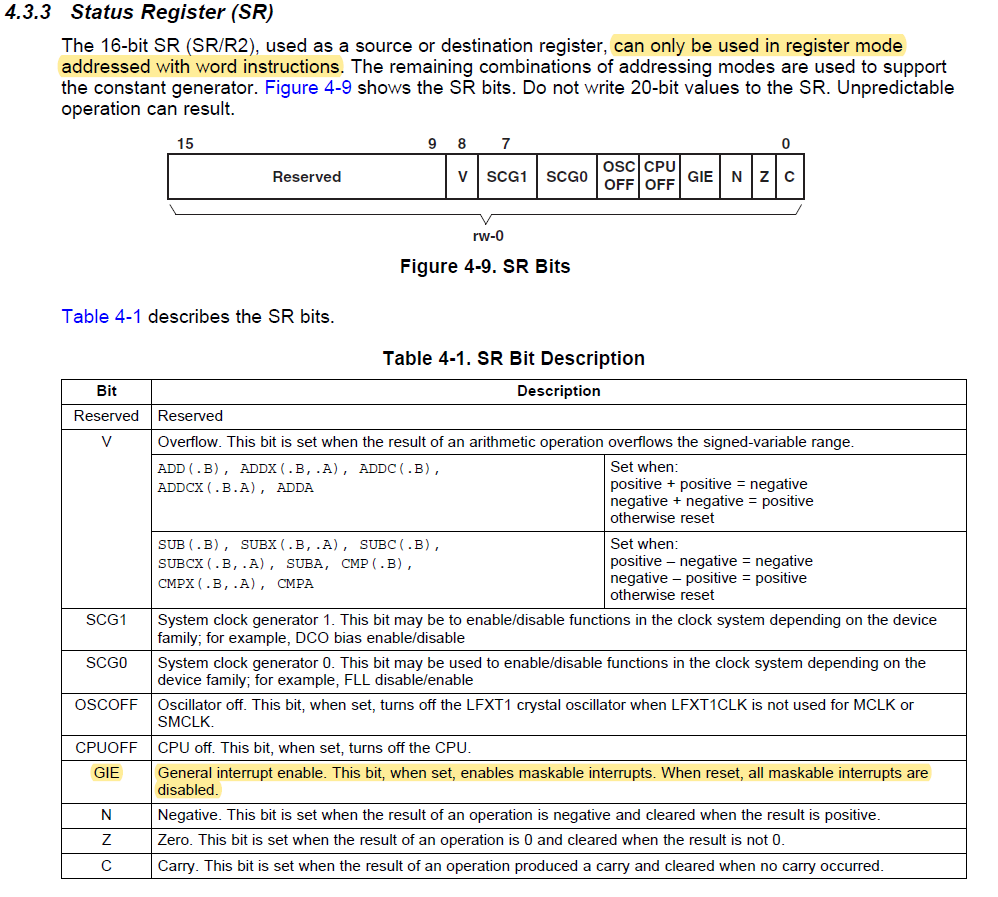
DCO = 111

Also set RSLx to 1111

The DCO can also be calibrated by a pre-generated define



# GIE



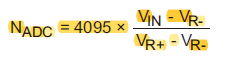
# ADC

The ADC core operates with an upper and a lower reference Voltage that can be defined by user.

If the Input Signal is equal or higher than the upper Voltage, the digital value is Full Scale (0x0FFF)

If the Input Signal is below the lower Voltage the digital value is low (0x0000)

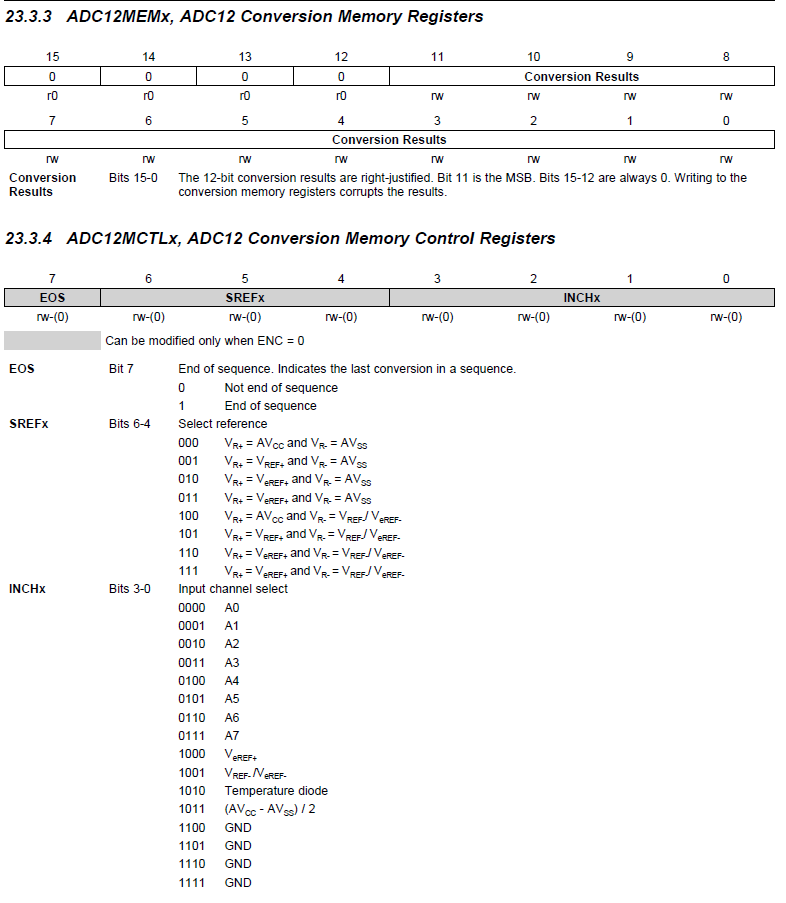
In between the Digital Value equals the calculated value NADC:

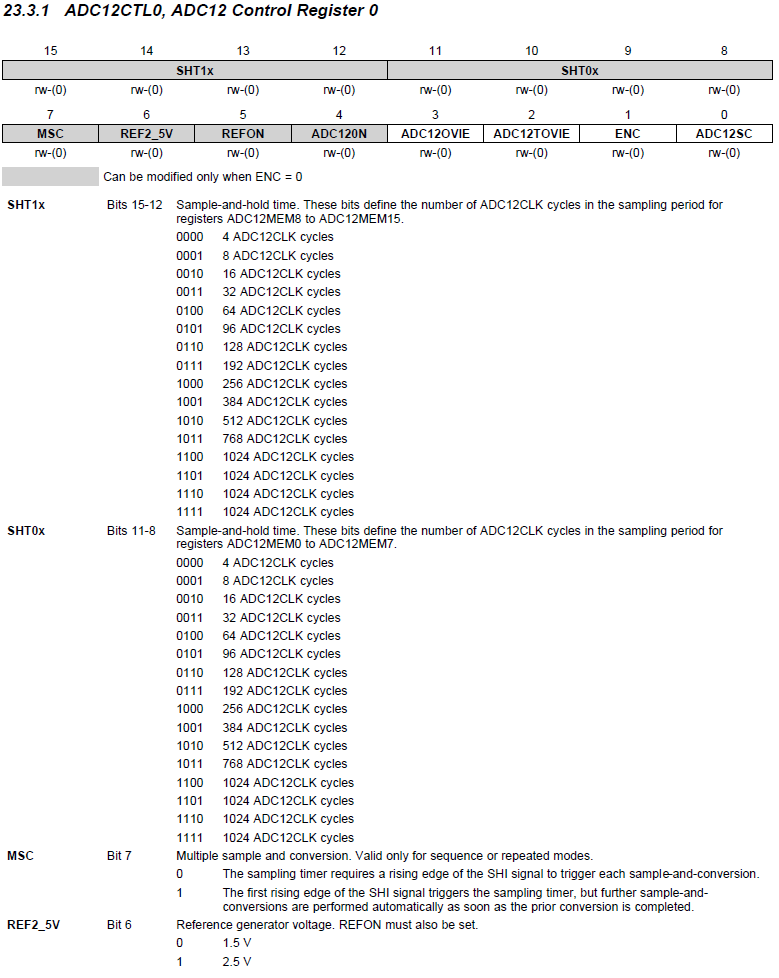


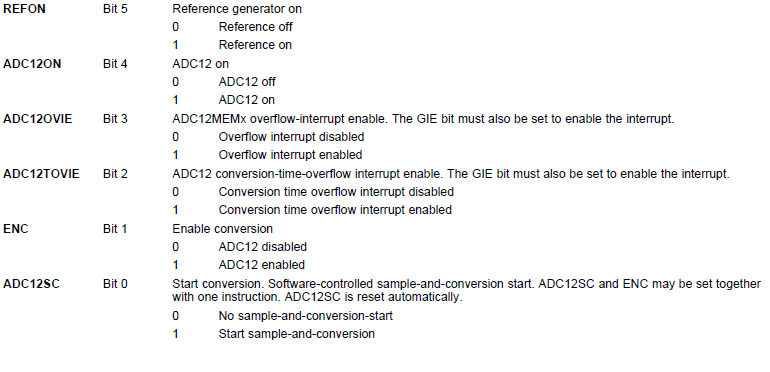
The ADC core is configured by only two registers: ADC12CTL0 and ADC12CTL1

