UPPGIFT 5.9

Projektet `asciidisplay` ska nu färdigställas och testas. Följande huvudprogram ska användas för teständamål.

```
int main( void )
{
 char *s ;
 char test1[] = "Alfanumerisk ";
 char test2[] = "Display - test";

 init_app();
 ascii_init();
 ascii_gotoxy(1,1);
 s = test1;
 while( *s )
    ascii_write_char( *s++ );
 ascii_gotoxy(1,2);
 s = test2;
 while( *s )
    ascii_write_char( *s++ );
 return 0;
}
```

- Bygg projektet och rätta eventuella fel.
- Testa ditt program med *CodeLite* och *SimServer*.

5.4 Drivrutiner för grafisk display

Gränssnittet mot den grafiska displayen på LCD-modulen är mycket likartat ASCII-displayen. Här använder vi därför drivrutiner som redan är inbyggda i *MD407*:s debugger, dessa är

```
void graphic_initalize(void);
void graphic_clear_screen (void);
void graphic_pixel_set (int x, int y);
void graphic_pixel_clear (int x, int y);
```

Drivrutinerna kan användas via den speciella instruktionen `SVC` (*SuperVisor Call*) som startar undantagshantering. Detaljerna för undantagshantering kommer vi att beskriva i kapitel 6, här nöjer vi oss med att se hur vi kan använda instruktionen.

En funktion för användning av undantagshantering kan i GCC allmänt definieras på följande sätt:

```
__attribute__((naked))
returtyp namn(parametrar)
{
   __asm volatile (" .HWORD  0xDFxx\n");
   __asm volatile (" BX LR\n");
}
```

`__attribute__((naked))` betyder som bekant att GCC inte ska generera kod exempelvis för lokala variabler eller utträde ur funktionen. Den resulterande assemblerkoden i detta fall blir därför:

```
namn:
   .HWORD  0xDFxx
   BX LR
```

Konstanten 0xDFxx kommer här att tolkas som operationskoden för instruktionen `SVC` där *xx* utgör 8 bitar som kan definieras av användaren. På detta sätt kan 256 olika inbyggda funktioner direkt definieras med hjälp av `SVC`, här använder vi fyra utav dessa möjligheter.

```
__attribute__((naked))
void graphic_initalize(void)
{
    __asm volatile (" .HWORD  0xDFF0\n");
    __asm volatile (" BX LR\n");
}
```

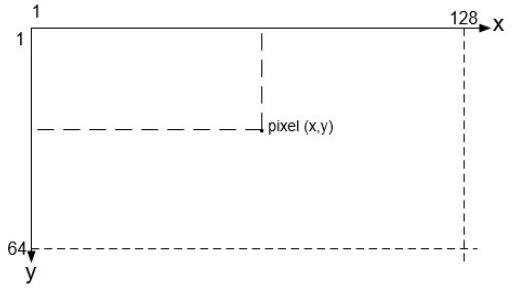
```
__attribute__((naked))
void graphic_clear_screen(void)
{
    __asm volatile (" .HWORD  0xDFF1\n");
    __asm volatile (" BX LR\n");
}
```

```
__attribute__((naked))
void graphic_pixel_set( int x, int y)
{
    __asm volatile (" .HWORD  0xDFF2\n");
    __asm volatile (" BX LR\n");
}
```

```
__attribute__((naked))
void graphic_pixel_clear( int x, int y)
{
    __asm volatile (" .HWORD  0xDFF3\n");
    __asm volatile (" BX LR\n");
}
```

FIGUR 5.11 IMPLEMENTERING AV DRIVRUTINER FÖR GRAFISK DISPLAY

Den grafiska displayens geometri framgår av följande figur:



FIGUR 5.12 GRAFIKDISPLAY BESKRIVEN MED LOGISKA KOORDINATER.

UPPGIFT 5.10

Skapa ett nytt projekt, `graphicdisplay`, med `startup` och en tom `main`-funktion.

- Kopiera fördröjningsrutinerna från föregående uppgift till den nya c-filen.
- Implementera drivrutinerna enligt figur 5.11 ovan. Observera att Port E initieras av `graphic_initialize`.
- Skapa nu ett huvudprogram enligt följande:

```
void main(void)
{
   int   i;
   graphic_initalize();
   graphic_clear_screen();
   for( i = 1; i <= 128; i++ )      /* rita en horisonell linje */
      graphic_pixel_set( i, 10 );
   for( i = 1; i <= 64; i++ )          /* rita en vertikal linje */
      graphic_pixel_set( 10, i );
   delay_milli( 500 );              /* vänta 0,5 sekunder */
   for( i = 1; i <= 128; i++ )
      graphic_pixel_clear( i, 10 ); /* sudda horisontella linjen */
   for( i = 1; i <= 64; i++ )
      graphic_pixel_clear( 10, i ); /* sudda vertikala linjen */
}
```

- Kompilera, rätta eventuella syntaxfel
- Testa nu funktionerna, programmet ska resultera i utskrift av en vertikal och en horisontell linje enligt figuren.

De uppritade linjerna visas i en halv sekund, därefter ska de raderas och återställa en blank display.

EXEMPEL 5.3 DATASTRUKTUR FÖR EN PUNKT I ETT PLAN

Eftersom vår display har en begränsad geometri ryms koordinaterna inom 8 bitar och vi kan definiera en lämplig typ (`POINT`) för en punkt, typdefinitionen definierar samtidigt en pekartyp (`PPOINT`) till datastrukturen:

```
typedef struct
{
  char x,y;
} POINT, *PPOINT;
```

UPPGIFT 5.11 TYPDEFINITION AV EN LINJE

Definiera en sammansatt typ `LINE`, och en pekartyp `PLINE` som representerar en linje, dvs. två punkter med namnen `p0` och `p1`. Använd den tidigare typdefinitionen av en punkt.

