

ATmega32

*software-ontwikkeling in
assembler en C*

Jesse op den Brouw

Eerste druk

Deze pagina is opzettelijk leeg gelaten.

1

Interrupts

In dit hoofdstuk worden interrupts en het interruptmechanisme van de ATmega32 behandeld.

1.1 Polling

De ATmega32 kan met behulp van een wachtlus reageren op signalen van externe bronnen. De externe bronnen kunnen zo aan de processor aangeven dat er bijvoorbeeld data beschikbaar is en dat afhandeling is gewenst. Dat werkt als volgt. De ATmega32 leest een byte in van een van de I/O-poorten. Vervolgens wordt de bit die getest moet worden met behulp van een bitmasker gefilterd. Is de geteste bit niet actief, dan wordt opnieuw een byte ingelezen en begint het testen opnieuw. Is de geteste bit wel actief, dan wordt de wachtlus verlaten en wordt de rest van het programma uitgevoerd. Deze methode wordt *polling* genoemd. In het vakgebied van de Operating Systems wordt dit *busy waiting* genoemd.

In listing 1.1 is een voorbeeld van zo'n wachtlus te zien. Eerst wordt de status van de pinnen van Port A ingelezen door middel van een `in`-instructie. De `andi`-instructie zorgt ervoor dat de minst significante bit wordt gefilterd; de overige bits worden op 0 gezet. De `andi`-instructie past ook de Z-vlag aan. Als blijkt dat de minst significante bit een 0 is dan wordt de Z-vlag op 1 gezet, anders wordt de Z-vlag op 0 gezet (merk op dat de overige bits op 0 gezet zijn). Als blijkt dat de minst significante bit een 0 is, dan zorgt de `breq`-instructie ervoor dat weer naar het begin van de wachtlus wordt gesprongen. Is de minst significante bit een 1, dan wordt de lus verlaten en wordt de eerstvolgende instructie uitgevoerd.

```
1 loop:    in    r16,PINA    ; read in pins Port A
2          andi  r16,0x01    ; filter LSB, Z-flag = 1 if bit = 0
3          breq  loop        ; again if bit is 0
```

Listing 1.1: Polling.

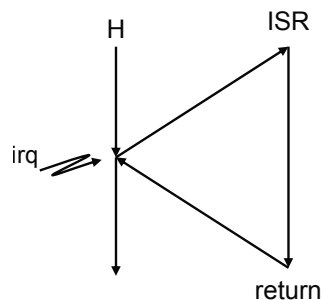
Deze manier van detectie werkt eenvoudig en snel. Het programma (de wachtlus) is eenvoudig en er zijn slechts enkele klokpulsen nodig om de status van een extern signaal te bepalen. Het nadeel is natuurlijk dat de processor niets anders kan doen.

1.2 Interrupts

Slimmer is om de hardware aan de processor door te laten geven dat de status van een signaal is veranderd, zodat de processor direct actie kan ondernemen. Het lopende programma wordt dan *geïnterrupteerd* (onderbroken). Dit wordt een *interrupt* genoemd. Een aanvraag voor een onderbreking (het signaal) wordt een *interrupt request* (IRQ) genoemd.

Nadat een interrupt request door de processor is herkend, wordt het lopende programma onderbroken (de huidige instructie wordt afgemaakt) en wordt een *interrupt service routine* (ISR) uitgevoerd. De ISR handelt dan de interrupt verder af. Daarna gaat de processor verder waar het gebleven was. Een andere naam is *Interrupt Handler*.

We beelden dit uit in figuur 1.1. Hierin stelt H de uitvoer van het lopende programma voor. Op een gegeven moment wordt er een interrupt geregistreerd door de processor. Het lopende programma wordt verlaten en de processor gaat de ISR uitvoeren. Aan het einde van de ISR wordt teruggekeerd naar het lopende programma H.



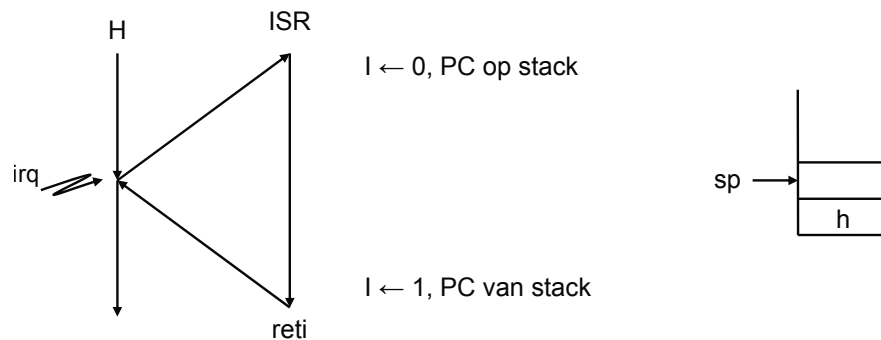
Figuur 1.1: Eenvoudige voorstelling van een interruptafhandeling.

ISR's lijken veel op gewone subroutines, maar er zijn verschillen:

- Een ISR wordt gestart door een interrupt, niet door een instructie¹.
- Direct nadat de interrupt is herkend, wordt de Global Interrupt Enable vlag (de I-vlag) in het Status Register op 0 gezet. Dit zorgt ervoor dat de ISR niet kan worden onderbroken door een (nieuwe) interruptaanvraag.
- Voor terugkeer moet de instructie `reti` (Return From Interrupt) gebruikt worden. Deze instructie zet de Global Interrupt Enable vlag weer aan.

Interrupts en subroutines hebben wel één overeenkomst: in beide gevallen wordt de program counter (PC) op de stack gezet. Dit wordt uitgebeeld in figuur 1.2. Omdat het terugkeeradres op de stack wordt gezet, moet de stack pointer (SP) worden geïnitieerd vóórdat de interrupts mogen worden afgehandeld. Dat betekent dat direct na het starten van het lopende programma, bijvoorbeeld na een reset, de interrupts nog niet mogen worden afgehandeld omdat de stack pointer nog niet is geïnitieerd.

¹ Er zijn processoren die instructies hebben die software-interrupts genereren, zoals de Intel x86-familie.



Figuur 1.2: Interruptafhandeling: de program counter wordt op de stack gezet.

Twee instructies beïnvloeden direct de I-flag:

`sei` - set Global Interrupt Enable flag, interrupts worden afgehandeld.

`clic` - clear Global Interrupt Enable flag, interrupts worden geblokkeerd.

