Laborationer med MC68
Realtidssystem

©GMV 2003, 2004

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 XCC32, Version 1.2 (med RTK 4.3 eller senare).
Notera speciellt att följande biblioteksfunktioner getts nya namn:
`get_system_time` heter nu `get_rtk_time`
`wait` heter nu `waitsem`
`signal` heter nu `signalsem`

Innan du börjar...

I detta häfte behandlas laborationer med enkortsdatorn *MC68* och laborationskortet *ML13*.

Handböcker som beskriver *MC68* och *ML13* finner du på GMV's hemsida, www.gbgmv.se.

Laborationerna genomförs med programutvecklingsmiljön *XCC32*.

I *XCC* finns simulatorer för såväl *MC68* som *ML13*. Hjälpsystemet i *XCC* innehåller beskrivningar av hur du konfigurerar simulatorn och ansluter en simulerad variant av *ML13*. Alla uppgifter kan lösas med hjälp av simulatorerna i *XCC*.

Det förutsätts att du har en grundläggande förståelse för mikroprocessorn (*MC68340*) 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:
- Parallellexekvering
- Klockrutinen
- MC68340's periodiska räknare

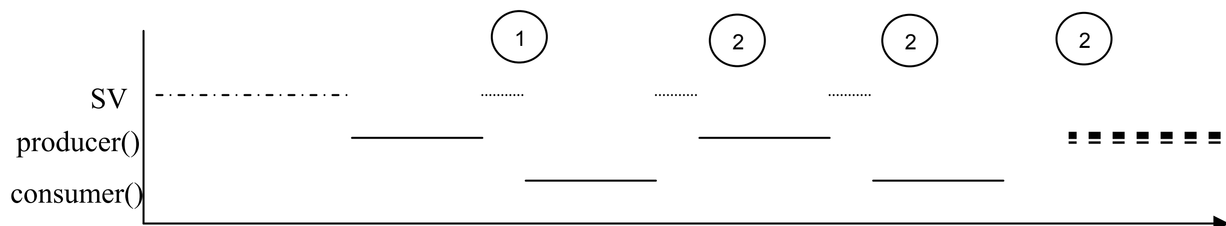
Under denna inledande laboration ska vi visa ett enkelt exempel på hur *tidsdelning* (*time-sharing*) kan utföras. Processortiden ska delas lika mellan två *applikationsprogram:*

- `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. Dessutom måste SV tillhandahålla rutinen '_putchar', men det är samma rutin som användes under introduktionsmomentet. Vi börjar med situation '2', den är enklast:

2) Ett program har blivit avbrutet av räknarkretsen, vi måste tillhandahålla avbrottsrutinen 'timer_interrupt', denna ska:

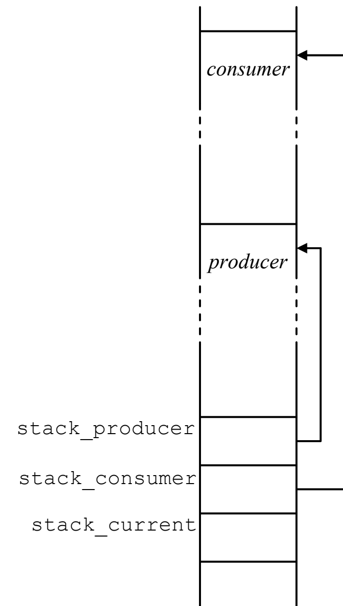
a) spara programmets flyktiga omgivning
b) starta det andra programmet

Programmets "flyktiga omgivning" utgörs av processorns registerinnehåll. Programräknaren (PC) och statusregistret (SR) sparas automatiskt av processorn vid avbrottet men hur är det med övriga registerinnehåll? Eftersom vi inte på förhand kan veta vilka register som används (dessa allokeras ju av kompilatorn) måste vi ta det säkra före det osäkra och spara samtliga registerinnehåll.