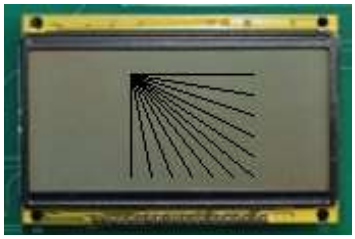
```
typedef struct
{


} LINE, *PLINE;
```

För att rita en rät linje mellan två godtyckliga punkter på displayen kan *Bresenham's algoritm* användas. Algoritmen använder bara heltalsaritmetik.

Bresenham's algoritm: line(x0, x1, y0, y1)

```
if( abs(y1 - y0) > abs(x1 - x0) )  then
   steep = 1;
else
   steep = 0;
if steep then
   swap(x0, y0);
   swap(x1, y1);
if (x0 > x1) then
   swap(x0, x1);
   swap(y0, y1);
deltax = x1 - x0;
deltay = abs(y1 - y0);
error = 0;
y = y0;
if  (y0 < y1) then
   ystep = 1;
else
   ystep = -1;
for x from x0 to x1
   if steep then
      graphic_pixel_set (y,x);
   else
      graphic_pixel_set (x,y);
   error = error + deltay;
   if  (2×error ≥ deltax)
      y = y + ystep;
      error = error – deltax;
```

UPPGIFT 5.12 PLOT AV EN RÄT LINJE



Du ska konstruera en funktion `int draw_line(PLINE l)` som sammanbinder två punkter med en rät linje. Funktionen använder `graphic_pixel_set` för att plotta linjens punkter på grafikdisplayen. Om linjen kan plottas korrekt, dvs. koordinaterna är giltiga för displayen, returneras 1, annars returneras 0. Endast heltalsaritmetik får användas i denna uppgift.

Skapa ett nytt projekt, `linetest`, med `startup` och en tom `main`-funktion.

- Kopiera fördröjningsrutiner och grafikdisplayens drivrutiner från uppgift 5.10 till den nya c-filen.
- Implementera `draw_line` enligt specifikationen.
- Skapa nu ett huvudprogram enligt följande testprogram som ritar linjer från en enskild punkt (x=40, y=10) i en solfjäderform på displayen.

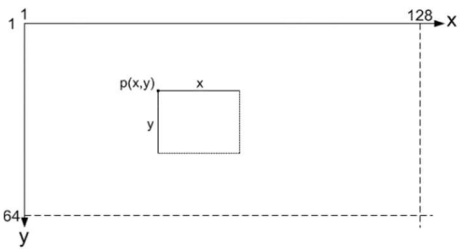
```
LINE lines[]={
  {40,10, 100,10},
  {40,10, 100,20},
  {40,10, 100,30},
  {40,10, 100,40},
  {40,10, 100,50},
  {40,10, 100,60},
  {40,10, 90,60},
  {40,10, 80,60},
  {40,10, 70,60},
  {40,10, 60,60},
  {40,10, 50,60},
  {40,10, 40,60}
};
```

```
void main(void)
{
  graphic_initalize();
  graphic_clear_screen();
  while( 1 )
  {
    for( int i = 0;i< sizeof(lines)/sizeof( LINE );i++)
    {
          draw_line( &lines[i] );
          delay_milli( 500 );
    }
     graphic_clear_screen();
  }
}
```

UPPGIFT 5.13 PLOT AV REKTANGEL

Med en fungerande `draw_line` funktion kan man snabbt implementera ytterligare geometrier.

En rektangel kan exempelvis lämpligen definieras av *origo* p(x,y), utsträckning i x-led och utsträckning i y-led enligt figuren till höger.



Definiera en sammansatt typ `RECT`, och en pekartyp `PRECT` som representerar en rektangel med en punkt `p` som origo och ytterligare två heltal `x` och `y` som anger rektangelns utsträckning i x respektive y-led relativt origo.Använd den tidigare typdefinitionen av en punkt för origo.

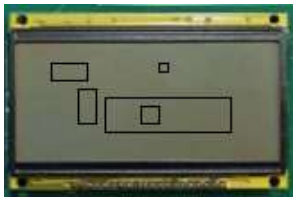
```
typedef struct
{


}
```

Algoritm: draw_rectangle( )

start.x = p(x), start.y = p(y); end.x = p(x)+x; end.y = p(y); line = (start,end); draw_line(line);
start.x = p(x)+x, start.y = p(y); end.x = p(x)+x; end.y = p(y)+y; line = (start,end); draw_line(line);
start.x = p(x)+x, start.y = p(y)+y; end.x = p(x); end.y = p(y)+y; line = (start,end); draw_line(line);
start.x = p(x), start.y = p(y)+y; end.x = p(x); end.y = p(y); line = (start,end); draw_line(line);

Skapa ett nytt projekt, `rectest`, med `startup` och en tom `main`-funktion.

- Kopiera fördröjningsrutiner, grafikdisplayens drivrutiner och `draw_line` från tidigare uppgifter.
- Implementera funktionen `int draw_rect( PRECT r )` som ritar en rektangel på displayen, returnerar 1 om hela rektangeln kunde ritas, returnerar annars 0.
- Skapa nu ett huvudprogram enligt följande testprogram som ritar fem olika rektanglar på displayen, det ska se ut som bilden till höger.



```
RECT rectangles[]={
{10,10, 20,10},
{25,25, 10,20},
{40,30, 70,20},
{60,35, 10,10},
{70,10, 5,5},
 };
```

```
void main(void)
{
  graphic_initalize();
  graphic_clear_screen();

  while( 1 )
  {
    for( int i = 0; i< sizeof(rectangles)/sizeof(
RECT ); i++)
    {
      draw_rect( &rectangles[i] );
      delay_milli( 500 );
       }
    graphic_clear_screen();
  }
}
```

En rektangel består alltså av fyra punkter sammanbundna med linjer mellan vilka vinklarna alltid är 90°. En *pentagon*, har i stället fem sammanbundna linjer, om det är en *regelbunden* (*reguljär*) pentagon är dessutom alla linjer lika långa. Samma resonemang ger oss definitioner av *hexagon* (sex linjer) *septagon* (sju linjer) *oktagon* (åtta linjer), osv.

