

Laborationer med MC12 Realtidssystem

©GMV 2004

Version 2011-08-12

Läromedel på elektronisk form, LOMEK, får kopieras fritt

Du ska redovisa dina laborationsresultat vid kontrollstationer. Då du nått en sådan, ska du därför tillkalla handledare som kontrollerar och fyller i följande tabell.

Kontroll station	Uppvisat för:	Godkänt (Datum)	Godkänt (Signatur)
1			
2			
3			
4			
5			
6			
7			
8			

Namn och grupp (skriv ditt namn och linje/klass/grupp tillhörighet här)

OBSERVERA:

Denna handledning förutsätter att du har installerat XCC12, Version 1.8

Notera speciellt att följande biblioteksfunktioner getts nya namn:

`wait` heter nu `waitsem`

`signal` heter nu `signalsem`

Innan du börjar...

I detta häfte behandlas laborationer med enkortsdatorn *MC12* och laborationskortet *ML13*.

Handböcker som beskriver *MC12* och *ML13* finner du på GMV's hemsida, www.gbgmv.se.

Laborationerna genomförs med programutvecklingsmiljön *XCC12*.

I *XCC* finns simulatorer för såväl *MC12* som *ML13*. Hjälpssystemet i *XCC* innehåller beskrivningar av hur du konfigurerar simulatören och ansluter en simulerad variant av *ML13*. Praktiskt taget alla uppgifter kan lösas med hjälp av simulatorerna i *XCC*.

Det förutsätts att du har en grundläggande förståelse för mikroprocessorn *MC68HC12* och att du tidigare bekantat dig med dess instruktionsrepertoir och dessutom självständigt genomfört viss assemblerprogrammering. Det förutsätts vidare att du har grundläggande programmeringskunskaper, speciellt i programspråket 'C'.

1. Tidsdelning

Under momentet studerar du:

- Parallelexekvering
- Klockrutinen
- MC68HCS12's periodiska räknare

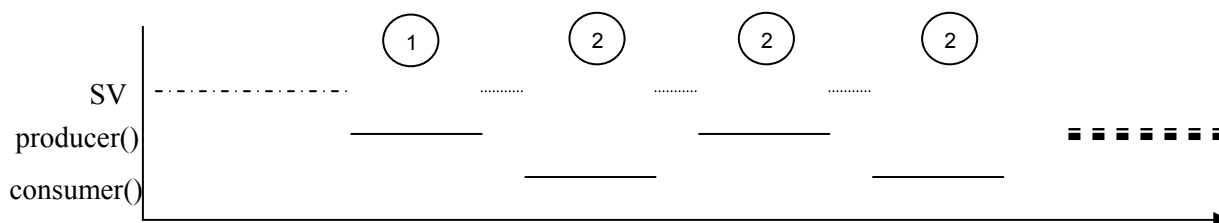
Under denna inledande laboration ska vi visa ett enkelt exempel på hur *tidsdelning* (*time-sharing*) kan utföras. Processortiden ska delas lika mellan två *applikationsprogram*:

- `producer()` producerar ASCII-tecknen 'a','b','c'.....osv och placerar tecknen efter hand i en buffert.
- `consumer()` läser tecken från buffert och skriver dessa till bildskärmen..

För att genomföra uppgiften måste du konstruera programrutiner för *stackhantering* och *avbrott*, låt oss först studera problemet i detalj.

Beskrivning av uppgiften

Programexekveringen illustreras av följande figur som visar hur processor'n tilldelas de olika programdelarna. Med 'SV' menas här *supervisor*, dvs de delar du själv ska konstruera:



SV hanterar situationerna '1' och '2' ovan. Vi börjar med situation '2', den är enklast:

- 2) Ett program har blivit avbrutet av systemets räknarkretsen, vi måste tillhandahålla en avbrottsrutin 'timer_interrupt', denna ska:
 - a) spara programmets flyktiga omgivning
 - b) starta det andra programmet

Programmets "flyktiga omgivning" utgörs av processorns registerinnehåll. Vid ett avbrott i HC12 sparas automatiskt samtliga registerinnehåll på stacken, vi behöver inte bekymra oss mer för detta.

Nästa steg blir att "starta nästa program", vilket är då detta??? ('producer' eller 'consumer'), vi behöver uppenbarligen en variabel, låt oss kalla den RUNNING, som anger vilket program som exekveras. Om RUNNING är 0, exekveras program 'producer()' om RUNNING är 1 exekveras program 'consumer()'. Vi är nu klara för detaljerna i den så kallade *klockrutinen*.

Klockrutinen

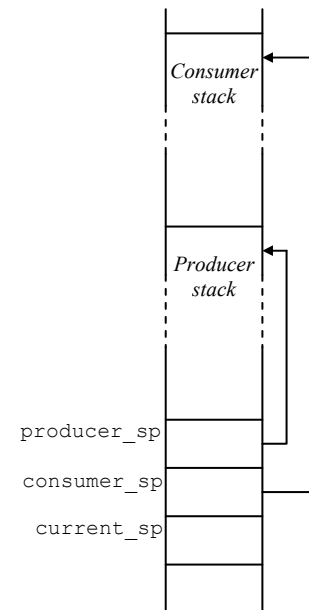
Algoritmen för 'timer_interrupt' måste utformas så att den klarar av att återstarta ett program. Eftersom vi sparar ett avbrutet program's flyktiga omgivning på något sätt måste vi också återställa det på motsvarande sätt. Varje program har sin egen stack i minnet enligt figur på nästa sida. För att kunna spara ett programs stackpekare behov ett antal variabler *current_sp*, *producer_sp*..

Algoritmen/strukturen för rutinen 'timer_interrupt' som skall se till att det andra programmet börjar exekveras där de avbröts tidigare:

timer_interrupt:

```

save processor context; ( sker automatiskt för MC12)
save current stackpointer in current_sp;
if(RUNNING==producer_id){
    producer_sp = current_sp;
    RUNNING = consumer_id;
    current_sp = consumer_sp;
}else{
    consumer_sp = current_sp;
    RUNNING = producer_id;
    current_sp = producer_sp;
}
restore current_sp;
restore processor context; ( sker automatiskt för MC12)
kvittera avbrottet; (RTC modulen i MC12 kräver detta)
return from interrupt;
```



Initieringar av stackar och periodiska räknaren

Du måste också hantera "situation 1" dvs åstadkomma alla nödvändiga initieringar. Då 'producer()' programmet startas första gången, bör detta bli på samma sätt som då avbrottsrutinen startar ett nytt program, det innebär att vi måste skapa en initial stack för båda programmen.

