Microcontroller Programming - 1

Registerniveau programmeren met de MSP4305969FR Launchpad via Code Composer Studio

Douwe T. Schotanus, Bart Snijder, Marco Winkelman

Augustus, 2024

EDMP.24

Reader EDMP1

Versie 1.0

Domein Techniek - Engineering en Design - Opleiding Elektrotechniek

Versiebeheer

Versie	Datum	Wijzigingen
0.1	10-06-24	Creeëren document
0.2	12-06-24	Overzetten GPIO, ISR, Clocks en Timers
0.3	21-06-24	Verbeteren van H1. Toevoegen uitleg theorie.
0.4	02-07-24	Toevoegen H4 en H5

Contents

0 Introductie

In dit document vind je de practicumbeschrijving van MP1 (Microcontrollerprogramming 1). Voordat je met het practicumgedeelte begint, is het verplicht om de voorbereidende opdrachten thuis te maken. De docent controleert hierop.

Voordat je begint:

1. Installeer de benodigde software. Zie hiervoor installatiehandleiding software. We gebruiken hiervoor code composer studio
2. Je programmeert je eigen bordje. Je mag samenwerken, maar je moet uiteindelijk zelf code schrijven.
3. De verwachte inspanning is 1,5 uur voorbereiding en 3 uur practicum per week
4. Er geldt een aanwezigheidsplicht bij het practicum
5. Elke week schrijf je een meetrapport. Hierin beantwoord je alle vragen en kopieer je de code die je hebt geschreven. Vul dit meetrapport zonder de practicumhandleiding te kopiëren in. Dit meetrapport lever je elke week individueel op Brightspace in.
6. Wanneer je een bron gebruikt zoals een datasheet, refereer je daarnaar.

In dit vak herhalen we een aantal onderdelen die je in uC-programmeren in het eerste jaar ook gehad hebt. We doen dit, zodat je op stoom komt voor deze microcontroller. Daarnaast gaan we geen gebruik meer maken van code generators, zoals Atmel Start of MCC. Dit vak heeft tot doel de opbouw van een processor, gebaseerd op de ARM Cortex M0+, te leren aan de hand van het schrijven van low-level (registerniveau) C programma's. De documentatie van de producent van de MSP430 processorfamilie, Texas Instrument (kortweg TI), wordt zoveel mogelijk gevolgd. Op Brightspace en op Internet zijn ontzettend veel bestanden te vinden over deze processor. We hebben alvast verschillende documenten klaargezet, die je kunt gebruiken. Dit zijn:

1. MSP430 workshop
2. MSP430 userguide
3. MSP430 datasheet
4. Studiehandleiding
5. Diverse voorbeeldprogramma's

Dat we zonder grafische tools en zonder bestaande libraries gaan programmeren heeft twee redenen:

1. Het voornaamste doel van dit vak is het leren begrijpen van een processor op het registerniveau. Hoe werken de registers, wat kan ik aanpassen, hoe pas ik dit aan? Als embeddedprogrammeur kun je soms een chip krijgen waarvoor geen bestaande libraries beschikbaar zijn. Hier moet jezelf op basis van een datasheet mee uit de voeten kunnen.
2. Daarnaast ga je vaak gebruik maken van sensoren of actuatoren waarvoor geen bestaande libraries zijn. Jouw taak is om uit te vogelen hoe deze sensor/actuator werkt en hoe deze aangestuurd wordt. Om die reden gaat het tweede deel van dit vak over het schrijven van ons eigen I2C-library.

Door de focus van dit vak is het om die reden niet toegestaan om bestaande libraries, anders dan de MSP430.h te gebruiken, tenzij dit expliciet toegestaan is door de docent. Het gebruik van driverlib

(Arduino-achtig library van TI is niet toegestaan). Dit heeft ermee te maken dat deze libraries veel van de onderliggende complexiteit verbergen. Dit is natuurlijk de bedoeling van zo'n library, maar dat maakt het begrip van hoe een bepaalde peripheral exact aangestuurd moet worden veel moeilijker.

Na het afronden van dit vak, ben je in staat om met een nieuwe microcontroller in relatief korte tijd uit de voeten te kunnen, daarnaast kun je deze microcontroller integreren in een simpele toepassing waarbij je zelf drivers programmeert voor I2C sensoren en actuatoren.

De onderdelen die we gaan behandelen zijn 1 GPIO + Interrupts, 2 Clocks + Timers en RTC, 3 ADC, 4 I2C, 5 I2C sensor. 6. Testen en versiebeheer.

We sluiten dit vak af met een assessment. In dit assessment demonstreer je dat je een microcontroller kunt programmeren om een I2C sensor uit te lezen met een zelfgemaakt I2C library.

1 Introductie, GPIO & ISR

Deze week herhalen we de basis van GPIO (General Purpose Input & Output en ISR (interupts). Lees de theorie van uC-programmeren uit jaar 1 over GPIO en ISR opnieuw door. Dit doe je voor de les.

We beginnen met wat theorievragen over GPIO en ISR, om je geheugen weer op te frissen.

In dit practicum gaan we de twee leds van het ontwikkelbordje aansturen. Daarnaast gaan we de twee knoppen uitlezen. Elke pin heeft een multiplexer waarmee de juiste functie (periferie) aan de fysieke pin wordt gekoppeld. Dit heb je al gedaan in uC-programmeren in jaar 1, waar we inputs, outputs of analoge functies op een pin kwijt konden. Welke functie dat is, wordt bepaalt door het P1SEL register. In het practicum van EDMP1 werk je met de FR5969. De userguide (SLAU367.pdf) van de MSP430 processorfamilie beschrijft hoe we deze functionaliteit op registerniveau kunnen instellen. Daarnaast moet je ook weten aan welk pootje van het IC een LED vast zit. Dit wordt in het datasheet beschreven (MSP4305969FR.pdf)

De meeste pinnen hebben twee of zelfs meer functies.

1. Pak het datasheet erbij en ga naar p7. Bestudeer de pin layout. Benoem de functionaliteiten van pin 32, 33 en 34.

Van elke pin kunnen we instellen welke functionaliteit gebruikt moet worden. We doen dit door de juiste registers in te stellen. Daarnaast hebben we memorymapped IO. Dit houdt in dat we o.b.v. waarden in het memory, alles aan de GPIO kunnen instellen, waaronder input/output, pullup weerstanden etc.