En *polygon*, dvs. *månghörning*, är en geometri som består av godtyckligt antal linjer sammanbundna av polygonens punkter. Vi skiljer vanligtvis mellan olika polygontyper:

- Hos en regelbunden (reguljär) polygon har alla linjer samma längd.
- En enkel konkav polygon innesluter en sammanhängande yta.
- En komplex polygon kan innesluta flera disjunkta ytor.

Vi ska nu konstruera en ritfunktion som plottar en godtycklig polygon (oavsett typ) på vår display. Det enda krav vi ställer är att polygonens samtliga punkter ryms inom displayen.

Eftersom en polygon har en mängd punkter som inte är statiskt bestämt vill vi här använda en typ som inte är bestämd av geometrins storlek. Vi väljer därför att skapa en *länkad lista* av punkter, som tillsammans beskriver linjedragningen för polygonen.

Vi utgår från vår tidigare definition av en punkt men lägger till en *länk* `next` till ytterligare en punkt:

```
typedef struct polygonpoint
{
  char x,y;
  struct polygonpoint *next;
} POLYPOINT, *PPOLYPOINT;
```

Vi kan då beskriva en polygon med ett antal punkter där `next` fältet är en pekare till nästa punkt eller 0 om detta är polygonens sista punkt. För att sluta polygonen låter vi den sista punkten sammanfalla med den första. Det behövs därför n+1 punkter för att beskriva en n-hörning.

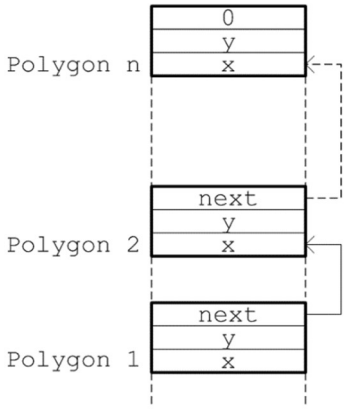
EXEMPEL 5.4 BESKRIVNING AV EN POLYGON MED FEM HÖRNEN

Följande deklarationer skapar en länkad lista för en polygon med fem hörnen.

```
POLYPOINT Polygon6 {30,30, 0 }
POLYPOINT Polygon5 {10,5, Polygon6 }
POLYPOINT Polygon4 {10,15, Polygon5 }
POLYPOINT Polygon3 {15,20, Polygon4 }
POLYPOINT Polygon2 {20,30, Polygon3 }
POLYPOINT Polygon1 {30,30, Polygon2 }
```

Algoritm: draw_polygon(*address of Polygon 1*)

```
p0.x = Polygon 1.x;
p0.y = Polygon 1.y;
ptr = Polygon 1.next
while( ptr != 0)
      p1.x = *(ptr).x
      p1.y = *(ptr).y
      line=(p0,p1)
      draw_line(line)
      p0.x=p1.x;
      p0.y=p1.y;
      ptr = *(ptr).next
```



UPPGIFT 5.14 PLOT AV POLYGON

Skapa ett nytt projekt, `polytest`, med `startup` och en tom `main`-funktion.

- Kopiera fördröjningsrutiner, grafikdisplayens drivrutiner och `draw_line` från tidigare uppgifter.
- Implementera funktionen `int draw_polygon( PPOLYPOINT p )` som ritar en polygon på displayen, returnerar 1 om hela polygonen kunde ritas, returnerar annars 0.
- Skapa nu ett huvudprogram enligt följande testprogram som ritar en polygon på displayen, det ska se ut som bilden till höger.

```
POLYPOINT pg8 = { 20,20, 0};
POLYPOINT pg7 = { 20,55, &pg8};
POLYPOINT pg6 = { 70,60, &pg7};
POLYPOINT pg5 = { 80,35, &pg6};
POLYPOINT pg4 = { 100,25, &pg5};
POLYPOINT pg3 = { 90,10, &pg4};
POLYPOINT pg2 = { 40,10, &pg3};
POLYPOINT pg1 = { 20,20, &pg2};
```

```
void main(void)
{
  graphic_initalize();
  graphic_clear_screen();

  while( 1 )
  {
    draw_polygon( &pg1 );
    delay_milli( 500 );
    graphic_clear_screen();
  }
}
```

▪▪▪

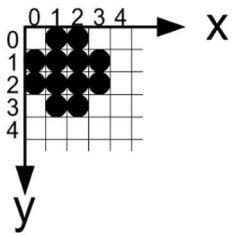
RÖRLIGA OBJEKT

Vi ska nu se hur man kan skapa objekt och få dem att röra sig över displayen. Objektet karakteristeras av statiska egenskaper, som exempelvis dess geometri och en rad dynamiska egenskaper som exempelvis position, rörelseriktning och hastighet.

Vi börjar med en definition av objektets utseende, `GEOMETRY`, som helt enkelt definierar objektet i form av pixels uttryckta i x och y-kordinater i ett logiskt koordinatsystem med position i origo. Objektet förutsätts här bestå av maximalt 30 pixlar. posten `numpoints` anger hur många av dessa som används för det aktuella objektet och posterna `sizex` respektive `sizey` anger objektets maximala utsträckning i x- respektive y-led.

```
#define MAX_POINTS 30
typedef struct
{
  int   numpoints;
  int   sizex;
  int   sizey;
  POINT px[ MAX_POINTS ];
} GEOMETRY, *PGEOMETRY;
```

EXEMPEL 5.5



Vi bygger upp ett objekt i form av en liten "boll" på följande sätt:

En vektor med x/y-par som innehåller alla objektets fyllda pixlar blir:

`(0,1)(0,2)(1,0)(1,1)(1,2)(1,3)(2,0)(2,1)(2,2)(2,3)(3,1)(3,2)`

Totala antalet fyllda pixlar är 12 st.