De I-vlag is gepositioneerd op bit 7 van het Status Register (SREG). Zie figuur 1.3.

7	6	5	4	3	2	1	0
I	T	H	S	V	N	Z	C
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Figuur 1.3: Het Status Register SREG.

Pas na het initialiseren van de stack pointer mag de instructie `sei` gegeven worden.

1.3 Interruptbronnen

De ATmega32 kent veel interruptbronnen. Zo kan de ATmega32 reageren op drie externe interrupts: INT0, INT1 en INT2. Verder kan de analoog-digitaal converter (ADC) een interrupt geven als de conversie klaar is. De seriële interface (USART) kent naast een interrupt wanneer een karakter is ontvangen ook een interrupt voor wanneer een karakter is verzonden. De nog te bespreken timer/counters (zie hoofdstuk ??) kunnen een interrupt afgeven wanneer een timer/counter “over de kop” gaat, dus wanneer een timer/counter van de hoogste stand naar de laagste stand gaat. Verder zijn er nog interrupts mogelijk van de EEPROM, de TWI- en de SPI-interface.

Al deze interruptbronnen hebben een eigen interrupt-enable-bit. Om een interrupt van een bron ook daadwerkelijk te laten plaatsvinden, moet deze bit geactiveerd zijn (logisch 1) én de I-vlag moet logisch 1 zijn.

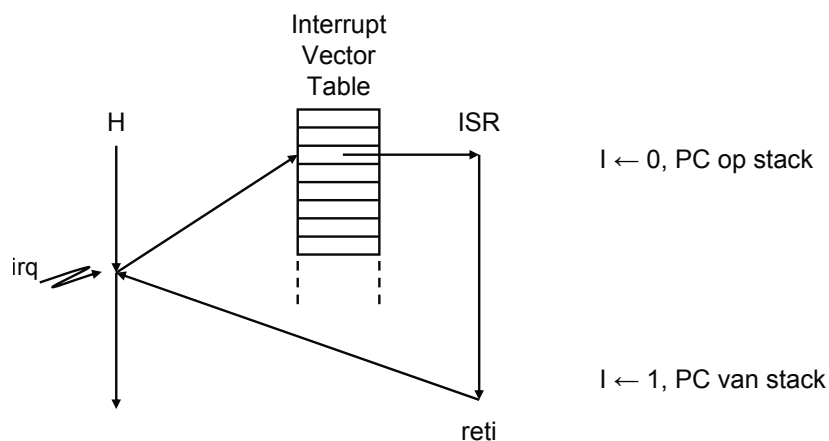
De ATmega32 heeft geen *software interrupts*. Dat zijn software-instructies die interrupts genereren. Deze zijn wel na te bootsen met INT0, INT1 en INT2.

1.4 De Interrupt Vector Table

We hebben nu besproken hoe het verwerken van een interrupt wordt uitgevoerd. Maar nu rijst de vraag: waar moet de ISR beginnen? We zouden hiervoor een vast adres in de

Flash-ROM kunnen kiezen, bijvoorbeeld adres 0x1000. Vervolgens reserveren we 512 bytes voor de ISR. De volgende ISR begint dan op adres 0x1200. Hoewel dit natuurlijk te realiseren is, kleven hier wel wat nadelen aan. Ten eerste moet de ISR altijd op adres 0x1000 beginnen. Ten tweede kan de ISR niet groter zijn dan 512 bytes. Als de ISR korter is dan 512 bytes, dan gebruiken we een gedeelte van de Flash-ROM niet (bedenk dat de volgende ISR op adres 0x1200 begint). Ten derde moet de Flash-ROM minimaal 4608 (= 0x1200) bytes groot zijn. We zien nu dat een vast adres en een vaste lengte voor een ISR niet flexibel is.

We kunnen deze drie problemen oplossen door gebruik te maken van een *Interrupt Vector Table*. Dit werkt als volgt. Aan elke interrupt is een zogenoemde *Interrupt Vector* gekoppeld. Dit is een vast adres in de Flash-ROM waar naartoe gesprongen wordt als een interrupt is herkend. Elke vector is twee words (vier bytes) groot. Na het herkennen van de interrupt wordt de PC dus geladen met het adres van de Interrupt Vector die bij de interrupt hoort. Aangezien elke Interrupt Vector slechts twee words groot is, zal op die plek een spronginstructie naar de eigenlijke ISR geplaatst worden. Het uitvoeren gebeurt dus in twee stappen: eerst naar de Interrupt Vector springen en van daaruit naar de eigenlijke ISR. Het gebruik van de Interrupt Vector Table is weergegeven in figuur 1.4.



Figuur 1.4: Interruptafhandeling: gebruik van de Interrupt Vector Table.

De Interrupt Vector Table ligt aan het begin van de Flash-ROM, van adres 0x000 t/m 0x029. Dat betekent dat de eerste vrije plaats 0x02a is. Vanaf hier kan het gebruikersprogramma geplaatst worden. Na een reset is de PC geladen met 0x000, dus wordt de instructieverwerking begonnen bij de eerste vector. Deze is dan ook gereserveerd voor de Reset Vector. Op de Reset Vector wordt dan een spronginstructie naar adres 0x02a geplaatst.

Natuurlijk heeft het gebruik van de Interrupt Vector Table een keerzijde. Zo zijn er extra geheugenplaatsen nodig voor het opslaan van de vectoren. De executietijd van de interrupt (Engels: interrupt latency) wordt met drie klokpulsen verhoogd omdat er een spronginstructie moet worden uitgevoerd.

Een overzicht van de Interrupt Vector Table is te vinden in tabel 1.1. Op adres 0x000 is de Reset Vector geplaatst. Daarna volgt op adres 0x002 de vector voor de eerste externe interrupt, dan voor de tweede externe interrupt, enzovoorts.