2. Pak de userguide erbij en ga naar H12. Bekijk de structuur (inhoudsopgave) van dit hoofdstuk. Wat valt je daarin op? Vergelijk dit ook met andere hoofdstukken in de userguide.
3. Ga naar het onderdeel waar alle registerinstellingen beschreven worden. Dit onderdeel bestaat uit verschillende subonderdelen met tabellen. Bekijk een aantal verschillende registers en bekijk wat voor waarden we allemaal toe kunnen kennen.
4. Op welke manier kunnen we pin (bijvoorbeeld P4.6) als output configureren?

Hierna volgen een aantal vragen over ISR.

5. Vaak slaapt de processor wanneer deze geen interrupt krijgt. Waarom is dit nuttig?
6. Beschrijf in je eigen woorden op welke manier een interrupt werkt.
7. Waarom is het belangrijk om ISR's kort te houden?
8. Wanneer verlaat de MSP430 een LPM (Low power mode)?
9. Waarom wordt polling meestal afgeraden, en is het verstandiger om interrupts te gebruiken?
10. Benoem 4 verschillende interruptbronnen (peripherals).
11. Op welke manier weet de processor dat een interrupt heeft plaatsgevonden?
12. Wat wordt er bedoeld met een interrupt port vector?
13. Waarom heeft elke interrupt vector een bepaalde prioriteit?
14. Bepaalde vectoren hebben een hogere prioriteit. Welke van de volgende twee vectoren heeft de hoogste prioriteit: GPIO Port 2 of WDT Interval Timer
15. Probeer in de header file alle interrupt vectors te vinden

1.1 GPIO

Het is natuurlijk lastig om vanuit niets te beginnen programmeren. Daarom gaan we gebruik maken van een aantal voorbeelden van TI.

16. Maak nieuw CCS (Code Composer Studio) project aan. Kies hiervoor Basic examples en Blink the LED.
17. Bouw de code en debug op target. Als het goed is knippert de groene led snel.

Zoals je ziet wordt een niet zo'n nette methode gebruikt om een delay in te bouwen. We gaan dit later op basis van een timer doen.

18. Bekijk de code goed om te begrijpen wat er gebeurt. Kijk op de launchpad op welke poorten de knoppen en leds aangesloten zitten.
19. We willen het knipperen wat interessanter maken door de rode led toe te voegen. Breid de code zo uit dat de rode led aan gaat als de groene uitgaat. Dus laat ze om en om knipperen. Zorg voor de juiste initialisatie van de richting van de poorten. Bouw de code en debug op target. Als het goed is knipperen de groene en rode led nu om en om. Neem de code mee in je logboek.

De MSP430 heeft net als alle andere uC's programmeerbare GPIO (General Purpose IO). Het programmeren van deze IO werkt net iets anders dan bij de AVR128db48 uit het eerste jaar. Zo zijn de poortnamen niet meer PA, PB etc. maar zijn ze P1, P2, P3 etc. In de tabel werken we met P1. De meest belangrijke registers hiervoor zijn opgenomen in tabel **??**. Let op dat je voor een knopje altijd de pull

Register	Functie	Voorbeeld
DIR	Zet de pin op input (0) of output (1)	$P1.DIR\| = BIT0;$
IN	Lees de inputwaarde van de pin	$a = P1.IN\&BIT1;$
OUT	Zet de outputwaarde van de pin (als output) Zet de pull resistor als pulldown (0) of pullup (1)	$P1.OUT\wedge = BIT0;$
REN	Zet de pull resistor uit (0) of aan (1)	$P1.REN\| = BIT1;$
SEL0	Zet de specifieke functionaliteit van de PIN	$P1.SEL0\| = BIT4\|BIT5;$
SEL1	nodig voor extra functionaliteit	$P1.SEL1\| = BIT4\|BIT5;$
SELC	nodig voor extra functionaliteit	
IV	Interrupt vector. Bevat alle pending interrupts	
IES	Interrupt edge select, zet rising (0) of falling (1) edge detectie	$P1.IES\| = BIT1;$
IE	Interrupt disable (0) of enable (1)	$P1.IE\| = BIT1;$
IFG	Interrupt flag not pending (0) of pending (1)	$P1.IFG\& =\sim BIT1;$

Table 1: Cheatsheet voor registerinstellingen van de GPIO

resistor gebruikt, anders blijft je knopje zweven en dat geeft problemen. Het is in dit vak de bedoeling dat je voortaan zelf op zoek gaat in datasheets om dit soort informatie boven tafel te krijgen.

We gaan nu een nieuw programma schrijven waarin we de rode led kunnen besturen met knop S1. We willen dat de rode led brand als je S1 indrukt, en uitstaat wanneer je dit niet doet.

20. Welke poort zit er aan S1?
21. Maak een nieuw project aan. Noem dit week1_button
22. Zet de poort die aan S1 zit als input.

23. Zorg voor de juiste pull configuratie.

24. Een schakelaar die aan ground zit doet niet zo veel. Waarom wordt het aanbevolen om dit te doen met alle poorten die als input geschakeld worden?

25. Lees S1 uit in de while loop.

26. Pas de waarde van de rode led zodanig aan, dat deze aanstaat in als S1 ingedrukt is, en uitstaat als deze dat niet is.

27. Neem je code op in je logboek.

We gaan nu het knipperen aanpakken. We willen dat de knoppen S1 en S2 gebruikt kunnen worden om de snelheid van het knipperen te wijzigen. Als je op S1 drukt gaat het knipperen steeds langzamer. Als je op S2 drukt gaat het knipperen steeds sneller. Als je beide knoppen tegelijk indrukt gaat het knipperen weer naar de oude snelheid terug.

28. Maak weer een nieuw project aan. Pak hiervoor weer het voorbeeldprogramma blink. Noem dit week1_variabeleknipper.

29. Declareer boven de main-functie een unsigned long variabele met een geschikte naam (knipper_delay) en initialiseer deze met de waarde 10000. Vervang de 10000 in de code van de wachtlus door deze variabele.

30. Maak initialisatie code om de poort die aan S2 zit als input te zetten zodanig dat de schakelaar S2 ook kan worden uitgelezen.

31. Voeg nu code toe om de waarde van de variabele te veranderen door te testen op de stand van de schakelaars S1 en S2.

32. Bouw de code en test het uit. Experimenteer met de waardes. Zorg dat de snelheid prettig aangepast kan worden.

33. Bekijk met de debugger de waarde van knipper_delay met Watch Expression en zet op die plek een breakpoint met als properties de action Refresh all windows. Je plaatst een breakpoint door dubbel te klikken in de kantlijn.

34. Wat merk je nu aan de knippersnelheid? Hoe komt dat denk je?

35. Neem je code op in je logboek.

We gaan later gebruik maken van timers en interrupts om delays in te bouwen en knoppen uit te lezen. Je hebt dan ook minder last van de debugger.