Objektets utsträckning, såväl i x-led som i y-led är 4 pixlar.

En instans av en sådan geometri kan då deklareras som:

```
GEOMETRY ball_geometry =
{
 12,       /* numpoints */
 4,4,      /* sizex,sizey */
 {
    /* px[0,1,2 ...] */
 {0,1},{0,2},{1,0},{1,1},{1,2},{1,3},{2,0},{2,1},{2,2},{2,3},{3,1},{3,2}
 }
};
```

Vi "kapslar in" objektet `GEOMETRY` i en större struktur `OBJECT`, där vi även lägger till tillståndsvariabler som objektets aktuella position, dess rörelseriktning och pekare till de funktioner som kan användas med objektet. Ett komplett objekt kan då definieras enligt:

```
typedef struct tObj{
   PGEOMETRY geo;
   int dirx,diry;
   int posx,posy;
   void (* draw ) (struct tObj *);
   void (* clear ) (struct tObj *);
   void (* move ) (struct tObj *);
   void (* set_speed ) (struct tObj *, int, int);
} OBJECT, *POBJECT;
```

Vi har här nöjt oss med fyra olika operationer på objektet:

`draw` – rita upp objektet i position posx,posy
`clear` – radera objektet
`move` – flytta objektet, detta är som regel den mest omfattande operationen
`set_speed` – sätt riktningsderivator för objektets rörelse.

De funktioner som opererar på objektet har här lagts som funktionspekare, i själva objektet. Detta gör det enklare om man sedan exempelvis vill använda olika *draw*-funktioner för olika objekt utan att tvingas till genomgripande ändringar i kod som använder objekten.

OBJEKTETS FUNKTIONER

Funktionen `void draw_ballobject(POBJECT o)` ritar bollen på aktuell position, vi använder funktionen `graphic_pixel_set` från tidigare:

Algoritm draw_ballobject(object):
För varje pixel 'p' i objektet:
graphic_pixel_set(object->aktuell x-position , object->aktuell y-position)

Funktionen `void clear_ballobject(POBJECT o)` raderar objektet på aktuell position, i princip samma som `draw_ballobject` men här raderas pixeln:

Algoritm clear_ballobject(object):
För varje pixel 'p' i objektet:
graphic_pixel_clear(object->aktuell x-position , object->aktuell y-position)

Den mest omfattande funktionen för vårt objekt, att objektet på displayen, är funktionen `void move_ballobject(POBJECT o )`. Här måste vi också specificera hur objektet beter sig då det kommer intill displayens sidor. I detta fall ska vårt objekt "studsa" mot displayens kanter. Objektets rörelse kan då illustreras enligt följande algoritm:

Algoritm: move_ballobject (object)
clear_ballobject(object);
Bestäm ny position genom att addera riktningskoordinater till aktuell position;
Om ny x-position är mindre än 1: (Betyder att objektet är på väg över kanten åt vänster)
skifta x-riktning 180 grader
Om ny x-position och objektets utsträckning i x-led är större än 128:
(Betyder att objektet är på väg över kanten åt höger)
skifta x-riktning 180 grader
Om ny y-position är mindre än 1: (Betyder att objektet är på väg över displayens övre kant)
skifta y-riktning 180 grader
Om ny y-position och objektets utsträckning i y-led är större än 64:
(Betyder att objektet är på väg över displayens nedre kant)
skifta y-riktning 180 grader
draw_ballobject(object);

Slutligen, `void set_ballobject_speed(POBJECT o, int speedx, int speedy)` sätter objektets rörelseriktning i x- och y-led.

EXEMPEL 5.6

Vi kan skapa en instans `ballobject` av ett objekt, i form av en boll som rör sig över skärmen och studsar mot kanterna:

```
static OBJECT    ballobject =
{
   &ball_geometry,     /* geometri för en boll */
   0,0,                /* initiala riktningskoordinater */
   1,1,                /* initial startposition */
   draw_ballobject,
   clear_ballobject,
   move_ballobject,
   set_ballobject_speed
};
```

UPPGIFT 5.15 MOVEPONG

- Skapa ett nytt projekt, `movepong`, med `startup` och en tom *main*-funktion och kopiera fördröjningsrutiner och grafikdisplayens drivrutiner från uppgift 5.10 till den nya c-filen.
- Implementera de tre typerna `POINT, GEOMETRY och OBJECT.`
- Implementera funktionerna:

```
void draw_ballobject(POBJECT);
void clear_ballobject(POBJECT)
void move_ballobject(POBJECT);
void set_ballobject_speed(POBJECT, int, int);
```

Skapa en instans (variabel) `ball` av ett objekt `balloject.`

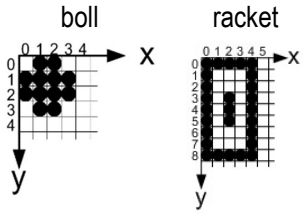
- Lägg till tangentbordsrutin från uppgift 4.2 så att du kan styra bollens rörelse över skärmen. Här behövs då `init_app` för att initiera D-porten för användning med tangentbordet.
- Använd följande program för att testa applikationen:

```
int main(void)
{
   char  c;
   POBJECT p = &ball;
   init_app();
   graphic_initalize();
   graphic_clear_screen ();
   while( 1 )
   {
      p->move( p );
      delay_milli(20);
      c = keyb();
      switch( c )
      {
         case 6:   p->set_speed( p, 3, 0); break;
         case 4:   p->set_speed( p, -3, 0); break;
         case 5:   p->set_speed( p, 0, 0); break;
         case 2:   p->set_speed( p, 0, -3); break;
         case 8:   p->set_speed( p, 0, 3); break;
      }
   }
}
```

UPPGIFT 5.16 SINGELPONG

En autonom boll rör sig över skärmen, den kan inte påverkas av spelare, riktningskoordinater är (4,1). Bollen studsar mot tre av displayens kanter. Ett racket (`PADDLE`) används för att förhindra att bollen försvinner över displayens högra kant.