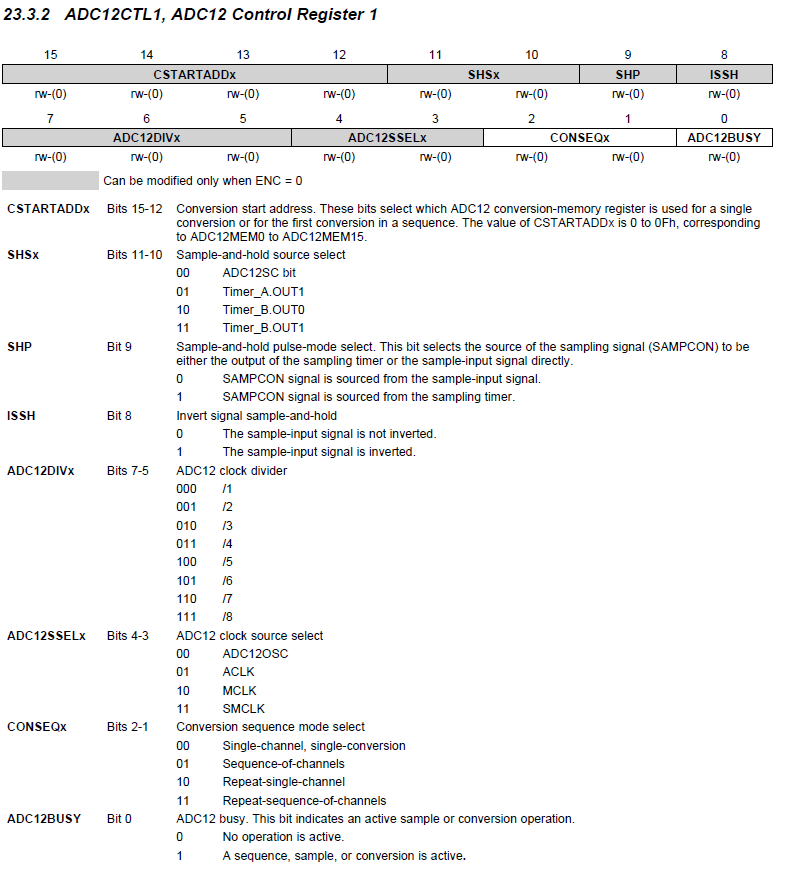
Its memory is stored in 16 registers therefore it can store up to 16-words of translation without any CPU intervention. The memory, that is to be saved, is controlled by another 16 Memory control registers. These decide what input channel is to be used and on what references the conversion has to be done.

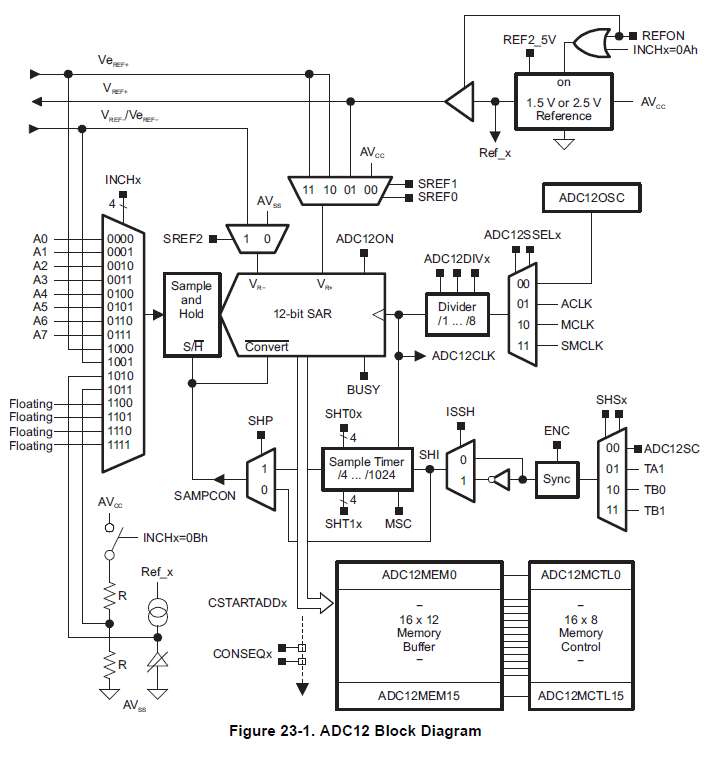
## Memory Register



Control Register





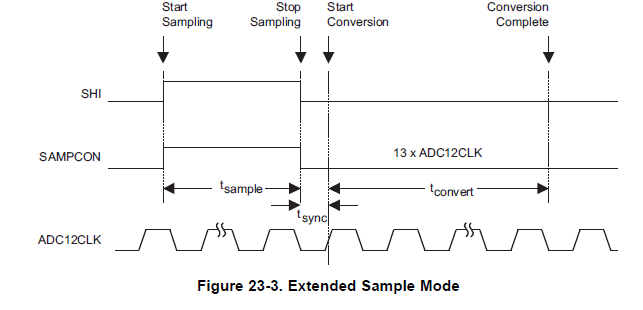


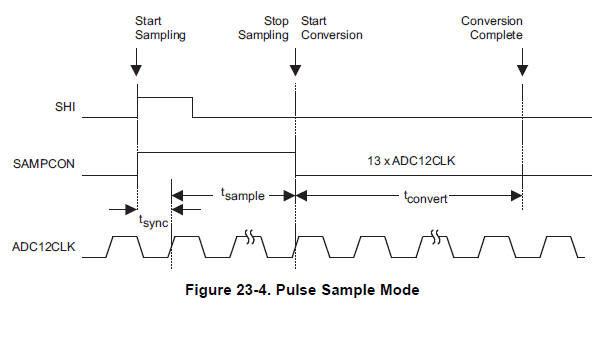
## Register explanations

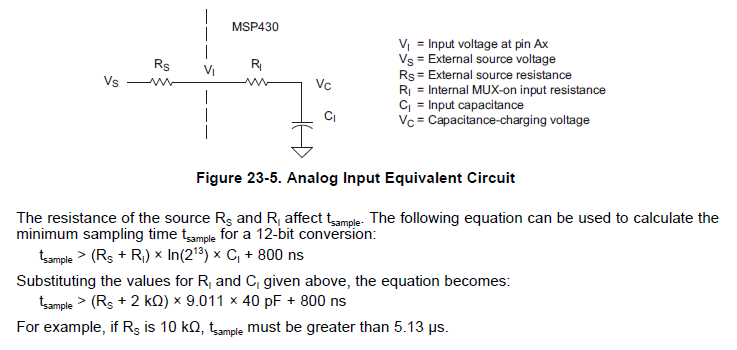
### SHP

This Bit decides between Extended Sample Mode and Pulse Sample Mode. In the Expanded Sample Mode, the conversion takes place as long as SHI is high. In the Pulse Sample Mode a pulse on SHI will trigger a Timer that will hold the sample go signal SAMPCON high as long as by the user defined in the SHT0 and SHT1 register. These two register will hold SAMPCON high for 4-1024 times the ADC12CLK. The 4.-Bit-SHT0 register controls sampling time for the ADC12MCTL 0-7 register and the 4-Bit-STH1 register for the ADC12MCTL 8-15 register.