1.2 ISR

We gaan nu proberen een ISR te configureren op een pin. Gebruik hiervoor het datasheet en/of tabel **??**.

36. Creëer een nieuw project week1_ISR.
37. Gebruik pin p1.1
38. Zet de pin als input
39. Configureer een pulldown resistor.
40. Laat de interrupt op een rising edge plaatsvinden.
41. Clear de individuele interruptflag.
42. Zet de GIE (Global Interrupt Enable) aan. Tip: __bis_SR_register(GIE);

Voor de MSP430 schrijf je iets andere ISR's dan voor de curiousity nano. We gebruiken hiervoor bijvoorbeeld:

Listing 1: ISR voor de MSP430

```
#pragma vector = ????
__interrupt void pushbutton_ISR (void)
{
        P1OUT ^= BIT0;                          // Toggle P1.0 (LED)
}
```

In bovenstaand voorbeeld houden we geen rekening met welke pin precies de interrupt veroorzaakt heeft.

43. Op welke manier zou je hier rekening mee kunnen houden?
44. Neem de ISR mee in je code en plaats de juiste portvector op de ????
45. Run de code en bekijk of de ISR werkt. Het zou kunnen zijn dat de ISR maar 1 keer werkt, en dat daarna de code niets meer doet. Waardoor zou dit kunnen komen?

Om te controleren wat er precies gaande is, gaan we gebruik maken van een debugfeature waarbij we in de registerwaardes van de MSP430 kunnen kijken.

46. Plaats een breakpoint aan het begin van je main, en laat het programma in debugmode runnen. Zorg ervoor dat deze stopt bij het breakpoint.
47. Open het registers window in CCS (view en dan registers)
48. Controleer of de registers correct ingesteld staan:

Register	Waarde	Functie
P1DIR.0	1	Output
P1DIR.1	0	Input
P1REN.1	1	Enable pullresistor
P1OUT.0	0	Set output
P1IE.1	1	Enable Interrupt
P1IES.1	0	Select edge
P1IFG.1	0	Interrupt Flag

49. Bekijk ook het status register (SR onder core registers)

50. Wanneer je nog niet voorbij GIE bent in je code (breakpoint), zou deze als het goed is nog uit moeten staan.

51. Gebruik een aantal keer de step-over functie om een aantal stappen te zetten in je code, totdat je voorbij de GIE komt. Bekijk het GIE register opnieuw.

Voor niet gegroepeerde interrupts, zal de CPU automatisch de IFG (interrupt flag) clearen. Dit gebeurt niet automatisch met gegroepeerde interrupts (group interrupts) zoals bijvoorbeeld bij een GPIO port.

52. Op welke manier kunnen we een IFG clearen voor een specifieke interrupt vanuit een groep? Houdt er rekening mee dat je niet per ongeluk andere (pending) interrupts cleared.

53. Bekijk het volgende stukje code, en neem dit over in je programma. Plaats op de plaatsen met ??? de juiste aanvulling.

Listing 2: ISR voor de MSP430

```
//*********************************************************************
// Interrupt Service Routines
//*********************************************************************
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
        switch(__even_in_range( ????, 0x10 )) {
                case 0x00: break; // None
                case 0x02: break; // Pin 0
                case 0x04: // Pin 1
                ??????????????????????;
                break;
                case 0x06: break; // Pin 2
                case 0x08: break; // Pin 3
                case 0x0A: break; // Pin 4
                case 0x0C: break; // Pin 5
                case 0x0E: break; // Pin 6
                case 0x10: break; // Pin 7
                default: _never_executed();
        }
}
```

2 Clocks, Oscillators & Timers

2.1 Voorbereiding

We gaan deze week aan de slag met een aantal Clock voorbeelden en gaan zelf een aantal clock parameters configureren, en nameten met led en oscilloscoop. Neem tijdens het meten foto's van je opstelling en metingen en voeg deze toe aan je logboek. De voorbeelden kun je vinden bij de Resource Explorer, die open je via Project/Examples.

Ter voorbereiding deze week pak je klokken en timers van uC-programmeren erbij uit jaar 1. Daarnaast bekijk je de userguide/datasheet over de klok (H3) en timerinstellingen. Lees beide hoofdstukken door.

Elke microcontroller heeft klokken. Deze zijn nodig voor de CPU maar ook voor bijna alle peripherals. De klokinstellingen van de MSP430 worden gegeven in H3. We kunnen deze klokinstellingen aanpassen met behulp van een aantal registers.

1. Waarom heeft de MSP430 3 verschillende klokken?
2. Geef de namen van de 3 verschillende klokken.
3. Geef de frequentie (Hz) van de $LF\_Crystal$ en de frequentie (Hz) van de $HF\_Crystal$.
4. Bekijk de launchpad en bekijk de locatie waar deze externe kristallen aanwezig zijn. Wat valt je hier op?
5. Benoem de 3 resetmodes van de MSP430.
6. Welke resetmode wordt er aangeroepen wanneer de Watchdogtimer (WDT) een timeout geeft?
7. Wat is een Timer?
8. Hoe ziet een PWM-signaal eruit?
9. Wat zijn de twee belangrijke eigenschappen van een PWM signaal?
10. Wat is het verschil tussen capture en compare mode voor een timer?
11. Wat is een timer countmode voor een timer? (zie datasheet)
12. Wat is een timer output mode? (tip bekijk TA0CTL1 in de datasheet)
13. Wat is de reden voor sleep?
14. Welke sleepmodes heeft de MSP430?
15. Pak de usermanual erbij en ga naar p103. Bestudeer de volgende registers:
    - CTL0
    - CTL1
    - CTL2
    - CTL3
16. Pak de usermanual erbij en ga naar p657. Bestudeer de volgende registers:
    - TAxCTL
    - TAxCCTL0
    - TAxCCR0
    - TAxIV

17. Hoeveel klokcycli duurt het tot een 16-bits timer afloopt?

18. Welke klok i.c.m. welke klokbron zouden we kunnen gebruiken om een timer na 2s te laten aflopen?

19. Geef een berekening van een klok i.c.m. klokbron en eventueel een prescaler divider die na 2s afloopt.

2.2 Clocks

We gaan een nieuw project beginnen op basis van een voorbeeld van TI.

