TSEA28 Datorteknik Y (och U)

Föreläsning 6

Kent Palmkvist, ISY

LINKÖPINGS UNIVERSITET

Dagens föreläsning

- I/O enheter
- Avbrott
  - Princip
- Avbrott på ARM

Praktiska kommentarer

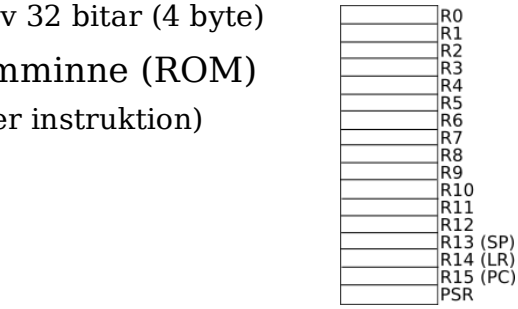
- Tillgång till labb utanför labtider
  - LiU-kort ska ge access till MUX1
  - MUX2 används BARA på distans
  - MUX1 används BARA på plats (i labbet)
- MUX2: Maskiner med distansutrustning
  - muxen2-101 - muxen2-116
  - Emulerar tangentbord etc. mha arduino (kommandon tsea28lab1, tsea28lab2 etc.)
  - Se Introduktion till Darma, video på lisam, samt förra föreläsningen för mer information.

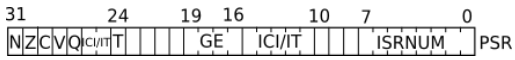
Praktiska kommentarer, forts.

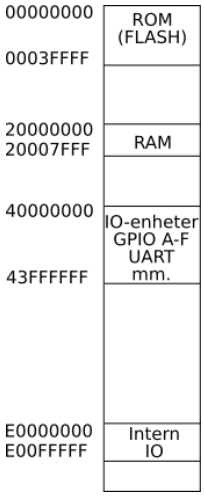
- Några redan godkända på lab1
  - Kan avboka sig på lab1b om ni vill (inget krav)
- Deadline för bokning lab1b
  - Måndag 5/2 kl 23:30

Programmerarens vy av DARMA

- CPU med 13 generella register, PC, LR, SP samt statusregister PSR
  - Varje register består av 32 bitar (4 byte)
- 256 KB Flash programminne (ROM)
  - Programkod (4 byte per instruktion)
  - Konstanter
- 32KB RAM för data
  - Variabler
  - Stack etc.
- Minnesmappad I/O







Register i processor vs I/O enheter

- I processorn finns 16 generella register internt (r0-r15)
  - Inklusive PC, LR, SP
  - Alla beräkningar görs på dessa register
- Minnesadresser som styr I/O-enheter (t ex GPIOF_GPIODATA) kallas slarvigt ibland för register
  - Är inte interna i processorn
    - Värden måste läsas/skrivas till/från processorns interna register
  - Hanteras som minnesadresser
    - Beter sig inte riktigt som vanliga minnesceller, kallas därför ofta för register i dokumentationen
  - Sammanblandningsrisk

I/O enheter

In och utmatning (I/O), generellt

- Alla datorer behöver kommunicera med omvärden (se studenträknarexemplet)
  - Skicka ut bitar via en speciell minnesadress kopplad till pinne
  - Läs av insignaler via en speciell adress kopplad till pinne
- Problem med den enkla uppsättningen
  - Måste veta i förväg hur många in och utsignaler som ska finnas
  - Kan bara skicka ut alternativt ta emot signal på en pinne
  - Varje pinne på kretsen kostar extra, vill utnyttja dom maximalt
  - Vissa funktioner är enkla att bygga hårdvara för men dyra att lösa programmeringsmässigt
  - Vissa funktioner har för hög hastighet för att processorn ska hinna med

In och utmatning, Parallellport (GPIO)

