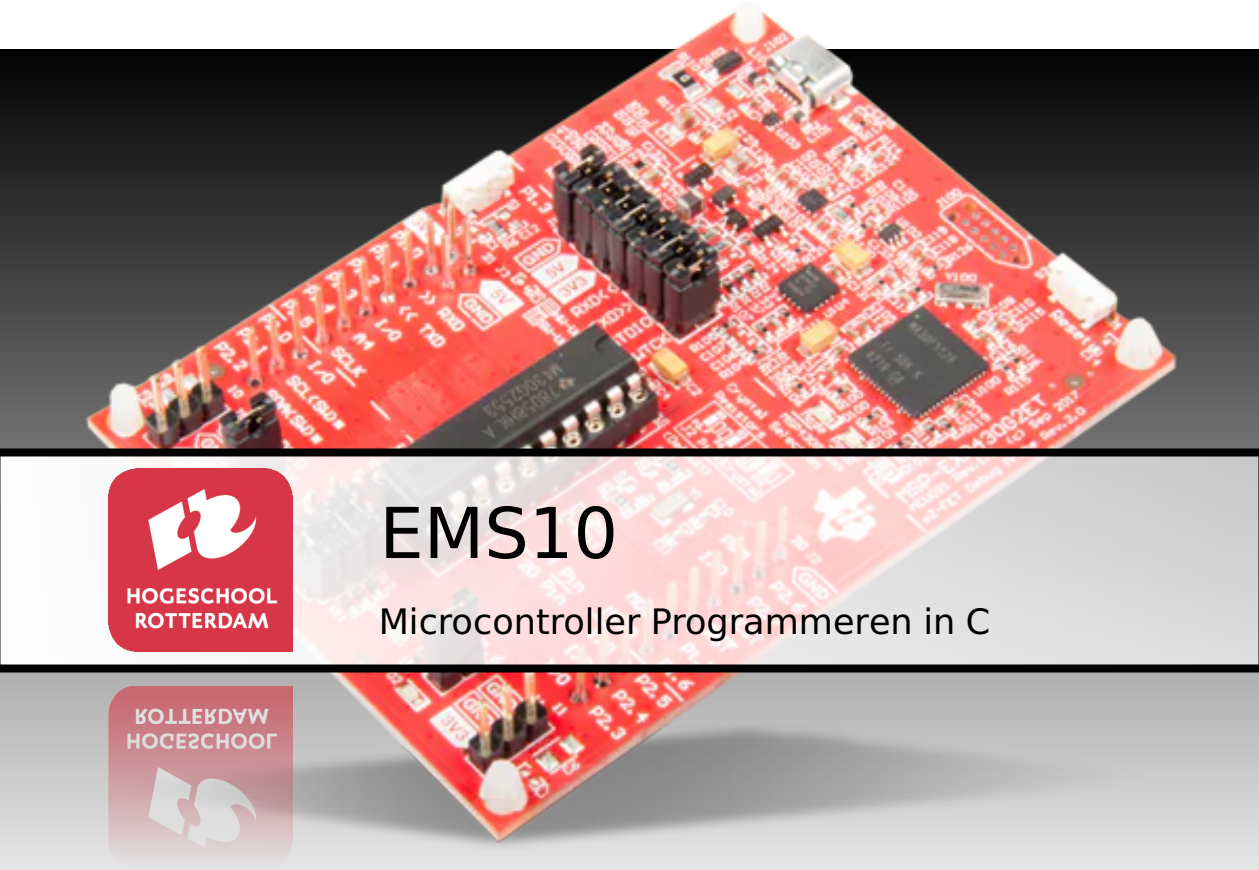


HOGESCHOOL ROTTERDAM

EMS10

Microcontroller Programmeren in C

Handboek

Versie 3.4a

D. Versluis

Versiehistorie

Datum	Versie	Omschrijving	Auteur
18-03-2022	3.4a	Bijlage ISR vectoren aangepast	BroJZ
27-02-2021	3.3	Aangepast voor nieuwe versie LaunchPad	BroJZ
02-03-2019	3.2	Isoleren van bits vervangen door maskeren van bits	BroJZ
02-03-2019	3.1	Diverse kleine verbeteringen	BroJZ
17-02-2018	3.0	Document is vertaald en omgezet naar de MSP430	VersD
20-08-2017	2.1	Added info to mask input bit	BroJZ
26-08-2016	2.0	Chronological chapter order	VlaMA
24-09-2015	1.0e	Simpified some parts	BroJZ
28-08-2015	1.0	Initial release version after peer-review	VersD
14-07-2015	0.1	Initial draft version	VersD

Inhoudsopgave

Voorbeeldcodes

1

Introductie

Dit handboek zal de student een introductie geven in de wereld van de 'embedded systems'. Dit zijn autonome systemen die slechts enkele simpele functies uitvoeren. Denk bijvoorbeeld aan de systemen in een moderne auto: de airbags moeten worden aangestuurd, de deursloten zijn elektronisch beveiligd, een ABS-systeem controleert de remmen enzovoorts. Tegenwoordig zijn er zelfs parkeerhulpen ingebouwd in de auto. Elk van deze systemen kan volledig autonoom en onafhankelijk van de andere systemen werken. Veel embedded systemen gebruiken erg weinig vermogen, zodanig weinig dat ze een lange tijd op een enkele batterij kunnen werken.

Embedded systemen hebben een mooie toekomst. Momenteel is het laatste buzzwoord 'Internet of Things'. Dit verwijst naar een grote hoeveelheid kleine (embedded) systemen die allemaal zijn verbonden met het internet. Miljoenen van deze systemen zouden een enorm netwerk van sensoren en actuatoren kunnen vormen. Denk aan de mogelijke toepassingen wanneer naar de statistieken van deze data wordt gekeken.

In dit deel van EMS10 zal de student de benodigde basis krijgen om een embedded systeem op te kunnen zetten. De student zal antwoord krijgen op diverse vragen zoals: "Hoe communiceer je met een embedded systeem?", "Hoe schrijf je de software ervoor?" en "Hoe kun je de in- en uitgangen lezen en aansturen?". Verschillende onderwerpen zullen worden behandeld: het aansturen van leds en andere actuatoren, en het uitlezen van knoppen en andere sensoren. Er zijn video's beschikbaar gemaakt die de student helpen om de technisch uitdagende onderwerpen te begrijpen.

De kern van een embedded systeem is vaak een microcontroller. Dit is een relatief eenvoudige processor die geïntegreerd is met geheugens en input- en outputmodulen, allemaal op één chip. Het programma, dat op de microcontroller is opgeslagen en door deze microprocessor wordt uitgevoerd, bepaalt het gedrag van het embedded systeem. Het vervolg van dit handboek zal de benodigde informatie verschaffen om te kunnen werken met microcontrollers.