Nästa steg blir att "starta nästa program", vilket är då detta???, 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 bör nu vara 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 (se figur i marginalen) och den slutliga algoritmen för 'timer_interrupt' blir:




```
timer_interrupt:
        save processor context;
        save current_stack;
        if(RUNNING==producer ){
                stack_producer = current_stack;
                running = consumer;
                current_stack = stack_consumer;
        }else{
                stack_consumer = current_stack;
                running = producer;
                current_stack = producer_stack;
        }
        restore current_stack;
        restore processor context;
        return from interrupt;
```

Uppenbarligen behövs variabler för att spara de olika stackpekarna. Implementera nu avbrottsrutinen 'timer_interrupt' i MC68000 assembler. (Det kan inte utföras i 'C')...

```
timer_interrupt:
      ...         spara omgivning
      ...         spara SP i current_stack
      utför if/else
      ...
      ...
      ...         återställ current_stack till SP
      ...         återställ omgivning
      RTE
```

Välj namnet:
mom1-low.s68
för denna källtext.

Initieringar

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 aktiveringspost motsvarande den som ges i satsen:

current_stack = producer_stack;

i avbrottsrutinen. Följande sekvens gör det jobbet:

```
init_producer:
      MOVEA.L      #producer,A0          BOS för 'producer'
      ADDA.L   #STKSIZE,A0               Sätt A0 = TOS producer
      MOVE.W   #0,-(A0)                  vector offset, format 0
      MOVE.L   #_producer,-(A0)  startadress 'producer' till stacken
      MOVE.W       #$2000,-(A0)                 initialt SR till stacken
      MOVEM.L      D0-D7/A0-A6,-(A0)            skapa 'flyktig omgivning'
      MOVE.L A0,(stack_producer).L              spara denna stackpekare
```

Motsvarande initiering 'init_consumer' görs även för 'consumer()'.

Nästa steg blir att initiera räknarkretsen så att den genererar avbrott med jämna mellanrum. Vi använder här den periodiska räknaren i MC68340's SIM-modul, den är avsedd för sådana här ändamål och därför också enkel att hantera.

Sim 40 – periodisk räknare

I SIM-40 modulen, dvs en del av den logik som ingår i microcontrollern MC68340, finns en ”periodisk räknare”. Till skillnad från de andra räknarkretsarna är den ganska enkel och i första hand avsedd att användas vid implementering av en ”realtidsklocka”.

Räknaren har två register, *Periodic Interrupt Control Register* (PICR), via detta register aktiveras räknaren, avbrottsnivå och avbrottsvektor tilldelas. Det andra registret, *Periodic Interrupt Timer Register* (PITR).

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 behöver ej kvitteras på något sätt. Efter initieringen behövs alltså inga läsningar/skrivningar utföras från/till dessa register.

PICR - Periodic Interrupt Control Register.

Används, tillsammans med PITR, för att implementera en realtidsklocka. Registret finns på offset +$22 i SIM-40 modulen. För en MC68 i standardutförande, där SIM-40 modulen relokeras till adress $FFFFF000, blir alltså registrets adress $FFFFF022. Se även exempel nedan.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	PIRQL2	PIRQL1	PIRQL0	PIV7	PIV6	PIV5	PIV4	PIV3	PIV2	PIV1	PIV0

Bit 10-8, PIRQL2-0; *Periodic Interrupt Request Level*.
Anger avbrottsnivå för den periodiska räknaren i SIM40.

PIRCL 1 2 0	Funktion
0 0 0	Genererar EJ avbrott
0 0 1	Avbrottsprioritetsnivå 1
0 1 0	Avbrottsprioritetsnivå 2
0 1 1	Avbrottsprioritetsnivå 3
1 0 0	Avbrottsprioritetsnivå 4
1 0 1	Avbrottsprioritetsnivå 5
1 1 0	Avbrottsprioritetsnivå 6
1 1 1	Avbrottsprioritetsnivå 7

Bit 7-0, PIV7-0; *Periodic Interrupt Vector*.
Vektornummer för den periodiska räknaren. Vektornumren $40-$FF kan väljas godtyckligt under förutsättning att det inte kolliderar med andra periferikretsar i systemet. Adressen till avbrottsrutinen för räknaren blir Vektornummer*4.