Följande sekvens gör det jobbet:

```

* Nedan skapas en initial aktiveringspost för 'producer'
LDS    #a_stack_producer
LDX    #_producer
PSHX                    PC - till producer
PSHY                    Y - innehållet är odefinierat
PSHX                    X - innehållet är odefinierat
PSHA                    A - innehållet är odefinierat
PSHB                    B - innehållet är odefinierat
* sist placerar vi CCR för process 1, här måste I-flaggan
* vara 0 för att processbyten ska kunna utföras senare. Vi kan dock
* inte nollställa I-flaggan ännu, ty avbrottshanteringen är ännu inte
* fullständigt initierad, vi använder A-registret för att skapa ett
* CC-innehåll på stacken
TPA                    CC -> A
ANDA    #%11101111    0 -> I-flagga
PSHA
STS    producer_sp
```

Motsvarande initiering görs även för 'consumer()'. Nästa steg blir att initiera systemets räknarkretsen så att den genererar avbrott med jämna mellanrum. Vi använder här den periodiska räknaren hos MC68HCS12, den är avsedd för sådana här ändamål och därför också enkel att hantera.

Periodiska räknaren

MC68HCS12 innehåller en modul CRG (*Clocks and Reset Generator*) med flera funktioner så som namnet antyder. Speciellt finns en RTI (*Real Time Interrupt*) funktion som vi ska använda här. Modulen har alltså fler funktioner men vi nöjer oss med att behandla RTI.

RTI-funktionen utgörs av en enkel räknare kopplad till centralenhetens avbrottsingång. Funktionen kontrolleras via tre register:

CRGINT (adress \$38)

Registret används för att aktivera avbrott

	7	6	5	4	3	2	1	0
RTIE	0	0	LOCKIE	0	0	SCMIE	0	

- RTIE: Aktivera avbrott från RTI-funktionen. Denna bit måste sättas till 1 för att avbrott ska genereras.
- LOCKIE, SCMIE, används ej här. Ska vara 0.

RTICTL (adress \$3B)

Registret används för att initiera en tidbas för den periodiska räknaren. En skrivning till detta register aktiverar RTI-funktionen.

	7	6	5	4	3	2	1	0
0	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0	

RTR-bitarna bestämmer avbrottsintervallet från räknaren. Systemets klockfrekvens (8 MHz) delas med ett tal specificerat av RTR-bitarna enligt följande tabell:

RTR [3:0]	RTR[6:4]							
	000 (OFF)	001	010	011	100	101	110	111
0000	OFF	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
0001	OFF	2×2^{10}	2×2^{11}	2×2^{12}	2×2^{13}	2×2^{14}	2×2^{15}	2×2^{16}
0010	OFF	3×2^{10}	3×2^{11}	3×2^{12}	3×2^{13}	3×2^{14}	3×2^{15}	3×2^{16}
0011	OFF	4×2^{10}	4×2^{11}	4×2^{12}	4×2^{13}	4×2^{14}	4×2^{15}	4×2^{16}
0100	OFF	5×2^{10}	5×2^{11}	5×2^{12}	5×2^{13}	5×2^{14}	5×2^{15}	5×2^{16}
0101	OFF	6×2^{10}	6×2^{11}	6×2^{12}	6×2^{13}	6×2^{14}	6×2^{15}	6×2^{16}
0110	OFF	7×2^{10}	7×2^{11}	7×2^{12}	7×2^{13}	7×2^{14}	7×2^{15}	7×2^{16}
0111	OFF	8×2^{10}	8×2^{11}	8×2^{12}	8×2^{13}	8×2^{14}	8×2^{15}	8×2^{16}
1000	OFF	9×2^{10}	9×2^{11}	9×2^{12}	9×2^{13}	9×2^{14}	9×2^{15}	9×2^{16}
1001	OFF	10×2^{10}	10×2^{11}	10×2^{12}	10×2^{13}	10×2^{14}	10×2^{15}	10×2^{16}
1010	OFF	11×2^{10}	11×2^{11}	11×2^{12}	11×2^{13}	11×2^{14}	11×2^{15}	11×2^{16}
1011	OFF	12×2^{10}	12×2^{11}	12×2^{12}	12×2^{13}	12×2^{14}	12×2^{15}	12×2^{16}
1100	OFF	13×2^{10}	13×2^{11}	13×2^{12}	13×2^{13}	13×2^{14}	13×2^{15}	13×2^{16}
1101	OFF	14×2^{10}	14×2^{11}	14×2^{12}	14×2^{13}	14×2^{14}	14×2^{15}	14×2^{16}
1110	OFF	15×2^{10}	15×2^{11}	15×2^{12}	15×2^{13}	15×2^{14}	15×2^{15}	15×2^{16}
1111	OFF	16×2^{10}	16×2^{11}	16×2^{12}	16×2^{13}	16×2^{14}	16×2^{15}	16×2^{16}

CRGFLG (adress \$37)

Statusregister, alla bitar är läsbara och varje bit representerar någon händelse.

7	6	5	4	3	2	1	0
RTIF	PORF	0	LOCKIF	LOCK	TRACK	SCMIF	SCM

- RTIF: Biten sätts till 1 vid avbrott från RTI-funktionen. Avbrottsignalen kvitteras genom att en etta skrivs till RTIF-biten.
- Övriga bitar, används ej här, nollor kan skrivas till dessa bitar utan att påverka någon funktion.

Då räknaren initierats/aktiverats, kommer den att räkna ned ett intervall och därefter begära avbrott, räknarvärdet initieras därefter på nytt automatiskt av kretsen och ett nytt intervall påbörjas. Avbrottet måste kvitteras för att återställa IRQ-signalen till en passiv nivå, detta görs alltså i en avbrottshanterinsrutin.

Oftast är det inte möjligt att åstadkomma tidbaser i basen 10. Följande exempel illustrerar hur man bestämmer RTR-bitarna för ett (approximativ) 10 ms avbrottsintervall:

EXEMPEL:

Önskat avbrottsintervall **10ms**

Klockfrekvens (MC12): **8 MHz**

Det gäller att:

$$8 \text{ MHz} = \text{RTR} * 10 \text{ ms}$$

dvs

$$\text{RTR} = 80000$$

Detta värde kan inte representeras (se tabell ovan) och vi måste därför hitta den bästa approximationen, i vårt fall:

$$\text{RTR} (= 100\ 1001 = \$49) \text{ som medför: } 10 \times 2^{13} = 81920$$

Eftersom detta värde är något större än det optimala, kommer vi att få en något längre periodtid, nämligen:

$$\text{avbrottsfrekvens} = 8 \cdot 10^6 / 81920 = 97.656 \text{ Hz}$$

vilket ger periodtiden

$$0.01024 \text{ s} = 10,24 \text{ ms.}$$

Klocka kommer alltså att "släpa efter" som en följd av detta systematiska fel.