SAMPCON = Sample Conversion

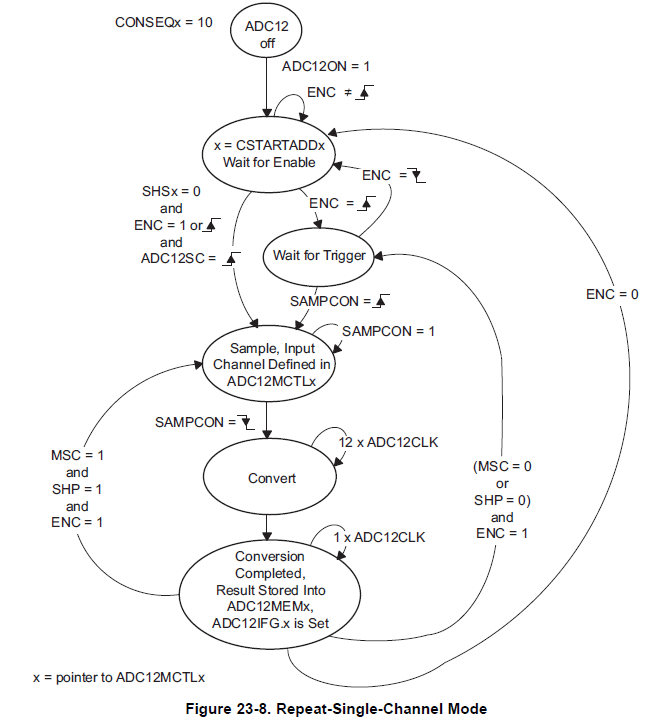






### Programming

For my use, the ADC will be initialized in the startup of the device and then will be triggered all 100m seconds by a Timer. When the conversion is complete, an interrupt will occur and the Controller can read the value and dim or brighten the LEDs.



ADC12CTL0

SHT0 = 256 ADC12CLK cycles (0x8) = 16us

ADC12ON = 1

ENC = 1

ADC12CTL1

SHSx = TIMER\_A (01)

SHP = 1

ADC12SSEL = MCLK (10)

CONSEQ = 10

ADC12MCTL0

Standard values

ADC12IE

ADC12IE0 = 1

## GPIO Interrupts

For enabling Interrupt capability the following Registers must be set according to implementation.

Interrupt Edge Select 🡪 PxIES every Bit in this Register stands for its Bit on the Port.

Bit = 0 🡪 PxIFGx is set on high to low transition

Bit = 1 🡪 PxIFGx is set on low to high transition

The interrupt gets only activated when there's a state change.

Interrupt Flag Register 🡪 PxIFG this needs to be set to 0 and in every ISR has to be reset by software

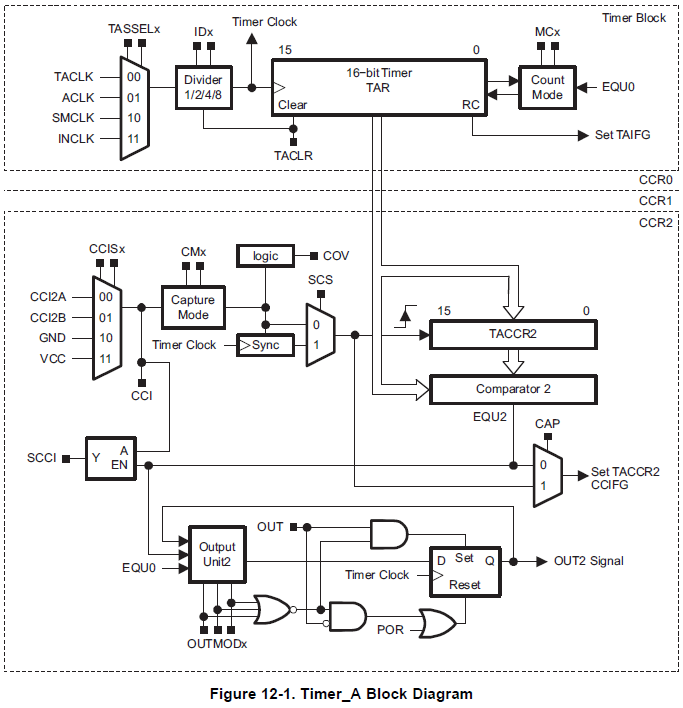
Interrupt Enable 🡪 PxIE Enables the interrupts according to Bits set in the Register.

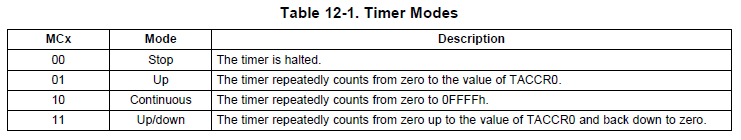
The whole system has also to be setup to enable interrupts: this can be done with the function call:

\_enable\_interrupt();

Or by enabling the GIE bit in the Status Register (SR).

## Timer

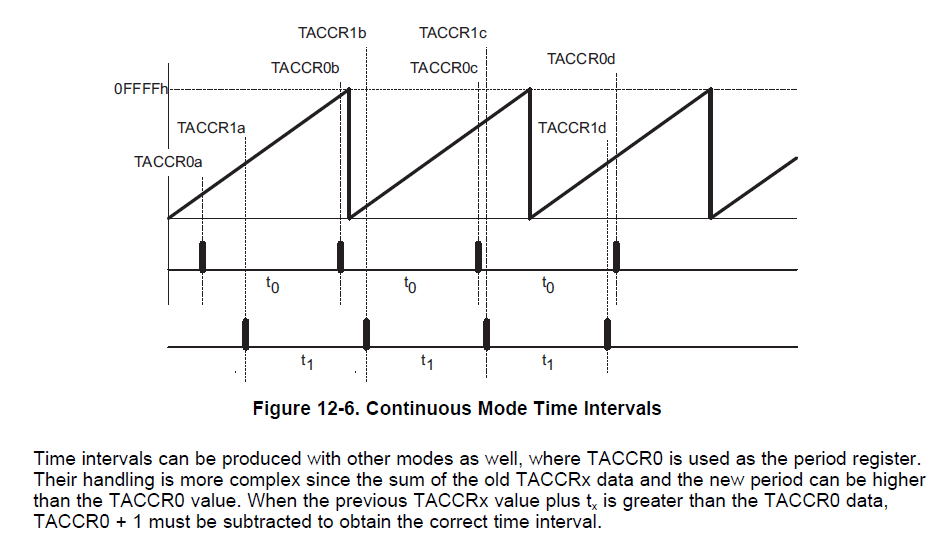


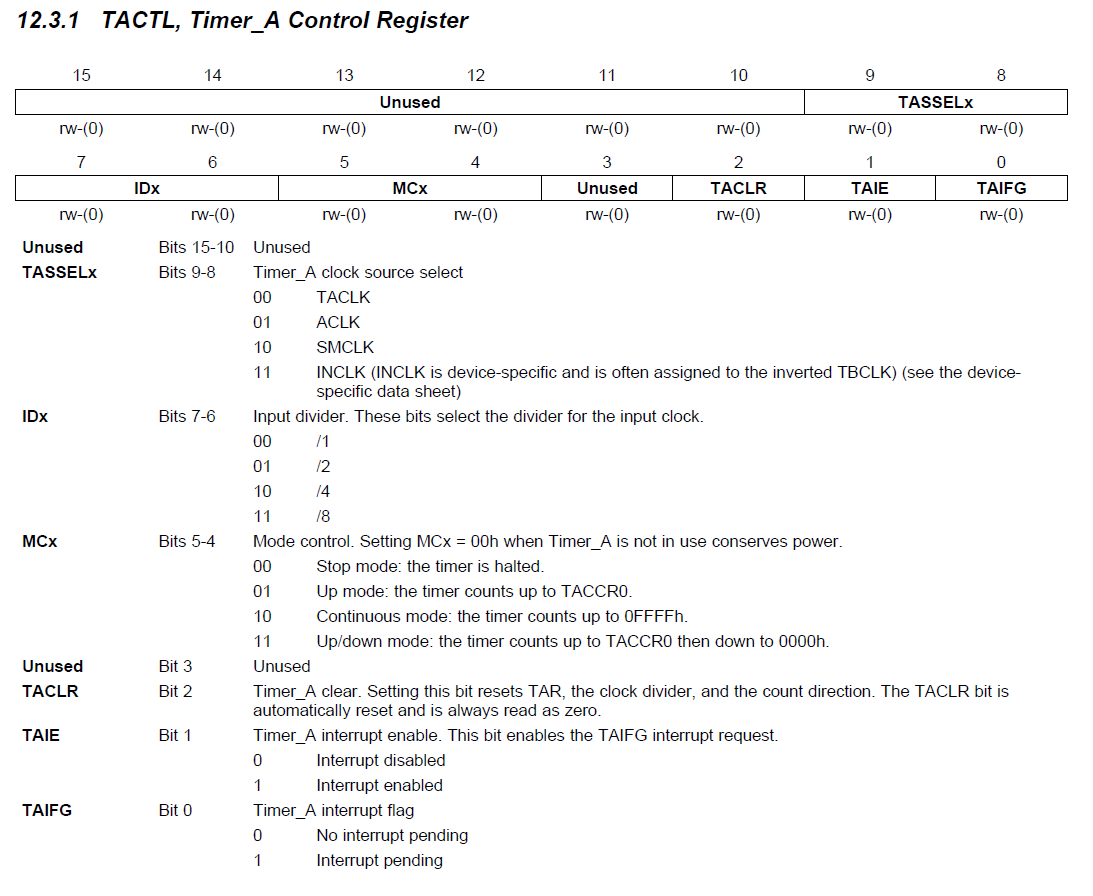
Timer\_A has one Timer Block and Three identical Capture/Compare Blocks. The Timer can be programmed to have one of the 4 Clock inputs shown above. The MCx defines in what Mode the Controller counts: 

The Capture\Control Blocks have one Timer A Capture Control register (TACCRx) in which a value can be stored. The reach of this Value triggers the Timer-intern Interrupt EQUx. The TACCR0 is used for secondary Output changes, so they can affect the whole Timer. It can either set or reset the output, affect the Timer to reset to 0 in the Up Mode or affect the Timer to change counting direction in the Up/Down mode.