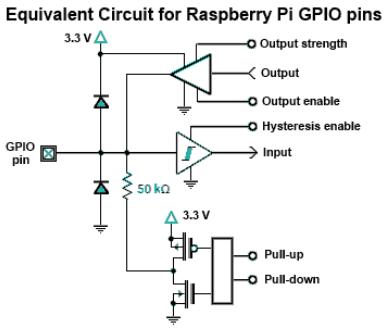
- Vanlig funktion är parallellporten (General Purpose IO)
  - Skicka ut en byte per gång
  - Läs av en byte per gång
- Att styra utsignaler
  - Värdet som lagras i utporten finns tillgänglig på datorns utsida
  - Varje gång nytt värde skrivs ändras alla utsignalerna på en gång
- I vissa datorer kan man även läsa ut utportens värde
  - Är det datat man skrivit till porten som läses, eller är det värdet på pinnen?
  - Ibland kan båda möjligheterna finnas (två olika adresser)

In och utmatning, Parallellport (GPIO), forts.

- Att läsa av en port är enkelt
  - Välj portens pinnar istället för minnet för någon specifik adress
- In och utgångar kan kombineras ihop
  - Möjliggör anpassning till aktuellt behov
  - Tillåter dubbelriktad kommunikation (databuss)
  - Måste kunna styra om en pinne ska fungera som ut eller ingång
  - Extra värden används (riktningsinformation) för varje bit
- Andra specialfunktioner
  - Drivning
  - Pull up/pull down
  - Tristate

Exempel BCM2835 (raspberry pi)

- Signaler till höger från/till olika register i I/O-enheten
- Varje pinne styrs av flera bitar
  - Värde som ska skickas ut
  - Välj om pinnen ska kunna skicka ut ett värde (tristate)
  - Välj om pinnen ska kunna skicka ut 0 och 1 eller bara 0 (open collector)
  - Välj om värdet alltid blir 0 eller 1 om pinnen inte ansluten (pull-up resp. Pull-down)
  - Kan styrka på utsignalen (max ström)
- Flera olika I/O-enheter kan anslutas till en pinne
  - Välj vilken funktion som ansluts via ytterligare ett register



http://www.mosaic-industries.com/embedded-systems/_media/microcontroller-projects/raspberry-pi/raspberry-pi-circuit-gpio-input-pins.png

GPIO kontrollregister, Darma

- Varje port har 36 register
  - 55 sidor i dokumentationen (Tiva TM4C123GH6PM Microcontroller Data Sheet)
  - 13 register styr funktion (in/ut, strömbegränsningar, funktionsval)
  - GPIODATA, värde att skicka ut och värde att läsa av
  - GPIODIR, om pinne är ut eller ingång
    - 0 = ingång, 1 = utgång
  - Titta gärna på lab1.asm och funktionerna initGPIOE respektive initGPIOF

Serieport (uart, RS232), Darma

- Tecken skickas från Darma till ansluten linux-maskin via uart
  - Översätter en byte till en ström av bitar med fast hastighet
    - Skickas över USB-sladden
  - Liten buffert finns inbyggd
    - Kan köa upp flera tecken och skicka dom allt eftersom
  - UART0_UARTDR, skickade respektive mottagna värden
    - Varje läsning tar bort ett värde ur kön av inkommande tecken
  - UART0_UARTFR, ange om data finns eller buffert full
    - Enskilda bitar i registret visar om inkommande tecken finns eller om det finns plats att skicka ut ett till

Andra I/O-enheter

- Kommunikation/styrning
  - Seriella standarder (RS232, SPI, I2C, USB, SATA)
  - Ethernet, bluetooth etc.
- GUI
  - Tangentbord, mus
  - Skärm
  - Ljud
- Diverse övriga
  - Kamera
  - Klocka, timer
  - Pulsviddsmodulering (PWM) för t ex motorstyrning
- Exempel: Raspberry pi
  - RS232 (UART)
  - Parallellport
  - USB (inkl. Ethernet via USB)
  - PCM/I2S ljud
  - I2C/SPI
  - PWM
  - Grafik (HDMI)
  - 205 sidors manual om I/O...
    - Saknar ändå mycket detaljer!