20. Ga naar de Resource Explorer en kies MSP430FR5969, en daarbinnen de Peripheral Examples/Register Level voorbeelden.
21. Import het voorbeeld $MSP430FR59xx\_cs\_01.c$.
22. Noem je nieuwe project $week2\_clock$
23. Lees goed de beschrijving (regel 47-59) om te zien wat de bedoeling van de code is en bestudeer het commentaar in de code.
24. Geef de registerniveaufunctie die de WDT stopt. Kijk hiervoor in de voorbeeldcode.
25. Bouw de code van het voorbeeld, debug en kijk naar de knippersnelheid van de groene led. Probeer te verklaren waarom deze snelheid knippert.
26. Op welke frequenties draaien de 3 klokken in het voorbeeld?
27. De SMCLK en ACLK wordt rechtstreeks als output op een pin geplaatst. Geef deze pinnen. Geef ook de code van het voorbeeld waar deze pinnen aan deze klokken gekoppeld worden.
28. Probeer de frequentie van de klokken te veranderen d.m.v. de registerinstellingen. Meet de frequentie na met door de oscilloscoop op de pin te zetten waar de SMCLK opstaat (de pin van ACLK is op de FR5969 Launchpad niet beschikbaar) . Doe voorzichtig, niet de klemmetjes van de oscilloscoop probe rechtstreeks op de pin van de Launchpad, maar doe dit via een female header draadje op de pin en zet daar de klemmetjes van probe op. Is de interne oscillator een beetje nauwkeurig?
29. Verander de code zodat de klokken weer op hun standaardfrequentie draaien. Je zult hier 3 lijnen voor moeten wegcommenten.
30. Test de code uit.
31. In de opdrachten van week 1 hebben we de klokken niet ingesteld. Toch werkte onze code. Leg uit waarom het niet nodig is om de klokken in te stellen voor werkzame code.
32. Leg uit waarom het meestal toch handiger is om wel de klokken in te stellen.
33. Probeer de knippersnelheid weer op de oude frequentie te krijgen door de *delay_cycles* aan te passen.
34. Hanteer nu weer de oorspronkelijke klokinstellingen van dit voorbeeld. Wat zie je gebeuren?
35. Pas de klokinstellingen zo aan zodat we de oorspronkelijke knipperfrequentie terugkrijgen.
36. Meet de pin waar de SMCLK op staat. Er valt je vast iets op, (of je hebt het in een keer goed gedaan). Wat moet je doen als je de SMCLK ook langzamer wilt maken?

2.3 Oscillators

XT1 is een kristal wat we als oscillatorbron kunnen gebruiken. Deze is zeer low power en zeer nauwkeurig.

37. Maak een nieuw project aan en noem dit *week2_oscillator*
38. Importeer de code uit voorbeeld $MSP430FR59xx\_cs\_03.c$ (TI Resource Explorer).
39. Lees goed de beschrijving (regel 46-65) om te zien wat de bedoeling van de code is en bestudeer het commentaar in de code.
40. Bekijk configuratie van de klokken in de code (regel 87 – 103)
41. Kopieer het stukje code van *cs_01* voor het knipperen van de LED (de while lus) in *cs_03*. Zet dit direct voor de regel $\_\_bis\_SR\_register(LPM0\_bits)$; Daarachter kom je nooit, want dan slaapt de FR5969.
42. Bouw en test. Wat is de knipper frequentie en wat is de SMCLK snelheid? Meet eventueel na met de scope.
43. Wat moet je aanpassen aan de klokconfiguratie om de SMCLK te koppelen aan het XT1 kristal? Koppel ook de SMCLK direct aan het kristal zodat je dit kunt nameten op de SMCLK pin.
44. Bouw en test. Wat is de knipper frequentie en wat is de SMCLK snelheid? Meet na met de scope.
45. Neem de code op in je logboek
46. Pas de while lus aan dat we weer de oorspronkelijke snelheid krijgen.

2.4 Timers

We gaan nu proberen een timer te configureren. Gegeven is de volgende timer instelling.

Listing 3: Een programma voor een timer

```
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;                    // Stop WDT

    // Configure GPIO
    P1DIR |= BIT0 | BIT1;                        // P1.0 and P1.1 output
    P1SEL0 |= BIT0 | BIT1;                       // P1.0 and P1.1 options select

    PM5CTL0 &= \sim LOCKLPM5;

    // Klok instellingen
    CSCTL0_H = CSKEY >> 8;                       // Unlock CS registers
    CSCTL1 = DCOFSEL_6;                          // Set DCO = 8MHz
    CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM_DCOCLK;
    CSCTL3 = DIVA__8 | DIVS__8 | DIVM__8;        // Set all dividers
    CSCTL0_H = 0;                                // Lock CS registers

    // Configure Timer0_A
    TA0CCR0 = 1000-1;                            // PWM Period CCR0
    TA0CCTL1 = OUTMOD_7;                         // CCR1 reset/set
    TA0CCR1 = 750;                               // CCR1 PWM duty cycle
    TA0CCTL2 = OUTMOD_7;                         // CCR2 reset/set
    TA0CCR2 = 250;                               // CCR2 PWM duty cycle
    TA0CTL = TASSEL__SMCLK | MC__UP | TACLR;     // SMCLK, up mode, clear TAR

    // Sleep and do nothing
    __bis_SR_register(LPM0_bits);                // Enter LPM0
    __no_operation();                            // For debugger
}
```

47. We zien een configuratie van de GPIO-pinnen. Wat gebeurt daar?
48. We stellen de klok en oscillatoren in. Wat doen we daar allemaal?
49. We stellen een aantal registers van TA0 in. Wat stellen we allemaal in? Beschrijf alles, met behulp van de datasheet.
50. In welke slaapmodus zitten we?
51. Sluit je vorige project en maak een nieuwe volgens bovenstaande stappen. Dit noemen we nu *week2_timer*.
52. Voeg bovenstaande code toe.
53. Voer de code uit en bestudeer het gedrag.
54. Demonstreer je begrip van de code, door deze als volgt aan te passen:
    - Verhoog de PWM frequentie 2x
    - Verander de klok die verbonden is met de timer

- Verander de dutycycle naar 10%
- Maak een whilelus waarin je de dutycycle elke 10 ms met 1% laat oplopen. Na die seconde moet je weer vanaf 0% beginnen.

We willen ook graag een timer maken die een interrupt kan genereren, zodat we wat in de code kunnen uitvoeren.