2

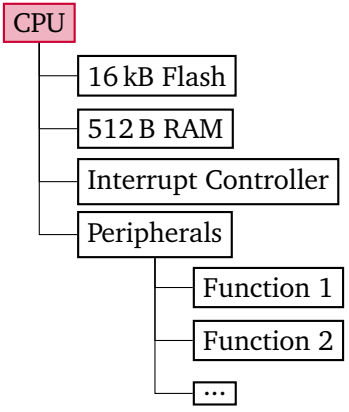
Microcontrollers

Dit hoofdstuk beschrijft de technologie achter microcontrollers.

2.1 Architectuur

Elk computersysteem heeft een centrale verwerkingseenheid nodig om te kunnen functioneren. Deze centrale verwerkingseenheid wordt in het Engels central processing unit of kortweg CPU genoemd. In het vervolg van dit handboek zullen we deze Engelse term gebruiken. Een embedded systeem bevat een kleine processor (8 tot en met 32 bit) die draait op een relatief lage frequentie (1 tot en met 100 MHz). Het geheugen waarmee deze kleine processor kan werken is veelal niet meer dan 32 kB aan RAM en zo'n 32 kB tot 1 MB aan ROM-geheugen (flash). Een pc bestaat uit meerdere onderdelen die losgekoppeld kunnen worden. Zo kan een geheugenmodule worden losgekoppeld en ook de harde schijf kan worden vervangen. Zelfs de CPU kan van het moederbord worden losgekoppeld. In een microcontroller (µC) zijn al deze onderdelen en nog een hoop extra schakelingen, geïntegreerd op een enkele chip. In deze cursus zul je leren werken met een µC van Texas Instruments: de MSP430G2xxx. Deze microcontrollers draaien op 16 MHz en hebben 16 kB aan flash geheugen en slechts 512 B aan RAM.

Een µC lijkt veel op een desktop-pc. Maar, als een µC geen keyboard heeft, hoe kan een gebruiker dan vertellen wat deze µC moet doen? En hoe kun je, zonder een beeldscherm, zien wat er gebeurt?



Figuur 2.1: Architectuur van een MSP430G2xxx microcontroller.

De processor (CPU) is een schakeling die berekeningen kan uitvoeren, beslissingen kan nemen en kan communiceren met de buitenwereld via andere schakelingen. De CPU moet echter wel worden geïnstrueerd wat te doen. Deze instructies worden ingelezen vanaf het flashgeheugen. De CPU begint met lezen op de geheugenplek met adres 0. Tenzij anders geïnstrueerd, zal de processor daarna elke opeenvolgende instructie van het flashgeheugen uitvoeren. Mocht er een instructie langskomen waarmee data moet worden onthouden, dan gebruikt de CPU het RAM-geheugen.

Naast deze geheugens is de processor ook verbonden met een hoop andere schakelingen die randapparaten worden genoemd. In het Engels (en dus in de datasheet) worden deze schakelingen 'peripherals' genoemd. Wij zullen in dit handboek in het vervolg dit Engelse woord gebruiken. De peripherals maken de microcontroller ook echt een 'controller' en voorzien deze van verschillende functionaliteiten. Het zijn elektronische, veelal digitale, schakelingen met specifieke functies waardoor de CPU kan worden ontlast.

Een µC kan bijvoorbeeld de volgende peripherals bevatten:

- digitale logica om een puls te genereren;
- elektronische schakeling om een analoog signaal in te lezen;
- logica om te kunnen communiceren via bepaalde protocollen (UART, SPI, I^2C, enzovoorts);
- logica waarmee tijd kan worden bijgehouden.

De CPU communiceert met deze schakelingen op dezelfde manier als met het geheugen. Er is een bepaald adres waarmee verschillende flip-flops kunnen worden ingesteld. Deze

flip-flops vormen vervolgens de in- en uitgangen van de peripherals (elektronische, veelal digitale, schakelingen). Hiermee kun je dus (de functie van) deze peripherals beïnvloeden! Paragraaf 2.3 zal hier dieper op ingaan.

Opdracht 1: CPU Peripherals

Gebruik de datasheet van de MSP430G2553 om uit te vinden welke peripherals zijn verbonden met de CPU.

2.2 Van instructies tot C-code

Voordat dieper ingegaan zal worden op een aantal peripherals, zal deze paragraaf beschrijven hoe een µC werkt. Om te beginnen zal de volgende vraag beantwoord worden: hoe worden de instructies voor de processor gevormd? Bij het Python-deel van deze cursus heb je geleerd dat de code wordt geïnterpreteerd door een stuk software, die vervolgens de processor (via het besturingssysteem) instrueert wat er moet gebeuren. Bij C-code werkt dit anders. Bij C is het nodig om eerst de code te compileren met een speciaal programma. Dit programma, de 'compiler', vertaalt de C-code naar instructies die de processor kan begrijpen. Bij het programmeren van een pc met de programmeertaal C, wordt de compiler eenmalig aangeroepen en daarna kan de software direct met de processor (via het besturingssysteem) communiceren zonder extra vertaalslag.

Bij het programmeren van een klein embedded systeem wordt de compiler niet op het embedded systeem zelf, maar op een pc uitgevoerd. Zo'n compiler wordt een 'cross compiler' genoemd. De door deze compiler geproduceerde code kan niet op een pc, maar wel op het betreffende embedded systeem worden uitgevoerd. Op het embedded systeem kan de code direct, zonder extra vertaalslag, worden uitgevoerd.

Deze code moeten dan eerst nog wel in het geheugen van de µC worden gezet. Wij maken gebruik van een zogenoemde 'integrated development environment' (IDE) die onder andere een editor en compiler bevat en waarmee je de gecompileerde code op het embedded systeem kunt laden en debuggen. We maken daarbij gebruik van een zogenoemd 'development board' van Texas Instruments dat MSP-EXP430G2ET LaunchPad wordt genoemd. De IDE communiceert met de logica op de LaunchPad om de instructies te schrijven naar het flashgeheugen van de te programmeren µC.

De geïntegreerde ontwikkelingsomgeving (IDE) genaamd Code Composer Studie (CCS) die wordt gebruikt is van Texas Instruments (TI), de producent van de µC die we gebruiken. CCS

is gebaseerd op Eclipse (een veelgebruikte open source IDE) en voorziet de gebruiker van alle benodigde tools om de microcontrollers van TI te kunnen programmeren en debuggen. In de opdrachten van EMS10 leer je CCS te gebruiken.