In my use I need a Interval Signal that generates an Impuls on the TIMER\_A Out 1 every 100ms.

16MHz 🡪 T = 625ns for 100ms that would be 1'600'000 cycles but the 16 Bit counter can only count untill 65'536. Even when divided by 8 it wouldn't be low enough so the Timer must be sourced by ACKL with it's 12kHz sourced from VLOCLK. 12kHz 🡪 83.33us for 100ms that woud be roughly 1'200 counts. That means I could take the continous Mode and set the TACCR1 to 0x4B0 and as soon as the value is reached I can increase the Value in the TACCR1 register by 0x4B0.

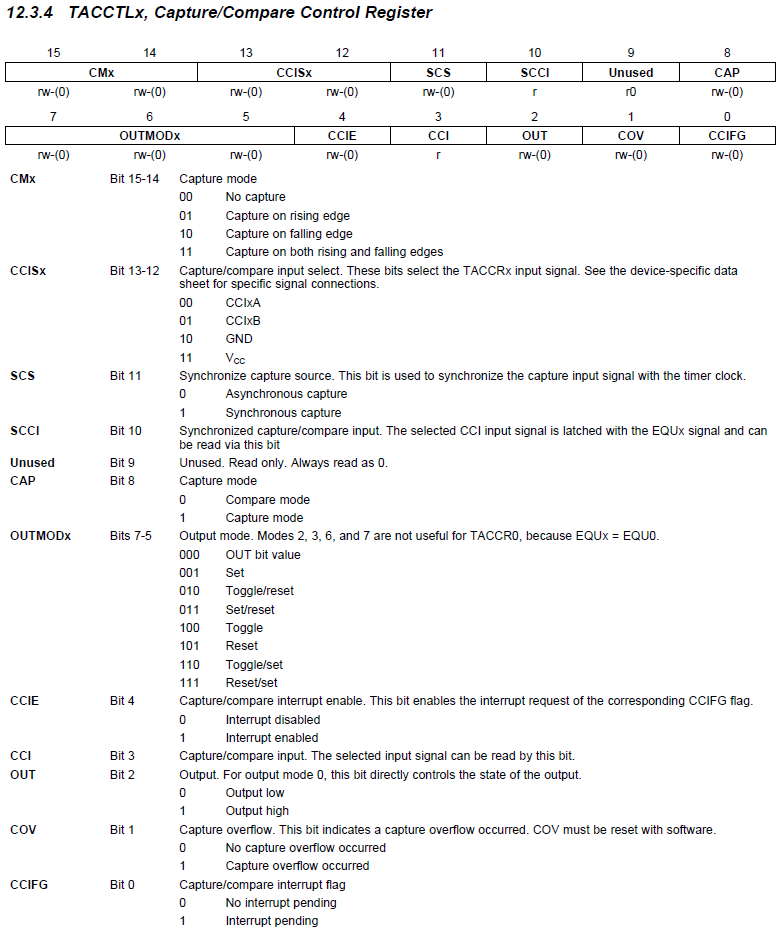




TASSELx = ACKL = 1

IDx = 00

MCx = Continuous = 2



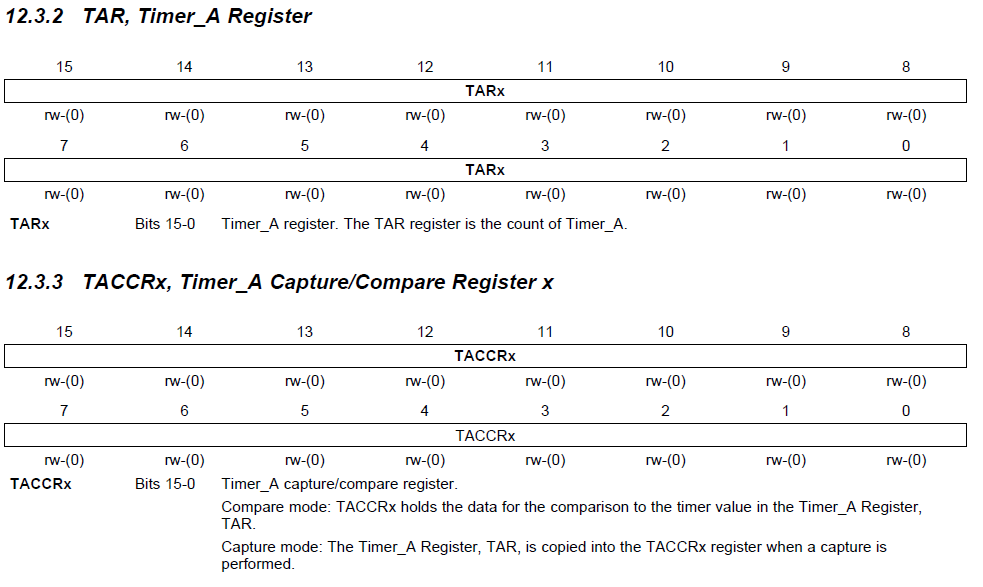
TACCTL1

CM = No capture = 0

CAP = Compare Mode = 0

OUMODx = Set/Reset = 3

CCIE = interrupt enable = 1

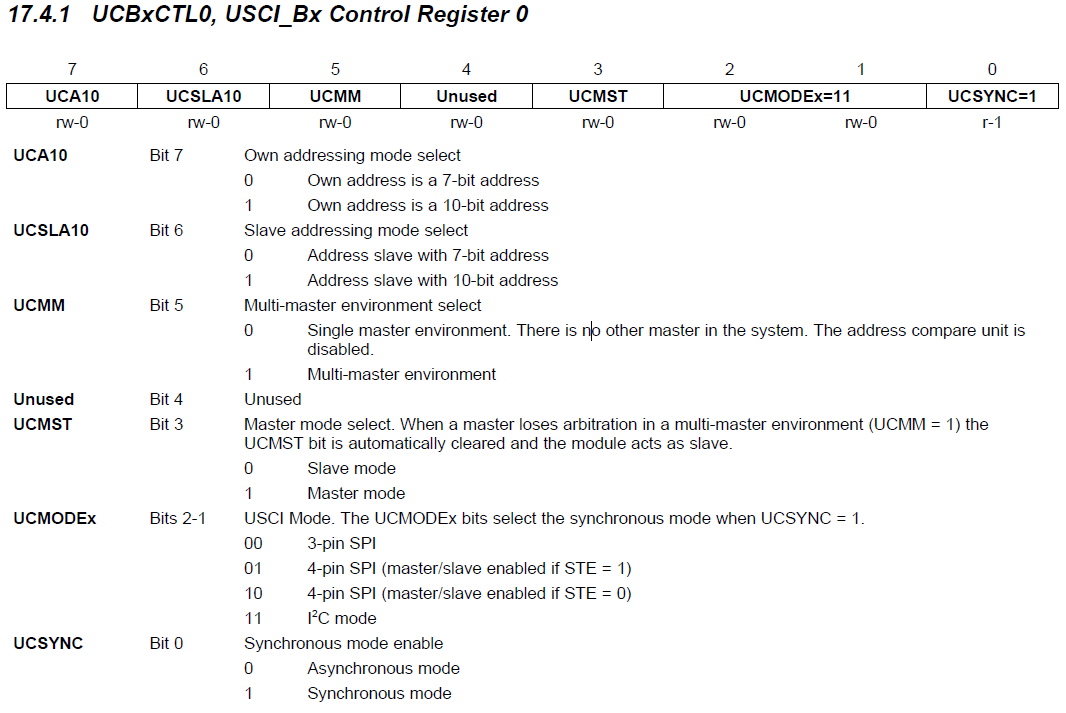


TACCR1 = 0x4B0 (gets increased by this value in each interrupt handling)

TACCR0 = 0x708 (gets increased by 0x4B0 in each interrupt Handling)

# I2C

The I2C is used for a 7Bit-Adressed device. The USCIB\_0 shall be used since it supports IC2 and USCIA\_1 and USCIB\_1 share an interrupt vector. Therefore if ASCIA\_1 is later used for SPI it would they would collide. Since the interrupt enable register (IE2) and the interrupt flag register (IFG2) are in the Special Function Register the bits should rather be set or cleared by the BIS.B or BIC.B instruction than the MOV.B or CLR.B instruction in order not to affect the other Bits.