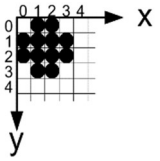
- Utgå från uppgift 5.15, skapa ett nytt projekt, `singlepong.`
- Lägg till ett object `PADDLE`, enligt figuren till höger.
- Skapa en uppsättning funktioner för `PADDLE`. Objektet kan bara flyttas i vertikalled längs displayens högra kortsida.
- Nedtryckt tangent 3 flyttar PADDLE uppåt, nedtryckt tangent 9 flyttar paddeln nedåt. Nedtryckt tangent 6 startar nytt spel.
- Experimentera med bollens hastighet och rackets storlek så att lagom svårighetsnivå uppnås.



UPPGIFT 5.17 SPINDELJAKT

I denna uppgift förekommer två objekt:

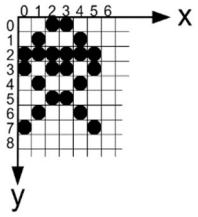
- En autonom boll rör sig över skärmen den kan inte påverkas av spelare, riktningskoordinater är (4,1).



- En "spindel" med geometri enligt figuren till höger. Spindeln kan manövreras horisontellt och vertikalt via tangentbordet, dess rörelsehastighet får vara max 2.

  Om spindeln manövreras så att den hamnar utanför skärmen (helt eller delvis) är spelet slut och spelaren har förlorat.

  Om spindeln manövreras så att bollen fångas (objekten överlappar varandra) är spelet slut och spelaren har vunnit.



- Utgå från uppgift 5.15, lägg till en ny rörelsefunktion för spindeln.
- Skapa en funktion som detekterar om objekten är i kontakt med varandra:

```
int objects_overlap( POBJECT o1, POBJECT o2 );
/* om object o1 överlappar object o2 returnera 1, annars 0 */
```

- Följande testprogram kan användas för att verifiera de nya funktionerna

```
int main(void)
{
    char   c;
    POBJECT victim = &ball;
    POBJECT creature = &spider;
    init_app();
    graphic_initalize();
    graphic_clearScreen();
    victim->set_speed( victim, 4, 1);

    while( 1 )
    {
        victim->move( victim );
        creature->move( creature );
        c = keyb();
        switch( c )
        {
            case 6: creature->set_speed( creature, 2, 0); break;
            case 4: creature->set_speed( creature, -2, 0); break;
            case 5: creature->set_speed( creature, 0, 0); break;
            case 2: creature->set_speed( creature, 0, -2); break;
            case 8: creature->set_speed( creature, 0, 2); break;
             default:
                 creature->set_speed( creature, 0, 0); break;
        }
         if( objects_overlap( victim, creature ))
         {
             // Game over
             break;
         }
        delay_milli(20);
    }
}
```

6.3 Räknarkretsar

Utöver systemkomponenten SysTick finns det en rad ytterligare räknarkretsar med olika egenskaper relaterade till hantering av tid. Detta är kretsar konstruerade bland annat för att:

- tillhandahålla enkla tidsbaser för nogrann tidmätning
- till att räkna pulser eller pulslängder eller mäta periodtid för vågformer hos en insignal till räknarkretsen (*input capture*)
- generera utsignaler för pulsbreddsmodulering (*PWM*)
- utlösa startsignaler för andra enheter (*output compare*)

KLASSIFICERING AV STM32-RÄKNARKRETSAR

Räknarkretsarna hos en *MD407* kan klassificeras enligt följande:

- Grundläggande enkla räknare (*basic timers, BT*)
- Räknare för allmänna ändamål (*general purpose timers, GP*)
- Avancerade räknare (*advanced timers, AT*)

De grundläggande enkla räknarna (BT) saknar I/O-kanaler för in- eller utsignaler och kan därför bara användas för att generera tidbaser.

Räknare för allmänna ändamål (GP) har I/O-kanaler för in- och utsignaler som kan konfigureras för en lång rad olika uppgifter (*input capture*, *PWM*, *output compare*).

Avancerade räknare kan också användas för allmänna ändamål men de har dessutom förmågan att generera *komplementära* PWM-signaler samt generera broms och dödtid för sådana signaler. Dessa funktioner gör dem lämpliga för applikationer relaterade till motorstyrning, växelriktare och andra kraftelektronikrelaterade tillämpningar.

MD407 har 14 räknarmoduler som betecknas TIM1, TIM2 ... TIM14. Räknarmodulerna delar inga resurser och är därför helt oberoende av varandra.

Kretstyp	Modul	Upplösning	Räknartyp	Prescaler	DMA	IO-kanaler	Komplemetär utgång	Max klockfrekvens IO	Max klockfrekvens räknare
Grundläggande	TIM6, TIM7	16-bit	Upp	Heltal mellan 1 och 65536	Ja	0	Nej	42 MHz	84MHz
Generella	TIM2, TIM5		Upp, Ned, Alternerande		Ja	4	Nej	42 MHz	84MHz
	TIM3, TIM4	16-bit	Upp, Ned, Alternerande		Ja	4	Nej	42 MHz	84MHz
	TIM9	16-bit	Upp		Nej	2	Nej	84MHz	168MHz
	TIM10, TIM11	16-bit	Upp		Nej	1	Nej	84MHz	168MHz
	TIM12	16-bit	Upp		Nej	2	Nej	42 MHz	84MHz
	TIM13, TIM4	16-bit	Upp		Nej	1	Nej	42 MHz	84MHz
Avancerade	TIM1, TIM8	16-bit	Upp, Ned, Alternerande		Ja	4	Ja	84MHz	168MHz