55. Maak een nieuw project en noem dit $week2\_timerISR$.
    - Stel de LED in als een output.
    - Configureer de ACLK. We willen dat deze CLK met een frequentie van 32768 Hz draait. Gebruik hiervoor eventueel een prescalar divider en vergeet niet om de klokinstellingen te unlocken en daarna weer te locken
    - Congigureer timer A1 zodat deze een interrupt kan geven op een overflow
    - Koppel de ACLK aan de Timer.
    - Zet de Timer in continuous mode en clear de timer.
    - Configureer de CCR (capture and compare) zo dat deze een compare interrupt kan genereren op CCR2. Tip: wellicht moet je hiervoor op zoek naar voorbeelden, of je moet in het datasheet kijken.
    - Zet de Global Interrupt Enable aan.
    - Laat de MSP430 draaien in sleep mode LPM0.
    - Voeg twee ISR's toe 1 voor vectors $TIMER1\_A0$ en 1 voor $TIMER1\_A1$. Tip: neem de code van slide 23 over.
56. Beide ISR's zijn bedoeld voor TIMER A1. Echter zullen ze om verschillende redenen afgaan. Licht toe wanneer elke ISR aangeroepen wordt.
57. Plaats onder de juiste plaats code zodat de LED aangezet wordt op het moment dat de CCR2-waarde bereikt wordt. De LED gaat weer uit op het moment dat er een timeroverflow plaatsvindt.
58. Wanneer we in plaats van een interrupt na een overflow in continuous mode een interrupt na een overflow in up mode (overflow o.b.v. waarde in CCR0 register) willen, wat moeten we dan allemaal aanpassen?

3 ADC

3.1 Voorbereiding

Bereid deze week voor door de relevante hoofdstukken van jaar 1 uC-programmeren erbij te pakken. Lees daarnaast de relevante hoofdstukken in de datasheet/userguide.

1. Wat doet een ADC?
2. Waarom heeft de ADC een referentiespanning nodig?
3. Wat wordt er bedoeld met samplerate?
4. Wat is de resolutie van een 12-bits ADC die meet tussen 0 en 3.6V?
5. Leg uit hoe we het ADC-resultaat in ons programma kunnen gebruiken, zonder gebruik te maken van polling.
6. We hebben meestal maar een ADC-peripheral op onze uC zitten. Op welke manier konden we toch verschillende pinnen uitlezen?
7. Pak de usermanual erbij en ga naar p374. Bestudeer de registers.
8. Wellicht zie je dat de MSP430 een hoop verschillende MCTL registers heeft. Waarvoor dienen deze registers?
9. Waar slaat de ADC zijn resultaat in op?

3.2 ADC

10. Maak een nieuw project aan $week3_ADC$.
11. Open de resource explorer en ga net als voorgaande weken naar de registerlevel voorbeelden voor de MSP430FR5969 toe.
12. Vervang de de code van je main met die van $msp430fr59xx\_adc12\_01.c$.
13. Bestudeer de code waarin we de input pinnen configureren.
14. Op welke pin wordt een analoge waarde uitgelezen?
15. Waarom is het nodig dat we deze pin als analoge pin moeten instellen? Op welke manier wordt dit gedaan? Geef de bijbehorende code.
16. Op regels 87-91 configureren we de adc. Bestudeer deze code regel voor regel.
17. We zien een aantal CTL (control) registers. Bekijk deze registers in het datasheet en licht voor alles wat we instellen toe wat de reden is dat we dit doen. (regel 87 t/m 89).
18. Wat stellen we in op regel 90? Licht toe.
19. Wat stellen we in op regel 91? Licht toe.
20. Op regel 96 starten we de ADC. We doen hier een start en enable conversion. Licht toe wat de verschillen zijn tussen start en enable. Tip: een van deze twee hadden we ook buiten de while lus kunnen aanzetten.
21. Waarom is het logisch dat er een apart start en enable instelling zijn?
22. Wordt de ADC softwarematig of hardwarematig gestart? Licht toe.
23. In welke LPM slapen we wanneer we geen sample nemen?

Vanaf regel 103 begint de code voor onze ISR. Ook de ADC kan een ISR geven. Dit heb je ook al gezien in uC-programmeren. We zien een stukje precompilercode op regels 103, 104, 106, 108 t/m 110 in onze ISR staan. Deze code wordt toegevoegd om ervoor te zorgen dat we een waarschuwing krijgen als we per ongeluk 2x een ISR definiëren. Daarnaast wordt er aangegeven dat de de GNUC compiler niet ondersteund wordt. In onze situatie doet deze precompilercode niets, maar het is toch goed om te weten waarom dit toegevoegd wordt. Het is niet nodig dit over te nemen, maar het kan ook geen kwaad om dit wel te doen.

24. Welke ISR-vector handelt deze ISR af?
25. Wat zou de reden kunnen zijn dat we $switch(\_\_even_in_range(ADC12IV, ADC12IV\_ADC12RDYIFG))$ op regel 107 gebruiken? Tip: Dit vervult een zeer belangrijke functie
26. We zien dat er heel veel verschillende interruptbronnen zijn voor deze ISR-vector. Leg uit waarvoor al deze bronnen dienen.

Bij een van de interruptbronnen is een stukje functionele code geplaatst. Hier staat ook een if statement waarbij de waarde van de respectievelijke ADC-waarde gecontroleerd wordt.

27. Welk register bevat (in dit geval) de ADC-uitlezing?
28. Wat gebeurt er in de code?
29. Wat gebeurt er op regel 125? Lees nauwkeurig, dit is niet dezelfde functie als op regel 98. Tip: Het kan misschien handig zijn om dit met de debugger stap voor stap te bekijken.

We willen nu gaan bekijken of we inderdaad een ADC-uitlezing krijgen. Hiervoor gaan we een potmeter aansluiten op de pin.

30. Sluit een potmeter aan op de pin waarop we de ADC uitlezen. Zorg ervoor dat de potmeter correct gevoed wordt.
31. Run de code en draai aan de potmeter. Wanneer het goed is zou je op een bepaalde hoek moeten kunnen zien dat de ADC 'werkt'.
32. Dit is natuurlijk nog niet heel verhelderend. Ga met behulp van een breakpoint en watch expression kijken wat de waarde van de ADC uitlezing is. Tip: We kunnen het breakpoint zo aanpassen, dat in plaats van dat hij pauzeert, de windows refresht. Hierdoor kunnen we vrij eenvoudig meekijken in de processor terwijl deze runt. Rechtermuisknop op je breakpoint, breakpoint properties en verandere 'remain halted' in 'refresh all windows'.
33. Maak een screenshot waarin je laat zien dat je meekijkt in de processor m.b.v. watch expression.

We gaan nu een nieuw project aanmaken om de ADC via een timer te laten starten. Hiermee kunnen we de ADC veel efficiënter uitlezen.