Följande kodsekvenser ger strukturen för en rutin som initierar den periodiska räknaren för att anropa en viss avbrottsrutin i *MC68HCS12*.

```

segment      abs

* Adressdefinitioner för RTC-kretsen i HC12

CRGFLG      equ    $37
CRGINT      equ    $38
RTICTL      equ    $3B
* Flagga
RTIF        equ    $80

timer_init:
* avbrottsvektor måste initieras under DBG12...
    LDX    #timer_interrupt
    STX    $3FF0      Avbrottsvektor RTC
* men vi skriver den också till 'rätt' adress för
* att även kunna använda exemplet i simulatorn...
    STX    $FFF0

* initiera RTC:n för önskad avbrottsfrekvens
    MOVB   #$49,RTICTL
* tidsbas 10,24 ms, se exempel ovan
* initiera RTC
    MOVB   #$80,CRGINT      aktivera avbrott

* nollställ I-flagga så att avbrott accepteras
    CLI

```

Not: Avbrottet måste alltid kvitteras genom att ettställa RTIF flaggan i CRGFLG-registret.

Programmet:

Det är nu dags att sätta ihop allt till ett 'main'-program. Följande "skelett" ger dig huvuddragen för programmet 'mom1-low.s12'. Applikationsprocesserna (producer/consumer) finner du nedan.

```

* mom1-low.s12
    segment      abs
* Definitioner för RTC-kretsen i HC12
CRGFLG      equ    $37
CRGINT      equ    $38
RTICTL      equ    $3B
* Flagga
RTIF        equ    $80

STKSIZE     EQU    $40      stackutrymmen för program

* Datadeklarationer
    bss
* 'STKSIZE' bytes stackutrymme för 'producer'
    RMB      STKSIZE-1
a_stack_producer RMB    1

'STKSIZE' bytes stackutrymme för 'consumer'
    RMB      STKSIZE-1
a_stack_consumer RMB    1

RUNNING     RMB    1      anger exekverande program
* temporär lagring stackpekare för 'RUNNING'
current_sp  RMB    2

* temporär lagring stackpekare för 'producer'
Producer_sp RMB    2

* temporär lagring stackpekare för 'consumer'
consumer_sp RMB    2

```

```
segment      text      starta KOD-segment

* Följande funktioner är definierade i 'mom1.c'
extern      _producer
extern      _consumer
*
*   Programexekveringen startar här
*
define      _main
entry      _main
_main:
    init_producer; (Skriv koden här ej som subrutin)
    init_consumer; ( dito )
    init_RTC;      ( dito )
*
*   utför nu start av 'producer()' så som
*   'timer_interrupt' gör det...
    CLR     RUNNING
    LDS     producer_sp
    RTI
*   Exekveringen kommer aldrig tillbaks hit...
```

producent-konsument processerna ska se ut på följande sätt:

```
/*
    mom1.c
    Enkel 'producent/konsument'
*/

#include      <_startup.h> // För '_outchar'

// 25 tecken i engelska alfabetet
#define BUFSIZE      25+1

char  buffer[BUFSIZE];
int  position;

void  producer(void)
{
    char  tecken;
    position = -1;
    tecken = 'a';
    while(1){ // oändlig slinga
        if(position < BUFSIZE-1){
            buffer[++position]=tecken;
            if(tecken == 'z')
                tecken = 'a';
            else
                tecken++;
        }
    }
}

void  consumer(void)
{
    char  tecken;
    while(1){ // oändlig slinga
        if(position >= 0){
            tecken = buffer[position--];
            _outchar( tecken);
        }
    }
}
```


Implementering och test

Du bör nu vara mogen att implementera och testa denna första enkla applikation. Du kan använda den inbyggda simulatoren i MCC12 tillsammans med IO-simulatoren för att testa programmet *innan* du går till laborationsplatsen. Följande arbetsgång kan vara lämplig:

- Starta MCC12
- Skapa ett nytt 'Workspace' - namnge det 'RTLAB'.
- Skapa ett nytt projekt, 'moment1', använd standardinställningarna som föreslås.
- Skapa källtexterna 'mom1-low.s12' respektive 'mom1.c' enligt tidigare anvisningar.
- Lägg de nya källtexterna till projektet.
- Välj 'Build All' för att skapa applikationen.
- Testa programmet med MCC12's debugger, använd simulatoren för att koppla 'Console' till seriekommunikationskretsen så att du kan se programmets utskrifter.

Tänk på att realtidsegenskaperna skiljer sig markant mellan simulator och den verkliga laborationsutrustningen. Allmänt gäller att det går betydligt långsammare i simulatoren. Använd kortast möjliga tidbas då du testar i simulatoren, annars får du vänta länge på utskrifter från programmet.

Vid laborationsplatsen:

Tänk på att du ska ha med dina egna källtextfiler till laborationsplatsen. Du gör nu på samma sätt som under dina tidigare förberedelser (implementering och test) men laddar det färdiga programmet till laborationsdatorn i stället för debuggern.

Provkör programmet med periodtid (10.24 ms) för klockrutinen, granska en utskriftsföljd som börjar på 'a', vad ser du? (Skriv upp de 10 påföljande tecknen):

Ändra nu initieringen av räknaren så att ett kortare avbrottsintervall används, kompilera om och ladda ned på nytt, provkör programmet. granska en utskriftsföljd som börjar på 'a', vad ser du? Utskrifterna kommer betydligt snabbare, avbryt laborationsdatorn genom att trycka på reset-knappen, granska en utskriftsföljd som börjar på 'a', vad ser du? (Skriv upp de 10 påföljande tecknen)

Granska ytterligare tre andra (godtyckligt valda) utskriftsföljder som börjar på 'a', skriv upp de 10 påföljande tecknen)

Är detta ett förväntat resultat?

Försök förklara likheter/skillnader?

Vilka slutsatser kan du dra av denna implementering av producent/konsument-problemet?

Kontrollstation 1

2. Tidsstyrd dörrautomat

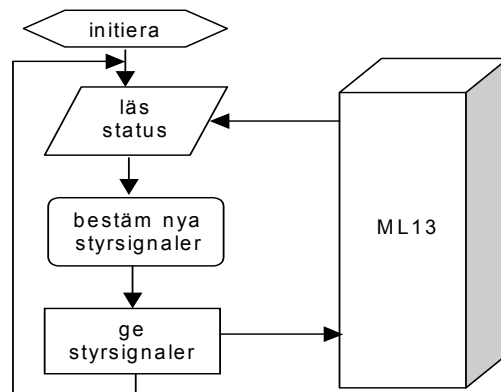
Under detta moment ska du implementera en 'dörrautomat' med ML13 utan att använda avbrott.