2.3 Peripherals

Om een echt programma te kunnen schrijven voor een µC is het nodig eerst meer te leren over hoe de CPU communiceert met alle peripherals binnen de µC. Zoals eerder beschreven in dit handboek, is de CPU verbonden met het geheugen en een aantal peripherals. De meeste peripherals zijn digitale schakelingen en dus opgebouwd uit logische poorten (AND, OR en NOT) en geheugenelementen (flip-flops). Exact zoals je geleerd hebt bij de cursus ELE10.

In dit geval zijn bepaalde in- en uitgangen van de schakelingen gekoppeld aan flip-flops en die flip-flops zijn gekoppeld aan de CPU via een bus. De CPU kan deze flip-flops bereiken via een bepaalde geheugenlocatie. Een stuk logica zorgt er daarbij voor dat een bepaalde geheugenlocatie wordt gekoppeld aan de juiste flip-flops. Dus door te schrijven naar een bepaald adres, worden bepaalde flip-flips hoog of laag gemaakt, waardoor de digitale schakeling van de peripheral een andere uitkomst zal krijgen, waardoor andere flip-flops weer een nieuwe waarde kunnen krijgen. De in- en uitgangen van de peripherals kunnen ook zijn gekoppeld aan de buitenwereld via elektronica bestaande uit transistors en weerstanden. Zo kan een microcontroller bijvoorbeeld een led aansturen of een knop inlezen. Het aansturen van een led gaat dan, vereenvoudigd weergegeven, bijvoorbeeld als volgt. Het programma dat wordt uitgevoerd op de CPU instrueert de CPU om te schrijven naar een bepaald adres. Dit adres is gekoppeld met een bepaalde peripheral die gebruikt kan worden om de I/O-pinnen van de µC aan te sturen. Deze schrijfactie zorgt ervoor dat bepaalde flip-flops in de peripheral geset of gereset worden. Het programma moet de juiste data naar het betreffende adres schrijven zodat een bepaalde flip-flop wordt geset. Deze flip-flop stuurt dan een transistor open die er voor zorgt dat er stoom kan gaan lopen door de uitgangspin van de µC die (via een weerstand) is verbonden met de led. Deze stroom zorgt ervoor dat de led gaat branden.

De peripherals gebruiken dus flip-flops, op specifieke geheugenlocaties, als in- en uitgangen. Deze flip-flops zijn, in de MSP430G2553, gegroepeerd in groepen van 8 of 16 bits die I/O-registers genoemd worden. Peripherals die met I/O-registers van 8 bits werken worden in de datasheet ‘peripherals with byte access’ genoemd. Peripherals die met I/O-registers van 16 bits werken worden in de datasheet ‘peripherals with word access’ genoemd. Elk groepje van 8 bits heeft een eigen geheugenlocatie. Nu rest alleen nog het beantwoorden van de volgende vragen. Over welke I/O-registers beschikt een bepaalde peripheral? Wat is de

betekenis van de verschillende bits in deze I/O-registers? Naar welke geheugenlocaties moet worden geschreven of gelezen om de I/O-registers van een bepaalde peripheral in te stellen of uit te lezen?

Opdracht 2: Geheugenadressen

Gebruik de datasheet van de MSP430G2553 om het bereik aan geheugenadressen te vinden waar de schakeling van de Timer0_A3 peripheral aan is gekoppeld.

In C-code is het mogelijk om de I/O-registers (op specifieke adressen) te benaderen via speciale namen. Het is de gewoonte dat de namen van deze registers met hoofdletters worden geschreven, de registernamen zelf zijn gedefinieerd in een zogenaamd *header*-bestand (`msp430.h` voor deze µC). Om bijvoorbeeld de waarde `0` te schrijven in het counter register van de Timer0_A3 schakeling kan deze C-code worden gebruikt:

```
TA0R = 0;
```

Hiermee worden alle flip-flops die de huidige tellerwaarde bevatten voor Timer0_A3 dus nul gemaakt.

De namen van de registers, die zijn gedefinieerd in het headerbestand, zijn hetzelfde als de namen die je kunt vinden in de datasheet van de microcontroller. Als de bits (flip-flops) van de registers de werking van de peripheral kan beïnvloeden, welke bits hebben dan welke functie?

Tot nu toe is de datasheet gebruikt om uit te zoeken welke peripherals zijn verbonden aan de processor en wat de adreslocaties van de verschillende registers zijn. Het is de vraag of de benodigde informatie over de verschillende bits van de registers ook is te vinden in de datasheets van de µC.

Alle benodigde informatie om de MSP430G2553 te kunnen programmeren is te vinden in twee verschillende documenten. Er is een algemeen document, de 'MSP430x2xx Family User's Guide', die geldig is voor een hele familie MSP430 microcontrollers. Hierin staan alle mogelijke peripherals en hun registers beschreven die een MSP430 microcontroller uit die specifieke familie kan bevatten. Er is ook een microcontroller specifiek deel: de datasheet. De datasheet bevat specifieke informatie over een bepaalde groep (binnen een familie) in ons geval de MSP430G2x53 en MSP430G2x13 microcontrollers. In deze datasheet wordt bijvoorbeeld beschreven welke peripherals in een bepaald type MSP430G2x53 of

MSP430G2x13 zijn te vinden en wat de pin-out is van de chips. Wij maken gebruik van de MSP430G2553.

De user's guide van de MSP430x2xx familie is modulair opgebouwd. Elke peripheral die te vinden is in deze familie heeft zijn eigen hoofdstuk. Binnen elk hoofdstuk is dan een algemene beschrijving te vinden over de betreffende peripheral. Er wordt uitgelegd welke stappen moeten worden ondernomen om de functionaliteit van de peripheral in te stellen zoals gewenst.

Misschien is de belangrijkste paragraaf in een hoofdstuk wel de paragraaf waar in de naam het woord "Registers" voorkomt. Deze paragraaf geeft alle registernamen voor de betreffende peripheral. Per register wordt vervolgens uitgelegd wat de functie is van elke bit binnen dat register. Als een peripheral onbekend voor je is, dan is het goed om te starten met de eerste paragrafen van het betreffende hoofdstuk.

In het volgende hoofdstuk volgt een algemene uitleg over de GPIO-peripheral van de MSP-430G2553.