34. Maak een nieuw project en noem dit $week3\_ADCTimer$.
35. Kopieer de code van het voorgaande project ($week3\_ADC$).
36. Programmeer een timer. Gebruik hiervoor de code die je vorige week geschreven hebt.
37. Zorg ervoor dat de timer op een overflow een interrupt geeft.
38. Start de ADC vanuit de timer ISR (gebruik hiervoor dezelfde code als bij het voorgaande project om een sample te nemen).
39. Neem je code over in het logboek.
40. Verhoog de frequentie van de timer zodat we meten met 8 samples per seconde. Bereken het gemiddelde van deze 8 samples. Tip: Je kunt alle samples bij elkaar optellen en na 8 keer de som 3 bits naar rechts schuiven.
41. Breid de code uit: Vergelijk dit gemiddelde met het vorige gemiddelde, als deze meer dan 5 (of kies ander getal) niveaus hoger is, laat dan de groene led branden, als meer dan 5 niveaus lager de rode led, daar tussenin geen led aan. Test het door je vinger op de diode te leggen of te blazen. Kies het niveau van 5 zo dat de leds niet branden als je niets met de diode doet en alleen de rode brandt als de diode opwarmt, en de groene als de diode snel afkoelt.

4 I2C & Softwareproject

Het doel van deze week is het bekwaam worden met de I2C interface. Dit begint met een stukje theorie, en daarna gaan we in het practicum verschillende I2C-sensoren en -actuatoren uitlezen en aansturen.

Tot nu toe heb je waarschijnlijk nog niet heel veel gewerkt met (embedded) communicatieprotocollen. Toch zul je dit als embeddedprogrammeur veel tegen gaan komen. De reden hiervoor is dat we vaak veel verschillende sensoren en of actuatoren willen uitlezen en aansturen op een bord. Vaak willen we dit met een grote nauwkeurigheid en hoge snelheid doen. Je kunt hier natuurlijk een analoge interface in combinatie met een nauwkeurige ADC voor gebruiken, maar dit levert bepaalde problemen op wanneer we dit voor veel sensoren willen doen, of wanneer de sensoren over een grotere afstand van elkaar geplaatst zijn. Wanneer je een analoge lijn wil uitlezen voor een sensor, dan moet je daar een verbinding tussen die sensor en tussen de uC maken. Dat betekent een extra aparte analoge lijn voor elke extra sensor. Dit levert problemen op, niet alleen omdat we zeer veel lijntjes op ons pcb krijgen: ons uC zal op een gegeven moment ook te weinig IO poorten hebben, ook zorgt dit voor EMI problemen en kan zeker wanneer deze lijnen langer zijn leiden tot problemen in de SnR (Signal-to-noise Ratio). Het grote voordeel van een digitale uitlezing is, dat de nauwkeurigheid van het signaal niet verandert. Wanneer we de kwaliteit van het signaal kunnen waarborgen, zal een langere lijn geen effect hebben op de nauwkeurigheid van de uitlezing, immers een 0 blijft een 0 en een 1 blijft een 1. Daarnaast zou het natuurlijk ook handig zijn wanneer we niet voor elke sensor een aparte signaallijn naar de uC hoeven te trekken. Dit is eigenlijk niet wenselijk. Het wordt lastig om al deze lijnen op je PCB te trekken, maar daarnaast is het ook zonde, want veel van de sensoren hoeven we helemaal niet continu uit te lezen. We nemen meestal maar om de zoveel ms een sample, terwijl de rest van de tijd deze lijn niet wordt gebruikt. Wanneer we ervoor zorgen dat we een communicatieprotocol hebben waarin we een bestaande lijn kunnen hergebruiken, zorgt dit voor veel efficiëntiewinst.

De interface naar de uC wordt hiermee sterk vereenvoudigt, en in het geval van I2C bestaat deze uit een datalijn, kloklijn, voeding en gnd. Allerlei sensoren en actuatoren kunnen dan gebruik maken van dezelfde lijn. De I2C-interface is in 1979 ontwikkeld door Philips en wordt tegenwoordig beheert door NXP, wat in 2006 van Philips afsplitste.

Zoals we eerder verteld hebben heeft de I2C-interface 4 lijnen. De data wordt verstuurd mbv de datalijn. Aan dezelfde I2C kunnen meerdere apparaten verbonden worden. Deze apparaten kunnen onderling met elkaar communiceren. Hiervoor geldt dat er n controllers en m aantal targets zijn.

4.1 Softwareproject

In het verleden heb je al heel wat verschillende projecten aangemaakt. Deze projecten bestonden bijna altijd alleen uit een main.c. De main.c is het startpunt van ons programma. Of we dat nu voor een computer of embedded platform schrijven. Nu gaan we de stap naar iets groters maken. We gaan proberen om een gestructureerd project te maken. Hiervoor houden we rekening met een vastomlijnde structuur. Deze structuur wordt in veel verschillende C-projecten aangehouden en helpt ons in de onderhoudbaarheid van onze code. Ook gaan we deze code committen naar een GIT-repository waardoor we onze code eenvoudig kunnen delen en aan versiebeheer kunnen doen.

Hoe we dit gaan aanpakken is als volgt. Je deelt jezelf op in groepen van 5 personen en gaat gezamenlijk de volgende libraries programmeren:

- GPIO + ISR
- ADC
- CLK

- Timer
- I2C

Jullie gaan een gemeenschappelijke GIT-repo onderhouden. Je wordt hiervan maintainer, wat je o.a. de rechten geeft om te committen/pushen. We gaan voor elk library zorgen dat er functies geschreven worden voor de belangrijkste registerinstellingen. Het doel van deze functies wordt beschreven per onderdeel. Hiernaast ga je ervoor zorgen dat een collegastudent jouw code kan gebruiken in een vervolg project. Daarom besteden we veel aandacht aan documentatie en gaan we ook onze code testen.

4.2 Projectopzet

1. Maak groepen van 5 studenten en geef de groep samenstelling door aan de docent
2. Geef iedereen een functie en verantwoordelijkheid:
    - a) Hoofdmaintainer + GPIO + ISR
    - b) ADC
    - c) CLK,
    - d) Timer
    - e) I2C

Er zijn een aantal taken die maar een iemand hoeft te doen. We geven dit aan met een letter (a). Wanneer er geen letter staat, voer je allemaal deze taak uit.