Att avbryta en aktivitet

Problem med nuvarande datormodell

- I många fall behöver maskinen göra flera saker samtidigt
  - Räkna tid och kontrollera sensorer
    - T ex blinka röd lysdiod och vänta på tangenttryck i lab 1
- Om beräkningar/andra aktiviteter samtidigt
  - Hinner kanske inte med att kontrollera sensorer i tid
  - Tillbringar för mycket tid att kontrollera sensorer
    - Varierande tidsräkning beroende på sensoraktivitet
  - Svårt att programmera
  - Måste om och om igen anropa rutin som läser av sensorer

Exempel på aktiviteter som måste hanteras direkt

- Tangentbordstryck
  - Kan trycka på ny tangent inom 50 ms
- Tecken som skickas över seriell kommunikationslänk
  - 115200 bitar/s => ~11000 tecken/s => ca 100 us per tecken
- Datapaket skickade över WLAN
  - Svar inom 16 us (eller snabbare)
- Olika applikationer har olika krav på max responstid

Ett försök att implementera samtidig kontroll

- Antag större beräkningsloop
  - Antag beräkningen kan beskrivas som 512 små steg (implementerat i subrutinen calculate_small_part)

```
        mov r5,#512
loop    bl calculate_small_part
        bl check_keyboard
        bl check_network
        subs r5,r5,#1
        bne loop
```

  - Subrutinerna måste returnera snabbt, annat missar man tecken
- Nackdel: Alla program måste hålla ordning på vad som ska kontrolleras (göra rätt anrop)
  - Måste kunna delas upp i små korta avsnitt

Analogi med ”riktiga världen”

- Antag ni ska äta frukost, läsa tidningen och samtidigt väntar telefonsamtal
- Lösning ett
  - Lyft luren och kontrollera om någon ringer till er
    - Om någon i andra änden: prata i telefonen, tills samtalet är över
  - Ta två tuggor flingor, svälj
  - Läs en artikel i tidningen
  - Börja om
- Lösning två
  - Ät två tuggor, läs en artikel i tidningen, börja om
  - Telefonen ringer vilket avbryter aktuell aktivitet (ni slutar äta, lämnar tidningen öppen)
    - Svara i telefon, prata
  - Återgå till att äta och läsa tidningen där ni var när telefonen ringde

Skillnader mellan metoderna

- Att kontinuerligt kontrollera/testa om något hänt är
  - Ineffektivt (många onödiga instruktioner som görs)
    - Speciellt om händelsen är sällsynt
  - Långsamt (måste vänta på att aktuell aktivitet slut innan byter)
  - Känt som ”programmed IO” eller ”busy wait I/O”
- Att bli avbruten (motsvarande att telefonen ringer) kräver
  - Möjlighet att mitt i exekvering av ett program hoppa till en annan rutin
    - Extra hårdvara (motsvarande ringklockan i telefonen)
  - Programmet som blir avbrutet ska inte märka att det hänt
    - Inga register, flaggor eller liknande i processorn som huvudprogrammet använder får förändras
  - Känt som avbrottsstyrd I/O

Avbrott

Att avbryta datorns huvudprogram

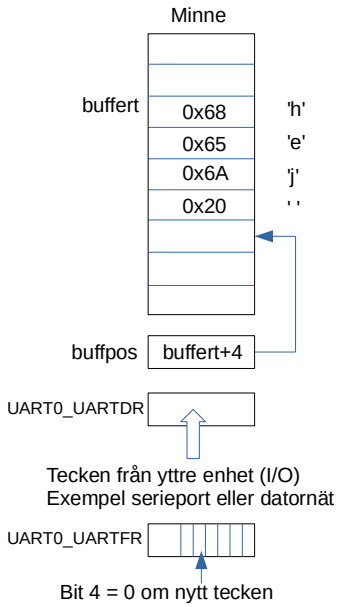
- En I/O-enhet (t ex parallellport) skickar signal
  - Kallas för en avbrottsbegäran
- Processorn avbryter nuvarande program
  - Gamla värdet på PC och statusregister (Z,C,N flaggor mm) sparas på stacken
  - Byt värde på PC till speciell avbrottsrutin
  - Ta hand om orsaken till avbrottet
  - Återställ statusregister och PC (hämta från stacken)
  - Programmet fortsätter som innan avbrottet
- Avbrottshantering liknar subrutinanrop
  - Skillnad: Även staturegister med flaggor sparas