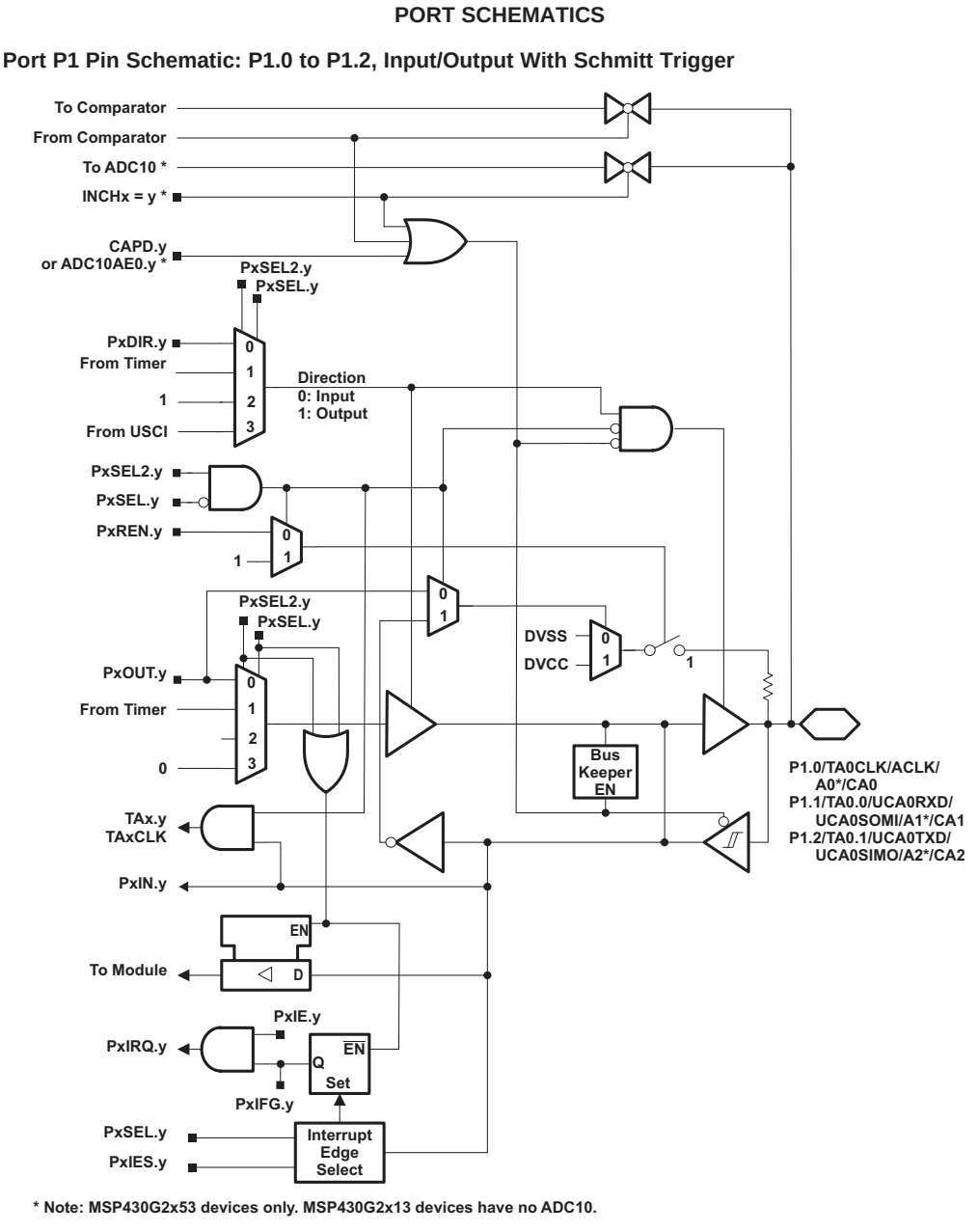
3

Digitale I/O

Er is een peripheral die aanwezig is in vrijwel elke microcontroller op de markt. Dit is de GPIO (General Purpose Input/Output) peripheral. Met hulp van deze peripheral kan de CPU elke I/O-pin aansturen die te vinden is op de µC. Hiermee zou je een pin kunnen instellen als ingang en de waarde op de pin kunnen inlezen (logische 1 of 0). Of je zou de pin als een uitgang kunnen instellen en een logische waarde kunnen schrijven naar de pin. Met deze peripheral kan elke taak worden uitgevoerd die te maken heeft met externe schakelingen. Het is de poort naar de buitenwereld.

De pinnen zijn opgedeeld in delen, zogenaamde poorten. De MSP430G2553 heeft een poort genaamd P1 en een poort genaamd P2. Elke poort heeft maximaal 8 pinnen. Hierdoor kunnen de pinnen aangestuurd worden met registers van 8 bits breed. Als bijvoorbeeld een led is gekoppeld aan P1.0 (poort P1, pin 0), dan kan deze led worden aangestuurd door bit 0 aan te passen van het betreffende poort 1 register. Om een schakelaar in te lezen op P2.2 (poort P2, pin 2), moet bit 2 worden uitgelezen van het betreffende poort 2 register.

Elke bit-positie in de GPIO-registers is dus gekoppeld aan een specifieke pin waarbij elke pin een eigen schakeling heeft. In figuur 3.1 is bijvoorbeeld de schakeling weergeven van P1.0 tot en met P1.2. Dit schema kun je terugvinden in de datascheet van de MSP430G2553. Daar kun je ook de schema's voor de overige I/O-pinnen van de MSP430G2553 vinden.



Figuur 3.1: De schakeling behorende bij P1.0 – P1.2.

3.1 De datarichting instellen

Wanneer een µC opstart, staat elke I/O-pin ingesteld als ingang. Dit voorkomt dat er per ongeluk een externe schakeling wordt aangestuurd en er gevaarlijke situaties ontstaan. Om een pin in te stellen als uit- of ingang kan gebruik worden gemaakt van het `PxDIR`-register. Om bijvoorbeeld P2.3 in te stellen als uitgang, moet bit 3 van het `P2DIR`-register op `1` worden gezet. Moet P2.2 een ingang zijn, dan moet bit 2 van het `P2DIR`-register `0` worden gemaakt. Dit staat overigens nog los van de spanning op de pin. Een uitgangspin kan een lage of hoge spanning hebben en ook een ingangspin ook een lage of hoge spanning hebben.

Als bijvoorbeeld `P1DIR` deze waarde krijgt: `0b00000000`[1], dan is elke pin van poort P1 een ingangspin. Als `0b11111111` wordt geschreven naar het `P1DIR`-register dan is elke pin van poort P1 een uitgangspin. Als `0b10101010` wordt geschreven naar `P1DIR`, dan zijn bepaalde pinnen van poort P1 uitgangspinnen en andere pinnen van poort P1 ingangspinnen. Om precies te zijn, P1.7, P1.5, P1.3 en P1.1 zijn dan uitgangspinnen en P1.6, P1.4, P1.2 en P1.0 zijn ingangspinnen.