3. Volg nu de GIT-handleiding (zie Brightspace) zodat je een GIT-account aan kunt maken
4. a) Maak een nieuw project aan en voeg de
5. a) Maak een repository aan en geef iedereen in je groep rechten zodat deze maintainer kan zijn.
6. git pull https://github.com/DTSchotanus/MP1A (vervang A door B, C, D, E afhankelijk van je groep)
7. git add *
8. git commit -m "informatieve boodschap"
9. git push -u origin master (-u origin master kan in het vervolg weggelaten worden)

Een git add * voegt alle wijzigingen toe aan een aanstaande commit. Een git commit creëert een lokale versie waarbij nieuwe wijzigingen gebundeld worden. De -m "informatieve boodschap" staat voortaan bij deze versie als wat deze commit inhoudt. De git push upload je wijzigingen naar het centrale repository.

We hebben alvast een structuur voor je klaargezet. Probeer deze structuur te hanteren.

- docs (hierin neem je documentatie op)
- include (hierin neem je de .h bestanden op)
- lib (hierin neem je externe libraries op)
- src (hierin neem je de .c bestanden op)

- tests (hierin neem je de testen op)

Het gebruiken van GIT in teams gaat natuurlijk versieconflicten opleveren. Het is jullie taak als team hier goed mee om te gaan. Een tip die hierin gegeven kan worden is de volgende werkwijze te hanteren:

- Werk met maar een persoon aan een document tegelijkertijd
- Doe eerst een pull voordat je bestanden gaat adden, committen en pushen
- Verwerk je wijzigingen altijd wanneer je klaar bent (pull, add, commit, push)
- Gebruik ipv add * alleen de add voor de specifieke bestanden die je aangepast hebt
- Gebruik informatieve beschrijvingen voor commits. Hanteer hier werkwoord zelfstandignaamwoord in het Engels in kleine letters (conventie). Bijvoorbeeld: add function to read adc12.

10. Zorg ervoor dat iedereen in je team toegang tot het GIT repository heeft en een lokale kopie heeft.
11. Open het project in CCS en bestudeer nu de inhoud van *src/main.c*, *include/gpio_lib.h* en *src/gpio_lib.c*.

Het lijkt misschien een beetje dubbelop, een .h en een .c bestand met precies dezelfde naam. De reden dat we dit toch doen, is om ervoor te zorgen dat alle functie declaraties (prototypes) beschikbaar zijn in de .h bestand. In basisprogrammeren heb je geleerd dat C eerst een prototype nodig heeft om ervoor te zorgen dat het weet hoe een bepaalde functie gebruikt moet worden. De implementatie van die functie (hoe werkt de functie) vindt plaats in het gelijknamige .c bestand. Een programmeur die een library ontwikkeld schermt hiermee de precieze implementatie van zijn functies af van de gebruiker van die functie. Hiermee kun je bugs voorkomen doordat je een duidelijke interface naar de buitenwereld creëert.

Wanneer je in de toekomst een nieuw code-bestand wil toevoegen, zorg je ervoor dat:

- Creëer m.b.v. CCS een nieuw *code_lib.h* in de map include
- Include msp430.h
- Creëer een nieuw *code_lib.c* in de map src
- Include $< include/gpio\_lib.h >$

CCS zorgt er automatisch voor dat de juiste bestanden gelinkt worden. In andere build/make omgevingen kan het nodig zijn om dit handmatig te doen (CMAKE). Dit gebruiken we nu niet. Je compiler ziet dus automatisch het nieuwe .c bestand, omdat dit in dezelfde folder bevindt. Daarna include je het juiste h bestand. Hiermee zorg je ervoor dat de header (prototypes) voor de implementatie komen. Soms kan het voorkomen dat bepaalde code meerdere keren geïnclude wordt. Hiervoor gebruiken we de volgende preprocessor-directives (in bijvoorbeeld *gpio_lib.h*).

- $\#ifndef INCLUDE\_GPIO\_LIB\_H\_$
- $\#define INCLUDE\_GPIO\_LIB\_H\_$
- *code*
- $\#endif /* INCLUDE\_GPIO\_LIB\_H\_ */$

12. Voor elk library hanteren we een .c en een .h bestand Laat iedereen zijn eigen .c en .h bestande aanmaken die nog ontbreken. Zorg ervoor dat iedereen de bestaande naming convention hanteert.
13. Laat iedereen stuk voor stuk zijn changes add, commit en pushen naar de centrale repo.
14. Wanneer je er niet uitkomt, neem dan contact op met de docent.

5 Libraries

Nu we snappen hoe we een gemeenschappelijk project kunnen draaien, gaan we aan de slag om gezamenlijk een stukje embedded software te ontwikkelen. Het doel is uiteindelijk om een eenvoudige scope te maken. We gebruiken hiervoor een OLED display waarop we een analoog signaal laten zien.

Dit systeem werkt als volgt: ADC neemt elke x ms een analoge sample. Dit sample wordt opgeslagen in een variabele. Deze variabele wordt in een array geschoven en deze array wordt geplot op de OLED. We hebben hiervoor meerdere peripherals nodig. Je krijgt misschien het idee, hoe je dit wil gaan aanpakken, toch willen we eerst graag dat je de libraries schrijft, en pas als deze werken, dat je aan de slag gaat met de functionele code. Het doel hier is het schrijven van onderhoudbare/gedocumenteerde/geteste code. Niet zozeer het maken van een goedwerkend programma. We zien dat meer als een mooie bijvangst.

Afhankelijk van je gekozen peripheral ga je nu aan de slag met een van de volgende paragrafen. Je hoeft dus alleen maar het specifieke subhoofdstuk van jouw peripheral te maken. In de initiële commit is een voorbeeld gegeven van hoe je een functie kunt programmeren in een extern library. Hier maken we een functie waar je de waarde van een pin kunt aanpassen door een register en registerwaarde mee te geven. Bestudeer deze code in de .c en .h bestanden om te controleren of je snapt hoe je externe functies kunt aanroepen.

Elk subhoofdstuk bevat een lijst van alle functies en de gewenste functionaliteit. De complexiteit van elke functie kan verschillen. Zorg ervoor dat je aan het einde van deze week jouw library afhebt, zodat jullie dit volgende week kunnen gebruiken om verder te gaan.

Een aantal tips voordat je begint. De registers die je wil veranderen zijn addresses van de MSP430FR5969. Je kunt de waarde van deze adressen vinden in $ccs\_base/msp430/include/msp430fr5969.cmd$.

Maak voor elk .c bestand een .h bestand aan. Begin in dit .h bestand met de prototypes van je functies.