Fler exempel på användning av avbrott

- Signal från I/O-enhet
  - Tangent nedtryckt (jfr strobe från tangentbord)
- Timeravbrott
  - En viss tid har förflutit
- Systemanrop
  - Applikationsprogram anropar ett operativsystem för att utföra en speciell funktion
- Fel i programmet
  - Division med 0, läsning av ord (16 eller 32 bitar) på udda adress, läsning i ej existerande minne, och liknande
- Ett datapaket har kommit in på nätverkskortet
- Datablock är inläst från hårddisk till minne

Exempel motivation till avbrott

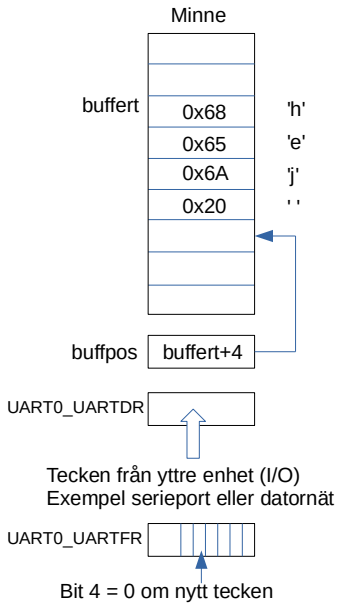
- Antag vi har en I/O enhet som tar emot tecken i hög hastighet.
  - Om UART0_UARTDR inte töms innan nästa tecken från I/O så tappas tecken bort
  - Spara inkommande tecken (finns i UART0_UARTDR) i en buffert i minnet
- Subrutin för att spara värde i buffert
  - Variabel buffpos pekar på nästa lediga plats i buffert
  - Subrutinen kontrollerar om nytt tecken finns, och i så fall placera det på lediga platsen i bufferten och uppdatera buffpos till nästa lediga plats.



Exempel motivation till avbrott

- Subrutin för att spara värde i buffert

```
savechar push {r0,r1,r2}                    ; Kommer använda registren
         mov r1,#(UART0_UARTFR & 0xffff)    ; Peka på
         movt r1,#(UART0_UARTFR >> 16)      ; statusregister
         ldr  r2,[r1]                       ; Hämta status
         ands r2,r2,#0x10                   ; Finns tecken?
         bne skipcopy                       ; nej (bit 4= 1), hoppa ur
         mov r1,#(UART0_UARTDR & 0xffff)    ; peka på
         movt r1,#(UART0_UARTDR >> 16)      ; dataregistret
         ldr  r2,[r1]                       ; läs tecken
         mov r1,#(buffpos & 0xffff)         ; peka på variabeln
         movt r1,#(buffpos >> 16)           ; buffpos
         ldr  r0,[r1]                       ; hämta värdet i buffpos
         strb r2,[r0],#1                    ; spara tecken i buffert, öka r0
         str  r0,[r1]                       ; öka adress i buffpos
skipcopy pop {r0, r1, r2}                   ; återställ register
         bx lr
```



Exempel motivation till avbrott, forts.