Table 8-2. Digital I/O Registers

Port	Register	Short Form	Address	Register Type	Initial State
P1	Input	P1IN	020h	Read only	-
	Output	P1OUT	021h	Read/write	Unchanged
	Direction	P1DIR	022h	Read/write	Reset with PUC
	Interrupt Flag	P1IFG	023h	Read/write	Reset with PUC
	Interrupt Edge Select	P1IES	024h	Read/write	Unchanged
	Interrupt Enable	P1IE	025h	Read/write	Reset with PUC
	Port Select	P1SEL	026h	Read/write	Reset with PUC
	Port Select 2	P1SEL2	041h	Read/write	Reset with PUC
	Resistor Enable	P1REN	027h	Read/write	Reset with PUC
P2	Input	P2IN	028h	Read only	-
	Output	P2OUT	029h	Read/write	Unchanged
	Direction	P2DIR	02Ah	Read/write	Reset with PUC
	Interrupt Flag	P2IFG	02Bh	Read/write	Reset with PUC
	Interrupt Edge Select	P2IES	02Ch	Read/write	Unchanged
	Interrupt Enable	P2IE	02Dh	Read/write	Reset with PUC
	Port Select	P2SEL	02Eh	Read/write	0C0h with PUC
	Port Select 2	P2SEL2	042h	Read/write	Reset with PUC
	Resistor Enable	P2REN	02Fh	Read/write	Reset with PUC

Figuur 3.2: De digital I/O peripheral registers.

[1] De prefix `0b` kan in de C-compiler voor de MSP430 gebruikt worden om een getal in binaire notatie in te voeren. In standaard C is dit echter *niet* mogelijk. Daarom is het beter om altijd de hexadecimale notatie te gebruiken, in dit geval `0x00` omdat deze notatie door elke standaard C-compiler wordt ondersteund. Bekijk eventueel de video over getalnotaties.

3.2 Overige instellingen

Wanneer de richting van de pin is ingesteld moet echter nog wel een spanning worden ingesteld of gelezen. Dit is mogelijk met respectievelijk de PxOUT- en PxIN-registers. Zoals in figuur 3.2 is te zien zijn er 9 verschillende I/O-registers beschikbaar per I/O-poort. In paragraaf 8.2 van de user's guide is gegeven welke functies de bits hebben in de betreffende registers.

De functies van de belangrijkste registers zijn gegeven in tabel 3.1.

Tabel 3.1: I/O-registers voor GPIO.

Register	Gebruik dit register om ...
PxDIR	... een pin in te stellen als uitgangpin of als ingangspin.
PxIN	... een spanning in te lezen van een bepaalde ingangspin (hoog of laag).
PxOUT	... een spanning in te stellen op een bepaalde uitgangspin (hoog of laag) of ... te kiezen tussen een pull-down of een pull-up weerstand als deze is aangeschakeld voor een ingangspin (zie ook PxREN).
PxSELx	... een specifieke pin te koppelen aan een andere functie of peripheral. Bijvoorbeeld aan de timer peripheral. Zie figuur 3.1.
PxREN	... de pull-up of pull-down weerstand aan te zetten voor een specifieke ingangspin.

Listing 3.1 laat zien hoe de pinnen van poort P1 als uitgang worden ingesteld en om en om laag (0 V) en hoog (3,0 V) worden gemaakt.

```
1 int main(void)
2 {
3     WDTCTL = WDTPW | WDTHOLD; // Stop de watchdog timer
4
5     P1DIR = 0b11111111; // Configureer alle pinnen van poort P1 ↩
  ↪ als uitgang
6     P1OUT = 0b10101010; // Stel de pinnen om en om in op een hoge ↩
  ↪ en lage spanning
7
8     return 0;
9 }
```

Listing 3.1: Voorbeeldprogramma om de pinnen van poort P1 om en om laag en hoog te maken.

Listing 3.2 laat zien hoe logische waarden kunnen worden ingelezen.

```
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop de watchdog timer

    P1DIR = 0b00000000; // Configureer alle pinnen van poort P1 als ingang

    if (P1IN == 0b10101010)
    {
        // De code binnen de if wordt alleen uitgevoerd wanneer P1IN gelijk is aan 0b10101010.
    }
    return 0;
}
```

Listing 3.2: Voorbeeldprogramma om te controleren of de logische waarden op de pinnen van poort P1 overeenkomen met 0b10101010.

4

Bit-operaties

Naast de interne werking van de µC is het ook belangrijk om een aantal software-aspecten te leren. Dit hoofdstuk legt uit hoe individuele bits binnen een byte kunnen worden aangepast. Met bit-operaties kunnen individuele bits worden aangezet, uitgezet en geïnverteerd of de waarde van een individuele bit kan worden bekeken.

4.1 Bitwise-operatoren

Een operator is een commando waarmee een specifieke functie wordt uitgevoerd en is gekoppeld aan een speciaal karakter of woord. De C taal voorziet in een aantal mogelijke bitwise-operatoren. Deze operatoren zijn gegeven in tabel 4.1. In deze tabel heeft `iVar` de waarde `0b00010100`.

Tabel 4.1: Bitwise operatoren met `iVar = 0b00010100`.

Operator	Beschrijving	Voorbeeld	Resultaat
<<	Schuif alle bits naar links	`result = iVar << 3`	0b10100000
>>	Schuif alle bits naar rechts	`result = iVar >> 4`	0b00000001
&	Bitwise AND	`result = iVar & 0x0F`	0b00000100
^	Bitwise XOR	`result = iVar ^ 0x0F`	0b00011011
|	Bitwise OR	`result = iVar | 0x0F`	0b00011111
~	Bitwise NOT	`result = ~iVar`	0b11101011

4.2 Bits aanzetten

Een enkele bit 1 maken is een veel voorkomende operatie. We noemen dit het *setten* van een bit. De moeilijkheid hierbij is om bij het instellen niet meteen alle andere bits aan te passen. Stel de 8-bit variabele `iVar` heeft de waarde `0b00010100` en je wilt alleen de meest significante bit (bit 7) 1 maken. Je wilt niet dat andere bits ook worden aangepast. Om dit voor elkaar te krijgen moeten er bitwise-operatoren gebruikt worden.

Eerst moet bit 7 worden geïsoleerd van de andere bits. Je kunt ook zeggen dat alle bits behalve bit 7 gemaskeerd moeten worden. Hiervoor kan een variabele `mask` worden aangemaakt[2].

```
uint8_t mask = 0b10000000;
```

Of om hetzelfde te bereiken op een meer leesbare wijze kan de shift-operator << worden toegepast:

```
uint8_t mask = 1<<7; // dit resulteert ook in 0b10000000
```

Nu een masker waarin alle bits behalve bit 7 gemaskeerd worden, kan de OR-operator worden gebruikt om alleen bit 7 1 te maken.

```
iVar = iVar | mask;
```

Stel dat de 8-bit variabele `iVar` de waarde `0b00010100` had voor de OR-operatie dan is nadien de waarde `0b10010100`.