EXEMPEL:
Första ”lediga” avbrottsnummer är $40, om räknarkretsen tilldelas detta ska avbrottsvektor $40*4 = $100, användas, dvs här ska adressen till räknarkretsens avbrottsrutin skrivas.

PITR - Periodic Interrupt Timer Register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	SWP	PTP	PITR7	PITR6	PITR5	PITR4	PITR3	PITR2	PITR1	PITR0

Bit 9, SWP; Software Watchdog Prescale.

1 = Software Watchdog är nerdelad med 512
0 = Normal,

Vi använder EJ "Watchdog" i detta exempel, sätt biten till '0'

Bit 8, PTP; Periodic Timer Prescale Control.
1 = Periodiska räknaren är nerdelad med 512
0 = Normal

Bit 7-0, PITR7-0; Periodic Interrupt Timer Register Bits.
Innehåller räknarvärdet för den periodiska räknaren, \$00 stänger av räknaren. Periodtiden bestäms enligt:

period = PITRvärde/(32768/prescaler värde)/4

Vilket också kan skrivas som:

period = *PITRvärde* / 8192 / *prescaler*,

där *prescaler* är 1 eller 512.

Följande tabell anger minimala/maximala periodtider för prescaler's olika värden. Observera att om PTP är 1 kan PITR anta värden 0-\$FF, annars måste PITR vara minst 1.

PTP	prescaler	min period	max period
0	1	122 µs	31,1 ms
1	512	62,5 ms	15,93 s

Oftast vill vi bestämma ett *PITRvärde* utifrån någon periodtid och vi får då:

PITRvärde = *period* * 8192 / *prescaler*

Om vi exempelvis vill bestämma räknarvärdet för avbrottsintervallet 1 s får vi:

$PITRvärde = 1 * 8192 / 512 = 16_{10} = 10_{16}$

eftersom PTP då ska vara 1 innebär detta:

```
MOVE.W    #$110,(PITR).L
```

Fyll i följande tabell med värden för olika avbrottsperioder. Observera att det inte alltid är möjligt att bestämma dessa exakt, i de fall detta inte går, ange också hur stor avvikelse (verklig tid) som ditt angivna räknarvärde ger:

Periodtid	Räknarvärde	Avvikelse +/-
1 ms		
10 ms		
100 ms		
250 ms		
500 ms		
1 s		

Avslutningsvis måste vi också kortfattat beskriva ett register som inte hör till räknarkretsen, det måste dock initieras för att SIM-40 modulen ska accteptera avbrott från någon av de interna periferienheterna.

MCR - Module Configuration Register.

Module configuration register används för att bestämma övergripande funktioner. Det kan ändras endast då CPU32 är i SM (*supervisor mode*).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	FRZ1	FRZ0	FIRQ	0	0	SHEN1	SHEN0	SUPV	0	0	0	IARB3	IARB2	IARB1	IARB0

Bit 3-0, IARB3-0; *Interrupt Arbitration Bits*. MC68340's I/O moduler måste ha olika prioriteter men externa avbrottsprioriteter kan vara de samma. Dessa bitar avgör hur bussen arbitreras vid samtidiga avbrott på samma prioritetsnivå, $F ger SIM40 högsta prioritet, $1 lägsta.

För att vår räknarkrets ska fungera initierar i IARB-fältet till $F

Konstruera nu en initieringssekvens för räknarkretsen. Under laborationstillfället kommer du att prova programmet med olika periodtider.