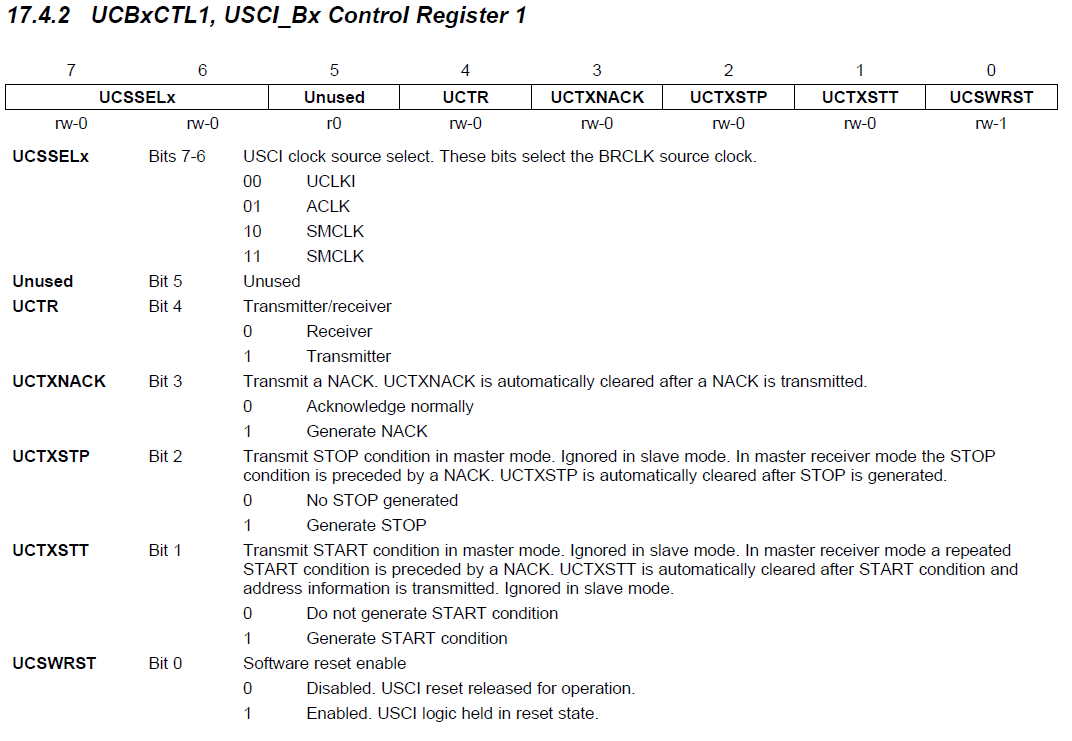


UCSLA10, slave addressing mode = 7-Bit = 0

UCMST = Master mode = 1

UCMODEx = I2C mode = 11b

UCSYNC = Synchronous =1



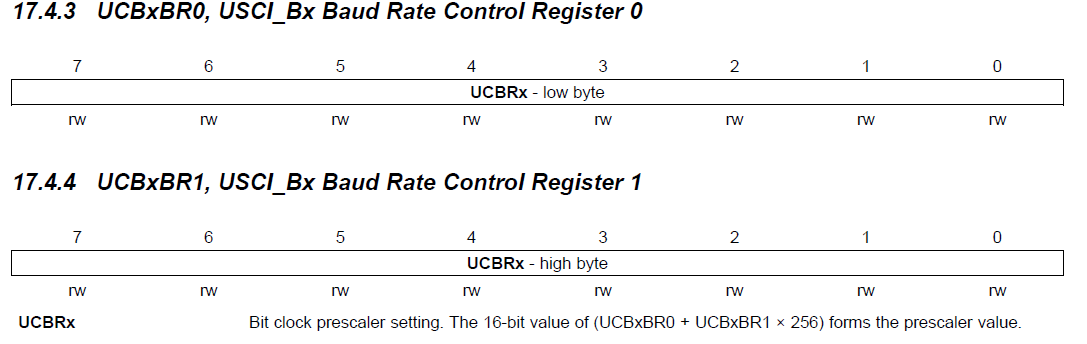
UCSSELx = SMCLK = 10b or 11b

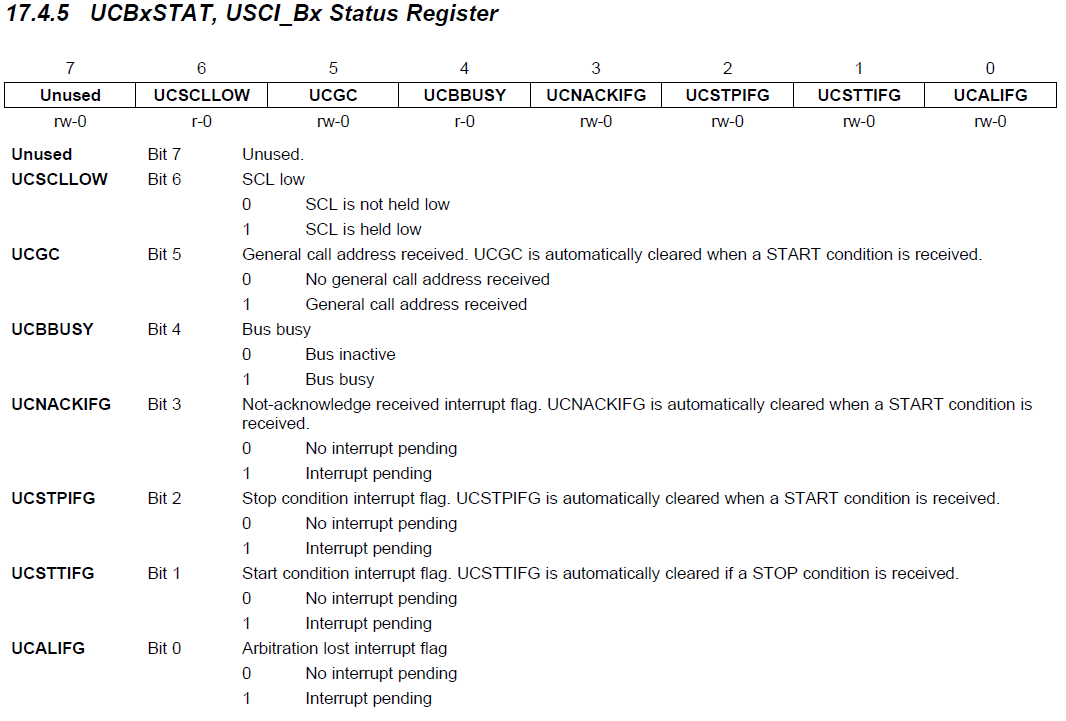
UCTR is to be set in operation when needed

UCTXSTP is to be set in operation when needed

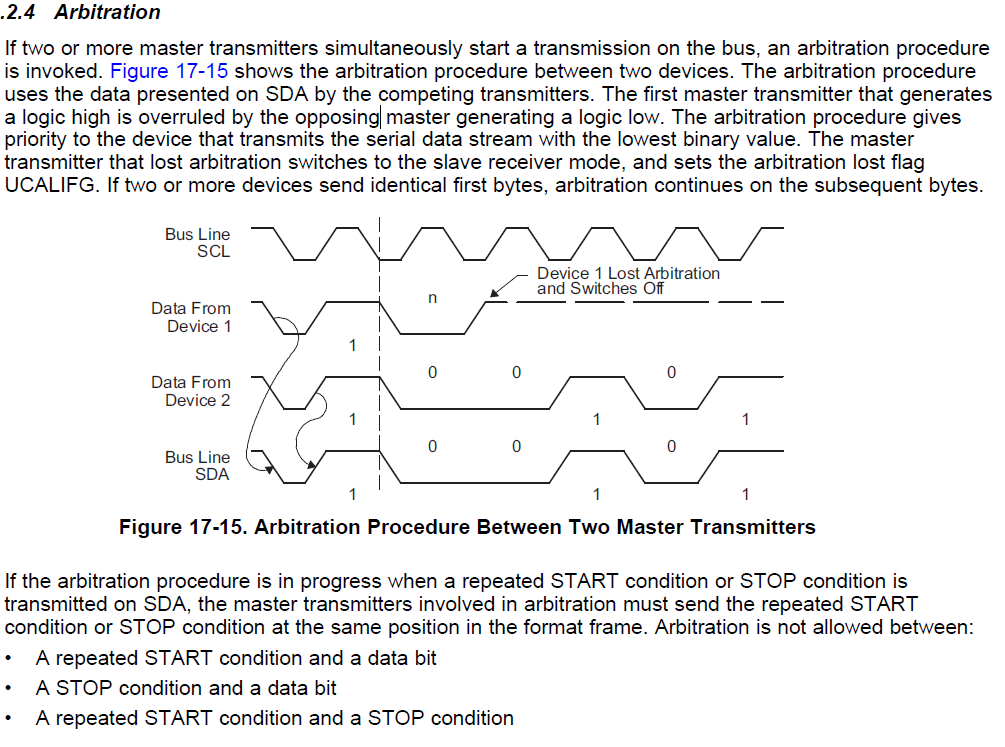
UCTXSTT is to be set in operation when needed