Deze regel kan ook korter worden genoteerd:

```
iVar |= mask;
```

Het kan nog compacter door ook het masker in dezelfde regel aan te maken:

```
iVar |= 1<<7;
```

Natuurlijk is deze manier niet gelimiteerd tot 1 bit aanzetten. Om bijvoorbeeld bit 6, bit 3 en bit 0 aan te zetten en de overige bits niet aan te passen, kan de volgende code worden gebruikt:

```
uint8_t mask = 0b01001001;
iVar = iVar | mask;
```

[2] Deze variabele is van het type `uint8_t`. Zie bijlage B als je dit type nog niet kent.

Of compacter:

```
iVar |= 1<<6 | 1<<3 | 1<<0;
```

4.3 Bits uitzetten

Het tegenovergestelde van een bit 1 maken is een bit 0 maken. We noemen dit het *resetten* van een bit. Deze operatie is iets lastiger omdat er twee operatoren voor nodig zijn: de &- en ~-operator. Wanneer de &-operator wordt gebruikt moet de waarde van het te isoleren bit 0 zijn. Omdat het een AND-bewerking betreft, moeten de andere bits in het masker 1 zijn om de waarde niet aan te passen. Want $1 \cdot A = A$. Om alle bits behalve bit 2 te maskeren voorafgaande aan een AND-bewerking kun je het volgende doen:

```
uint8_t mask = 0b11111011;
```

Een slimmere en meer leesbare manier om de variabele mask te initialiseren is om de bitwise NOT-operator ~ te gebruiken:

```
uint8_t mask = ~(1<<2); // dit resulteert ook in 0b11111011
```

De haakjes zijn nodig omdat de ~-operator een hogere prioriteit heeft dan de <<-operator[3].

Nu kun je de AND-operator toepassen om de niet gemaskeerde bit 0 te maken in iVar:

```
iVar = iVar & mask;
```

Stel iVar was 0b00010100 voor de AND-operatie, dan is nu de waarde 0b00010000.

Deze regel kan ook korter worden genoteerd:

```
iVar &= mask;
```

Het kan nog compacter door ook het masker in dezelfde regel aan te maken:

```
iVar &= ~(1<<2);
```

[3] Zie voor de prioriteiten van de verschillende operatoren in C: http://en.cppreference.com/w/c/language/operator_precedence.

Natuurlijk is deze manier niet gelimiteerd tot 1 bit uitzetten. Om bijvoorbeeld bit 6, bit 3 en bit 0 uit te zetten en de overige bits niet aan te passen, kan de volgende code worden gebruikt:

```
uint8_t mask = 0b10110110;
iVar = iVar & mask;
```

Of compacter:

```
iVar &= ~(1<<6 | 1<<3 | 1<<0);
```

4.4 Inverteren van bits

Met hulp van de XOR-operator `^` is het mogelijk om bepaalde bits te inverteren, ook wel *toggling* of *flipping* genoemd. Om bijvoorbeeld bit 6, bit 3 en bit 0 te inverteren en de overige bits niet aan te passen, kan de volgende code worden gebruikt:

```
uint8_t mask = 0b01001001;
iVar = iVar ^ mask;
```

Of compacter:

```
iVar ^= 1<<6 | 1<<3 | 1<<0;
```

4.5 Een bit testen

Om een enkele bit te bekijken is het zaak om de overige bits te maskeren op de gebruikelijke manier. Met de AND-operator kan dan worden gekeken of alleen deze bit 1 is.

Als voorbeeld, om alleen bit 6 te bekijken van `iVar`:

```
if ((iVar & 1<<6) == 1<<6) {
    // code die wordt uitgevoerd als bit 6 van iVar 1 is
}
// of:
if (iVar & 1<<6)
{
    // code die wordt uitgevoerd als bit 6 van iVar 1 is
}
```

De haakjes in de voorwaarde van de eerste **if**-instructie zijn nodig omdat de ==-operator een hogere prioriteit heeft dan de &-operator.

5

Ontdenderen

Dit hoofdstuk legt verschillende software technieken uit om knoppen te ontdenderen. Het woord ontdenderen suggereert dat er iets aan het denderen is en daarmee moet stoppen. Dit denderen (Engels: bouncing) wordt veroorzaakt door de veertjes binnen knoppen. Wanneer een mechanische knop wordt ingedrukt of losgelaten zal de veer het contact laten denderen en hiermee een aantal keren een elektrische connectie maken. Hierdoor zou een enkele druk kunnen worden gezien (door de µC) als het meerdere keren, snel achter elkaar, indrukken van de knop.

Figuur 5.1: Mogelijk signaal van een denderende knop.

Er zijn verschillende manieren om het signaal afkomstig van een denderende knop te ontdenderen. De gemakkelijkste manier is een condensator en weerstand toe te voegen om hiermee een laagdoorlaatfilter te realiseren. Hoogfrequent gedender wordt dan weggefilterd.

In dit hoofdstuk worden (goedkopere) softwarematige oplossingen behandeld. Het idee blijft hetzelfde: filter de snelle veranderingen uit het signaal. De beschreven methodes zijn afgeleid uit *Debounce Code - One post to rule them all* [1].

Vanaf nu zal het Engelse woord 'debouncing' worden gebruikt. Elke methode van debouncing moet meerdere malen naar de knopwaarde kijken. Het liefst wordt dit gekoppeld aan een counter/timer schakeling, maar het kan ook in een simpele **while**-loop.

5.1 Knoppen verwerken

Naast het debouncen van een signaal moet het signaal nog steeds worden geïnterpreteerd door de software. Als we willen reageren op het indrukken van de knop, dan kan beter niet naar de gefilterde waarde van input-pin worden gekeken, maar alleen naar de verandering hiervan. We zijn geïnteresseerd in de flank van het knopsignaal. Dit interpreteren van het knopsignaal staat volledig los van het debouncen. Debouncen levert een gefilterd knopsignaal op en vervolgens moet naar het gefilterde signaal worden gekeken om te bepalen wat er moet gebeuren.

