Laborationer med MC11
Realtidssystem

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 XCC Educational, Version 1.3 eller XCC11 Pro, Version 1.0.

Innan du börjar...

I detta häfte behandlas laborationer med enkortsdatorn *MC11* och laborationskortet *ML13*.

Handböcker som beskriver *MC11* och *ML13* finner du på GMV's hemsida, www.gbgmv.se.

Laborationerna genomförs med programutvecklingsmiljön *XCC11*.

I *XCC* finns simulatorer för såväl *MC11* 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 *MC68HC11* 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
- MC68HC11's periodiska räknare

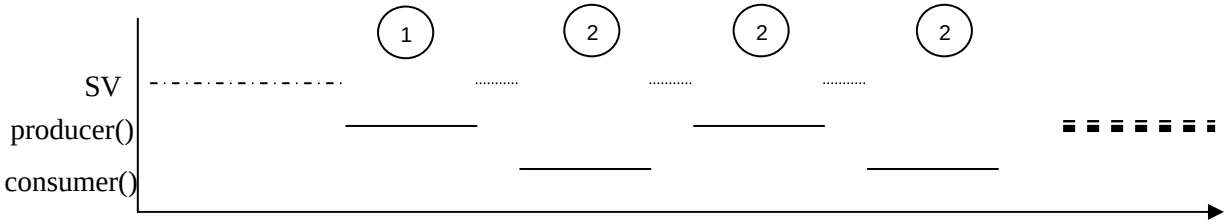
Under denna inledande laboration ska vi visa ett enkelt exempel på hur *tidsdelning* (*time-sharing*) kan utföras. Processortiden ska delas lika mellan två *applikationsprogram:*

- `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 räknarkretsen, vi måste tillhandahålla avbrottsrutinen 'timer_interrupt', denna ska:
   a) spara programmets flyktiga omgivning
   b) starta det andra programmet

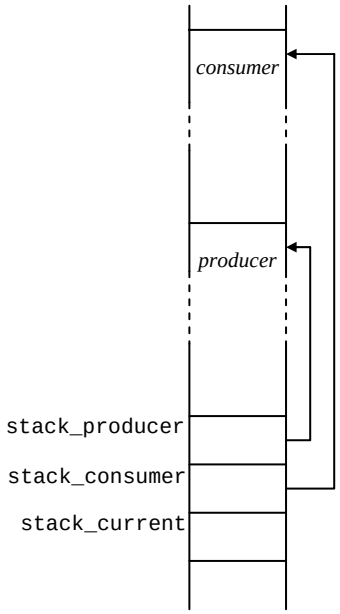
Programmets "flyktiga omgivning" utgörs av processorns registerinnehåll. Vid avbrott sparas automatiskt samtliga registerinnehåll på stacken, vi behöver inte bekymra oss mer för detta.

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 (se figur i marginalen) och 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 assemblerspråk. (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
      RTI
```

Välj namnet:
mom1-low.s11
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:

```
* skapa 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   stack_producer
```

Motsvarande initiering görs även för 'consumer()'.

Nästa steg blir att initiera räknarkretsen så att den genererar avbrott med jämna mellanrum. Vi använder här den periodiska räknaren hos *MC68HC11*, den är avsedd för sådana här ändamål och därför också enkel att hantera.

periodisk räknare

Räknaren kontrolleras via tre register, *Pulse Accumulator Control* (PACTL), två bitar i detta register bestämmer periodiciteten för avbrott, *Timer Interrupt Mask Register 2* (TMSK2), via detta register aktiveras räknarens avbrottsmekansim. *Timer Interrupt Flag Register 2* (TFLG2), registret innehåller en statusbit som anger om ett helt intervall räknats. Genom att skriva till samma bit kvitteras avbrott. Observera att registren har fler funktioner men vi nöjer oss här med det som behövs för att åstadkomma en *realtidsklocka*.

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.

PACTL - Pulse Accumulator Control.

Används, för att skapa tidbasen för avbrottsintervall.

7	6	5	4	3	2	1	0
x	x	x	x	0	0	RTR1	RTR0

Bitar som anges med 'x' har ovidkommande funktion. I vår tillämpning ska dessa vara noll. RTR-bitarna (*Real Time Interrupt Rate select*) bestämmer tiden mellan varje avbrott. Periodiciteten beror av systemets klockfrekvens och kan avläsas ur följande tabell:

RTR1	RTR0	8 MHz	4 MHz
0	0	4.10 ms	8.19 ms
0	1	8.19 ms	16.38 ms
1	0	16.38 ms	32.77 ms
1	1	32.77 ms	65.54 ms

TMSK2 - Timer Interrupt Mask Register 2.

Aktiverar avbrottsystemet i RTC.

7	6	5	4	3	2	1	0
x	RTII	x	x	0	0	x	x

Bitar som anges med 'x' har ovidkommande funktion. I vår tillämpning ska dessa vara noll. För att aktivera avbrott från RTC'n ska denna bit ettställas.

TFLG2 - Timer Interrupt Flag Register 2.

Statusregister för RTC.

7	6	5	4	3	2	1	0
x	RTIF	x	x	0	0	0	0

RTIF sätts till 1 vid avbrott från RTC. Avbrottsbegäran kvitteras genom att något värde (0 eller 1) skrivs till denna bit.

Följande kodsekvenser ger exempel på användning av RTC i *MC68HC11A*.

```
* Definitioner för RTC-kretsen
RTC_TMSK2     EQU    $1024
RTC_TFL2      EQU    $1025
RTC_PACTL     EQU    $1026

timer_init:
* avbrottsvektor måste initieras under BUFFALO...
      LDX    #timer_interrupt
#ifdef SP2
      STX    $7FF0         Avbrottsvektor RTC
#else
      STX    $3FF0         Avbrottsvektor RTC
#endif
* men vi skriver den också till 'rätt' adress för att
* även kunna använda exemplet i simulatorn...
      STX    $FFF0

* initiera RTC avbrottsfrekvens
* tidsbas (0,1,2,3 där 0 är den kortaste perioden)
* enligt tabell ovan
      LDAA   #3            32.77 ms/8MHz
      STAA   RTC_PACTL

* starta klocka
      LDAA   #$40
      STAA   RTC_TMSK2

* nollställ I-flagga så att avbrott accepteras
      CLI

      RTS
```

VIKTIGT:
Avbrottsvektorn måste skrivas till rätt minnesadress, denna kan skilja sig mellan olika MC11-system beroende på om systemet uppgraderats (”sp2”) eller inte. Vi illustrerar detta här med villkorlig assemblering.

Vid avbrott måste som sagt detta kvitteras, det följande ger exempel på en minimal avbrottsrutin för RTC'n.