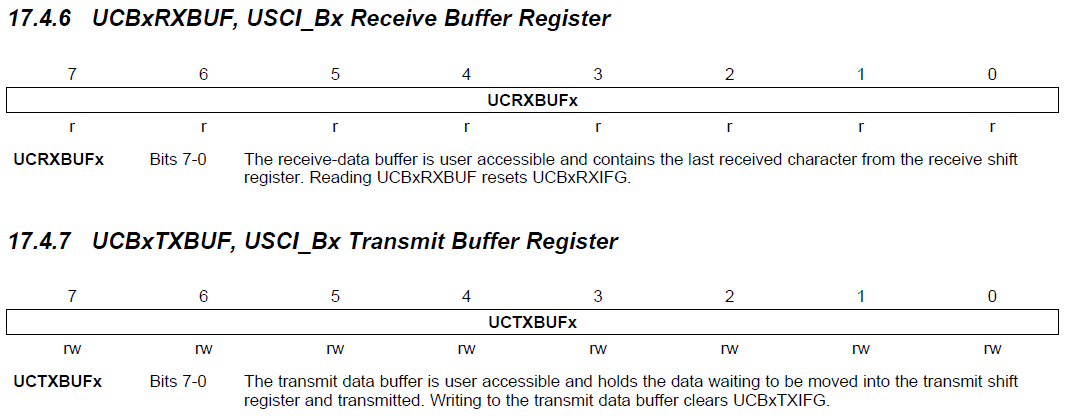
UCSWRST is to be cleared on beginning to enable the module and then set in operation when needed

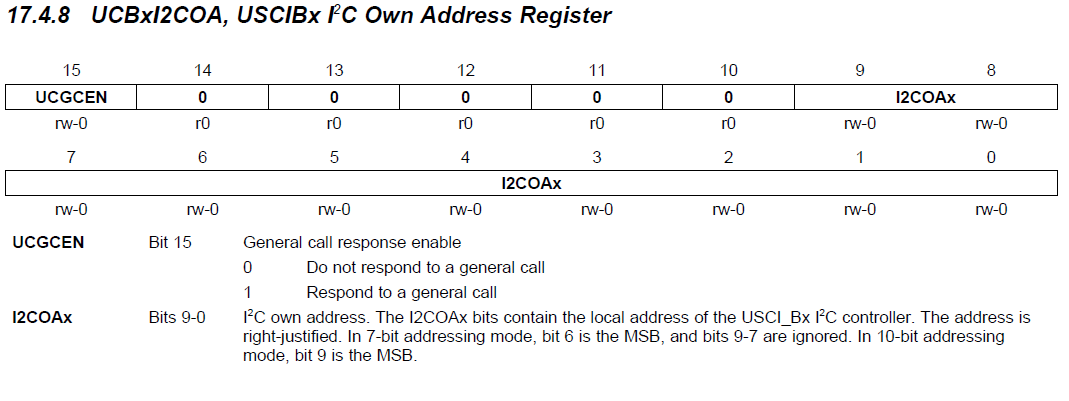




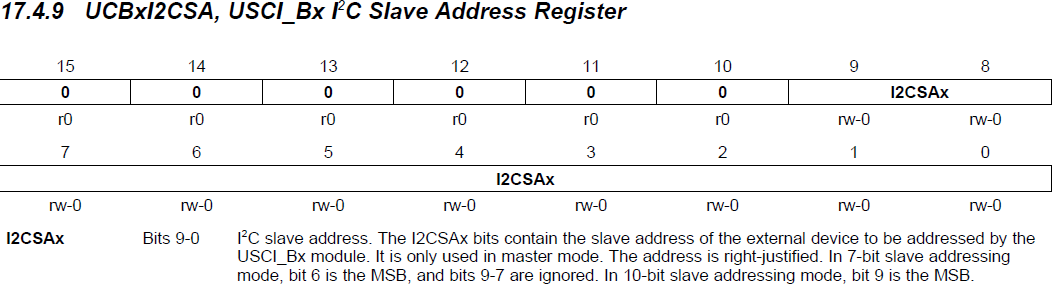
Use this register to control communication flow. The Arbitration los interrupt flags indicates that there was a lost of information transmitting on the Bus due to two Master strating to transmit at the same time.



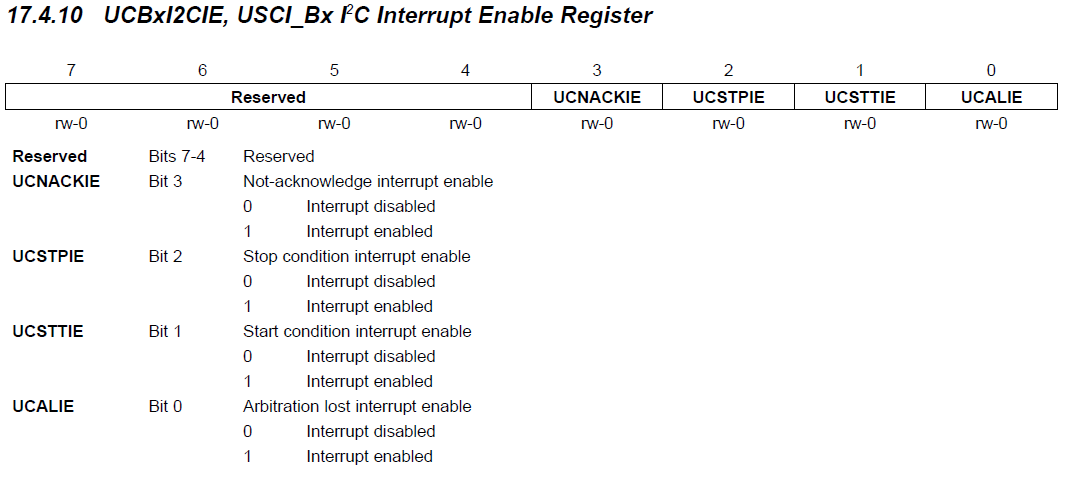




We can ignore this register because we are not working in slave mode on our device.

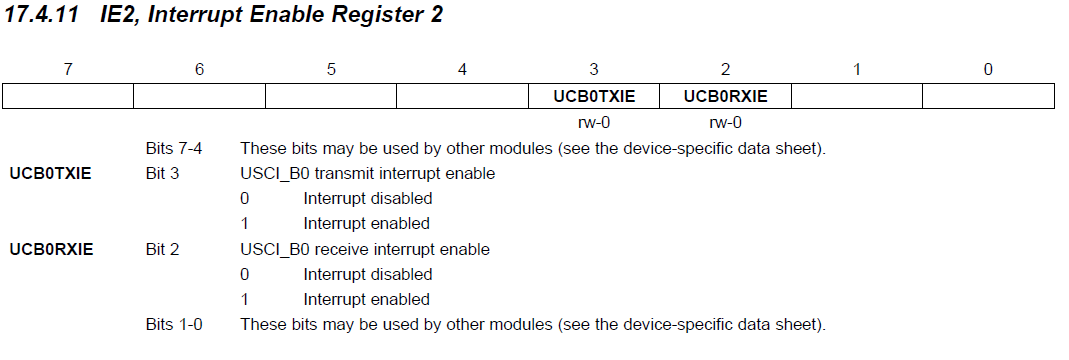


UCB0I2CSAx = our device address = 0011100b = 0x1C

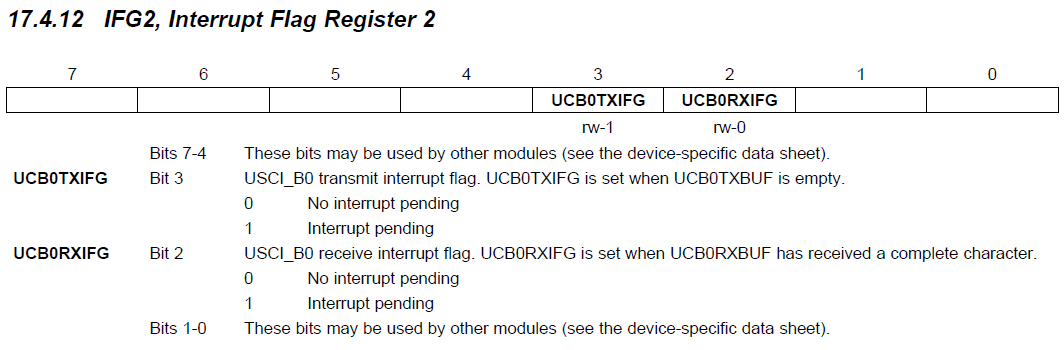


We only need the NOTACK IFG for dealing with communication errors.