En enkel beskrivning av ML13 finns under hjälpen i XCC12

Den tidsstyrda implementeringen av en dörrautomat ska fungera på följande sätt:

- Om någon närmar sig dörrutrymmet ska dörren öppnas.
- Efter att ha varit öppen (i några sekunder) ska dörren stängas automatiskt.
- Dörren får inte stängas om det finns någon kvar i dörrutrymmet
- Om någon närmar sig dörrutrymmet under tiden dörren stängs ska den omedelbart öppnas igen.

Huvudprogrammets struktur beskrivs grovt av följande illustration:



Implementeringen ska göras, helt och hållet, i programspråket 'C'.

Tips:

För att läsa från *ML13*'s IO-portar direkt i 'C' kan först följande "macros" definieras:

```
#define ML13_Status      0xB00
#define read_control  *((char *)ML13_Status)
```

macroet kan sedan användas, exempelvis på följande sätt:

```
if( read_control & 0x03 ){ ....}
```

där if-satsen utförs om någon av bit 0 eller bit 1 i *ML13*'s statusregister är 1.

För att skriva till *ML13*'s IO-portar direkt i 'C' kan först följande "macros" definieras:

```
#define ML13_Control      0xB00
#define set_control(x)  *((char *)ML13_Control)=x
```

macroet kan sedan användas, exempelvis på följande sätt:

```
set_control(0x1);
```

satsen skriver värdet 1 till *ML13*'s styrregister

Implementering och test

Läroboken ger ett utförligt förslag på hur denna uppgift ska lösas'.

- Skapa ett nytt projekt 'moment2' i workspace 'RTLAB'. Använd standardinställningarna som föreslås.
- Skapa en källtextfil MOM2.C med ett huvudprogram som löser laborationsuppgiften. Lägg till denna fil till projektet.
- Testa programmet med XCC12's debugger, använd simulatoren för att koppla ML13 till adress \$B00 så att du kan observera "dörrens" beteende.

Vid laborationsplatsen

Tänk på att du ska ha med dina egna källtextfiler till laborationsplatsen. Du gör nu på samma sätt som under dina tidigare förberedelser (implementering och test) men laddar det färdiga programmet till laborationsdatorn i stället för debuggern.

Kontrollstation 2

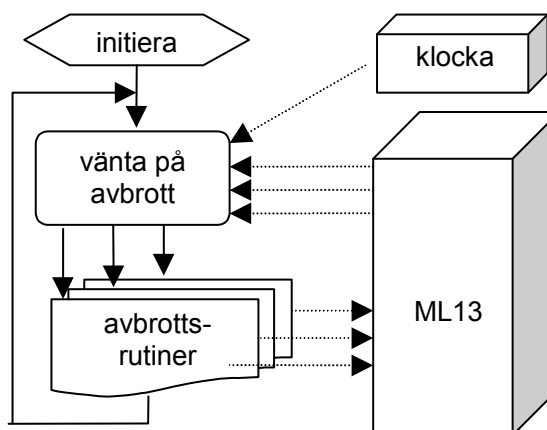
3. Händelsestyrd dörrautomat

Under detta moment tar vi hjälp av ML13's avbrottsmekanismer och implementerar en händelsestyrd dörrautomat.

Den händelsestyrda implementeringen av dörrautomaten ska ha samma funktion som den tidsstyrda implementeringen, dvs:

- Om någon närmar sig dörrutrymmet ska dörren öppnas.
- Efter att ha varit öppen (i några sekunder) ska dörren stängas automatiskt.
- Dörren får inte stängas om det finns någon kvar i dörrutrymmet
- Om någon närmar sig dörrutrymmet under tiden dörren stängs ska den omedelbart öppnas igen.

Huvudprogrammets struktur beskrivs denna gång av följande illustration:



Uppgift 3a:

Ni skall i huvudsakligen skriva programmet i 'C'. Vi väljer dock initialt att skriva en del rutiner i assembler för att klara avbrottshanteringen. Nedan följer ett assemblerprogram som ni får givet.

Ni skall som första uppgift skriva ett lämpligt C-program som tillsammans med det givna assemblerprogrammet löser den aktuella uppgiften.

```
*
*   Assemblerrutiner för
*   händelsestyrd "dörrautomat"

        segment      abs
ML13_IRQ_Status EQU   $B01
ML13_IRQ_Control EQU  $B01

NO_IRQ_TYPE      EQU   0
SENSOR            EQU   1
CLOSED_DOOR       EQU   2
OPENED_DOOR       EQU   4
CLOSING_DOOR      EQU   8
OPENING_DOOR      EQU  16
TIME_OUT          EQU  32

*
* Definitioner för RTC-kretsen i HC12
CRGFLG            EQU   $37
CRGINT            EQU   $38
RTICTL            EQU   $3B
* Flagga
RTIF              EQU   $80

* tidsbas ( se CRG-modul HCS12)
* Med laborationsmaskin använder vi korrekt
* tidsbas i simulatorn ger den dock allt för långa
* fördröjningar
#ifdef SIM
TimeBase          EQU   %00010000    kortast möjliga
#else
TimeBase          EQU   %01111001    längsta möjliga
#endif

        segment      text
        define        _standby
        entry          _standby
_standby:
        WAI                        vänta på avbrott
        RTS
        exit _standby

*
*
*
        extern        _interrupt_type
*   sätts av avbrottshanterare

        define        _init_irq
*   void init_irq(void);
*   sätt upp för avbrott från ML13
        entry _init_irq
_init_irq:
        CLR    ML13_IRQ_Control    kvittera
avbrott
* avbrottsvektor måste initieras under DBG12...
        LDX    #ptimirq
        STX    $3FF0              Avbrottsvektor RTC
* men vi skriver den också till 'rätt' adress för
* att även kunna använda exemplet i simulatorn...
        STX    $FFF0

* För normalt IRQ... (från ML13)
        LDX    #ML13_irq
        STX    $3FF2
        STX    $FFF2

        CLI                        acceptera avbrott
        RTS
        exit _init_irq
```

```

*
* Avbrottsrutin för ML13
ML13_irq:
    CLRA
    LDAB    ML13_IRQ_Status
    BITB    #1
    BEQ     n1
    LDAB    #OPENED_DOOR
    STD     _interrupt_type
    BRA     n7

n1:    BITB    #2
    BEQ     n2
    LDAB    #CLOSED_DOOR
    STD     _interrupt_type
    BRA     n7

n2:    BITB    #4
    BEQ     n3
    LDAB    #SENSOR
    STD     _interrupt_type
    BRA     n7

n3:    BITB    #8
    BEQ     n4
    LDAB    #SENSOR
    STD     _interrupt_type
    BRA     n7

n4:    BITB    #$10
    BEQ     n5
    LDAB    #OPENING_DOOR
    STD     _interrupt_type
    BRA     n7

n5:    BITB    #$20
    BEQ     n6
    LDAB    #CLOSING_DOOR
    STD     _interrupt_type
    BRA     n7

n6:    LDAB    #NO_IRQ_TYPE
    STD     _interrupt_type
n7:    CLR     ML13_IRQ_Control    kvittera avbrott
    RTI

    segment    text
    define _set_timeout
    void set_timeout(int sekunder)
    *    max 255 sekunders fördröjning ...
_set_timeout:
    PSHY
    TSY
    LDD     4,Y    antal sekunders fördröjning

*
*    Med 8 MHz HC12
*    Längsta tidbas är 10 * 2**16
*    = 655360 cykler = 0,08192 sekunder
*    = 12,2 avbrott/sekund...
    LDAA    #12
    MUL                     antal avbrott innan timeout
* Rimlighetskontroll ...
    BEQ     set_tim_exit
    STD     delay_count
    LDAA    #TimeBase        se ovan
    STAA    RTICTL
    MOVB    #$80,CRGINT set RTIE
set_tim_exit
    PULY
    RTS