Det är nu dags att sätta samman allt sammans till ett 'main'-program. Följande "skelett" ger dig huvuddragen för 'mom1-low.s11'. Applikationsprocesserna (producer/consumer) finner du nedan.

```
*
*  mom1.low.s68
*
*  MC68 applikation som skiftar mellan två program
*  en gång per sekund.
*
*  Introduktion till "time-sharing"
*
*
        segment     abs
MODBASE equ         $FFFFF000
mc68_sim40_mcr      equ   MODBASE+$000
mc68_sim40_picr     equ   MODBASE+$022
mc68_sim40_pitr     equ   MODBASE+$024

STKSIZE EQU   256   stackutrymmen för program


*
        segment     bss
*  segment för globala variabler


a_stack_producer    DS.B  STKSIZE
*'STKSIZE' bytes stackutrymme för program A
a_stack_consumer    DS.B  STKSIZE
*'STKSIZE' bytes stackutrymme för program B

RUNNING        DS.L  1     anger exekverande program
current_stack DS.L  1
*temporär lagring stackpekare för 'RUNNING'

stack_producerDS.L  1           stackpekare för 'producer'
stack_consumerDS.L  1           stackpekare för 'consumer'

        segment     text        segment för kod...

* Följande funktioner är definierade i 'mom1.c'
        extern      _producer
        extern      _consumer
```

```
*
      define      _main
      entry       _main
_main:
      init_producer;
      init_consumer;
      init_RTC;

* starta 'producer'
      CLR.L       (RUNNING).L
      MOVE.L      (stack_producer).L,(current_stack).L
      MOVEA.L     (current_stack).L,SP
      MOVEM.L     (SP)+,D0-D7/A0-A6
      RTE
```

Observera att programexekveringen aldrig upptas igen efter instruktionen RTE.

Producent-konsument processerna ska se ut på följande sätt:

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

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

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

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 simulatorn i *XCC32* tillsammans med IO-simulatorn för att testa programmet *innan* du går till laborationsplatsen. Följande arbetsgång kan vara lämplig:

- Starta XCC32
- Skapa ett nytt 'Workspace' - namnge det 'RTLAB'.
- Skapa ett nytt projekt, 'moment1', använd standardinställningarna som föreslås.
- Skapa källtexterna 'mom1-low.s68' respektive 'mom1.c' enligt tidigare anvisningar.
- Lägg de nya källtexterna till projektet.
- Välj 'Build All' för att skapa applikationen.
- Testa programmet med *XCC32*'s debugger, använd simulatorn 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 simulatorn.

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 längsta periodtid (1 s) för klockrutinen, granska en utskriftsföljd som börjar på ’a’, vad ser du? (Skriv upp de 10 påföljande tecknen):

Ändra nu initieringen av räknaren till 1 ms 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 förklara likheter/skillnader?

____

____

____

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

____

____

____

Kontrollstation 1
kontakta din laborationshandledare som kontrollerar hur du utfört uppgifterna i detta avsnitt.

Uppvisat för________________________den ____/____

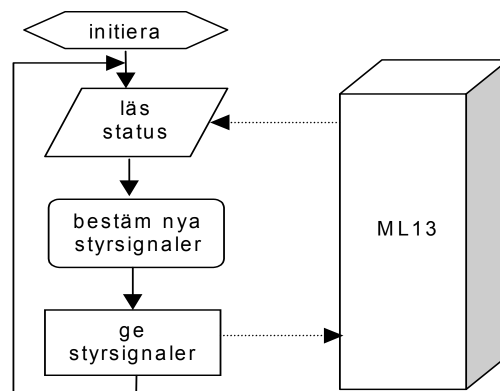
2 Tidsstyrd dörrautomat

Under detta moment ska du implementera en 'dörrautomat' med ML13 utan att använda avbrott.
Beskrivning av ML13 finns i separat handbok, en enklare beskrivning finns även under hjälpsystemet i XCC32

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     0x8B000
#define read_control *((char *)ML13_Status)
```

macrot 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    0x8B000
#define set_control(x) *((char *)ML13_Control)=x
```

macrot 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 *XCC32*'s debugger, använd simulatorn för att koppla *ML13* till adress $8B000 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
kontakta din laborationshandledare som kontrollerar hur du utfört uppgifterna i detta avsnitt.