Tabel 1.1: Interruptvectortabel voor de ATmega32.

vector #	ROM-adres	Bron	Omschrijving
1	0x000	RESET	Reset vector
2	0x002	INT0	External Interrupt request 0
3	0x004	INT1	External Interrupt request 1
4	0x006	INT2	External Interrupt request 2
5	0x008	TIMER2_COMP	Timer/Counter2 Compare Match
6	0x00A	TIMER2_OVF	Timer/Counter2 Overflow
7	0x00C	TIMER1_CAPT	Timer/Counter1 Capture Event
8	0x00E	TIMER1_COMPA	Timer/Counter1 Compare Match A
9	0x010	TIMER1_COMPB	Timer/Counter1 Compare Match B
10	0x012	TIMER1_OVF	Timer/Counter1 Overflow
11	0x014	TIMER0_COMP	Timer/Counter0 Compare Match
12	0x016	TIMER0_OVF	Timer/Counter0 Overflow
13	0x018	SPI, STC	Serial Transfer Complete
14	0x01A	USART, RXC	USART, Rx Complete
15	0x01C	USART, UDRE	USART Data Register Empty
16	0x01E	USART, TXC	USART, Tx Complete
17	0x020	ADC	ADC Conversion Complete
18	0x022	EE_RDY	EEPROM Ready
19	0x024	ANA_COMP	Analog Comparator
20	0x026	TWI	Two-wire Serial Interface
21	0x028	SPM_RDY	Store Program Memory Ready

Noot: het ROM-adres is in words.

1.5 Interruptprioriteit

Als twee interrupt tegelijk binnenkomen, wordt de interrupt met de laagste Interrupt Vector als eerste uitgevoerd. Er zijn dus prioriteiten aan de interrupts gekoppeld. De interrupt die op dat moment niet wordt uitgevoerd, moet wachten totdat de uitgevoerde interrupt afgehandeld is. Komt er echter in de tussentijd een interrupt binnen met een lagere Interrupt Vector dan wordt die weer als eerste uitgevoerd.

1.6 Stappen in de interruptverwerking

Als een interruptsignaal bij de processor binnenkomt, worden de volgende stappen genomen:

1. De processor maakt de instructie af die op dat moment wordt uitgevoerd. Dit is een instructie in het lopende programma.
2. De processor slaat het adres van de volgende instructie op de stack op. Dit adres staat in de program counter. De Global Interrupt Enable vlag wordt op 0 gezet.
3. De processor springt naar de interrupt vector. Dit is een vaste geheugenlocatie in de Flash-ROM.
4. Vanuit de interrupt vector wordt gesprongen naar het adres van de daadwerkelijke

ISR.

5. Bij terugkeer uit de ISR wordt de program counter geladen met het adres dat op de stack gezet was. De Global Interrupt Enable vlag wordt op 1 gezet.
6. De processor vervolgt het uitvoeren van het programma. Er wordt altijd één instructie uitgevoerd voordat een eventuele nieuwe interrupt wordt verwerkt.

1.7 Interruptresponstijd

De responstijd voor alle ingeschakelde interrupts van de ATmega32 is minimaal vier klokcycli. Tijdens deze vier klokcycli wordt de program counter op de stack geplaatst (twee bytes) en wordt de program counter geladen met het vectoradres van de interrupt. Na vier klokcycli wordt gesprongen naar het vectoradres van de interrupt die wordt verwerkt. De instructie op de vector is normaal gesproken een sprong naar de interrupt-routine en deze sprong duurt drie klokcycli voor een `jmp`-instructie en twee klokcycli voor een `rjmp`-instructie.

Als een interrupt optreedt tijdens de uitvoering van een instructie met meerdere klokcycli dan wordt deze instructie afgemaakt voordat de interrupt wordt verwerkt. Staat de ATmega32 in slaapstand dan wordt de responstijd met vier klokcycli verhoogd. Deze verhoging komt bovenop de opstarttijd van de geselecteerde slaapmodus.

Een terugkeer (`return`) uit een interruptroutine duurt vier klokcycli. Tijdens deze vier klokcycli wordt de program counter van de stack gehaald (twee bytes) en wordt de stack pointer met twee verhoogd. Tevens wordt de I-bit in het statusregister op 1 gezet.

1.8 Context switch

Een interrupt kan op elk willekeurig moment plaatsvinden. We noemen zulke interrupts *asynchroon*. Asynchroon houdt in dat het moment van de interrupt niet (helemaal) te voorspellen is². Stel dat het lopende programma een complexe berekening aan het uitvoeren is. De berekening zorgt ervoor dat de vlaggen en de registers worden aangepast. Als nu een interrupt binnenkomt en de ISR wordt gestart dan is er een grote kans dat in de ISR ook registers en de vlaggen worden aangepast. Bij terugkeer naar het lopende programma moeten dan de originele waarden van de registers en de vlaggen weer beschikbaar zijn om de berekening correct te laten verlopen, d.w.z. dat de *context* weer hersteld moet worden. Het is daarom belangrijk om in de ISR de gebruikte registers en de vlaggen op te slaan. Dat kan door gebruik te maken van de stack.

In listing 1.2 is te zien hoe de vlaggen op de stack geplaatst worden. De vlaggen zijn op te vragen via I/O-register `SREG`. Nu kan de inhoud van een I/O-register niet direct op de stack geplaatst worden. Dit moet via een register. Dat betekent dat de inhoud van dat register eerst op de stack geplaatst moet worden. Daarna worden de vlaggen ingelezen in het register en op de stack gezet.

Aan het einde van de ISR moeten de registers en de vlaggen weer van de stack gehaald worden. Let daarbij op de volgorde van de registers.

² Interrupts van bijvoorbeeld de timer/counters zijn natuurlijk wel uit te rekenen.


```

1 ISR:    push r16          ; push R16
2         in  r16,SREG      ; SREG in I/O memory ...
3         push r16         ; ... so push flags via R16
4
5         ; rest of the ISR code
6
7         pop  r16          ; pop flags via R16
8         out  SREG,r16
9         pop  r16          ; pop R16
10        reti              ; return

```

Listing 1.2: Opslaan van de registers en vlaggen.

1.9 Externe interrupts

De ATmega32 kent drie externe interrupts genaamd INT0, INT1 en INT2. Deze interrupts zijn fysiek verbonden met Port D, bit 2 (PD2, INT0), Port D, bit 3 (PD3, INT1) en Port B, bit 2 (PB2, INT2). INT0 en INT1 kunnen zowel op een laag niveau als op flanken reageren. INT2 kan alleen op flanken reageren. Zoals te zien in tabel 1.1 worden de vectoradressen 0x002, 0x004 en 0x006 gebruikt voor respectievelijk INT0, INT1, en INT2. We merken op dat INT0 dus een hogere prioriteit heeft dan INT1 en INT2.

Om een externe interrupt te herkennen, moeten de bijbehorende pinnen als ingang gedefinieerd worden. Maar er moet opgemerkt worden dat interrupts ook worden herkend als de bijbehorende pin als uitgang gedefinieerd is. Op deze manier is het mogelijk om vanuit de software een interrupt te genereren.