```

```
* avbrott från realtidsklockan
ptimirq:
* kvittera avbrott
    BSET    REGBASE+CRG+CRGFLG, #RTIF
    LDD     delay_count
    BEQ     timeout
    SUBD    #1
    STD     delay_count
    RTI

* vid timeout, stanna timerkretsen
timeout:
    CLR     RTICTL
    LDD     #TIME_OUT
    STD     _interrupt_type
    RTI

    BSS
delay_count:
    RMW     1      static, exporteras inte...
```

Implementering och test

Läroboken ger ett utförligt förslag på hur huvudprogrammet i denna uppgift kan utformas.

- Skapa ett nytt projekt 'moment3' i workspace 'RTLAB'. Använd standardinställningarna som föreslås.
- Skapa en källtextfil MOM3.C med ett huvudprogram som löser laborationsuppgiften. Lägg till denna fil till projektet.
- Hämta filen MOM3-LOW.S12 (assembler) från kurshemsidan. Lägg även denna fil till själva projektet.
- Testa programmet med XCC12's debugger, använd simulatorn för att koppla ML13 till adress \$B00 så att du kan observera "dörrens" beteende. Tänk på att ML13-simulatoren kan konfigureras för att generera avbrott, detta är samma sak som då du ansluter avbrottsignalen från ett verkligt ML13 till avbrottsingången hos MC12.

Du *får* förstås använda det förslag till assembler-program som givits i detta moment men du *måste* inte göra det...

Vid laborationsplatsen

Inför laborationen måste en avbrottsutgång på ML13 kopplas till rätt avbrottsingång på MC12 med hjälp av en extern kopplingskabel.

Uppgift 3b:

Ni skall nu ersätta en del av de givna assemblerrutinerna med motsvarande rutiner fast skrivna i C.

Minimikravet är att ni skriver om avbrottsrutinen ML13irq så att den helt är skriven i C. Övriga delar kan vara kvar i assembler omni vill. De nya rutinerna i C skall sparas i en ny fil mom3-irq.c

Kontrollstation 3

Under detta moment får du stifta bekantskap med en enkel realtids-kärna, *RTK12*.

Du ska använda *RTK12* för att utföra tre processer under en "non-pre-emptive" schemalägningsstrategi.

4. Non-Pre-Emptive Scheduling

I detta moment ska vi se hur *RTK12* kan användas med en *Non-Pre-Emptive* schemalägningsstrategi. Vi illustrerar samtidigt det huvudprogram som varje applikation för *RTK12* måste innehålla.

Realtidskärnan finns tillgänglig som programbibliotek ("*librtkd.e12*" och "*librtk.e12*") under *XCC12*.

Läs om *RTK12* under *XCC12*'s hjälpsystem.!

RTK12's funktioner

Varje *RTK12*-applikation innehåller ett huvudprogram och ett antal funktioner som kommer att behandlas som processer. *RTK12* tillhandahåller ett antal funktioner för processhantering och synkronisering.

```
void InitKernel(int, void(*)());
```

initierar *RTK12*'s interna datastrukturer. Parameter 1 är ett heltal som sätter systemets "timeslice", dvs antal klockavbrott mellan varje anrop av applikationens avbrottshanterare. Parameter 2 är en pekare till den funktion (inga parametrar, inget returvärde), som utgör applikationens avbrottshanterare. Jämför med parameterlistan till `set_timer()`.

```
void StartKernel(void);
```

inga parametrar, aktiverar realtidsklockan och startar den första processen. Om ingen process är exekverbar anropas `ExitKernel()`.

```
void ExitKernel(void);
```

avslutar *RTK12*, stänger av realtidsklockan, kontrollerar processköns så att ingen process finns kvar.

```
int CreateProcess(char *, void(*)());
```

skapar process under *RTK12*. Parameter 1 är en pekare till en teckensträng med processens namn, denna används för diagnostiska ändamål. Parameter 2 är en pekare till den funktion (inga parametrar, inget returvärde) som ska registreras som process. Antalet processer som kan skapas under *RTK12* är statiskt bestämt av konstanten `MAX_PROCESSES` i `_rtk.h`. Returvärdet är -1 om processen inte kan skapas.

```
void TerminateProcess(int);
```

avslutar anropande process och sparar status om processen. Processen kommer efter detta inte att exekveras men processkontrollblocket sparas så att detta kan undersökas av någon annan process, alternativt inspekteras då `ExitKernel()` utförs.

En applikation för *RTK12* består av ett huvudprogram och ett antal processer. Ett huvudprogram har alltid samma struktur:

- Initiera kärnan
- Skapa alla processer
- Initiera eventuella semaforer (beskrivs i senare moment)
- Starta kärnan.

Processer

En *RTK12*-process har samma utseende som en C-funktion utan parametrar. Det finns dock några viktiga skillnader som man måste tänka på då man programmerar processerna:

- En process är inte en funktion i den meningen att den kan anropas från någon annan process (eller funktion).
- Exekveringen av en process kan, när som helst, komma att avbrytas, för att vid ett senare tillfälle återupptas.
- Flera processer kan dela samma funktioner, dvs anropa samma subrutiner. Observera då att globala variabler som delas av flera processer (eller används av funktioner som delas av flera processer) som regel måste skyddas mot inkonsistent uppdatering (se kurslitteraturen).