```
timer_interrupt:
* kvittera avbrott från RTC
      LDAA   #$40
      STAA   RTC_TFL2
      RTI
```

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

```
* mom1-low.s11
      segment      abs
* Definitioner för RTC-kretsen i HC11
RTC_TMSK2     EQU    $1024
RTC_TFL2      EQU    $1025
RTC_PACTL     EQU    $1026

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_stack       RMW    1

* temporär lagring stackpekare för 'producer'
stack_producer      RMW    1

* temporär lagring stackpekare för 'consumer'
stack_consumer      RMW    1
```

```
       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;
       init_consumer;
       init_RTC;
*
*    utför nu start av 'producer()' så som
*    'timer_interrupt' gör det...
       CLR    RUNNING
       LDS    stack_producer
       RTI
*      Exekveringen kommer aldrig tillbaks hit...
```

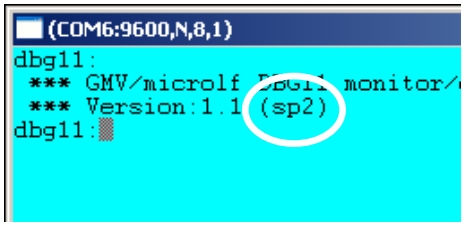
producent-konsument processerna ska se ut på följande sätt:

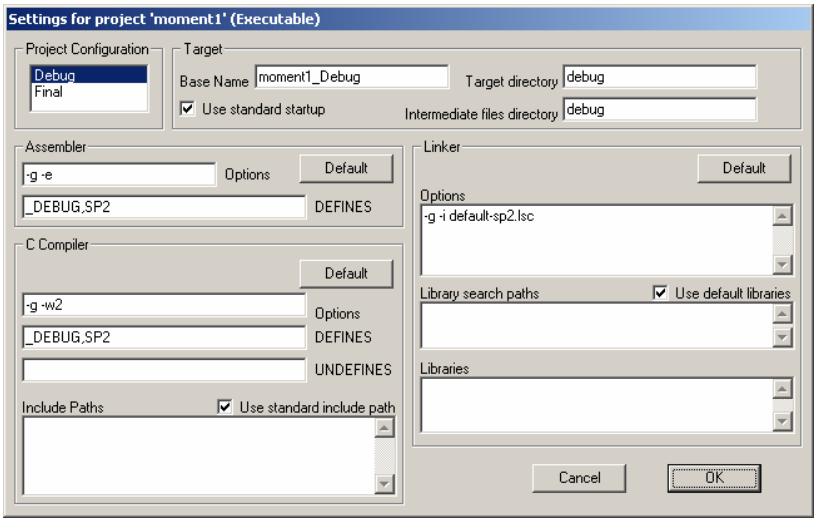
```
/*
       mom1.c
       Enkel 'producent/konsument'
*/

#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

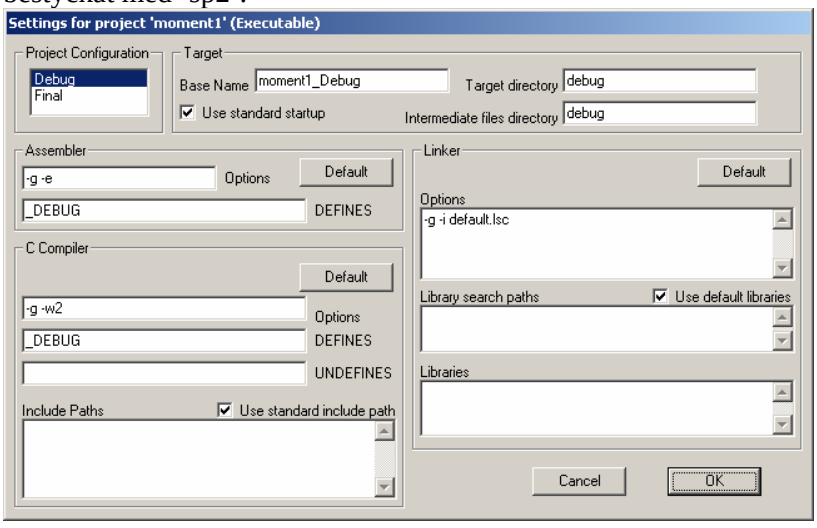
Du bör nu vara mogen att implementera och testa denna första enkla applikation. Du kan använda den inbyggda simulatorn i *XCC11* tillsammans med IO-simulatorn för att testa programmet *innan* du går till laborationsplatsen. Följande arbetsgång kan vara lämplig:

- Starta *XCC11*
- Skapa ett nytt 'Workspace' - namnge det 'RTLAB'.
- Skapa nu ett nytt projekt, 'moment1'. Projektets inställningar beror av den version av *MC11* ditt laborationssystem är uppbyggt kring. Är du osäker bör du fråga din lärare. Om du har ett laborationssystem tillgängligt kan du enkelt kontrollera, tryck ’RESET’ på laborationssystemet, *DBG11* identifierar sig med en utskrift, efter ’Version’ står texten ’sp2’ (se marginalen) för system som uppgraderats med sådan hårdvara. Följande inställningar är lämpliga om ditt laborationssystem är bestyckat med ’sp2’.





Använd följande inställningar om ditt laborationssystem *inte* är bestyckat med ’sp2’.



- Skapa källtexterna 'mom1-low.s11' respektive 'mom1.c' enligt tidigare anvisningar.
- Lägg de nya källtexterna till projektet.
- Välj 'Build All' för att skapa applikationen.

- Testa programmet med *XCC11*'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 (32.77 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 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

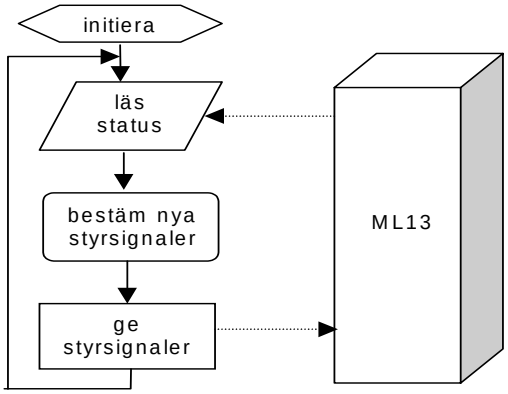
Under detta moment ska du implementera en 'dörrautomat' med ML13 utan att använda avbrott.