Het General Interrupt Control Register (GICR) heeft drie bits waarmee de INT's kunnen worden geactiveerd. Een logische 1 in de betreffende bit activeert de INT. Merk op dat de I-vlag in het Status Register 1 moet zijn om interrupt ook daadwerkelijk te laten plaatsvinden. De indeling van het GICR is te vinden in figuur 1.5. Alleen de bits 5, 6 en 7 zijn van belang.

7	6	5	4	3	2	1	0
INT1	INT0	INT2	—	—	—	IVSEL	IVCE
R/W	R/W	R/W	R	R	R	R/W	R/W
0	0	0	0	0	0	0	0

Figuur 1.5: Het GICR register.

In listing 1.3 is te zien hoe INT0 geactiveerd wordt. Merk op dat de betreffende bit in het Data Direction Register D (DDRD, bit 2) op 0 moet staan om de pin als ingang te definiëren.

```

1    ldi r16,0b01000000 ; activate INT0 (pin PD2)
2    out GICR,r16

```

Listing 1.3: Het activeren van INT0.

Een flankgevoelige interrupt (opgaande flank, neergaande flank of beide flanken) wordt door de ATmega32 “ingevangen”. Dit wordt bijgehouden in het General Interrupt Flag Register (GIFR). Dat betekent dat als een flankgevoelige interrupt wordt herkend, de bijbehorende bit in het GIFR-register op 1 wordt gezet. Als de ISR wordt uitgevoerd, wordt de bit door de hardware op 0 gezet. Zolang een interruptaanvraag niet wordt uitgevoerd blijft de bit dus 1. In figuur 1.6 is de indeling van het GIFR-register te zien. Merk op dat een niveaugevoelige interrupt (INT0, INT1) de bijbehorende bits in het GIFR-register niet op 1 zet. Bij niveaugevoelige interrupts wordt direct de status van de pinnen ingelezen.

7	6	5	4	3	2	1	0
INTF1	INTF0	INTF2	—	—	—	—	—
R/W	R/W	R/W	R	R	R	R	R
0	0	0	0	0	0	0	0

Figuur 1.6: Het GIFR register.

Merk op dat een bit in het GIFR-register door software op 0 kan worden gezet door er een 1 naar toe te schrijven. Dit principe geldt voor alle interrupts die de ATmega32 kent.

Triggermogelijkheden van INT0 en INT1

Zowel INT0 als INT1 kunnen op meerdere mogelijkheden geactiveerd worden. Ze kunnen reageren op een laag niveau, op een opgaande flank, op een neergaande flank of op beide flanken. Ze kunnen niet reageren op een hoog niveau. De mogelijkheden worden aangegeven met de ISC-bits in het MCU Control Register (MCUCR). De indeling van dit register is te zien in figuur 1.7.





7	6	5	4	3	2	1	0
SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Figuur 1.7: Het MCUCR register.

Bit 3, 2 – ISC11, ISC10: Interrupt Sense Control 1 bit 1 en bit 0

De ISC11- en ISC10-bits definiëren de wijze waarop INT1 een interrupt genereert. De INT1-pin (PD3) wordt bemonsterd voordat de flank wordt gedetecteerd. Als flankgevoelige triggering is geselecteerd, worden pulsen groter dan één klokcyclus herkend. Als een puls korter is dan één klokcyclus dan is er geen garantie dat de puls (en dus de flank) wordt herkend. Als triggering van een laag niveau wordt gebruikt, dan moet het lage niveau aangehouden worden totdat de huidige instructie uitgevoerd is. De langst-durende instructie is een `call`-instructie. Deze instructie duurt op een ATmega32 vier klokcycli. Dat betekent dat het lage niveau op de INT1-pin dus minimaal vier klokcycli moet duren. De interrupt wordt (steeds) herkend zolang het lage niveau aangehouden wordt. In figuur 1.2 is te zien hoe een bepaalde triggering ingesteld moet worden.

Tabel 1.2: *Triggermogelijkheden van INT1.*

ISC11	ISC10		Omschrijving
0	0		Een laag niveau op de INT1-ingang genereert een interrupt.
0	1		Een opgaande en neergaande flank op de INT1-ingang genereert een interrupt.
1	0		Een neergaande flank op de INT1-ingang genereert een interrupt.
1	1		Een opgaande flank op de INT1-ingang genereert een interrupt.

In listing 1.4 is te zien hoe INT1 wordt ingesteld voor het herkennen van een opgaande flank om een interrupt te genereren.

```

1  ldi r16,0b00001100 ; activate rising edge INT1
2  out MCUCR,r16





```

Listing 1.4: *Instellen van de opgaande flank voor INT1.*

Bit 1, 0 – ISC01, ISC00: Interrupt Sense Control 0 bit 1 en bit 0

De ISC01- en ISC00-bits definiëren de wijze waarop INT0 een interrupt genereert. De werking is identiek aan die van de ISC-bits van INT1. Ook de timing is identiek. Zie tabel 1.3.

Tabel 1.3: *Triggermogelijkheden van INT0.*

ISC01	ISC00		Omschrijving
0	0		Een laag niveau op de INT0-ingang genereert een interrupt.
0	1		Een opgaande en neergaande flank op de INT0-ingang genereert een interrupt.
1	0		Een neergaande flank op de INT0-ingang genereert een interrupt.
1	1		Een opgaande flank op de INT0-ingang genereert een interrupt.

Triggermogelijkheden van INT2

Externe interrupt INT2 kan alleen op de opgaande flank of op de neergaande flank triggeren. Dit is in te stellen in het MCU Control and Status Register (MCUCSR). De indeling van dit register is te zien in figuur 1.8. INT2 wordt *asynchroon* verwerkt. Asynchroon betekent hier dat het interruptsignaal buiten de klok om wordt “ingevangen”. De minimale pulsduur van INT2 is 50 ns, maar de puls wordt dus niet bemonsterd op klokflanken van de systeemklok. De aanvraag wordt wel verwerkt op de flanken van de systeemklok.

7	6	5	4	3	2	1	0
JTD	ISC2	—	JTRF	WDRF	BORF	EXTRF	PORF
R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Figuur 1.8: Het MCUCSR register.

Bit 6 - ISC2: Interrupt Sense Control 2

Bit ISC2 geeft aan op welke flank INT2 moet reageren. Een 0 geeft aan dat INT2 op een neergaande flank triggert, een 1 geeft aan dat INT2 op een opgaande flank triggert.