- Vill samtidigt som tecken kommer kunna beräkna summa n+(n-1)+....+1. (se exemplet tidigare i föreläsningen)
  - Lägg till anrop till test om tecken tillgängligt

```
         mov     r0,#66
         bl      nsum
          ...

nsum:    push    {r1,lr}
         mov     r1,#0
nextturn: add    r1,r1,r0
         bl      savechar
         subs    r0,r0,#1
         bne     nextturn
         mov     r0,r1
         pop     {r1,lr}
         bx      lr
```

```
savechar push {r0,r1,r2}                ; Kommer använda registren
         mov r1,#(UART0_UARTFR & 0xffff)  ; Peka på
         movt r1,#(UART0_UARTFR >> 16)    ; statusregister
         ldr r2,[r1]                      ; Hämta status
         ands r2,r2,#0x10                 ; Finns tecken?
         bne skipcopy                     ; nej (bit 4= 1), hoppa ur
         mov r1,#(UART0_UARTDR & 0xffff)  ; peka på
         movt r1,#(UART0_UARTDR >> 16)    ; dataregistret
         ldr r2,[r1]                      ; läs tecken
         mov r1,#(buffpos & 0xffff)       ; peka på variabeln
         movt r1,#(buffpos >> 16)         ; buffpos
         ldr r0,[r1]                      ; hämta värdet i buffpos
         strb r2,[r0],#1                  ; spara tecken i buffert, öka
r0
         str r0,[r1]                      ; öka adress i buffpos
skipcopy pop {r0, r1, r2}                 ; återställ register
         bx lr
```

- Kan inte placera anrop var som helst
  - Om mellan SUB och BNE kraschar programmet

Exempel på busy wait istället för avbrott

- För varje varv i loop (tidigare 3 instruktioner) som utförs 8 extra instruktioner för att testa om tecken behöver tas om hand
  - 1/4 av hastigheten mot tidigare
- Kan minska förlusten av beräknings-hastighet genom t ex behålla konstanter i register som behövs för test.
  - Måste hålla ordning på vilka register som förstörs av savechar
- Kan minska förlusten av beräknings-hastighet genom att t ex räkna två steg i loop innan test om tecken finns, etc.
  - Komplicerad kod, massa olika gränsfall

```
        mov     r0,#66
        bl      nsum
        ...

nsum:   push {r1,lr}
        mov r1,#0
nextturn: add   r1,r1,r0
        bl   savechar
        subs r0,r0,#1
        bne  nextturn
        mov  r0,r1
        pop  {r1,lr}
        bx  lr

savechar: push {r0,r1,r2}           ; använda registren
        mov  r1,#(UART0_UARTFR & 0xffff)    ; Peka på
        movt r1,#(UART0_UARTFR >> 16)     ; statusregist
        ldr  r2,[r1]                ; Hämta status
        ands r2,r2,#0x10            ; Finns tecken?
        bne skipcopy          ; nej (bit 4= 1), hoppa ur
        mov r1,#(UART0_UARTDR & 0xffff)     ; peka på
        movt r1,#(UART0_UARTDR >> 16)     ; dataregistr
        ldr  r2,[r1]                ; läs tecken
        mov r1,#(buffpos & 0xffff)  ; peka på variabeln
        movt r1,#(buffpos >> 16)   ; buffpos
        ldr  r0,[r1]                ; hämta buffpos
        strb r2,[r0],#1          ; spara i buffert, öka r0
        str  r0,[r1]                ; spara ny buffpos
skipcopy: pop  {r0, r1, r2}             ; återställ register
        bx  lr
```


Exempel på avbrott, inte riktigt rätt

- Lägg till speciell funtion i datorn
  - Låt subrutinen savechar starta så fort ett tecken kommit
  - Speciell hårdvara sparar aktuellt värde i PC på stacken och placerar adressen savechar i PC
  - När subrutinen är slut (bx lr utförs) fortsätter nsum som vanligt och nya avbrott tillåts
- Problem: lr förstörs
- Ytterligare problem: Vad händer om tecken kommer efter SUBS instruktionen?
  - savechar ändrar Z-flaggan!

```
        mov     r0,#66
        bl    nsum
        ...

nsum:     push {r1,lr}
        mov     r1,#0
nextturn: add     r1,r1,r0
        bl  savechar ; sköts av avbrottshårdvara
        subs r0,r0,#1
        bne     nextturn
        mov     r0,r1
        pop     {r1,lr}
        bx  lr

savechar: push {r0,r1,r2}
        mov     r1,#(UART0_UARTFR & 0xffff)
        movt r1,#(UART0_UARTFR >>16)
        ldr  r2,[r1]
        ands r2,r2,#0x10
        bne     skipcopy
        mov     r1,#(UART0_UARTDR & 0xffff)
        movt r1,#(UART0_UARTDR >> 16)
        ldr  r2,[r1]
        mov     r1,#(buffpos & 0xffff)
        movt r1,#(buffpos >> 16)
        ldr  r0,[r1]
        strb    r2,[r0],#1
        str  r0,[r1]
skipcopy: pop     {r0, r1, r2}
        bx  lr
```