Wanneer we in een functie verschillende registers willen aanpassen, kunnen we de adressen en offsets van die registers gebruiken. Zie hiervoor onderstaande tabel:

Register	Address	Offset
P1OUT	0x0202	0x0000
P2OUT	0x0203	0x0001
P3OUT	0x0222	0x0020
P4OUT	0x0223	0x0021

Dit kunnen we gebruiken om onze functies eenvoudiger te maken. Wanneer je bijvoorbeeld een functie maakt om de PxOUT registers aan te passen, kun je met behulp van het base address van P1 + offset P2, P3 of P4 aanroepen. Je doet dit als volgt:

$*(\&P1OUT + 0x0021) | = BIT6$; Deze regel code is equivalent aan: $P4OUT | = BIT6$;. We pakken het adres van P1OUT (&). Hier tellen we de offset bij op. Dit dereferencen we weer (*). Daarmee krijgen we P4OUT wat we kunnen gebruiken om P4.6 hoog te maken. Deze offset truc kunnen we bij alle Portregisters toepassen.

We kunnen de offset als variabele meegeven aan een functie. Denk bijvoorbeeld aan het definiëren van een ENUM waarin je deze offsetwaardes een naam geeft. Je kunt dan een enkele functie aanroepen die o.b.v. de meegegeven portoffset het juiste portregister (P1,P2,P3 of P4) aanpast.

Gegeven is het volgende stukje code (functie aanroep, waarbij port en bit 2 variabelen zijn). $IES?(*(\&P1IES + port) | = bit) : (*(\&P1IES + port) \& = \sim bit)$;. We gebruiken hier de conditional operator. Deze heb je in het eerste jaar gehad. Dit lijkt op een if/else statement waarbij IES de conditie is die geëvalueerd wordt.

Daarna volgt eerst de if (conditie is waar), en dan de else (conditie is niet waar) uitvoercode. In dit geval gaan we afhankelijk van de waarde van IES of een specifieke bit hoog zetten, of laag maken. Port is de offset van de specifieke port (bijvoorbeeld 0x0020), bit is de specifieke bit van dit register wat we willen aanpassen (bijvoorbeeld BIT5).

5.1 GPIO + ISR

5.1.1 GPIO

$voidpinSet(uint16\_tport, uint16\_tbit, boolval)$
Past port.bit aan naar val. Val kan 0 of 1 zijn. We zouden bijvoorbeeld P4.2 naar 1 kunnen zetten, of P1.2 naar 0.

$voidpinToggle(uint16\_tport, uint16\_tbit)$
Togglet de specifieke pin (port.bit)

$pinGet(uint16\_tport, uint16\_tbit)$
Krijg waarde van specifieke pin (port.bit)

$pinConfigInput(uint16\_tport, uint16\_tbit, boolpullResistor, boolpullUP, boolIES, boolIE)$
Zet Pin als input Configureert pull resistor (pullResistor) Selecteert pull down of pull up (pullUP) Selecteert de juiste interrupt edge (IES) Zet de interrupt aan (IE)

$pinConfigOutput(uint16\_tport, uint16\_tbit, uint16\_tval)$
Zet Pin als output (port.bit) Geeft initiële waarde mee aan pin met waarde val

$pinConfigAnalog(uint16\_tport, uint16\_tbit)$
Configureert pin als analoge pin

$pinConfigTimer(uint16\_tport, uint16\_tbit)$
Configureert pin met timer als output

$pinConfigCLK(uint16\_tport, uint16\_tbit)$
Configureert pin met clock als output

5.1.2 ISR

Het ISR library is bijzonder, omdat de ISR niet als software functies, maar als hardware functies aangeroepen worden. Omdat er een stukje speciale precompiler code voor nodig is, is het makkelijker om een .c bestand aan te maken met alle lege voorgeprogrammeerde ISR's. Deze kun je dan later gaan programmeren.

5.2 ADC

$ADCconfig()$

- Configureert de sample and hold time (ADC12SHT0 en ADC12SHT1)
- Zet de ADC als softwaretriggered of timertriggered (ADC12SHP)
- Zet de klokinstellingen van de ADC: klokbron (ADC12SSELx), clock divider (ADC12DIVx), prescaler divider (ADC12PDIV)

- Stelt de ADC in de juiste bit resolutie in (ADC12RES)

$ADCconfigMEM(ADC12MCTLx)$

- Zet de juiste referentiespanning (ADC12VRSEL) voor dit specifieke memory ctl register (ADC12MCTLx)
- Zet de juiste inputchannel (ADC12INCHx) voor dit specifieke memory ctl register (ADC12MCTLx)

$ADCenable()$
enabled/disabled de ADC (ADC12ENC)

$ADCon()$
Zet de ADC aan/uit (alleen mogelijk wanneer adc enabled is) (ADC12ON)

$ADCsample()$
start een softwarematige sample met de ADC (ADC12SC)

$ADCgetMem()$
Leest juiste memory register uit

$ADCIERenable()$
Zet op het juiste register een IER

5.3 CLK

$CLKunlock()$ unlocked de clock

$CLKunlock()$ locked de clock

$CLKsetDCO()$ Past de DCO aan naar gewenste snelheid

$CLKsetACLK()$ Past de ACLK aan naar gewenste oscillatorbron en snelheid

$CLKsetSMCLK()$ Past de SMCLK aan naar gewenste oscillatorbron en snelheid

$CLKsetMCLK()$ Past de MCLK aan naar gewenste oscillatorbron en snelheid

5.4 Timer

$TAInit()$ initialiseert TA instantie (0 of 1) Koppelt juiste clock Bepaalt prescaler divider Bepaalt juiste mode (up, continuous ...) Bepaalt of er een interrupt op een overflow plaatsvindt Cleart de timer counterwaarde Cleart de timer interruptflag

$TACCTLConfig()$ initialiseert CCTL register instantie Selecteert of deze in capture danwel compare zit Bepaalt juiste output mode Bepaalt of er een interrupt plaatsvindt Cleart de cc interruptflag

5.5 I2C

6 Help het werkt niet!

Wanneer je programma niet werkt, is het jouw taak als programmeur om dit op te lossen. Docenten kunnen je hierbij helpen, maar een belangrijke vaardigheid is om zelf problemen op te sporen, te kunnen isoleren en te verhelpen.

Veel voorkomende problemen, die wij door de jaren heen gezien hebben:

- De watchdogtimer wordt niet gestopt $WDTCTL = WDTPW|WDTHOLD$;
- De high-impedance mode van de GPIO wordt niet uitgezet $PM5CTL0\& =\sim LOCKLPM$
- Je gebruikt = i.p.v | =.
- Je gebruikt &= i.p.v. & =∼
- De global interrupt enable staat niet aan
- Je vergeet de flag van een interrupt uit te zetten in een grouped ISR.