Bij het wijzigen van de ISC2-bit kan een interrupt optreden. Daarom is het nodig om INT2 eerst uit te schakelen door het uitschakelen van de INT2-bit in het GICR-register. Vervolgens kan de ISC2-bit worden gewijzigd. Daarna moet een eventuele interrupt-aanvraag worden gewist door een logische 1 naar de INTF2-bit in het GIFR-register te schrijven voordat de interrupt opnieuw wordt ingeschakeld. In listing 1.5 is te zien hoe van flank kan worden gewisseld.

```

1      ; Disable INT2, leave INT0/INT1 untouched
2      in    r16,GICR
3      andi  r16,0b11011111
4      out   GICR,r16
5
6      ; Change falling edge to rising edge, don't touch other flags
7      in    r16,MCUCSR
8      ori   r16,0b01000000
9      out   MCUCSR,r16
10
11     ; Clear INTF2 flag
12     ldi   r16,0x00100000
13     out   GIFR,r16
14
15     ; Enable INT2, leave INT0/INT1 untouched
16     in    r16,GICR
17     ori   r16,0b00100000
18     out   GICR,r16

```

Listing 1.5: Omschakelen van INT2 van neergaande naar opgaande flank.

In listing 1.6 is een voorbeeld te zien van het gebruik van INT0. Als taak heeft de ISR om de bits van Port B te inverteren. We beginnen met het aangeven van de reset-vector (regel 5) en de INT0-vector (regel 8). Daarna wordt de stack geïnitieerd (regel 14 t/m 17). In regel 20 en 21 wordt Port B als uitgang ingesteld. In regel 24 en 25 wordt gekozen voor de opgaande flank van INT0. In regel 25 en 26 activeren we INT0. Daarna worden de interrupts vrij gegeven middels een `sei`-instructie (regel 32) en blijven we in een eeuwig durende lus wachten (regel 35).

In de ISR wordt de context (de processorstatus) op de stack gezet (regels 39 t/m 31). Daarna wordt Port B ingelezen, geïnverteerd en weer terugschreven (regels 44 t/m 46). Vlak voor het terugkeren wordt de context weer van de stack gehaald (regels 49 t/m 41).

```

1 .def temp = r16
2
3 .org 0x000
4     ; reset vector
5     rjmp reset
6 .org 0x002
7     ; ISR vector for INT0
8     rjmp int0_isr
9
10 .org 0x02a
11     ; main program starts here
12 reset:
13     ; set up stack pointer
14     ldi temp, low(RAMEND)
15     out SPL, temp
16     ldi temp, high(RAMEND)
17     out SPH, temp
18
19     ; All pins Port B are outputs
20     ldi temp, 0xff
21     out DDRB, temp
22
23     ; set up rising edge INT0
24     ldi temp, 0b00000011
25     out MCUCR, temp
26
27     ; set up INT0
28     ldi temp, 0b01000000
29     out GICR, temp
30
31     ; enable interrupts
32     sei
33
34     ; forever....
35 loop: rjmp loop
36
37 int0_isr:
38     ; save regs and flags
39     push temp
40     in temp, SREG
41     push temp
42
43     ; Invert all Port B bits
44     in temp, PORTB
45     com temp
46     out PORTB, temp
47
48     ; restore regs and flags and return from interrupt
49     pop temp
50     out SREG, temp
51     pop temp
52     reti

```

Listing 1.6: Voorbeeldprogramma INT0 met opgaande flank en ISR.

Daarna keren we weer terug naar het lopende programma door middel van een `reti`-instructie.

1.10 Interrupts binnen interrupts

Als een interrupt wordt afgehandeld, wordt de bijbehorende ISR gestart. Net voordat ISR wordt gestart, wordt de Global Interrupt Enable vlag (I-vlag) op 0 gezet. Dit zorgt ervoor dat nieuwe interruptaanvragen (nog) niet afgehandeld worden. Deze aanvragen moeten wachten totdat de ISR klaar is. Bij het terugkeren wordt de I-vlag weer op 1 gezet, zodat wachtende interruptaanvragen kunnen worden afgehandeld.

Het is mogelijk om in een ISR de I-vlag weer op 1 te zetten zodat een interruptaanvraag kan worden bediend. De ISR wordt dus onderbroken door een interrupt en er wordt een (andere) ISR gestart. We spreken dan over een interrupt binnen een interrupt.

Het kan handig zijn om een ISR te laten onderbreken door een interrupt. Zo kan een ISR voor de afhandeling van een toetsaanslag prima onderbroken worden door een interruptaanvraag van de netwerkkaart. We kunnen ongeveer drie toetsaanslagen per seconden maken maar netwerkpakketten komen met 1 Gbps (gigabit per seconde) binnen. Het is dan belangrijk om deze interrupt eerst af te handelen. Nadat de ISR van de netwerkkaart is afgelopen, wordt weer verder gegaan met de ISR van het toetsenbord.

Voorzichtigheid is wel geboden met interrupts binnen interrupts, vooral bij level-triggered interrupts. Zolang de logische waarde (level) wordt aangehouden, wordt de ISR onderbroken door een interruptaanvraag. Daardoor wordt elke keer als een ISR wordt gestart het terugkeeradres op de stack gezet. De stack loopt zodoende snel vol wat resulteert in een *stack overflow*.

1.11 Software interrupts

De ATmega32 kent geen instructies om een interrupt te genereren. Dit kan echter wel nagebootst worden middels de externe interrupts zoals te zien is in listing 1.7. In dit voorbeeld wordt INT0 gebruikt voor het genereren van een interrupt. Dit kan gerealiseerd worden door de externe interrupt pin PD2 (Port D, pin 2) als uitgang te definiëren. Door het schrijven van een logische 1 naar PD2 wordt een opgaande flank gerealiseerd die door de ATmega32 gezien wordt als een interruptaanvraag voor INT0.

In de listing worden de vectoren en de stack geïnitieerd (regels 1 t/m 10). In regel 12 en 13 wordt PD2 als uitgang gezet. In regels 15 t/m 19 wordt INT0 als flankgevoelige interrupt (opgaande flank) ingesteld. Daarna worden de interrupts vrijgegeven. Om niet continu een interrupt te genereren is een korte wachtlus ingebouwd. Die zorgt ervoor dat eens in de 767 klokcycli een interrupt gegenereerd wordt. Eerst wordt PD2 op een logische 0 gezet (regel 24). Daarna wordt de wachtlus gestart (regels 26 en 27). In regels 29 en 30 wordt PD2 logisch 1 gemaakt zodat een opgaande flank wordt aangeboden aan de interrupthardware. Daarna wordt gesprongen naar regel 23 zodat weer een logische 0 op PD2 gezet wordt en wordt de wachtlus weer uitgevoerd.