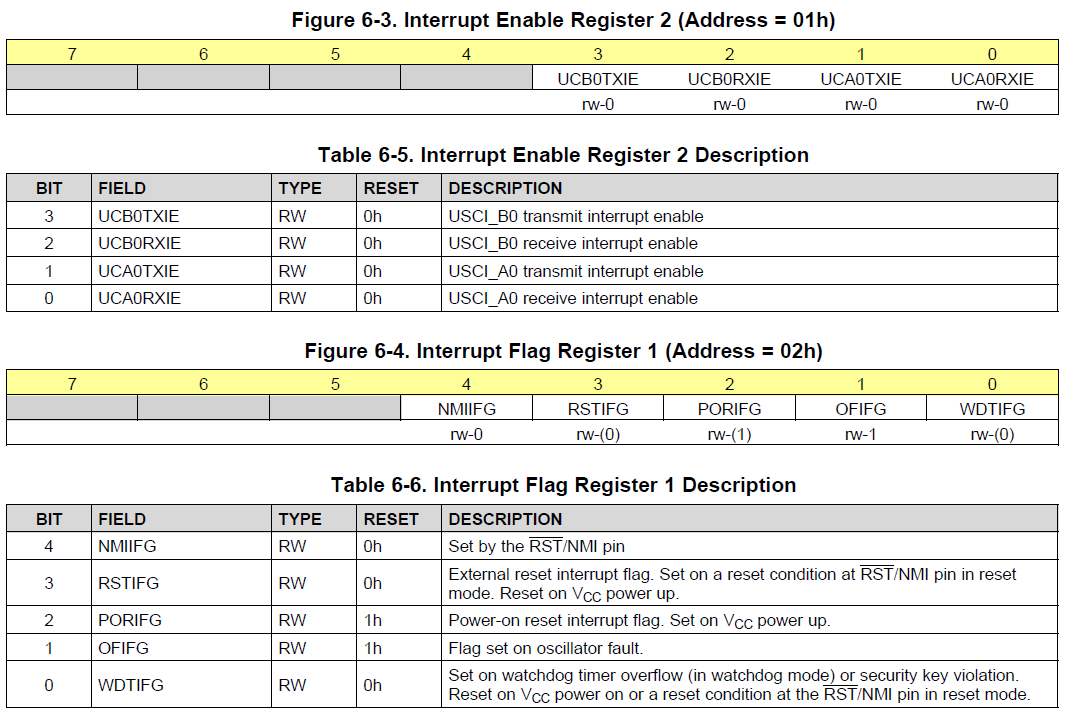
UCNACKIE = 1



To begin with enable the receive interrupt



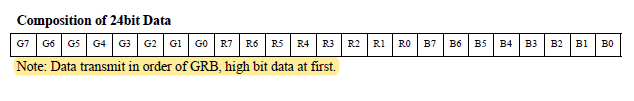
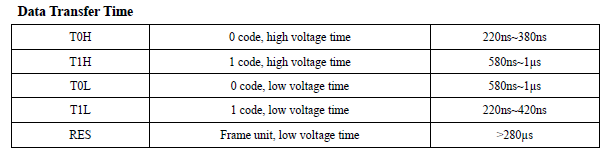
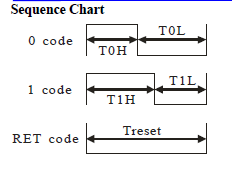
Device specific\*\*



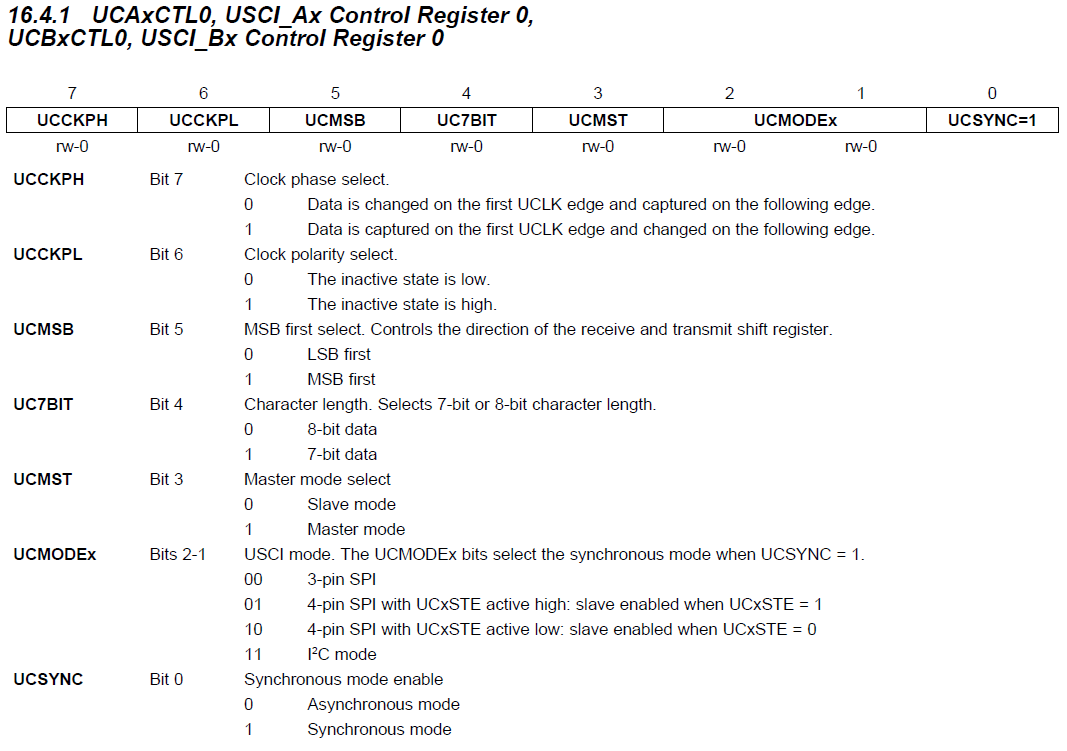
# SPI

We Abuse the SPI MOSI to write to the LEDs. For that use we do not need to read from anywhere and we do not need to pay attention to the CLK, only to make sure the MOSI works without interruption. We only need to send Bits on the MOSI Pin whenever we need it.

LED specification:



To send the Times reqiuered for the LEDs the SPI can be used in

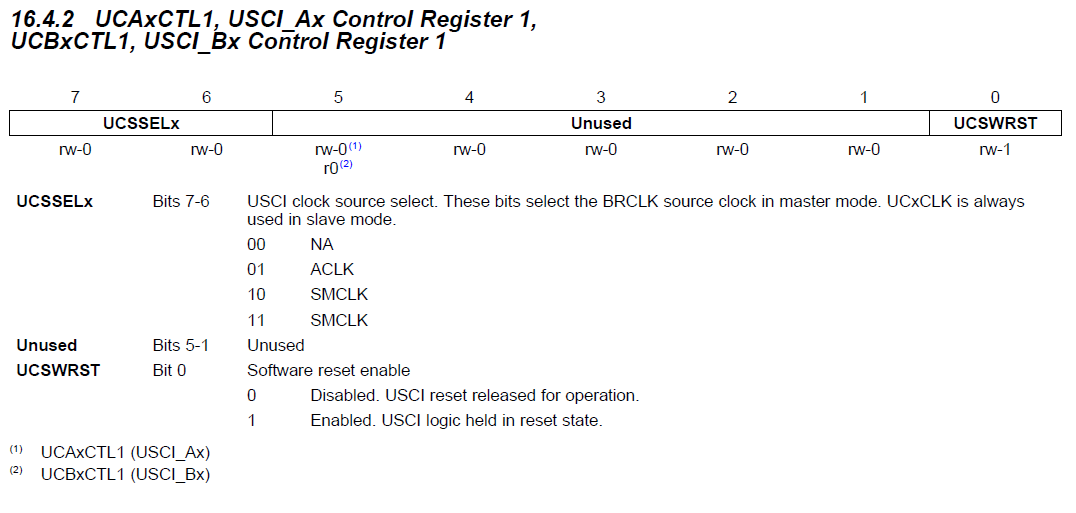


UCMSB = MSB first = 1

UCMST = Master Mode = 1

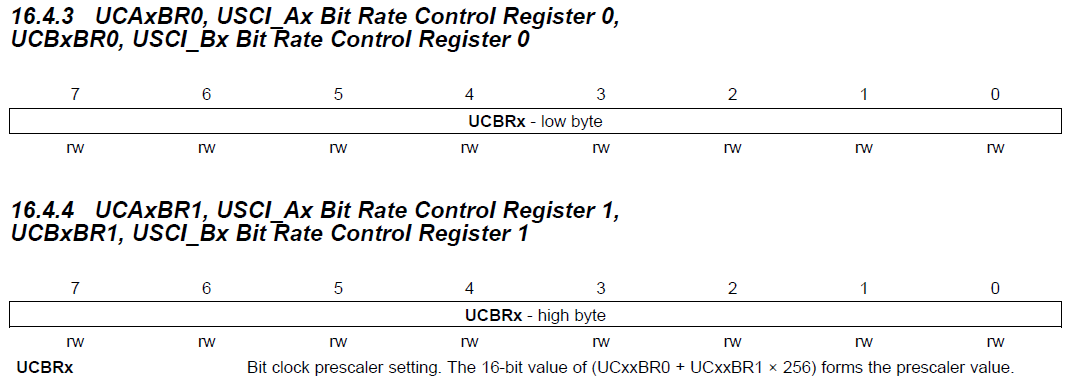
UCMODEx = SPI mode = 00

UCSYNC = synchronous = 1 ??? asynchronous for not noting the CLK ???



UCSSEx = SMCLK = 10 or 11

UCSWRST disable to enable module and enable again during operation when needed

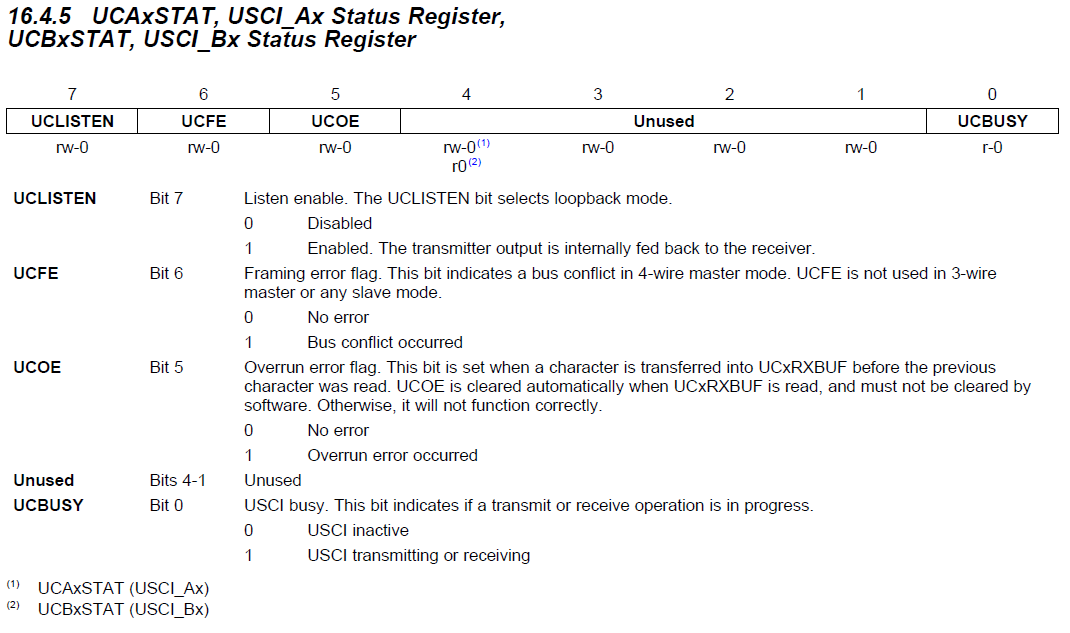


Set prescaler to 4 so the SMCLK of 16MHz will be scaled down to 4MHz.

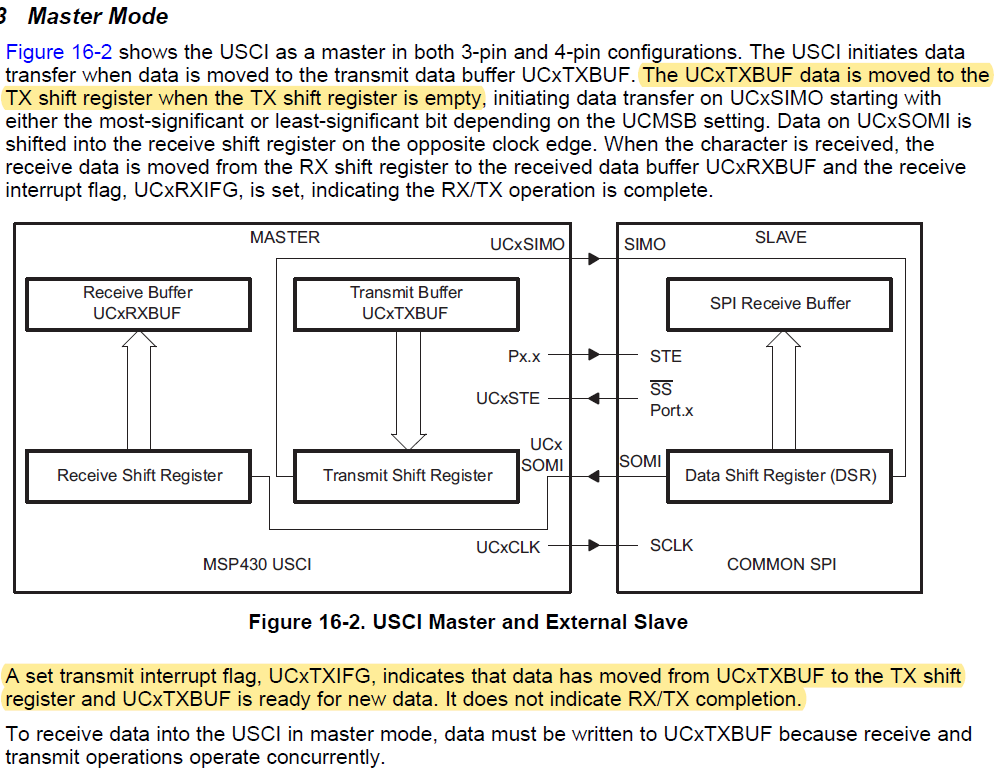
To send a T0H or a T1L it will take 1 cycle.

To send a T1H or a T0L it will take 3 cycles.

To send a RES it will take 12 cycles.



UCLISTEN = maybe can be used for debugging



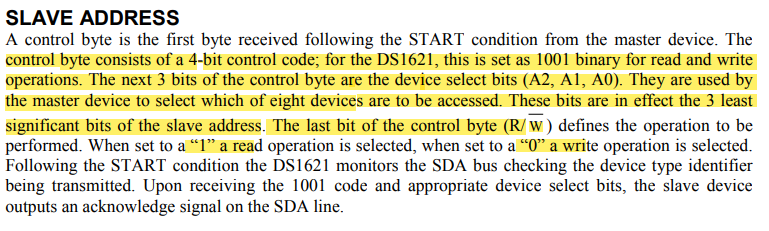
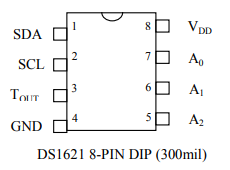
To abuse the SPI as PWM generator we need to send the Data packages to the TXBUF register, wait for it to be stored to the shift register indicated by the UC1TXIFG, and then send the next data package to the TXBUF.

To use the SPI output, define Port 5 Pin 1 as SIMO. For this set the Port Select Port 5 Bit 1 to indicate its use as secondary function.

# Temperature sensor

I actual don't even need a temperature sensor I just want to test out the I2C Bus with a simple device so let us hop into that for a brief moment.

It has the following Pinout :



The A0, A1 and A2 pins are only useful if you have more than one of these devices so they do not matter to us. Accordingly, the slave address will be 10010001 for a read and 10010000 for a write addressing.

This device has a command protocol that is used to issue the wanted command. To write to the device the Master has to generate a Start condition send the device specific address byte and after the acknowledge the device can send the command for the wanted write function. After an the repeatedly send acknowledges the Data bytes can be send.

To read from the device the Master has to generate a Start condition send the device specific address byte and after the acknowledge the device can send the command for the wanted read function. Now the Master has to generate an REPEATED START and end the address byte once again but now as a read address. After the device acknowledge the device starts transmitting either one data Byte with an following NACK and stop condition or two data bytes where the master firs generates an ACK and after the second byte the NACK and STOP.