Uppvisat för______________________den ____/____

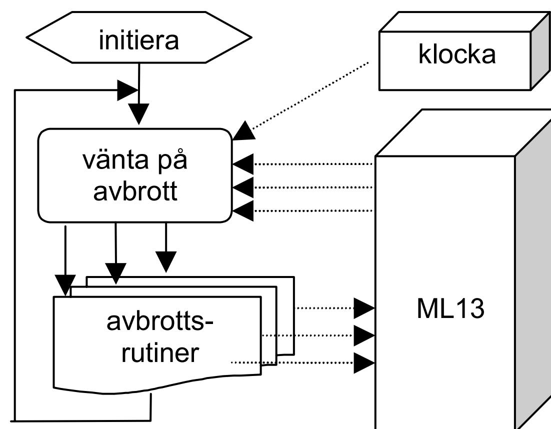
3 Händelsestyrd dörrautomat

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

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

- 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:



Förberedelser:

Inför laborationen måste en avbrottsutgång på *ML13* kopplas till rätt avbrottsingång på *MC68*. Dessutom måste signalerna från IRQ-STATUS registret på *ML13* kopplas till rätt avbrottsnivå.

Vi använder avbrottsnivå 1 hos *MC68*. Detta innebär att vi kopplar en en-polig kopplingskabel mellan stift 8 på *MC68's* 26-poliga kontakt till J8 "IRQ1" på *ML13*.

Programmet ska huvudsakligen skrivas i 'C'. Vi tvingas dock använda en del assemblerkod för att klara avbrottshanteringen. I det följande beskrivs en konstruktion som, tillsammans med ett lämpligt 'C'-program kan vara användbart för laborationen.

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


* Definitioner av portadresser till ML13

      segment     abs
ML13_IRQ_Status         EQU   $8B001
ML13_IRQ_Control        EQU   $8B001

* Definitioner av avbrottstyp från ML13
* Observera att dessa måste överensstämma med
* definitioner i 'C'-programmet

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


* Importerade symboler
      segment     text
* variabeln "interrupt_type" ska deklareras i MOM4.C (global)
* variabeln sätts av avbrottshanteringsrutinen (nedan)
            extern      _interrupt_type

* Exporterade symboler

* Följande funktioner definieras i denna fil.
* De används av huvudprogrammet och ska följaktligen
* externdeklareras där.

define      _standby
* prototyp: void  standby(void);

            define      _init_irq
*     prototyp: void    init_irq(void);

define      _set_timeout
* prototyp: void  set_timeout(int);

* Lokal variabel, används för tidshantering
            BSS
delay_count:            DS.L        1

            TEXT
* Hjälprutiner:
*
* Funktion: void  standby(void);
* Sätter processorns avbrottsmask till 0
* processorn väntar därefter till nästa avbrott

_standby:
            STOP        #$2000
            RTS
```

```
*
* void  init_irq(void);
* sätt upp för avbrott från ML13

_init_irq:
 CLR.B       (ML13_IRQ_Control).L   nollställ eventuella avbrott
*
* Följande initieringar gäller SIM40-modulen i MC68
* de ändrar initieringen för port B så vi kan använda dessa pinnar
* som avbrottsingångar
 ORI.W       #$0100,($FFFFF000).L   full IRQ mode
 ORI.B       #2,($FFFFF01F).L       irq nivå 1 (pin 8,26 pol kontakt)
 ORI.B       #2,($FFFFF006).L       detta är en autovektor-nivå...

* sätt avbrottsrutin för autovektornivån
 MOVE.L      #ML13_irq,($64).L
 RTS




* void  set_timeout(int sekunder)
* se även moment 1

_set_timeout:
 MOVE.L      (4,sp),(delay_count).L  antal sekunders fördröjning
 ORI.W       #$000F,($FFFFF000).L   aktivera intern arbitrering
 MOVE.W      #$0640,($FFFFF022).L   irqnivå 6, avbrottsvektor $40
 MOVE.L      #ptimirq,($100).L      installera avbrottsrutin
 MOVE.W      #$110,($FFFFF024).L    generera 1 avbrott/sekund
 RTS

* Avbrottshantering

* avbrott från periodisk räknare
ptimirq:
 SUBQ.L      #1,(delay_count).L
 BEQ         timeout
 RTE

* vid timeout, stanna timerkretsen
timeout:
 MOVE.W      #0,($FFFFF024).L
 MOVE.L      #TIME_OUT,(_interrupt_type).L
 RTE

* avbrott från ML13
* kontrollera ML13's statusregister och sätt global variabel
* till motsvarande avbrottstyp.