Beskrivning av ML13 finns i separat handbok, en enklare beskrivning finns även under hjälpsystemet i XCC11

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)
```

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

macrot kan sedan användas, exempelvis på följande sätt:

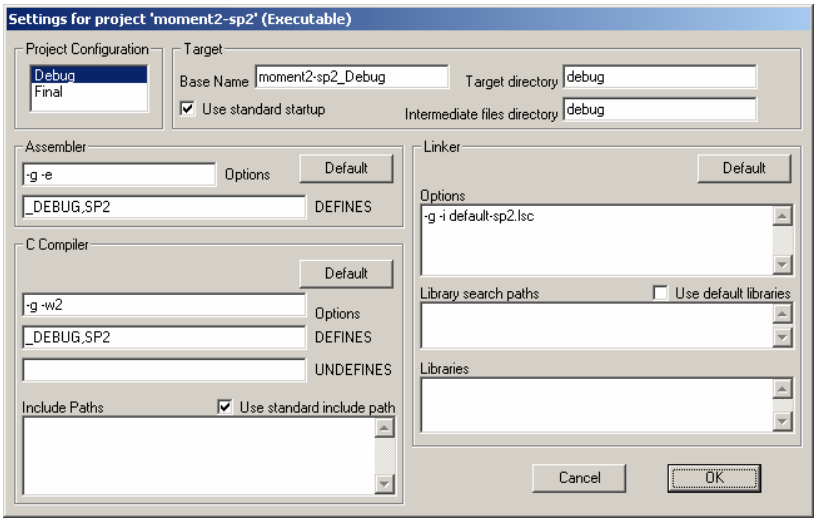
```
          set_control(0x1);
```

satsen skriver värdet 1 till *ML13's* styrregister

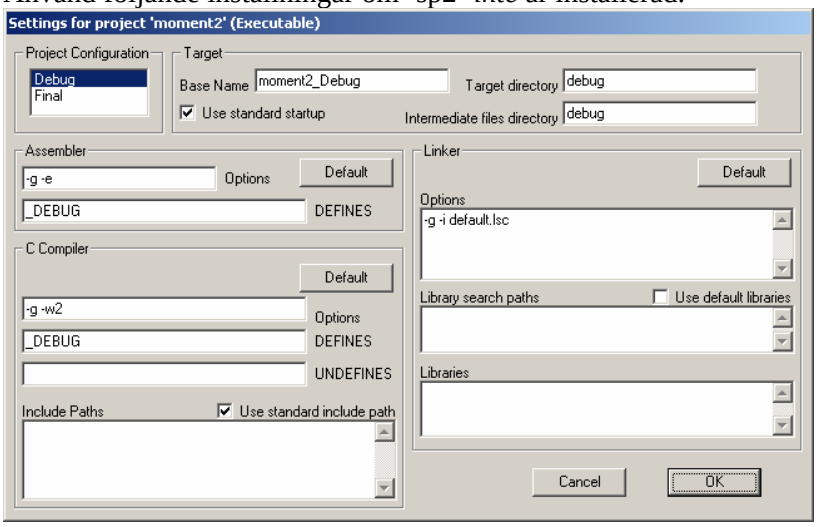
Implementering och test

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

- Skapa ett nytt projekt 'moment2' i workspace 'RTLAB'. Använd följande inställningar för ’sp2’.



Använd följande inställningar om ’sp2’ *inte* är installerad.



- Skapa en källtextfil `MOM2.C` med ett huvudprogram som löser laborationsuppgiften. Lägg till denna fil till projektet.
- Testa programmet med *XCC11*'s debugger, använd simulatorn 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
kontakta din laborationshandledare som kontrollerar hur du utfört uppgifterna i detta avsnitt.

Uppvisat för_____den ____/____

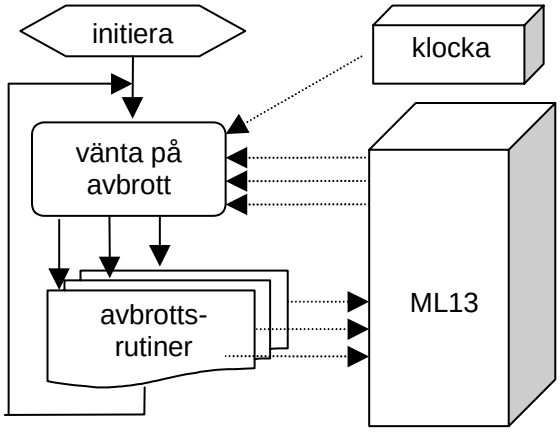
3. Händelsestyrd dörrautomat

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:



Förberedelser:

Programmet ska huvudsakligen skrivas i 'C'. Vi väljer först att använda 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"

       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

*
```

```
       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:
* återställ avbrottskällor
       CLR    ML13_IRQ_Control
       CLR    RTC_TMSK2
* avbrottsvektor måste initieras under debugger...
       LDX    #ptimirq
#ifdef SP2
       STX    $7FF0        Avbrottsvektor RTC
#else
       STX    $3FF0        Avbrottsvektor RTC
#endif
* 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
#ifdef SP2
       STX    $7FF2
#else
       STX    $3FF2
#endif
       STX    $FFF2

       CLI                 acceptera avbrott
       RTS
       exit   _init_irq

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      abs
* Definitioner för RTC-kretsen i HC11
RTC_TMSK2    EQU    $1024
RTC_TFL2     EQU    $1025
RTC_PACTL    EQU    $1026

       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 HC11 och RTR1/RTR0 =11 får vi 32.77 ms
* mellan avbrotten, detta ger c:a 30
* avbrott/sekund
       LDAA   #30
       MUL
* Rimlighetskontroll ...
       BEQ    set_tim_exit
       STD    delay_count
       LDAA   #3           tidsbas
       STAA   RTC_PACTL
* starta klocka
       LDAA   #$40
       STAA   RTC_TMSK2
set_tim_exit:
       PULY
       RTS

* avbrott från realtidsklockan
ptimirq:
       LDAA   #$40         kvittera avbrott från RTC
       STAA   RTC_TFL2
       LDD    delay_count
       BEQ    timeout
       SUBD   #1
       STD    delay_count
       RTI

* vid timeout, stanna timerkretsen
timeout:
       CLR    RTC_TMSK2
       LDD    #TIME_OUT
       STD    _interrupt_type
       RTI

       bss
delay_count: RMW    1

* Dummy variabel för att kringgå WAI-problematiken
has_irq      RMB    1
```