5.2 Debounce voorbeeld 1: Vertraging

Deze methode is gebaseerd op een blokkerende wachtfunctie. Dit betekent dat zolang de wachtfunctie wordt uitgevoerd, de CPU hiermee bezig is en niets anders kan doen. Deze methode is hierdoor alleen geschikt voor eenvoudige projecten. Elke toepassing waarbij meer wordt gedaan dan het schakelen van een led heeft een betere oplossing nodig.

```
1  ...
2      while(1)
3      {
4          WDTCTL = WDTPW | WDTHOLD; // Stop de watchdog timer
5
6          if (leesKnop()) // Als de knop ingedrukt is
7          {
8              wacht_ms(20); // Even wachten
9              if (leesKnop()) // Als de knop nog steeds ingedrukt is
10             {
11                 // Doe iets
12             }
13         }
14     }
15 ...
```

Listing 5.1: Debounce voorbeeld 1.

De benodigde wachttijd van de blokkerende wachtfunctie in listing 5.1 is voor elke knop verschillend. Het denderen kan maximaal ergens tussen de 10 ms en 160 ms duren [2]. Maar bij de meeste schakelaars blijft de dendertijd beperkt tot hooguit 10 ms. We zouden bijvoorbeeld een seconde kunnen wachten zodat we helemaal zeker weten dat het denderen achter de rug is, maar dit betekent dat de gebruiker de knop een seconde lang moet indrukken voordat er iets gebeurd. Dat laatste is natuurlijk niet wenselijk. Om die reden wordt er meestal gebruik gemaakt van een wachttijd van ongeveer 20 ms.

5.3 Debounce voorbeeld 2: Integrator

Een nettere manier om te debouncen wordt in deze paragraaf uitgelegd. Het doel van listing 5.2 is om het knopsignaal te integreren door een variabele op te hogen als het knopsignaal hoog is en te verminderen als het knopsignaal laag is. Wanneer de integrator variabele een maximum bereikt is de knop echt ingedrukt. Wanner de integrator weer `0` is, wordt de knop gezien al losgelaten. Ook deze code gaat ervan uit dat het herhaaldelijk wordt aangeroepen, idealiter op basis van een timer interrupt, bijvoorbeeld elke ms. Interrupts en timers worden later uitgelegd in respectievelijk hoofdstuk 6 en hoofdstuk 7.

```
1  /************************************************************
2  debounce.c - written by Kenneth A. Kuhn
3  ************************************************************/
4  ...
5      // Gebruikte variabelen:
6      uint16_t integrator; // De integrator variabele
7      uint8_t  gefilterdeKnop; // De gefilterde knopwaarde
8
9      // Maximum waarde voor de integrator variabele
10     const uint16_t MAXIMUM = ....;
11
12     if (leesKnop()) // Als de knop ingedrukt is
13     {
14         if (integrator < MAXIMUM) //en integrator is kleiner dan ↩
   ↪ MAXIMUM
15         {
16             integrator++; // verhoog integrator
17         }
18     }
19     else  // Als de knop niet ingedrukt is
20     {
```

```
21          if (integrator > 0) // en integrator is groter dan 0
22          {
23              integrator--; // verlaag integrator
24          }
25      }
26      if (integrator == MAXIMUM) // bovengrens bereikt
27      {
28          gefilterdeKnop = 1;
29      }
30      else if (integrator == 0) // ondergrens bereikt
31      {
32          gefilterdeKnop = 0;
33      }
34  ...
```

Listing 5.2: Debounce voorbeeld 2.

De benodigde maximum waarde voor de integrator variabele in listing 5.2 is afhankelijk van de maximale dendertijd van een knop en de herhalingsfrequentie waarmee de integrator wordt aangeroepen.

6

Interrupts

Verschillende peripherals in de µC kunnen ook op eigen houtje communiceren met de processor. Deze communicatie is echter beperkt tot een enkel signaal. Zo'n signaal kan bijvoorbeeld aangeven dat de timer 1 ms heeft gemeten, of dat de analoog-digitaal converter (ADC) een nieuwe sample klaar heeft staan.

Maar wat gebeurt er precies als de CPU wordt gesignaleerd? Hoe reageert de software dan op zo'n signaal?

In de µC zit een interruptsysteem. Elke keer als een signaal binnenkomt van een peripheral, dan zorgt het interruptsysteem ervoor dat de processor springt naar een functie om dit signaal af te kunnen handelen. Zo'n functie heet een interruptserviceroutine (ISR) en is een specifieke C-functie die wordt geschreven door de programmeur. De programmeur moet bij het definiëren van de functie aangeven dat het om een ISR gaat en bij welke interrupt deze ISR uitgevoerd moet worden.

Hoewel de term 'signaal' veel gebruikt wordt, is het voor programmeurs normaal om het te hebben over 'interrupts' (Nederlands: onderbrekingen). Hoe weet het interruptsysteem welke functie (ISR) moet worden uitgevoerd? Een interrupt van de ADC moet immers anders worden afgehandeld dan een interrupt van een timer.

Zoals eerder besproken staan de instructies voor de CPU in het flashgeheugen. Het interruptsysteem maakt gebruik van een tabel waarin voor elke interruptbron een functielocatie is gedefinieerd. Deze tabel bevindt zich op een vaste plaats in het flashgeheugen. Wanneer een interruptsignaal binnenkomt bekijkt het interruptsysteem welke functielocatie hoort bij de

bron van die specifieke interrupt. Deze functielocatie wordt doorgegeven aan de CPU en de CPU begint op deze locatie instructies uit te voeren. Na afloop van de ISR gaat de CPU verder op de plaats waar de uitvoering van het programma was onderbroken door de interrupt. Het is aan de linker (dat is een onderdeel van de compiler) om op de juiste locaties de juiste functies en verwijzingen te zetten. Deze functieverwijzingen worden interruptvectoren genoemd. Zie bijlage A voor een opsomming van alle interruptvectoren van de MSP430G2553. In listing 6.1 is een voorbeeld van een ISR gegeven.

```
1  #pragma vector = ADC10_VECTOR
2  __interrupt void mijnADCInterruptRoutine(void)
3  {
4      // deze code wordt uitgevoerd bij een ADC interrupt
5  }
6
7  int main(void)
8  {
9      // ...
10     ADC10CTL0 |= ADC10IE; // lokale interrupt enable van de ADC10 ↩
   ↪ peripheral
11     __enable_interrupt(); // zet interruptsysteem aan
12     // ...
13 }
```

Listing 6.1: Voorbeeld van een interruptserviceroutine.

Om het interruptsysteem in te schakelen is een macro (functie) beschikbaar gemaakt genaamd `__enable_interrupt`. Deze macro moet worden aangeroepen voordat interrupts verwerkt kunnen worden, zie listing 6.1.