ML13_irq:
 BTST.B      #0,(ML13_IRQ_Status).L
 BEQ         n1
 MOVE.L      #OPENED_DOOR,(_interrupt_type).L
 BRA         n7

n1:     BTST.B      #1,(ML13_IRQ_Status).L
 BEQ         n2
 MOVE.L      #CLOSED_DOOR,(_interrupt_type).L
 BRA         n7

n2:     BTST.B      #2,(ML13_IRQ_Status).L
 BEQ         n3
 MOVE.L      #SENSOR,(_interrupt_type).L
 BRA         n7
n3:     BTST.B      #3,(ML13_IRQ_Status).L
```

```
            BEQ         n4
            MOVE.L      #SENSOR,(_interrupt_type).L
            BRA         n7

n4:   BTST.B      #4,(ML13_IRQ_Status).L
            BEQ         n5
            MOVE.L      #OPENING_DOOR,(_interrupt_type).L
            BRA         n7

n5:   BTST.B      #5,(ML13_IRQ_Status).L
            BEQ         n6
            MOVE.L      #CLOSING_DOOR,(_interrupt_type).L
            BRA         n7

n6:   MOVE.L      #NO_IRQ_TYPE,(_interrupt_type).L
n7:   CLR.B       (ML13_IRQ_Control).L                kvittera avbrott
            RTE
```

Implementering och test

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

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

- 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.
- Skapa en källtextfil `MOM3-LOW.S68` (assembler) med de rutiner som lämpligen skrivs i assembler. Lägg denna fil till projektet.
- Testa programmet med *XCC32*'s debugger, använd simulatorn för att koppla *ML13* till adress $8B000 så att du kan observera "dörrens" beteende. Tänk på att *ML13*-simulatorn kan konfigureras för att generera avbrott, detta är samma sak som då du ansluter avbrottsignalen från ett verkligt *ML13* till avbrottsingången hos *MC68*.

Vid laborationsplatsen

Inför laborationen måste en avbrottsutgång på *ML13* kopplas till rätt avbrottsingång på *MC68*.

> 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 3
kontakta din laborationshandledare som kontrollerar hur du utfört uppgifterna i detta avsnitt.

Uppvisat för______________________den ____/____

4 Non-Pre-Emptive Scheduling

Under detta moment får du stifta bekantskap med en enkel realtidskärna, *RTK32*.

Du ska använda RTK32 för att utföra tre processer under en "non-pre emptive" schemaläggningsstrategi.

I detta moment ska vi se hur *RTK32* kan användas med en *Non-Pre-Emptive* schemaläggningsstrategi. Vi illustrerar samtidigt det huvudprogram som varje applikation för *RTK32* måste innehålla.

Realtidskärnan finns tillgänglig som ett programbibliotek ("librtkd.e32" och "librtk.e32") under *XCC32*.

Läs om RTK32 under XCC32's hjälpsystem.!

RTK32's funktioner

Varje *RTK32*-applikation innehåller ett huvudprogram och ett antal funktioner som kommer att behandlas som processer. *RTK32* tillhandahåller ett antal funktioner för processhantering och synkronisering.

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

initierar *RTK32*'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 *RTK32*, stänger av realtidsklockan, kontrollerar processkön så att ingen process finns kvar.

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

skapar process under *RTK32*. 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 *RTK32* 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 *RTK32* 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 *RTK32*-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 *RTK32* 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){
        printl("\nCan't create process");
        exit();
   }
   if( CreateProcess("P2", P2) == -1){
        printl("\nCan't create process");
        exit();
   }
   StartKernel();
}
```

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

OBSERVERA:
För att kunna använda realtidskärnan måste MC68 vara bestyckad med minst 128 kB RWM.
På samma sätt gäller att du måste modifiera simulator-inställningarna:

R/W Memory banks	Base	Size
0	0	20000
1		
2		
3		

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 *XCC32*'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 4
kontakta din laborationshandledare som kontrollerar hur du utfört uppgifterna i detta avsnitt.