Applikationen mom4.c

Applikationen för detta moment består av huvudprogram, två processer (P1 och P2) och en avbrottshanterare. Huvudprogrammet följer exakt den struktur som beskrivits ovan. Observera speciellt att en avbrottshanterare *måste* tillhandahållas även om, som i detta fall, ingenting speciellt ska utföras vid avbrott. Vi vill illustrera *non-preemptive scheduling* och skriver därför två processer, som ska utföras sekvensiellt. Observera att i allmänhet kan man inte göra något antagande om *i vilken ordning* processerna kommer att startas av realtidskärnan. För *RTK12* gäller dock regeln att processerna startas i samma ordning som de skapats. (Se källtexter "CreateProcess()" och "dispatch()").

Processerna P1 och P2 är likartade, de ser ut på följande sätt:

```
void P1(void)
{
    int i;
    for(i=0;i<1000;i++)
        _outchar('1');

    TerminateProcess(0);
}

void P2(void)
{
    int i;
    for(i=0;i<1000;i++)
        _outchar('2');

    TerminateProcess(0);
}
```

Dvs, P1 skriver ut 1000 ettor till bildskärmen och terminerar därefter, P2 skriver ut 1000 tvåor till bildskärmen och terminerar.

Avbrottshanteraren gör i detta fall ingenting, men måste finnas med:

```
void AtInterrupt(void)
{
}
```

Huvudprogrammet...

```
main()
{
    InitKernel(TIMESLICE, AtInterrupt);
    if( CreateProcess("P1", P1) == -1){
        printf("\nCan't create process");
        exit();
    }
    if( CreateProcess("P2", P2) == -1){
        printf("\nCan't create process");
        exit();
    }
    StartKernel();
}
```

Nu kan det vara hög tid att prova detta ...

Implementering och test

Läroboken ger ett utförligt förslag på hur huvudprogrammet i denna uppgift ska utformas.

- Skapa ett nytt projekt 'moment4' i workspace 'RTLAB'. Använd standardinställningarna som föreslås.
- Skapa en källtextfil MOM4.C med ett huvudprogram, avbrottshanterare och processer enligt. Lägg till denna fil till projektet. Använd värdet 100 för TIMESLICE.
- Testa programmet med XCC12's debugger, använd simulatoren för att koppla 'Console' till seriekommunikationskretsen så att du kan se programmets utskrifter.

Vid laborationsplatsen

Tänk på att du ska ha med dina egna källtextfiler till laborationsplatsen. Du gör nu på samma sätt som under dina tidigare förberedelser (implementering och test) men laddar det färdiga programmet till laborationsdatorn i stället för debuggern.

Kontrollstation 4

5. Pre-Emptive Scheduling - Timesharing

Under detta moment ska vi illustrera *Pre-Emptive-Scheduling*, dvs en schemalägningsstrategi där processer tillfälligt avbryts för att senare återstartas. Detta förfarande är mycket vanligt i realtids-sammanhang och förtjänar speciellt stor uppmärksamhet.

Oändliga processer

Vi använder här processer som aldrig terminerar "frivilligt", dvs de exekveras i oändlighet, endast avbrutna av realtidskärnan. Ett enkelt sätt att åstadkomma en sådan programkonstruktion är:

```
while(1)
{
    /* satser */
}
```

Eftersom villkoret i while-satsen alltid är uppfyllt (dvs skilt från 0) kommer iterationen aldrig att brytas och "satser" exekveras gång på gång, endast avbrutna av realtidskärnan för processbyten.

I `_rtk.h` har vi definierat makrot:

```
#define DO_FOREVER while(1)
```

En "oändlig" process kan då skrivas som:

```
#include <_rtk.h>
void infinity(void)
{
    DO_FOREVER
    {
        /* satser */
    }
}
```

Applikationen mom5.c

Även för denna applikation använder vi mycket enkla processer P1,P2 och P3. De definieras av följande:

```
void P1(void)
{
    DO_FOREVER
    {
        _outchar('1');
    }
}

void P2(void)
{
    DO_FOREVER
    {
        _outchar('2');
    }
}

void P3(void)
{
    DO_FOREVER
    {
        _outchar('3');
    }
}
```

Vi har alltså tre "oändliga" processer som aldrig (frivilligt) terminerar. För att åstadkomma enkel rättvisa fördelar vi processortiden lika mellan dessa, processbytet gör vi efter varje *timeslice*, dvs i vår avbrottshanterare:

```
void AtInterrupt(void)
{
    insert_last(Running, &ReadyQ);
    Running = remque(&ReadyQ);
}
```

Implementering och test

Läroboken ger ett utförligt förslag på hur huvudprogrammet i denna uppgift ska utformas.

- Skapa ett nytt projekt 'moment5' i workspace 'RTLAB'. Använd standardinställningarna som föreslås.
- Skapa en källtextfil MOM5.C med ett huvudprogram, avbrottshanterare och processer enligt. Lägg till denna fil till projektet. Använd värdet 100 för TIMESLICE.
- Testa programmet med XCC12's debugger, använd simulatoren för att koppla 'Console' till seriekommunikationskretsen så att du kan se programmets utskrifter.

Vid laborationsplatsen

Tänk på att du ska ha med dina egna källtextfiler till laborationsplatsen. Du gör nu på samma sätt som under dina tidigare förberedelser (implementering och test) men laddar det färdiga programmet till laborationsdatorn i stället för debuggern.

Lägg till ytterligare en process 'P4' med samma beteende som P1, P2 och P3. Låt P4 skriva ut siffran 4.

Ändra konstanten `TIMESLICE`, tills du får ett fåtal (5-20) utskrifter mellan varje processbyte.

Får du alltid samma antal utskrifter från alla processer?

Uppskatta hur lång tid det tar att utföra en iteration (dvs skriva ut ett tecken) i P1.

Ledning: Bestäm längden av `TIMESLICE` (se moment 1) Varje process tilldelas en `TIMESLICE` åt gången.

Svar: _____

Kontrollstation 5

6. Semaforoperationer

I föregående moment såg vi exempel på hur man kan implementera en parallell programmeringsmodell i ett time-sharing system. Genom att betrakta varje enskilt program som en process och låta en realtidskärna administrera dessa processer kan vi alltså, i princip, bygga upp ett fleranvändarsystem (*multi-user*) och/eller ett så kallat *multi-tasking* system. Om systemet består av ett antal oberoende processer blir implementeringen tämligen enkel, detta är dock praktiskt taget aldrig fallet i verkligheten. I själva verket finns det oftast mycket nära beroenden mellan de olika processerna i ett realtidssystem. Ofta består dessa beroenden av att globala data delas men det kan också finnas speciella tidsberoenden. För att klara av dessa beroenden måste processerna på något sätt *synkroniseras* med varandra.