Software interrupt zijn in de ATmega32 niet echt nodig. Maar in de wereld van de Operating Systems vormen ze een schakel tussen een gebruikersprogramma en de *kernel*. Een gebruikersprogramma mag bijvoorbeeld niet zomaar naar de harddisk schrijven

```

1      .org 0x000
2      rjmp start
3      .org 0x002
4      rjmp int0_isr
5
6      .org 0x02a
7 start: ldi r16,low(RAMEND)    ; stack
8      out SPL,r16
9      ldi r16,high(RAMEND)
10     out SPH,r16
11
12     ldi r16,0b000000100    ; PD2 as output
13     out DDRD,r16
14
15     ldi r16,0b000000011    ; INT0 pos. edge
16     out MCUCR,r16
17
18     ldi r16,0b01000000    ; INT0 active
19     out GICR,r16
20
21     sei                    ; free interrupts
22
23 again: ldi r16,0x00
24     out PORTD,r16          ; Port D2 off
25
26 delay: subi r16,0x01       ; delay 767 clocks
27     brne delay
28
29     ldi r16,0b000000100    ; generate interrupt
30     out PORTD,r16
31
32     rjmp again
33
34 int0_isr:
35     reti

```

Listing 1.7: Externe interrupt INT0 wordt gebruikt als software interrupt.

maar moet deze acties via de kernel uitvoeren. Om in *kernel mode* te komen, moet een software interrupt worden uitgevoerd. De kernel bepaalt dan welke interrupt is aangevraagd en wat de te verwachten actie is. Deze kernel routines worden *system calls* genoemd. In programmeertalen zoals 'C' zijn software interrupts niet direct te gebruiken. In plaats daarvan zijn er functies zoals `read` en `write` voorhanden. Een gebruikersprogramma voert bijvoorbeeld een `read` uit en in die functie wordt dan de software interrupt uitgevoerd.

1.12 Interrupts in 'C'

De programmeertaal 'C' is niet uitgerust om ISR's te programmeren. De GNU AVR-C compiler heeft echter functies waarmee dit wel mogelijk is. Dit zijn dus uitbreidingen op de taal 'C' en kunnen niet gebruikt worden op bijvoorbeeld een PC of een iMac.

Om gebruik te maken van de interruptfaciliteiten moet een header-bestand geladen worden waarin functies zijn gedeclareerd. Dit is het bestand `avr/interrupt.h`. Aangezien de interrupts geactiveerd worden via de I/O-registers moet ook `avr/io.h` geladen worden. Dit is te zien in listing 1.8. Het gebruik van header-bestand `avr/cpufunc.h` wordt verderop uitgelegd.

```
1 #include <avr/io.h>
2 #include <avr/interrupt.h>
3 #include <avr/cpufunc.h>
```

Listing 1.8: Header-bestanden voor het gebruik van interrupts.

Na het laden van header-bestand `avr/interrupt.h` is voor het coderen van een ISR de macro `ISR(...)` beschikbaar. De definitie van deze macro is complex en wordt verder niet besproken. Als argument aan de macro moet in ieder geval de interruptvectornaam gegeven worden. Optioneel kunnen enkele *attributes* opgegeven worden.

In tabel 1.4 zijn de interruptvectornamen te zien. Deze namen moeten gebruikt worden bij de `ISR`-macro. Merk op dat de reset geen vectornaam heeft. Het is namelijk niet mogelijk om een ISR voor de reset te coderen. De resetvector wordt door de C-compiler ingevuld en zorgt ervoor dat, na initialisatie van variabelen, de functie `main()` wordt aangeroepen.

Tabel 1.4: Interruptvectornamen voor ISR in 'C' voor de ATmega32.

vector #	Bron	Vectornaam
1	RESET	—
2	INT0	INT0_vect
3	INT1	INT1_vect
4	INT2	INT2_vect
5	TIMER2_COMP	TIMER2_COMP_vect
6	TIMER2_OVF	TIMER2_OVF_vect
7	TIMER1_CAPT	TIMER1_CAPT_vect
8	TIMER1_COMPA	TIMER1_COMPA_vect
9	TIMER1_COMPB	TIMER1_COMPB_vect
10	TIMER1_OVF	TIMER1_OVF_vect
11	TIMER0_COMP	TIMER0_COMP_vect
12	TIMER0_OVF	TIMER0_OVF_vect
13	SPI, STC	SPI_STC_vect
14	USART, RXC	USART_RXC_vect
15	USART, UDRE	USART_UDRE_vect
16	USART, TXC	USART_TXC_vect
17	ADC	ADC_vect
18	EE_RDY	EE_RDY_vect
19	ANA_COMP	ANA_COMP_vect
20	TWI	TWI_vect
21	SPM_RDY	SPM_RDY_vect

In listing 1.9 is een compleet voorbeeld gegeven van het gebruik van de INT0 ISR. De taak van de ISR is om pin PB0 (Port B, bit 0) te laten toggelen. Na het laden van de header-bestanden wordt Port B als uitgang gedefinieerd en wordt INT0 geïnitieerd als een flankgevoelige interrupt op de opgaande flank (regels 7 t/m 10). In regel 12 worden de interrupts vrijgegeven door het aanzetten van de I-vlag middels het `sei()`-statement. Daarna wordt in een eeuwig durende lus gewacht middels het `while()`-statement.

```

1 #include <avr/io.h>
2 #include <avr/interrupt.h>
3 #include <avr/cpufunc.h>
4
5 int main(void) {
6
7     DDRB = 0xff;    /* Port B as output */
8
9     MCUCR = 0x03;   /* INT0 rising edge */
10    GICR = 0x40;    /* INT0 active */
11
12    sei();           /* Free interrupts */
13
14    while (1) {
15    }
16 }
17
18 ISR(INT0_vect) {   /* INT0 ISR */
19     _NOP();         /* To set a breakpoint */
20     PORTB ^= 0x01;  /* Flip PB0 */
21     _NOP();         /* To set a breakpoint */
22 }

```

Listing 1.9: Voorbeeld van het gebruik van de INT0 interrupt service routine.