TABELL 6.3 ÖVERSIKT AV RÄKNARKRETSAR I MD407

GRUNDLÄGGANDE ENKLA RÄKNARKRETSAR

Dessa är TIM6 och TIM7, och dom används typiskt för att generera en tidsbas för periodiska avbrott. Efter att ett räknarintervall har slutförts genereras en *uppdateringshändelse* (*update event*). Räknarkretsarna kan också fås att generera en startsignal för så kallad *direktminnesöverföring* (*Direct Memory Access*, *DMA*) men detta behandlar vi inte här. Räknarkretsarna används med åtta olika register:

- Styrregister, TIMx_CR1 och TIMx_CR2, för att bestämma räknarens funktionssätt.
- Styrregister TIMx_DIER för att kontrollera avbrottsfunktionerna.
- Statusregister TIMx_SR, med information om en *update*-händelse har skett.
- Styrregister TIMx_EGR används för att programmässigt aktivera *update*-funktionen.
- Register TIMx_CNT innehåller räknarkretsens aktuella räknarvärde
- Register TIMx_PSC, multiplikator för räknaren.
- Register TIMx_ARR, initiallt räknarvärde

TIM6: 0x4000 1000 - 0x4000 13FF

TIM7: 0x4000 1400 - 0x4000 17FF

offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0																																	TIMx_CR1
4																																	TIMx_CR2
8																																	
0xC																																	TIMx_DIER
0x10																																	TIMx_SR
0x14																																	TIMx_EGR
0x18																																	
0x1C																																	
0x20																																	
0x24																																	TIMx_CNT
0x28																																	TIMx_PSC
0x2C																																	TIMx_ARR

TIM6 och TIM7 styrregister CR1 (Control Register 1)

offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0x00									ARPE				OPM	URS	UDIS	CEN	TIMxCR1

ARPE: *Auto-reload preload enable*
0: TIMx_ARR register ändras via ett buffertregister.
1: TIMx_ARR register ändras omedelbart.
Då registret uppdateras med ett nytt värde kan detta överföras omedelbart, eller efter nästa kompletta räknarintervall är klart.

OPM: *One-pulse mode*
0: Räknaren fortsätter efter ett räknarintervall (*kontinuerligt arbetssätt*)
1: Räknaren stoppas efter ett räknarintervall genom att CEN-biten nollställs.

URS: *Update request source*
Biten kontrollerar källorna till UEV (*update event*).
0: Någon av följande händelser kan generera *update interrupt* eller DMA-begäran.
– Räknaren når max eller min-värde. (*overflow* eller *underflow*)
– UG-biten i EGR (se nedan) sätts till 1 av programvara.
– *Update generation* från en slav-räknare
1: Endast räknare *overflow/underflow* genererar *update interrupt* eller DMA-begäran.

UDIS: *Update disable*
Biten bestämmer om UEV (*update event*) kan genereras.
0: UEV är möjligt och kan genereras. Se även URS. Register ges begynnelsevärden från buffertar.
1: UEV ör avstängd, inget UE (*update event*) genereras. Skuggregister (hos buffrade register ARR och PCS) uppdateras bara då UG-biten sätts till 1.

CEN: *Counter enable*
0: Räknarkretsen är deaktiverad
1: Räknarkretsen är aktiverad

TIM6 och TIM7 styrregister CR2 (Control Register2)

offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0x04											MMS [2:0]						TIMxCR2

MMS[2:0]: *Master mode selection*

Dessa bitar avgör hur räknarkretsen ska fungera då den försätts i *master mode*. Detta arbetssätt används inte här och bitarna ska därför alltid sättas till 000.

TIM6 och TIM7 styrregister DIER (DMA/interrupt enable register)

offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0x0C								UDE								UIE	TIMxDIER

UDE: *Update DMA request enable*
0: *Update* vid DMA-begäran deaktiverad.
1: *Update* vid DMA-begäran aktiverad.

UIE: *Update interrupt enable*
0: *Update* genererar inget avbrott.
1: *Update* genererar avbrott.

TIM6 och TIM7 statusregister SR

offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0x10																UIF	TIMxSR

UIF: *Update Interrupt Flag*
0: *update event* har ej inträffat
1: *update event* har inträffat

TIM6 och TIM7 styrregister EGR (Event Generation Register)

offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0x14																UG	TIMxEGR

UG: *Update generation*
Ett program kan fås att generera en *update*-händelse. Typiskt används det för teständamål men andra tillämpningar är naturligtvis också möjliga. Biten kan sättas av programvara, återställs sedan av hårdvara.

0: Ingen åtgärd.
1: Genererar en *update*-händelse på samma sätt som om den hade genererats av den autonoma räknaren.

Register för räknarvärden

PSC och ARR ger tillsammans räknarintervallet, dvs. maximala antalet pulser som räknas vilket då ger periodtiden till ett *update event*. CNT innehåller det aktuella räknarvärdet.

TIM6 och TIM7 räknare CNT (Counter)

offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0x24								CNT[15:0]									TIMxCNT

TIMx_CNT innehåller det aktuella räknarvärdet. Registret kan bara läsas.

TIM6 och TIM7 räknare PSC (Prescaler)

offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0x28								PSC[15:0]									TIMxPSC

TIM6 och TIM7 räknare ARR (Auto Reload Register)

offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0x2C								ARR[15:0]									TIMxARR

TIMxPSC och TIMx_ARR anger räknarkretsens tidsbas, dvs. räknarintervall till *update event*. Räknarkretsarna använder samma klocka som tidsbas, antalet klockpulser under 1 sekund, dvs frekvensen för UE, bestäms av:

$$UE\ Hz = \frac{CLK\ \ Hz}{((PSC + 1)(ARR + 1))}$$

Räknarens klocka är här hälften av systemets klockfrekvens och för *MD407* innebär detta:

$$UE\ Hz = \frac{84\ \ MHz}{((PSC + 1)(ARR + 1))}$$

Vi ska längre fram, bland annat i exempel 6.10 och exempel 6.11 hur detta används.