Exempel på avbrott, rätt sätt

- Se till att även spara alla flaggor (Z,C,N,V etc.) dvs PSR registret, när savechar startas
  - Avbrott är mer än ett automatisk subrutinanrop
- Måste återställa flaggor när savechar är slut
  - Speciell återhoppsadress
- Även LR behöver återställas
- Vill inte behöva bestämma i förväg exakt vilken adress rutinen startar på
  - Fix minnesadress innehåller adress till rutin (PC läses från speciell minnesadress vid avbrottets start där adressen till savechar ligger lagrad)

```
            mov  r0,#66
            bl   nsum
             ...
nsum:       push {r1,lr}
            mov  r1,#0
nextturn:   add      r1,r1,r0
            bl   savechar
            subs r0,r0,#1
            bne      nextturn
            mov r0,r1
            pop      {r1,lr}
                bx   lr
savechar:   push {r0,r1,r2}
            mov  r1,#(UART0_UARTFR & 0xffff)
            movt r1,#(UART0_UARTFR >>16)
            ldr  r2,[r1]
            ands r2,r2,#0x10
            bne      skipcopy
            mov r1,#(UART0_UARTDR & 0xffff)
            movt r1,#(UART0_UARTDR >> 16)
            ldr  r2,[r1]
            mov r1,#(buffpos & 0xffff)
            movt r1,#(buffpos >> 16)
            ldr  r0,[r1]
            strb     r2,[r0],#1
            str  r0,[r1]
skipcopy:   pop      {r0, r1, r2}
            bx   lr
```

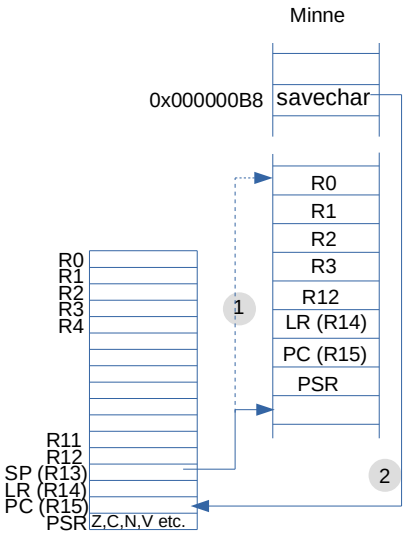

Avbrott i Darma

Avbrott på ARM

- ARM har olika sätt att hantera avbrott beroende på processormodell
  - Boken beskriver Cortex-A och tidigare modeller
    - Byter vissa register till nya när avbrott sker (banked registers)
    - Har väldigt få avbrott definierade
  - Darma använder Cortex-M som har en lite annorlunda avbrotthantering
- Gemensamt för alla
  - Avbrott signaleras från någon enhet till processorn
  - Avbrott medför att nuvarande program avbryts
    - Register sparas och/eller byts ut mot nya versioner
  - En adress ur vektortabell (i programminnet) anger var avbrottsrutinen ligger som ska startas

Avbrott på ARM Cortex-M4 (Darma)

- Avsnitt 2.5 i Tiva TM4C123GH6PM Microcontroller Data Sheet.
  - Kortbeskrivning i DARMA-dokument avsnitt 4.3
- När avbrottssignal aktiveras
  - Automatisk push av PSR, PC, LR, R12, R3, R2, R1, R0 på stack (SP (R13) minskas) (steg 1)
  - Läs fix minnexadress (t ex 0x000000B8-0x000000BB) som innehåller adress till avbrottrutin (steg 2)
  - Hantera avbrottet (t ex läs tangentbord)
  - Stäng av källan till avbrott
  - Återställ automatiskt register med pop av PSR, PC, LR, R12, R3, R2, R1 och R0 när bx lr körs. (steg 1 åt andra hållet)