Uppvisat för______________________den ____/____

5 Pre-Emptive Scheduling - Timesharing

Under detta moment ska vi illustrera *Pre-Emptive-Scheduling*, dvs en schemaläggningsstrategi 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 *XCC32*'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.

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
kontakta din laborationshandledare som kontrollerar hur du utfört uppgifterna i detta avsnitt.

Uppvisat för_______________________den ____/____

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.

Processynkronisering i *RTK32* sker med hjälp av *semaforer*. Semaforerna är implementerande 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()`.

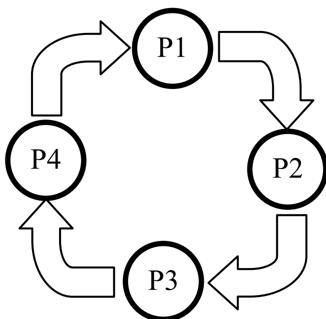
```
void waitsem( int );
```

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.

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 *XCC32*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
kontakta din laborationshandledare som kontrollerar hur du utfört uppgifterna i detta avsnitt.

Uppvisat för________________________den ____/____

I detta avsnitt skapar vi nya typer av applikationer för *RTK32*

Här illustreras meddelandeskickning.

Meddelanden

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 proceduren PutMessage(). Konsumentprocessen läser ett meddelande via proceduren GetMessage() som slänger meddelandet (samma meddelande får alltså inte läsas två gånger) och skriver meddelandet till bildskärmen. Procedurerna 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 procedurerna GetMessage() och PutMessage() enligt specifikationen ovan.
- Skriv ett huvudprogram med processerna *Producer*/*Consumer*.
- Sätt konstanten TIMESLICE så att *RTK32* 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 också använder PutMessage(). Provkör programmet och beskriv kortfattat dina iakttagelser:

____

____

____

____

____

Uppgift 2 - Moment 6C

Avslutningsvis ska du nu modifiera GetMessage() och PutMessage() så att inga meddelanden går förlorade eller läses flera gånger.
Ledning: Använd en lösning med *två* semaforer.

Kontrollstation 7
kontakta din laborationshandledare som kontrollerar hur du utfört uppgifterna i detta avsnitt.

Uppvisat för________________________den ____/____

7 Processynkronisering

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.

Dörrautomaten (*MC68/ML13* med programvara) ska nu implementeras under realtidskärnan *RTK32*.

- Dörren ska kunna ”låsas” genom att en tangent ’l’ (lock) trycks ned på tangentbordet. Låsning kan endast ske samtidigt som dörren är helt öppen. Om dörren är låst ska den kunna ”låsas upp” genom att en tangent ’u’ (unlock) trycks ned på tangentbordet.