EXEMPEL 6.6 GENERERING AV SLUMPTAL

Genom att starta en *free-running timer*, kan man senare genom att läsa av räknarvärdet vid någon godtycklig tidpunkt skapa ett slumptal (i någon mening). Följande kod visar hur TIM6 kan användas för en sådan funktion.

```
/* Timer 6 */
#define      TIM6_CR1  ((volatile unsigned short *) 0x40001000)
#define      TIM6_DIER    ((volatile unsigned short *) 0x4000100C)
#define      TIM6_CNT  ((volatile unsigned short *) 0x40001024)
#define      TIM6_ARR  ((volatile unsigned short *) 0x4000102C)
#define      UDIS   (1<<1)
#define      CEN    (1<<0)

void timer6_init( )
{
   *TIM6_CR1 &= ~CEN;  /* Stoppa räknarmodul */
   *TIM6_DIER |= UDIS; /* Vi behöver inget ”update”-event.. */
   *TIM6_ARR = 0xFFFF; /* Sätt räknarregister till maxvärde */
   *TIM6_CR1 |= CEN;   /* Starta räknarmodul */
}
void main(void)
{
   short random = 0;
   timer6_init();
   while( 1 )
   {
      random = *TIM6_CNT;
      ....
   }
}
```

UPPGIFT 6.4 UNDERSÖK SLUMPTALSGENERERING

- Skapa ett projekt `random_number`, och redigera källtexten enligt exempel 6.6.
- Lägg till en funktion `void gpio_init(void)` som sätter up portD bit 0-7 som en utport.

Koppla en Double Hexadecimal Display till GPIOD pin 0-7 och prova räknaren med följande testprogram:

```
void main(void)
{
   char random = 0;
   gpio_init();
   timer6_init();
   while( 1 )
   {
      random = (char) *TIM6_CNT;
      *GPIO_D_ODRLOW = random;
   }
}
```

- Stega genom programmet sats för sats. Ange de 5 första slumptalen:
- Ändra nu satserna i huvudprogrammet enligt följande:

```
*GPIO_D_ODRLOW = random;
random = (char) *TIM6_CNT;
```

Vilken slumptalsföljd får du nu?

Räknarkretsen kan också användas för att skapa en fast tidsbas. Olika tidmätningsfunktioner kan sedan använda tidsbasen för hantering av *verklig* tid, det som också kallas *realtid*. Här måste vi då använda avbrottsmekanismen hos räknarkretsen. Användning av avbrott från periferikretsar kontrolleras via modulen NVIC (*Nested Vectored Interrupt Controller*), vi beskriver därför nu först denna modul och hur den används.

ÖVERSIKT NVIC

Upp till 81 olika avbrott hanteras av NVIC. Dessa avbrott kan ges en prioritet från 0 t.o.m 15, där 0 är den högsta prioriteten. För varje prioritet finns också ett subprioritetsfält, vilket gör att man kan skapa prioritetsgrupper där avbrotten i sin tur har en inbördes prioritet. Följande tabell beskriver samtliga register i NVIC-modulen.

offset	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	mnemonic
0x000	SETENA[31:0] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ISER0
0x004	SETENA[63:32] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ISER1
0x008	Reserverat | SETENA[80:64] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ISER2
0x080	CLRENA[31:0] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_CER0
0x084	CLRENA [63:32] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ CER1
0x088	Reserverat | CLRENA [80:64] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ CER2
0x100	SETPEND[31:0] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ ISPR0
0x104	SETPEND [63:32] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ ISPR1
0x108	Reserverat | SETPEND [80:64] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ ISPR2
0x180	CLRPEND[31:0] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ICPR0
0x184	CLRPEND [63:32] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ICPR1
0x188	Reserverat | CLRPEND [80:64] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ ICPR2
0x200	ACTIVE[31:0] r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r	NVIC_ IABR0
0x204	ACTIVE [63:32] r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r	NVIC_IABR1
0x208	Reserverat | ACTIVE [80:64] r r r r r r r r r r r r r r r r	NVIC_IABR2
0x300	IP[3] | IP[2] | IP[1] | IP[0] rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw rw	NVIC_ IPR0
...	...	...
0x320	Reserverat | IP[80] rw rw rw rw rw rw rw rw	NVIC_IPR20

Varje avbrott kan hanteras individuellt och totalt åtgår alltså 4 register för att implementera varje function i NVIC. Den bit i NVIC-registren som kontrollerar respektive avbrott bestäms av avbrottets nummer i vektortabellen. För varje avbrott kontrollerat av NVIC finns alltså följande funktioner:

- *Interrupt Set Enable Registers*, NVIC_ISERx
- *Interrupt Clear Enable Registers*, NVIC_ICERx
- *Interrupt Set Pendig Registers*, NVIC_ISEPRx
- *Interrupt Clear Pending Registers*, NVIC_ICPRx
- *Interrupt Active Bit Registers*, NVIC_IABRx
- *Interrupt Priority Registers*, NVIC_IPRx
- *Software Interrupt Trigger Register*, NVIC_STIR

Interrupt set-enable bit: NVIC_ISERx:
Skrivning:
0: ingen effekt/ 1: möjliggör avbrott
Läsning:
0: avbrott avstängt/ 1: avbrott möjligt

Interrupt clear-enable bit: NVIC_ICERx
Skrivning:
0: ingen effekt/ 1: omöjliggör avbrott
Läsning:
0: avbrott avstängt/ 1: avbrott möjligt

Interrupt set pending bit: NVIC_ISEPRx
Skrivning:
0: ingen effekt/ 1: ändrar status till ”avvaktande”
Läsning:
0: avbrottstatus är inte ”avvaktande”
1: avbrottstatus är ”avvaktande”

Interrupt clear pending bit: NVIC_ICPRx:
Skrivning:
0: ingen effekt/ 1:avlägsnar avbrottstatus”avvaktande”
Läsning:
0: avbrottstatus ej ”avvaktande”
1: avbrottstatus”avvaktande”

Interrupt active bit: NVIC_IABRx:
Läsning:
0: avbrottstatus ej ”aktivt”/ 1: avbrottstatus är ”aktivt”

Interrupt priority : NVIC_IPRx:

Varje prioritetsfält kan ha ett prioritetsvärde , 0-255. Ju lägre värde, desto högre prioritet för motsvarande avbrott. Processorn implementerar bara bitar [7: 4] för varje fält, bitar [3: 0] läses som noll och ignoreras vid skrivning.