Het `sei()`-statement wordt vertaald naar een assembler `sei`-instructie. Er bestaat ook een `cli()`-statement dat vertaald wordt naar een assembler `cli`-instructie. In feite zijn dit macro's met de volgende definitie:

```

1 #define sei()    __asm__ __volatile__ ("sei" ::: "memory")
2 #define cli()    __asm__ __volatile__ ("cli" ::: "memory")

```

Listing 1.10: Definitie van de `sei()`- en `cli()`-statements.

De constructie `__asm__` signaleert aan de C-compiler dat wat volgt een assemblerinstructie is. De constructie `__volatile__` geeft aan dat de C-compiler de assemblerinstructie letterlijk moet overnemen en geen optimalisatie mag toepassen³.

De ISR is gecodeerd in de regels 18 t/m 22 en heeft drie statements. De macro `_NOP()` in de regels 19 en 21 wordt vertaald naar een assembler `nop`-instructie. Deze macro

³ De `sei`- en `cli`-instructies passen geen geheugenplaatsen en registers aan. De compiler zou hieruit de conclusie kunnen trekken dat de beide instructies dus netto geen effect hebben en dat ze dus verwijderd kunnen worden uit de gecompileerde code.

wordt gedefinieerd door het header-bestand `avr/cpufunc.h`. Het gebruik van deze macro is handig bij het zetten van een breakpoint tijdens debuggen. Het statement in regel 20 zorgt ervoor dat pin PBO (Port B, bit 0) steeds van waarde veranderd (toggelen). Het “dakje”-teken `^` (Nederland: accent circonflexe, Engels: circumflex) is de “C”-taalconstructie voor de EXOR-operatie. Het gehele statement werkt als volgt. De waarde van `PORTB` wordt gelezen, de bits 1 t/m 7 wordt onveranderd overgenomen en bit 0 wordt geïnverteerd. Daarna wordt de nieuwe waarde teruggestreven naar `PORTB`.

1.12.1 Bad Interrupt Handler

Wat gebeurt er als een bepaalde interruptbron geactiveerd wordt maar er is geen ISR beschikbaar, d.w.z. er is geen ISR geprogrammeerd. Als de betreffende interrupt geregistreerd wordt, zal de normale verwerking van de interrupt plaatsvinden. Er wordt gesprongen naar het vectoradres van de interrupt en daar wordt dan met de instructieverwerking begonnen. Maar er is geen ISR beschikbaar dus kan er niet naar de ISR gesprongen worden. De C-compiler lost dit op door alle niet gebruikte interrupts om te leiden naar de zogenoemde Bad Interrupt Handler. Deze *catch-all* ISR is als volgt in te stellen:

```

1 #include <avr/interrupt.h>
2
3 ISR(BADISR_vect) {
4     /* user code here */
5 }

```

Listing 1.11: De Bad Interrupt Handler.

De standaard actie van de Bad Interrupt Handler is dat er naar de reset-vector wordt gesprongen. Dit kan leiden tot een problematische uitvoering van het programma. Het is daarom aan te bevelen om de Bad Interrupt Handler altijd op te nemen in het programma. Merk op dat het ontbreken van een ISR duidt op een programmeerfout.

1.12.2 Interrupt prologue en epilogue

In paragraaf 1.8 is al beschreven dat tijdens het uitvoeren van de ISR de gebruikte registers en de vlaggen op de stack gezet moeten worden. De C-compiler zorgt hier automatisch voor. Dit wordt de *prologue* van de ISR genoemd. Vlak voordat de ISR verlaten wordt, worden de registers en de vlaggen weer van de stack gehaald. Ook hier zorgt de C-compiler voor. Dit wordt de *epilogue* genoemd. Merk op dat de C-compiler ook de `reti`-instructie automatisch toevoegt.

1.12.3 Naked interrupts

Soms is het handig om de prologue en de epilogue niet door de C-compiler te laten genereren, bijvoorbeeld als de interrupt met maximale snelheid moet worden afgehandeld. Dat kan door het toevoegen van de *attribute* `ISR_NAKED`. Dit is te zien in listing 1.12. Omdat er geen epilogue wordt gegenereerd, moet de ISR afgesloten worden met een `reti`-instructie. Daar zorgt de `reti()`-macro voor.

Merk op dat het statement in regel 4 wordt omgezet in een `sbi`-instructie, maar alleen

```

1 #include <avr/interrupt.h>
2
3 ISR(INT0_vect, ISR_NAKED) {
4     PORTB |= _BV(0);      /* results in SBI instruction which */
5                           /* does not affect the Status Register */
6     reti();               /* return from interrupt */
7 }

```

Listing 1.12: Een naked interrupt voor INT0.

als compiler-optimalisatie is ingeschakeld.

1.12.4 Interrupt attributes

Met attributes is de codegeneratie van de ISR aan te passen. Er zijn er vier:

ISR_BLOCK	Deze attribute geeft aan dat de ISR wordt uitgevoerd zonder dat de Global Interrupt Enable vlag weer op 1 wordt gezet. Hierdoor wordt de ISR <i>atomisch</i> uitgevoerd, d.w.z. dat de ISR niet kan worden onderbroken door een andere interrupt.
ISR_NOBLOCK	Deze attribute geeft aan dat de ISR wordt uitgevoerd met de Global Interrupt Enable vlag op 1. De C-compiler genereert hierdoor een <i>sei</i> -instructie zodat interrupts weer worden uitgevoerd. De lopende ISR kan dus worden onderbroken door een andere interruptaanvraag.
ISR_NAKED	De prologue en de epilogue worden niet door de C-compiler gegenereerd.
ISR_ALIASOF(<i>vector</i>)	De ISR voor <i>vector</i> wordt omgeleid naar de vector die is opgegeven in <code>ISR()</code> . Hierdoor is het mogelijk om twee interrupts door dezelfde ISR af te laten handelen. Voorbeeld:

```
ISR(INT1_vect, ISR_ALIASOF(INT0_vect));
```

De vector voor INT1 wijst nu naar dezelfde ISR als die van INT0.

1.13 Codegeneratie voor de ISR

Het is interessant om te weten wat voor assemblercode door de compiler wordt gegenereerd. Hiervoor wordt het programma uit listing 1.9 gebruikt. Het eerste deel is te zien in listing 1.13. Natuurlijk is het eerste stuk gereserveerd voor de interrupt vectoren (regels 2 t/m 21). In regel 2 is de sprong naar het begin van het programma te zien (de resetvector). In regel 3 is de spronginstructie naar de ISR van INT0 te zien. Omdat de overige interrupts niet gebruikt worden, zijn hier spronginstructies naar de Bad Interrupt Handler te zien. De actie van de Bad Interrupt Handler laat niets te raden over. Er wordt namelijk gesprongen naar de resetvector (regel 35).