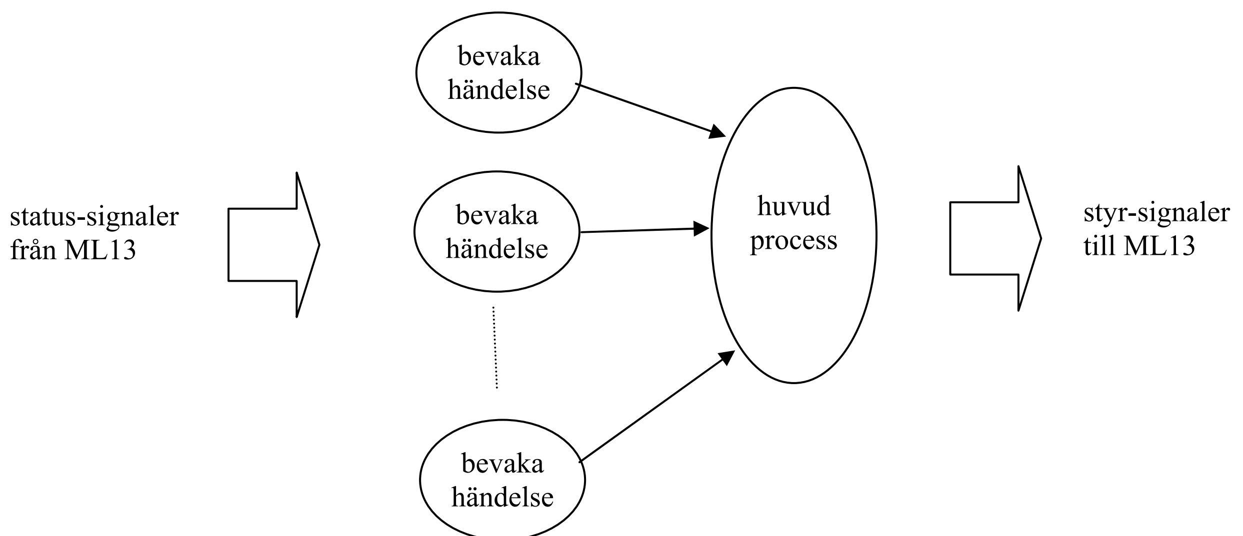
- Med en ”låst” dörr menas att den inte kan öppnas genom att någon av tryckknapparna på *ML13* trycks ned.

- Dörren ska vara låst från början.

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

Exekveringsmodell

Följande figur beskriver systemets processer:



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

```
PROCESS manage_door()
      message to tty ”Press ’u’ to unlock door”
      wait for ’u’ from tty
      message to tty ”Door unlocked”
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
      if ’l’ from tty begin
            enable ”close_door_event”
            message to tty ”Door closing”
            wait for ”door-closed-signal”
            wait for ’u’ from tty
            message to tty ”Door unlocked”
      end
end
```

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:

```
PROCESS open_door_event:
```

```
begin do forever:
      wait for event monitoring enabled;
      begin do forever
            if (open the door )
                  signal (open_the_door_sem)
                  break inner loop
            else
                  abandon cpu
      end
end
```

Funktionen ”yield”

Observera speciellt ”abandon cpu”, dvs processen överlåter resten av sin TIMESLICE till någon annan körbar process. En sådan funktion kallas traditionellt ”yield”. I *RTK32* finns ingen sådan funktion, du måste skriva den själv. Funktionen yield måste:

'yield' är normalt en standardfunktion i realtidskärnor.
Vi har utelämnat den i RTK32 så att du ska få möjlighet att implementera den själv.

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

Ledning:
Studera källtexterna för de färdiga funktionerna 'wait', 'insert_last', 'suspend' och 'dispatch' i *RTK32*.

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

'sleep' är normalt också en standardfunktion i realtidskärnor.

Här har du ett förslag på en enkel implementering.

Läs om 'get_rtk_time' i RTK32's hjälpsystem.

```
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 */
```

Implementering och test

Du kan implementera och testa hela applikationen med hjälp av XCC32's debugger och de olika simulatorerna.

Uppgiften är omfattande. Börja med att implementera och testa 'yield' och 'sleep'.

- Skapa ett nytt projekt 'moment7' i workspace 'RTLAB'. Använd standardinställningarna som föreslås.
- Skapa en källtextfil `MOM7.C` med ett huvudprogram, avbrottshanterare, processer och funktionerna yield/sleep enligt ovan. Lägg till denna fil till projektet.
- Testa programmet med *XCC32*'s debugger, använd simulatorn för att koppla 'Console' till seriekommunikationskretsen så att du kan se programmets utskrifter. Koppla även *ML13* på samma sätt som tidigare.
- Tänk på att simulatorns *ML13* normalt genererar en rad olika typer av avbrott. Du måste stänga av dessa i denna applikation. Försäkra dig om att IOSIMULATOR's *ML13* avbrottsmekanism är avstängd.

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 *MC68*.

Kontrollstation 8
kontakta din laborationshandledare som kontrollerar hur du utfört uppgifterna i detta avsnitt.

Uppvisat för________________________den ____/____