Kodning av avbrottsrutin i 'C'

De visade rutinerna kan förstås också kodas 'C'. Även avbrottshanteringsrutinen kan alternativt kodas i 'C'. Du måste dock deklarera rutinen på speciellt sätt. Observera att detta är en utvidgning av programspråket 'C'. Sådana utvidgningar förekommer i praktiskt taget alla C-kompilatorer. En nackdel är dock att de inte är standardiserade.

EXEMPEL:

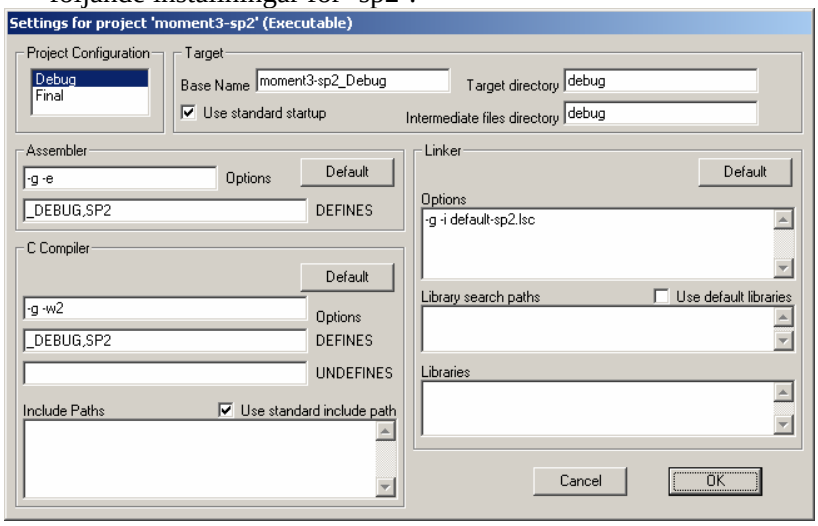
```
__interrupt void ptimirq( void )
{
        *((char *) RTC_TFL2 )= 0x40;
        if( delay_count == 0 )
        {       /* timeout ... */
                *((char *) RTC_TMSK2 )= 0;
                interrupt_type = TIME_OUT;
        }else{
                delay_count--;
        }
}
```

Implementering och test

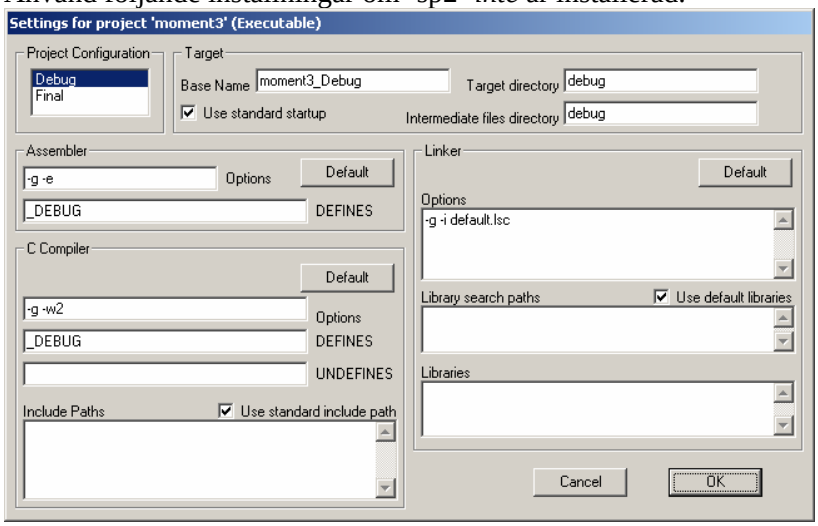
Läroboken ger ett utförligt förslag på hur huvudprogrammet i denna uppgift ska utformas.

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 följande inställningar för ’sp2’.



Använd följande inställningar om ’sp2’ *inte* är installerad.

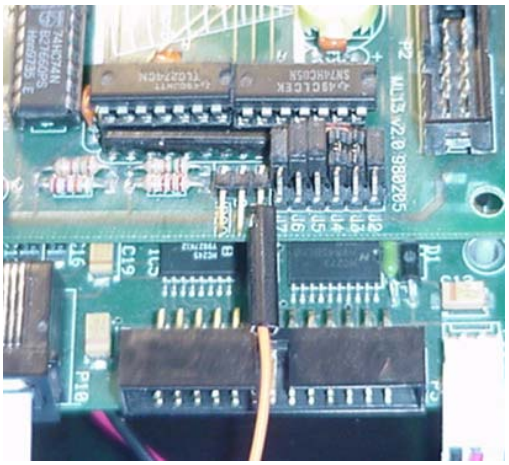


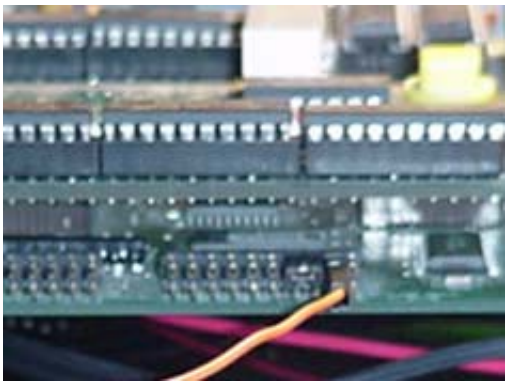
- Skapa en källtextfil `MOM3.C` med ett huvudprogram som löser laborationsuppgiften. Lägg till denna fil till projektet.
- Om du väljer att koda i assembler enligt förslaget ovan, skapa en källtextfil `MOM3-LOW.S11` med de rutiner som lämpligen skrivs i assembler. Lägg denna fil till projektet.

- Testa programmet med *XCC11*'s debugger, använd simulatorn för att koppla *ML13* till adress $B00 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 *MC11*.