Viktigt: osynlig avbrottshanterare

- Inga register (R4-R11) får ha ändrats när avbrottsrutin avslutas
  - Programmet som avbrutits får inte märka att avbrott skett
  - R0-R3 och R12 sparas redan automatiskt vid start av avbrott
  - Andra använda register måste spara undan först
- Vanligt att spara register först, återställa register sist

```
avbrottsrutin:push {r4,r5,r6}
              ....              ; aktivitet, tex läs tangent
              pop {r4,r5,r6}
              bx  lr            ; bx lr
```

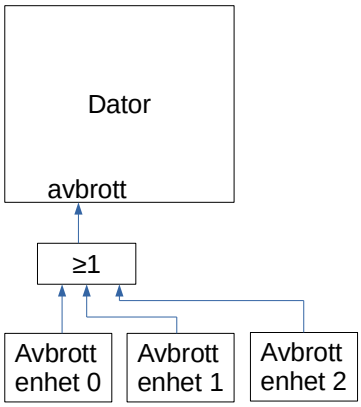

Spara energi med hjälp av avbrott

- I studenträknarexemplet kontrollerades sensorerna hela tiden (busy wait)
  - En loop som alltid läste nuvarande sensorläge, kollade värdet, och om inte aktivt provade igen
- Med avbrott kan processorn stanna
  - Ställ in hårdvaran så avbrott fås när knapp trycks
  - Inga instruktioner behöver avkodas innan sensorn skapar ett avbrott
  - Kan stoppa processorn (sleep mode)
    - Speciell instruktion/hårdvara behövs för att det ska fungera
    - WFI (wait for interrupt) (<- speciell instruktion i Darma)
  - Kan minska strömförbrukningen
    - Inga minnesaccesser, ingen instruktionsavkodning, inga registeruppdateringar

Flera avbrottsorsaker

- I exemplen tidigare kunde många saker ge avbrott
  - Tangenttryck på ett tangentbord
  - Paket till nätverkskort
- Enklaste implementeringen: Låt alla avbrottsorsaker köra igång samma avbrottsrutin
  - Måste hitta vilken/vilka enheter som orsakade avbrottet
  - Avbrottsrutinen startar alltid med att kontrollera orsak till avbrott

```
avbrottsrutin:
        var det enhet 1?
            ja, hoppa till rutin för enhet 1
        nej inte enhet 1. Var det enhet 2?
            ja, hoppa till rutin för enhet 2
         :
```