Processsynkronisering i *RTK12* sker med hjälp av *semaforer*. Semaforerna är implementerade som *Blockerande/Kö* (se även läroboken). Maximala antalet semaforer ges av konstanten `MAX_SEM_ID` definierad i `_rtk.h`. Följande operationer kan utföras på en semafor:

```
void initsem(int id, int count);
```

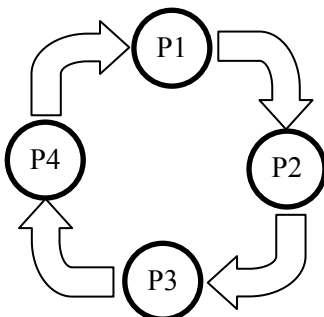
Den första parametern är ett identifikationsnummer 1 t.o.m `MAX_SEM_ID`. Den andra parametern ger ett initialvärde till semaforvariabeln. Varje semafor initieras en gång av huvudprogrammet, detta sker mellan utförande av `InitKernel()` och `StartKernel()`.

```
void waitsem( int );
```

Parametern är ett identifikationsnummer 1-`MAX_SEM_ID`. Processen utför *wait* på den semafor som anges av parametern. (se även läroboken). Rutinen förutsätter att semaforen initierats av `initsem()`.

```
void signalsem( int );
```

Parametern är ett identifikationsnummer 1-`MAX_SEM_ID`. Processen *signalerar* på den semafor som anges av parametern. Rutinen förutsätter att semaforen initierats av `initsem()`.



Uppgift: Tidssynkronisering av processer

Uppgiften består i att konstruera en *händelsekedja* av synkroniserade processer. Processerna "P1", "P2", "P3" och "P4" från föregående moment ska exekveras *i denna ordning*. Detta ska upprepas i en oavslutad kedja.

Processerna ska modifieras så att de bara skriver ett tecken åt gången till skärmen.

Tips: Man kan anropa semaforen (`waitsem(Sx)`) i en process medan man släpper samma semaforen (`signalsem(Sx)`) i en annan process. På detta sätt kan man synkronisera processers exekveringsordning.

Modifiera processerna i föregående moment så att utskriften från processerna ändras till '1234123412341234...' osv. Synkroniseringen ska ske med hjälp av semaforer.

Implementering och test

Läroboken ger ett utförligt förslag på hur huvudprogrammet i denna uppgift ska utformas.

- Skapa ett nytt projekt 'moment6a' i workspace 'RTLAB'. Använd standardinställningarna som föreslås.
- Skapa en källtextfil MOM6A.C med ett huvudprogram, avbrottshanterare och processer enligt. Lägg till denna fil till projektet. Använd värdet 100 för TIMESLICE.
- Testa programmet med XCC12's debugger, använd simulatorn för att koppla 'Console' till seriekommunikationskretsen så att du kan se programmets utskrifter.

Vid laborationsplatsen

Tänk på att du ska ha med dina egna källtextfiler till laborationsplatsen. Du gör nu på samma sätt som under dina tidigare förberedelser (implementering och test) men laddar det färdiga programmet till laborationsdatorn i stället för debuggern.

Kontrollstation 6

Meddelanden

I detta avsnitt skapar vi nya typer av applikationer för *RTK12*

Här illustreras meddelandeskickning.

I detta avsnitt ska vi simulera en anslagstavla. Två processer *producer()* och *consumer()* är givna, se nedan. Producentprocessen sätter med jämna mellanrum upp ett meddelande bestående av ett ASCII-tecken (A-Z) på tavlan med hjälp av en funktion *PutMessage()*. *PutMessage()* skriver alltid in ett nytt tecken, dvs skriver över befintligt. Konsumentprocessen läser ett meddelande via funktionen *GetMessage()* som slänger meddelandet (samma meddelande får alltså inte läsas två gånger) och skriver meddelandet till bildskärmen. Läsning och skrivning på anslagstavlan (gemensamt minne) ska synkroniseras med hjälp av *en* semafor.

```

PROCESS    Producer(void)
{
    int    i;
    char    msg, j;
    j = 'A';
    DO_FOREVER
    {
        for(i=0; i<PRODUCER_DELAY; i++);
        PutMessage(j);
        if(j=='Z')
            j = 'A';
        else
            j = j+1;
    }
}

PROCESS    Consumer(void)
{
    int    i;
    char    msg;

    DO_FOREVER
    {
        GetMessage(&msg);
        _outchar(msg);
        for(i=0; i<CONSUMER_DELAY; i++);
    }
}

```

Definiera själv de olika konstanterna *PRODUCER_DELAY* och *CONSUMER_DELAY*.

Uppgift 1- Moment 6B

- Skriv funktionerna *GetMessage()* och *PutMessage()* enligt specifikationen ovan.
- Skriv ett huvudprogram och processerna *Producer/Consumer*.
- Vid test i hårdvara sätt konstanten *TIMESLICE* så att *RTK12* byter process 10 ggr/sekund.

Betrakta resultatet av programkörningen. "Konsumeras" alla tecken som "produceras"?

Du kan öka takten hos producentprocessen genom att variera längden hos "busy-wait"-slingan. Låt producentprocessen producera tecken dubbelt så fort som konsumentprocessen konsumerar dom.

Beskriv iakttagelser:

Skapa ytterligare en producentprocess *producer2()* som producerar ASCII-tecknen 'a-z' och som också använder PutMessage(). Provkör programmet och beskriv kortfattat dina iakttagelser. Försök även förklara iakttagelserna.

Uppgift 2 - Moment 6C

Avslutningsvis ska du nu modifiera funktionerna GetMessage() och PutMessage() så att inga meddelanden går förlorade eller läses flera gånger. Räcker om du gör detta för fallet med en producent-process.

Ledning: Använd en lösning med *två* semaforer.

Kontrollstation 7

7. Processsynkronisering

Efter ett antal moment som syftat till att du ska göra dig bekant med en realtidskärna och få en uppfattning om vad den kan användas till är det nu dags för dig att lösa en smärre uppgift.

Övergripande funktionsbeskrivning .

Styrningen av dörrautomaten *MC12/ML13* ska nu implementeras i ett C- program med stöd av realtidskärnan *RTK12*.