Vid laborationsplatsen

Inför laborationen måste en avbrottsutgång på *ML13* kopplas till rätt avbrottsingång på *MC11*.





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 realtids-kärna, *RTK11*.

Du ska använda RTK11 för att utföra tre processer under en "non-pre emptive" schemaläggnings-strategi.

I detta moment ska vi se hur *RTK11* kan användas med en *Non-Pre-Emptive* schemaläggningsstrategi. Vi illustrerar samtidigt det huvudprogram som varje applikation för *RTK11* måste innehålla.

Realtidskärnan finns tillgänglig som ett programbibliotek ("librtkd.e11" och "librtk.e11") under *XCC11*.

Läs om RTK11 under XCC11's hjälpsystem.!

RTK11's funktioner

Varje *RTK11*-applikation innehåller ett huvudprogram och ett antal funktioner som kommer att behandlas som processer. *RTK11* tillhandahåller ett antal funktioner för processhantering och synkronisering.

`void InitKernel(int, void(*)());`
initierar *RTK11*'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 *RTK11*, stänger av realtidsklockan, kontrollerar processkön så att ingen process finns kvar.

`int CreateProcess( char *, void(*)() );`
skapar process under *RTK11*. 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 *RTK11* 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 *RTK11* 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 *RTK11*-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 *RTK11* 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...

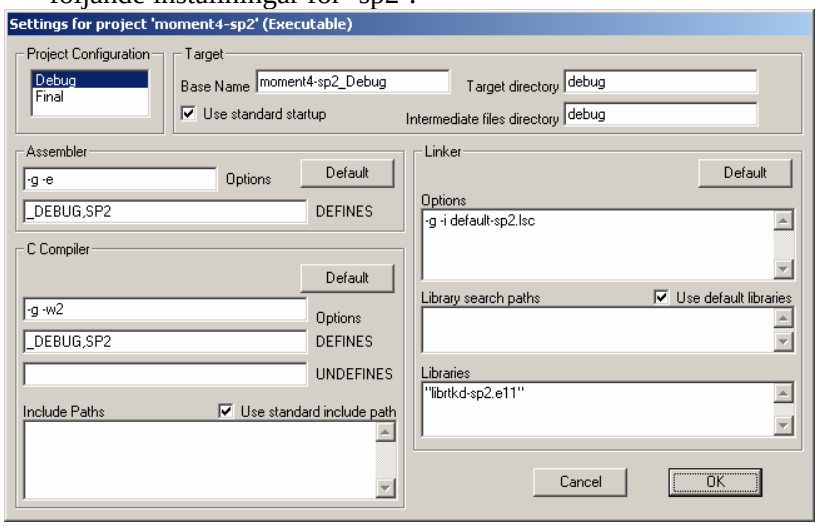
```
main()
{
      InitKernel(TIMESLICE, AtInterrupt);
      if( CreateProcess("P1", P1) == -1){
            printl("\nCan't create process");
            ExitKernel();
      }
      if( CreateProcess("P2", P2) == -1){
            printl("\nCan't create process");
            ExitKernel ();
      }
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

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

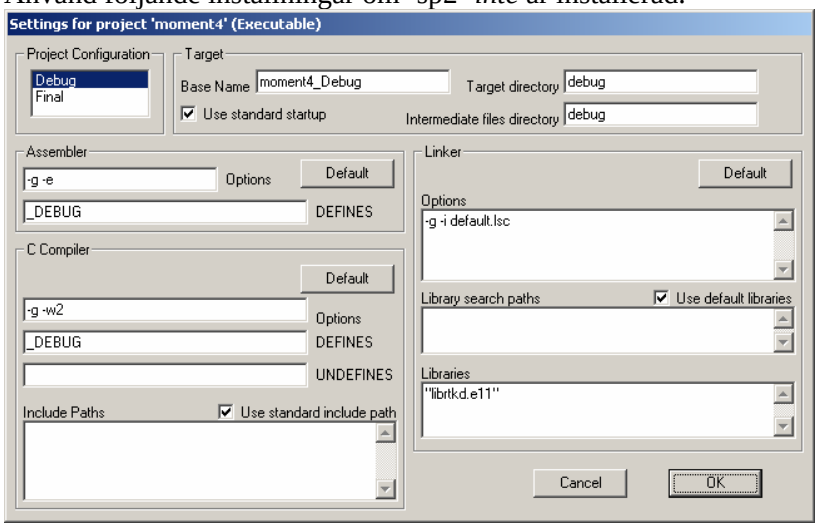
Implementering och test

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

- Skapa ett nytt projekt 'moment4' i workspace 'RTLAB'. Använd följande inställningar för ’sp2’.



Använd följande inställningar om ’sp2’ *inte* är installerad.



- 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 *XCC11*'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');
      }
}
```

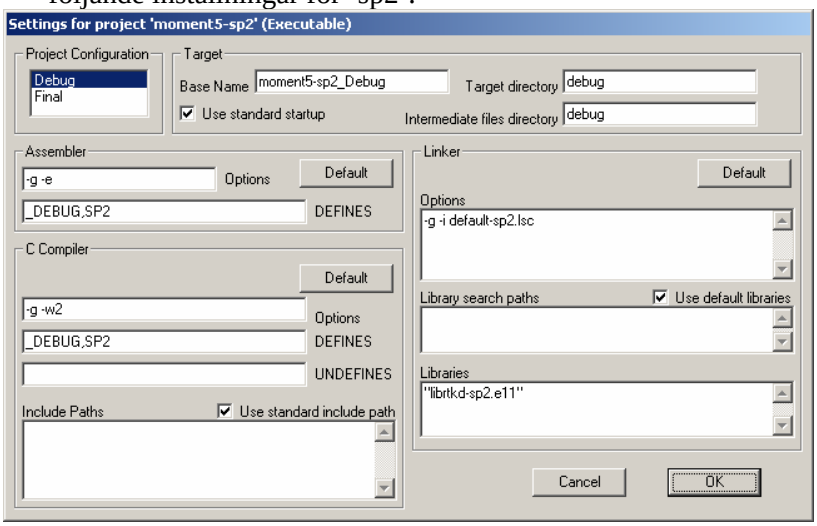
Vi har alltså tre "oändliga" processer som aldrig (frivilligt) terminerar. För att åstadkomma enkel rättvisa fördelar vi processortiden lika mellan dessa, processbytet gör vi efter varje *timeslice*, dvs i vår avbrottshanterare:

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

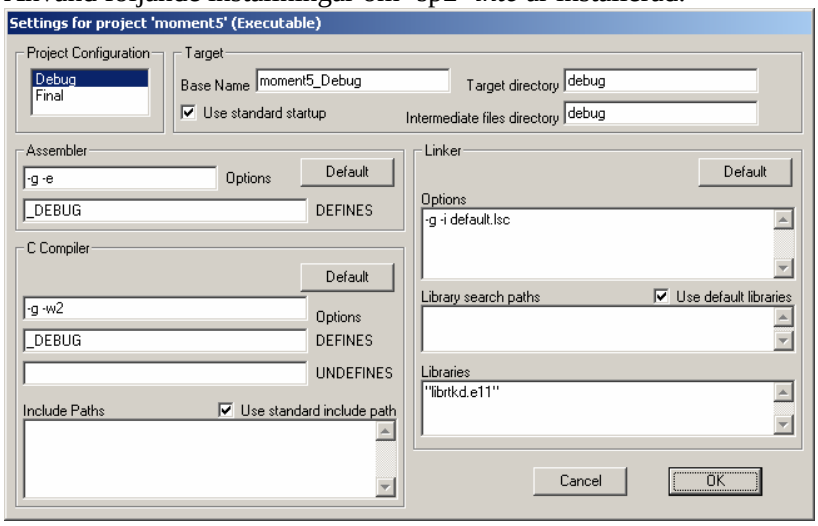
Implementering och test

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

- Skapa ett nytt projekt 'moment5' i workspace 'RTLAB'. Använd följande inställningar för ’sp2’.



Använd följande inställningar om ’sp2’ *inte* är installerad.



- 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 *XCC11*'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.

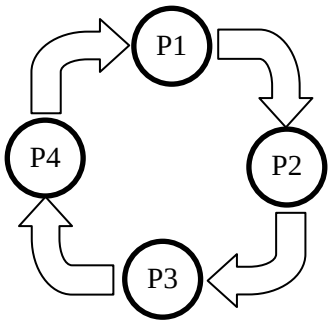
Processynkronisering i *RTK11* sker med hjälp av *semaforer*. Semaforerna är implementerande som *Blockerande/Kö* (se även läroboken). Maximala antalet semaforer ges av konstanten `MAX_SEM_ID` definierad i `_rtk.h`. Följande operationer kan utföras på en semafor:

```
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.

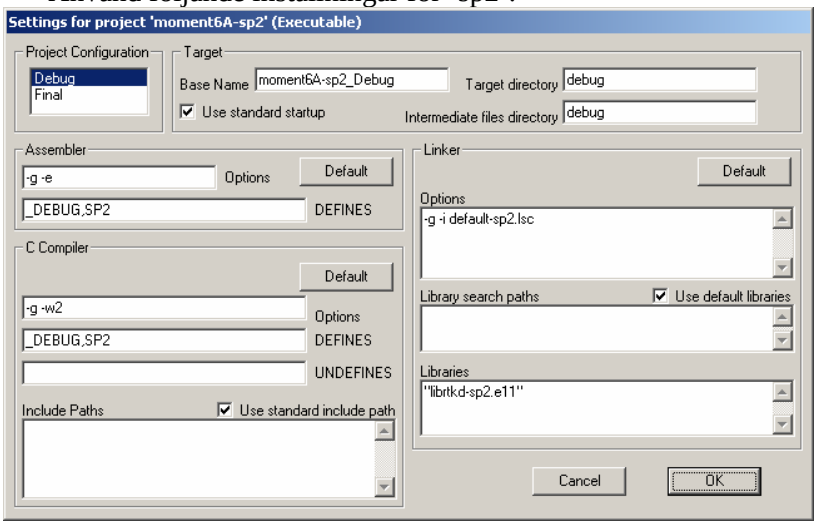
Processerna ska modifieras så att de bara skriver ett tecken åt gången till skärmen.

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

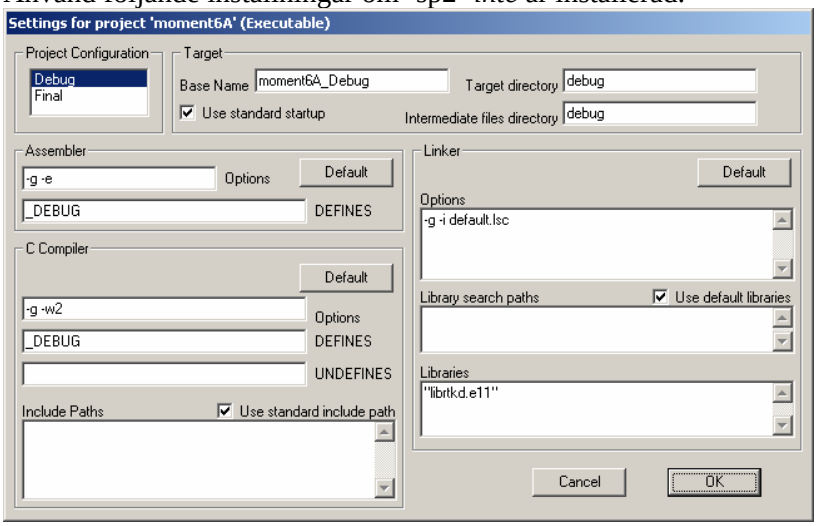
Implementering och test

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 ett nytt projekt 'moment6a' i workspace 'RTLAB'. Använd följande inställningar för ’sp2’.



Använd följande inställningar om ’sp2’ *inte* är installerad.



- 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 *XCC11*'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 ____/____

Meddelanden

I detta avsnitt skapar vi nya typer av applikationer för *RTK11*

Här illustreras meddelandeskickning.

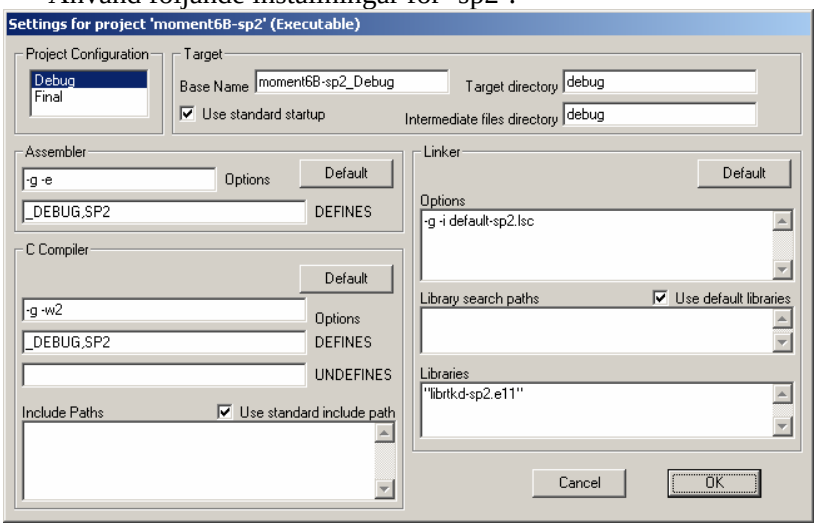
I detta avsnitt ska vi simulera en anslagstavla. Två processer *producer()* och *consumer()* är givna, se nedan. Producentprocessen sätter med jämna mellanrum upp ett meddelande bestående av ett ASCII-tecken (A-Z) på tavlan med hjälp av proceduren PutMessage(). Konsumentprocessen läser ett meddelande 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++);
      }
}
```