Een lokale 'interrupt enable'-bit kan gebruikt worden om een specifieke interruptbron in te schakelen. Dit bit kan meestal worden gevonden in het configuratieregister van de betreffende peripheral. Het kan zo zijn dat er slechts één interruptvector beschikbaar is voor meerdere interruptbronnen. Dit is bijvoorbeeld het geval bij de interrupts voor de pinnen. In zo'n geval moet binnen de ISR worden gekeken welke pin de interrupt heeft veroorzaakt. Dit wordt aangegeven door zogenoemde interrupt flags die te vinden zijn in een bepaald register van de betreffende peripheral.

7

Timers

In dit hoofdstuk wordt de algemene werking van de timer peripheral uitgelegd. Een timer is een simpele binaire counter (net als bij ELE10) die optelt of aftelt met een bepaalde frequentie. Het kloksignaal wat deze timer krijgt, is vaak afgeleid van de CPU klok. In het configuratieregister kan bijvoorbeeld de klokbron worden gekozen en de frequentie hiervan worden gedeeld door 8, 4, 2 of 1. Dit bepaalt de frequentie van de timer. Op basis van deze frequentie is het dan mogelijk om een verschil in tijd bij te houden (Δt).

In plaats van de CPU bezig te houden met het controleren van de timerwaarde kan de timer zelf de processor signaleren via het interruptsysteem. Het is bijvoorbeeld mogelijk om een interrupt te genereren wanneer de timer de waarde 100 heeft bereikt. Deze waarde is te zetten in het capture/compare register. Door dan de capture/compare interrupt enable bit in te schakelen geeft de timer peripheral een signaal wanneer de timerwaarde 100 heeft bereikt.

8

State Machines

Zoals in de introductie is beschreven, voeren embedded systemen slechts simpele taken uit. Maar zelfs een simpele taak moet soms meerdere sensoren uitlezen en meerdere actuatoren aansturen. Vaak gebeurt dit niet op hetzelfde moment. Wat dan nodig is, is een manier om tijd en toestanden te kunnen beheren.

Een mogelijke manier om de tijd/toestanden te managen is een toestandsmachine (Engels: state machine). Een toestandsmachine kan worden geïmplementeerd met een `switch-case`-structuur, zie listing 8.1. Deze structuur kan herhaaldelijk worden aangeroepen (bijvoorbeeld elke 50 ms) om de huidige toestand te evalueren en eventueel te schakelen naar een andere toestand. Op deze manier heeft de programmeur een framework om de tijd te kunnen beheren. Deze manier werkt beter dan bijvoorbeeld `while(wachtOpGebeurtenis)`. Wachten kan in een toestandsmachine gedaan worden door de iteraties te tellen. Na 20 iteraties van 50 ms is bijvoorbeeld 1 seconde voorbij gegaan.

Het is belangrijk om er zeker van te zijn dat de code binnen de cases niets blokkeert. Het gebruikt van `while()` moet dus worden voorkomen binnen de toestandsmachine.

```
// Enumeratie van alle mogelijke toestanden
typedef enum {BLINK, COUNT, WAIT, STOP} toestand;

...
{
    ...
    toestand huidigeToestand = STOP;

    while (1)
    {
        // Onderstaande code telkens uitvoeren na een signaal van bijvoorbeeld de timer
        switch (huidigeToestand)
        {
            case STOP:
                // Code die moet worden uitgevoerd in toestand STOP.
                if (eenGebeurtenis) huidigeToestand = BLINK;
                break;
            case BLINK:
                // Code die moet worden uitgevoerd in toestand BLINK.
                break;
            case ...
        }
    }
}
```

Listing 8.1: Switch-case structuur om een toestandsmachine te implementeren.

Bibliografie

[1] Hack A Day. *Debounce Code - One post to rule them all*. 2010. URL: http://hackaday.com/2010/11/09/debounce-code-one-post-to-rule-them-all/ (geciteerd op p. 23).

[2] Jack G. Ganssle. *A Guide to Debouncing*. 2008. URL: https://pubweb.eng.utah.edu/~cs5780/debouncing.pdf (geciteerd op p. 25).

A

ISR Vectoren

In tabel A.1 zijn een aantal veelgebruikte interruptvectoren van de MSP430G2553 te vinden. Deze vectoren zijn gedefinieerd in de include-file `msp430g2553.h`.

Tabel A.1: Veelgebruikte interruptvectoren van de MSP430G2553.

Vector	Interrupt afkomstig van:
`PORT1_VECTOR`	Port 1 pin change
`PORT2_VECTOR`	Port 2 pin change
`ADC10_VECTOR`	ADC10
`USCIAB0TX_VECTOR`	USCI A0/B0 Transmit
`USCIAB0RX_VECTOR`	USCI A0/B0 Receive
`TIMER0_A1_VECTOR`	Timer0_A CC1, CC2, TA0
`TIMER0_A0_VECTOR`	Timer0_A CC0
`TIMER1_A1_VECTOR`	Timer1_A CC1, CC2, TA1
`TIMER1_A0_VECTOR`	Timer1_A CC0

B

Verschillende datatypen

In dit handboek worden een aantal datatypes gebruikt die je misschien nog niet eerder hebt gezien. Deze bijlage heeft als doel deze datatypen uit te leggen. Bij het includeren van <stdint.h> in de code ontstaat de mogelijkheid om verschillende speciale datatypen te gebruiken. Deze datatypen zijn gedefinieerd aan de hand van de hoeveelheid bits en het numerieke bereik.

Bijvoorbeeld:

```
#include <stdint.h>
...
    uint8_t var1;  // Bereik: 0 t/m 255
    uint16_t var2; // Bereik: 0 t/m 65535
```

Dit stuk code maakt twee variabelen aan, var1 heeft een bitbreedte van 8 en var2 heeft een bitbreedte van 16. De letter u in de naam van het variabeletype betekent dat het een **unsigned** (positief) type is.

Je kunt ook een **signed** (two's complement) variabelen definiëren. Bijvoorbeeld:

```
    int8_t var3;  // Bereik: -128 t/m +127
    int16_t var4; // Bereik: -32768 t/m +32767
```

Het gebruik van deze typen stelt de programmeur in staat om systeemonafhankelijke code te schrijven. Als de typenaam **int** wordt gebruikt dan bepaalt het systeem waarvoor de code wordt gecompileerd, hoeveel bits de variabele is. Op een microcontroller kan het een 16-bit waarde zijn, maar op een pc kan het een 32-bit waarde zijn! Bij het programmeren van

embedded systemen is het belangrijk om zelf het aantal bits van een variabele te kunnen bepalen. Vaak is de programmeur namelijk bezig om direct de hardware aan te spreken, dan is het dus handig om te weten hoeveel bits een variabele bevat. Bovendien heeft het invloed op de snelheid: twee 64-bits integers optellen op een 16-bit systeem kost veel meer kloktikken dan twee 16-bit integers optellen.