Noot: merk op dat de C-compiler de adressen in bytes presenteert en niet in words. Dit is goed te zien bij de resetvector. Hier staat de instructie `jmp 0x54` en `0x54` gedeeld door 2 is `0x2a`, het einde van de Interrupt Vector Table.

```

1 00000000 <__vectors>:
2   0:  0c 94 2a 00      jmp 0x54      ; 0x54 <__ctors_end>
3   4:  0c 94 3e 00      jmp 0x7c      ; 0x7c <__vector_1>
4   8:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
5   c:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
6  10:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
7  14:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
8  18:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
9  1c:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
10 20:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
11 24:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
12 28:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
13 2c:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
14 30:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
15 34:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
16 38:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
17 3c:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
18 40:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
19 44:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
20 48:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
21 4c:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
22 50:  0c 94 34 00      jmp 0x68      ; 0x68 <__bad_interrupt>
23
24 00000054 <__ctors_end>:
25  54:  11 24              eor r1, r1
26  56:  1f be              out 0x3f, r1      ; 63
27  58:  cf e5              ldi r28, 0x5F      ; 95
28  5a:  d8 e0              ldi r29, 0x08      ; 8
29  5c:  de bf              out 0x3e, r29      ; 62
30  5e:  cd bf              out 0x3d, r28      ; 61
31  60:  0e 94 36 00      call 0x6c          ; 0x6c <main>
32  64:  0c 94 52 00      jmp 0xa4          ; 0xa4 <_exit>
33
34 00000068 <__bad_interrupt>:
35  68:  0c 94 00 00      jmp 0              ; 0x0 <__vectors>

```

Listing 1.13: De interrupt vector table en opstartcode.

In regels 25 en 26 wordt register R1 op 0 gezet en worden de vlaggen op 0 gezet. De C-compiler zorgt er trouwens voor dat R1 altijd op 0 gezet blijft. Daarna wordt de stack pointer geïnitieerd (regels 27 t/m 30). Vervolgens wordt de functie `main` aangeroepen. De C-compiler is hier strikt in; `main` is een functie, dus wordt het aangeroepen. Als uit `main` wordt teruggekeerd, wordt naar het einde van het programma gesprongen.

In listing 1.14 is de assemblercode van functie `main` te zien. Er zijn slechts vijf statements. In de regels 6 en 7 wordt Port B als uitgang ingesteld. In de regels 10 t/m 14 wordt INT0 als flankgevoelige interrupt voor de opgaande flank ingesteld. Daarna worden de interrupts vrij gegeven en wordt in een eeuwig durende lus gewacht (regels 17 en 18).

```

1 0000006c <main>:
2
3 int main(void) {
4
5     DDRB = 0xff;      /* Port B as output */
6 6c:  8f ef          ldi r24, 0xFF    ; 255
7 6e:  87 bb          out 0x17, r24    ; 23
8
9     MCUCR = 0x03;     /* INT0 rising edge */
10 70:  83 e0         ldi r24, 0x03    ; 3
11 72:  85 bf         out 0x35, r24    ; 53
12     GICR = 0x40;     /* INT0 active */
13 74:  80 e4         ldi r24, 0x40    ; 64
14 76:  8b bf         out 0x3b, r24    ; 59
15
16     sei();           /* Free interrupts */
17 78:  78 94         sei
18 7a:  ff cf         rjmp    .-2        ; 0x7a <main+0xe>

```

Listing 1.14: De functie *main*.

De C-code van de ISR bestaat uit slechts een aantal instructies maar in listing 1.15 is te zien dat er behoorlijk wat assemblercode is geproduceerd. De prologue van de ISR is te zien in de regels 4 t/m 10. Eerst worden register R0 en R1 op de stack gezet. Daarna worden de vlaggen via R0 op de stack gezet. Vervolgens wordt de inhoud van R1 op 0 gezet. Kennelijk worden register R24 en R25 in de ISR gebruikt, want de prologue wordt afgesloten met twee `push`-instructies die R24 en R25 op de stack zetten.

De `_NOP()`-macro wordt vertaald naar een assembler `nop`-instructie (regels 12 en 19).

De regels 14 t/m 17 laten zien hoe bit 0 van Port B wordt geïnverteerd. Port B wordt eerst geladen in register R25. Register R24 wordt geladen met de constante 0x01. Daarna wordt een EXOR-operatie met R24 en R25 uitgevoerd. Vervolgens wordt de nieuwe waarde weer naar Port B geschreven. Merk op dat er geen `eori`-instructie (EXOR immediate) bestaat zodat altijd een register moet worden gebruikt die met een constante moet worden geladen.

In de epilogue worden register R24 en R25 weer van de stack gehaald. Daarna worden de vlaggen via R0 vanuit de stack weer geladen. Vervolgens worden register R0 en R1 van stack gehaald. Als laatste wordt de `reti`-instructie uitgevoerd zodat de ISR wordt afgesloten.

```

1 0000007c <__vector_1>:
2
3 ISR(INT0_vect) {      /* INT0 ISR */
4   7c:  1f 92          push    r1
5   7e:  0f 92          push    r0
6   80:  0f b6          in      r0, 0x3f      ; 63
7   82:  0f 92          push    r0
8   84:  11 24          eor     r1, r1
9   86:  8f 93          push    r24
10  88:  9f 93          push    r25
11   _NOP();          /* To set a breakpoint */
12  8a:  00 00          nop
13   PORTB ^= 0x01;    /* Flip PB0 */
14  8c:  98 b3          in      r25, 0x18      ; 24
15  8e:  81 e0          ldi     r24, 0x01      ; 1
16  90:  89 27          eor     r24, r25
17  92:  88 bb          out     0x18, r24      ; 24
18   _NOP();          /* To set a breakpoint */
19  94:  00 00          nop
20 }
21 96:  9f 91          pop     r25
22 98:  8f 91          pop     r24
23 9a:  0f 90          pop     r0
24 9c:  0f be          out     0x3f, r0      ; 63
25 9e:  0f 90          pop     r0
26 a0:  1f 90          pop     r1
27 a2:  18 95          reti

```

Listing 1.15: De ISR voor INT0.