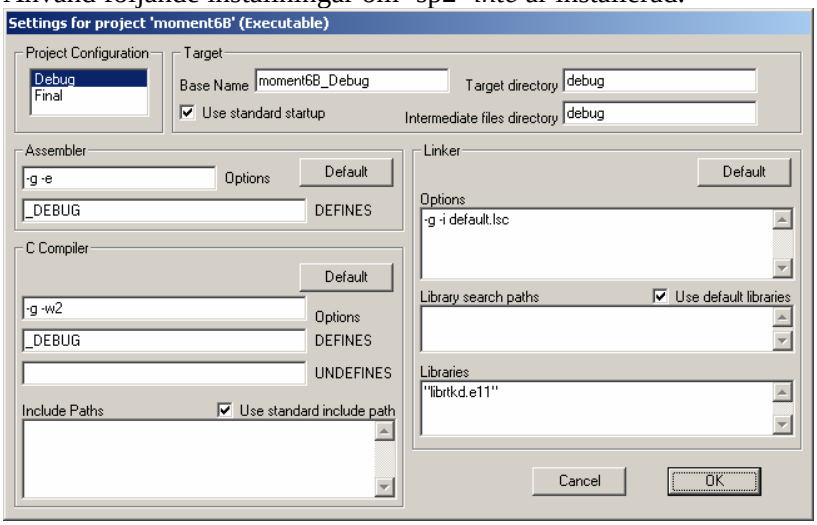
Definiera själv de olika konstanterna `PRODUCER_DELAY` och `CONSUMER_DELAY`.

Uppgift - Moment 6B

- Skapa ett nytt projekt 'moment6b' i workspace 'RTLAB'. Använd följande inställningar för ’sp2’.



Använd följande inställningar om ’sp2’ *inte* är installerad.



- Skapa en ny källtextfil ”`MOM6B.C`”. Skriv procedurerna GetMessage() och PutMessage() enligt specifikationen ovan.
- Skriv ett huvudprogram med processerna *Producer*/*Consumer*.
- Sätt konstanten TIMESLICE så att *RTK11* 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.

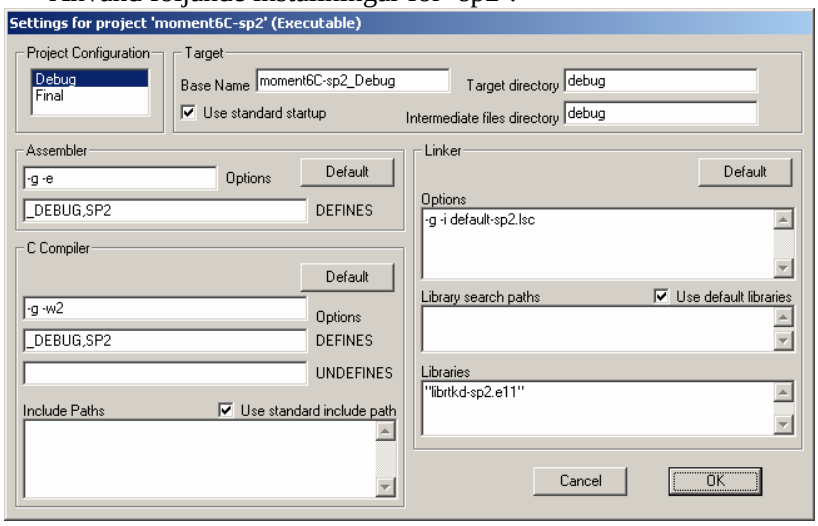
Beskriv iakttagelser:

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

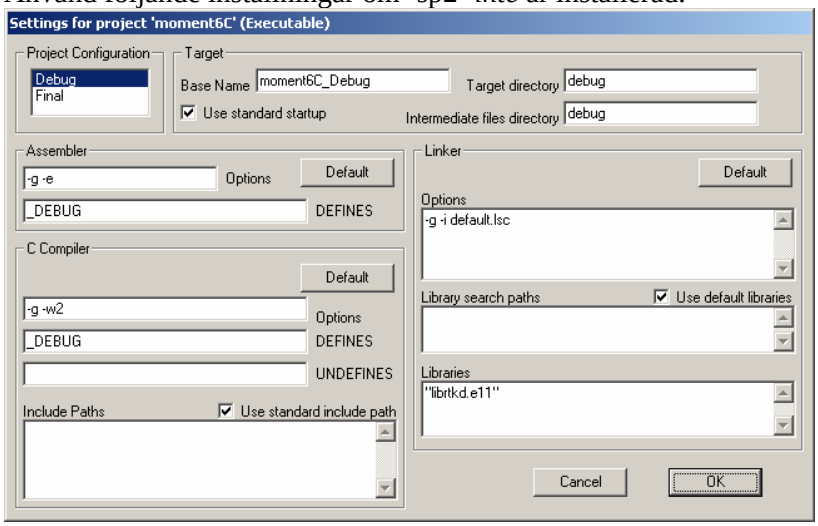
Uppgift - Moment 6C

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.