Flera källor till avbrott

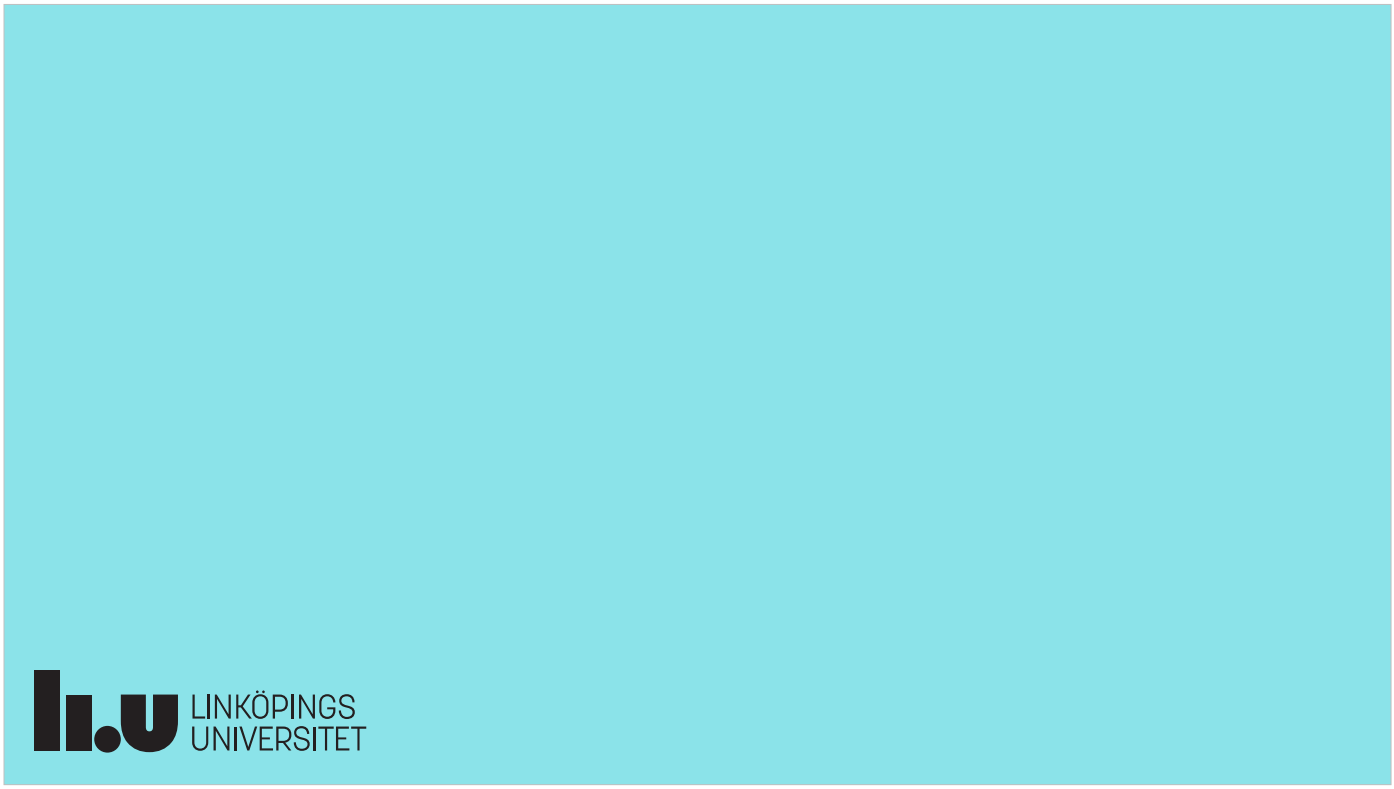
- Avbrott behöver åtgärdas olika fort beroende på källa
  - Mikrosekunder för WLAN
  - Millisekunder för tangentbord
- Avbrottrutinen får inte vara för lång
  - Nästa avbrott från en annan källa kan behöva tas om hand
  - Avbrott som kräver snabb åtgärd kan behöva bryta rutiner som åtgärdar avbrott med lång tid på sig
- Jämför frukost, när en kompis ringer på dörren, och sedan ringer telefonen
  - Besöket avbryter frukost, och telefonen avbryter samtalet med kompisen

Flera avbrottsnivåer

- Cortex-M4 (Darma) har flera avbrottsprioriteter
  - Sköts av en speciell enhet, Nested Vectored Interrupt Controller (NVIC)
  - Viktigaste: Reset (prioritet -3)
  - NMI (prioritet -2), speciell avbrottssignal
  - Hard fault (prioritet -1), t ex om adress till avbrottvektor inte kan läsas
  - Prioritet 0-7: avbrottsprioritet som kan väljas i NVIC
    - Prioritet 0: Högst avbrottsprioritet
    - Prioritet 7: Lägst avbrottsprioritet
- Avbrott med högre prioritet (lägre nummer) kan avbryta lägre prioriterat avbrott (högre nummer)

Avbrott kan avbryta pågående avbrott

- Vanlig användning av avbrott: kör kod med visst tidsintervall
  - Realtidsklocka i en dator
  - Skicka ut ljud på högtalare
- Ett avbrott med lägre priortet kan avbrytas
  - Det nya avbrott av högre prioritet kommer spara adress, flaggor etc på stacken på samma sätt som för avbrottet med låg prioritet
  - När högre priortetsavbrottet avslutas så återställs flaggor etc, och avbrottet med låg prioritet fortsätter


LINKÖPINGS
UNIVERSITET