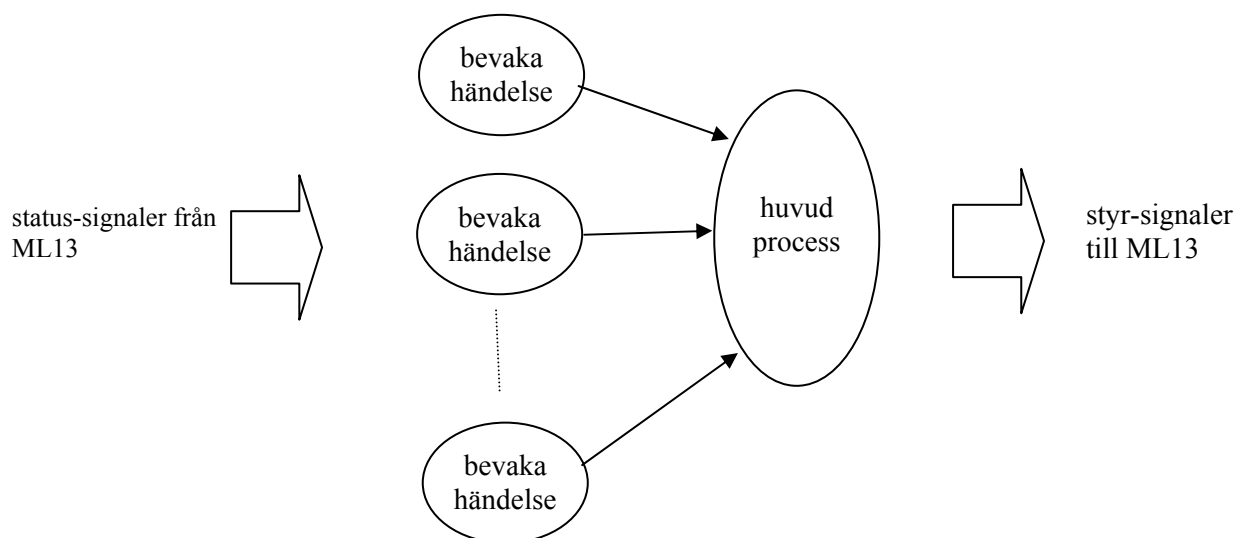
Utöver att dörren skall kunna öppnas och stängas som tidigare skall:

- Dörren kunna ”låsas” genom att en tangent ’l’ (lock) trycks ned på datorns tangentbord. Låsning skall endst kunna ske när dörren är stängd.
- Om dörren är låst ska den kunna ”låsas upp” genom att en tangent ’u’ (unlock) trycks ned på tangentbordet.
- Med en ”låst” dörr menas att den inte kan öppnas genom att någon av tryckknapparna på *ML13* trycks ned. Förslagsvis använder man en global variabel (int locked) som indikerar läget locked/unlocked.
- Dörren ska vara låst från början. (Se förslag på programstruktur senare).

Du kommer att finna en förslag på hur du kan principiellt kan lösa uppgiften. Du är dock inte bunden att i detalj följa dessa anvisningar, dock måste ”exekveringsmodellen” (se nedan) efterliknas.

Exekveringsmodell

Programmet skall byggas upp av ett antal processer som synkroniseras med ett antal semaforer och med en modell enligt nedan figur:



I centrum har vi en huvudprocess som ger alla styrsignaler till dörren. Huvudprocessen kan synkroniseras med de processer som tar emot statussignaler från dörren, lämpligtvis med hjälp av semaforer. Följande pseudo-kod anger en tänkbar lösning:

```
PROCESS manage_door() {
    begin do forever{
        enable "open_door_event"
        wait for "open-door-signal"
        open the door;
        enable "door_opened_event"
        wait for "door-opened-signal"
        wait for 5 seconds
        close the door
    }
}
```

Observera speciellt hur semaforer kan användas för "enable-events".

Exekveringsmodellen visar hur vi har ett antal processer som bevakar de olika händelser som kan inträffa:

- "öppna dörr"
- "dörren är helt öppen"
- "dörren är helt stängd"

En sådan process kan utformas exempelvis enligt nedan:

```
PROCESS open_door_event:
```

```
begin do forever:
    wait for event monitoring enabled;
    begin do forever
        if (open the door and not locked)
            signal (open_the_door_sem)
            break inner loop
        else
            abandon cpu
    end
end
```

Funktionen "yield"

Observera speciellt "abandon cpu", dvs processen överläter resten av sin TIMESLICE till någon annan körbar process. En sådan funktion kallas traditionellt "yield". I *RTK12* finns ingen sådan funktion så du måste skriva den själv. Funktionen yield måste:

- Spara processens flyktiga omgivning
- Placera processen sist i Ready-kön
- Starta nästa process i Ready-kön.

Ledning:

Studera källtexterna för de färdiga funktionerna 'waitsem', 'insert_last', 'suspend' och 'dispatch' i *RTK12*.

Funktionen "sleep"

För att åstadkomma en bestämd fördröjning konstrueras funktionen 'sleep', som alltså blir ytterligare en generell del av realtidskärnan. En process ska kunna suspendera sig för ett bestämt tidsintervall genom att anropa funktionen enligt:

```
....
sleep( intervall *100 ms );
...
```

Följande exempel visar en enkel implementering av "sleep"-funktionen

```
void sleep(int delay)
{
int    wakeup;
    /* sleep for delay * TIMESLICE */
    wakeup = get_rtk_time() + delay;

    while(1){
        if( wakeup > get_rtk_time() )
            yield(); /* not yet... */
        else
            return;
    }
}
```

Anropssekvensen för funktionen 'sleep' blir:

```
...
sleep (50); /* 5 sec. delay */
```

Funktionen "get_rtk_time" (finns i *RTK12*) returnerar aktuell systemtid. (Läs om denna i *XCC12*'s hjälpsystem).

Implementering och test

Uppgiften är omfattande. Arbete förslagsvis enligt nedan ordning.

- Skapa ett nytt projekt 'moment7' i workspace 'RTLAB'. Använd standardinställningarna som föreslås.
- Skapa en källtextfil *MOM7.C* med ett huvudprogram, avbrottshanterar och processer. Avvakta tills vidare med funktionerna *yield/sleep*. Lägg denna fil till projektet.
- Testa programmet med *XCC12*'s debugger, använd simulatoren för att koppla 'Console' till seriekommunikationskretsen så att du kan se programmets utskrifter. Koppla även *ML13* på samma sätt som tidigare.
- Lägg till funktionerna *yield()* och *sleep()* och testa slutlig funktion med dessa.
- Tänk på att simulatorns *ML13* normalt genererar en rad olika typer av avbrott. Du måste stänga av dessa i denna applikation.

Vid laborationsplatsen

Tänk på att du ska ha med dina egna källtextfiler till laborationsplatsen. Du gör nu på samma sätt som under dina tidigare förberedelser (implementering och test) men laddar det färdiga programmet till laborationsdatorn i stället för debuggern.

Ladda ned och testa din realtidsapplikation. I detta fall ska du *inte* ansluta avbrottsutgången på *ML13* till *MC12*.

Kontrollstation 8