- Skapa ett nytt projekt 'moment6c' i workspace 'RTLAB'. Använd följande inställningar för ’sp2’.



Använd följande inställningar om ’sp2’ *inte* är installerad.



- Kopiera din tidigare lösning ”MOM6B.C” till en ny fil ”MOM6C.C”. Modifiera procedurerna GetMessage() och PutMessage() enligt specifikationen ovan.

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 (*MC11/ML13* med programvara) ska nu implementeras under realtidskärnan *RTK11*.

- 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.

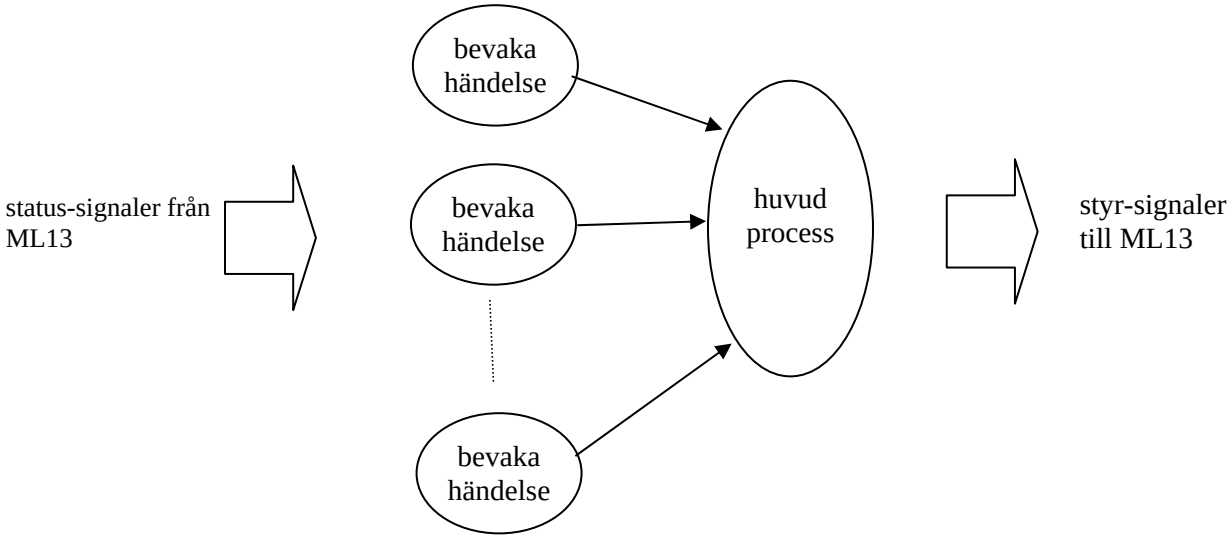
- Med en ”låst” dörr menas att den inte kan öppnas genom att någon av tryckknapparna på *ML13* trycks ned.

- 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) efterliknas.

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 *RTK11* finns ingen sådan funktion, 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 *RTK11*.

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:

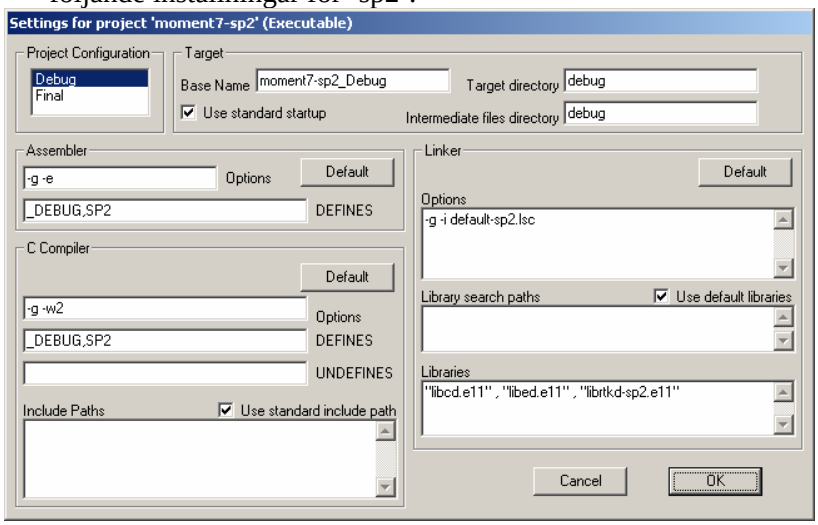
```
...
            sleep (50);          /* 5 sec. delay */
```

Funktionen ”get_system_time” (finns i *RTK11*) returnerar aktuell systemtid. (Läs om denna i *XCC11*'s hjälpsystem).

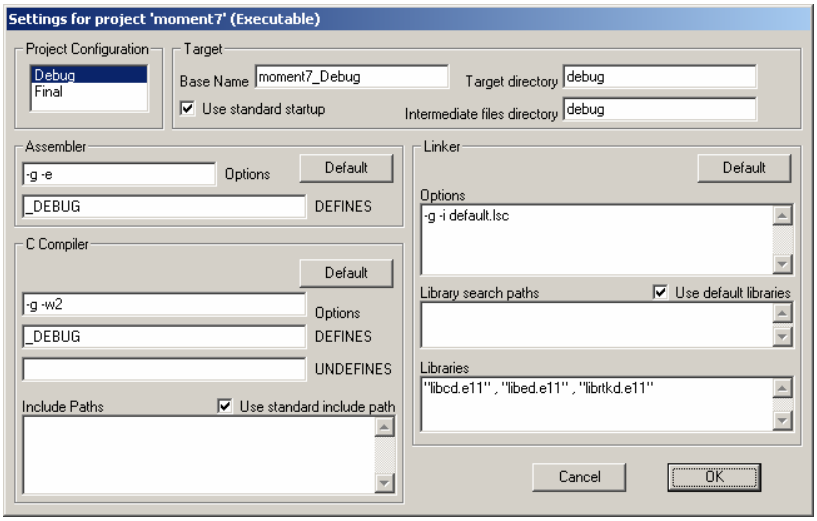
Implementering och test

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

- Skapa ett nytt projekt 'moment7' i workspace 'RTLAB'. Använd följande inställningar för ’sp2’.



Använd följande inställningar om ’sp2’ *inte* är installerad.



- 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 *XCC11*'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.

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 *MC11*.

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

Uppvisat för_____den ____/____