Den bit som kontrollerar respektive avbrottsnummer (IRQ) bestäms enligt:

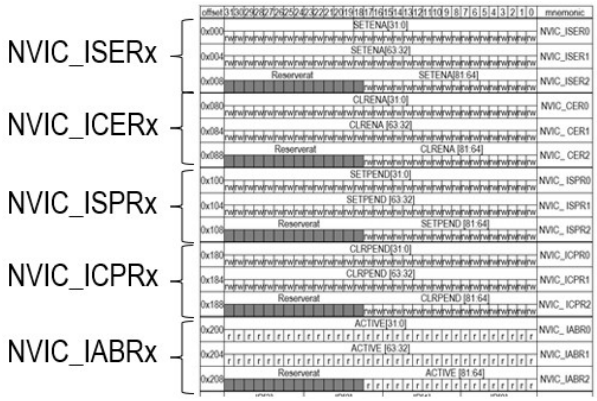
För vektor IRQ=n (n = 0..81):

`x` = n/32

dvs. `x` kan vara 0,1,2

`bit` = n – (32*x),

dvs. `bit` kan vara 0..31



EXEMPEL 6.7 BESTÄM NVIC-INITIERING OCH AVBROTTSVEKTOR FÖR TIM6

Vi letar upp TIM6 i vektortabellen (kolumn 3). Den första kolumnen anger avbrottets nummer i NVIC, dvs. 54. Kolumnen längst till höger anger avbrottsvektorns offset i vektortabellen:

54	settable	TIM6_DAC	TIM6 global interrupt, DAC1 and DAC2 underrun error interrupts	0x0000 0118

I detta fall identifierar vi alltså TIM6 global interrupt som avbrott nummer 54 och får:
`x` = 54/32 = 1 och `bit` = 54 -32 = 22. Dvs: bit 22 i register NVIC_ xxxx1
Bitmasken för denna bit är 0x00400000,

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

vilket också kan skrivas (1<<22), i NVIC-register 2. Vi kan då skapa definitioner av såväl adresser till den bit som ska sättas i NVIC:s *Set Enable* register:

```
#define NVIC_TIM6_IRQ_BPOS  (1<<22)
#define NVIC_TIM6_ISER      ((volatile unsigned int *) 0xE000E104)
```

Aktivering av avbrott från TIM6, dvs. *Set Enable*, görs nu enligt:

```
*NVIC_TIM6_ISER |= NVIC_TIM6_IRQ_BPOS;
```

Om vi senare vill deaktivera, dvs. stänga av avbrott från TIM6 görs detta genom att sätta motsvarande bit i *Clear Enable*-registret, för detta gör vi ytterligare definitionen:

```
#define NVIC_TIM6_ICER      ((volatile unsigned int *) 0xE000E184)
```

Deaktivering av avbrott från TIM6, dvs. *Clear Enable*, görs enligt:

```
*NVIC_TIM6_ICER |= NVIC_TIM6_IRQ_BPOS;
```

Vi kan fortsätta med att definiera ytterligare NVIC-register för TIM6:

```
#define NVIC_TIM6_ISPR      ((volatile unsigned int *) 0xE000E204)
#define NVIC_TIM6_ICPR      ((volatile unsigned int *) 0xE000E284)
#define NVIC_TIM6_IABR      ((volatile unsigned int *) 0xE000E304)
```

vi återkommer längre fram till hur dessa registren är avsedda att användas.

Avbrottsvektorns placering i vektortabellen är en offset. Till denna adress läggs innehållet i registret *Vector Table Offset Register* (VTOR). Vid RESET är innehållet i VTOR 0, och minnesområdet från adress 0 är i sin tur speglat av det interna FLASH-minnet. I *MD407* har monitor-programmet som tidigare sagts flyttat vektortabellen till adress 0x2001C000, dvs. läs- och skrivbart minne, vilket innebär att avbrotts-vektorer kan ändras av ett program. Avbrottsvektorn för TIM6 är därför placerad på adress 0x2001C118.

EXEMPEL 6.8 SKAPA EN TIDBAS

En tidbas används för att antingen skapa en realtidsfördröjning eller en periodicitet för *update event*. Tidbasen bestäms genom att registren TIMx_PSC och TIMx_ARR ges värden och tidbasens frekvens bestäms av:

$$UE\ Hz = \frac{CLK\ Hz}{((PSC + 1)(ARR + 1))}$$

där *CLK* anger den klockfrekvensen för räknarkretsen. I datablad finner man att denna ges av systemklockan/2.

Antag att vi vill skapa tidbasen 1 ms. Frekvensen (*Hz*) för periodtiden 1 ms = 10^{-3} *s* är alltså 10^3. Systemklockan hos *MD407* är 168MHz och klockningen av TIM6 och TIM7 därför: 84 MHz. Om vi nu därför väljer PSC=83 så kommer valet av ARR att bestämma antal mikrosekunder i fördröjningen. 1 ms är 1000 mikrosekunder varför vi får ARR = 999. Dvs.

$$UE\ Hz = \frac{CLK\ Hz}{((PSC+1)(ARR+1))} = \frac{84\ 10^6 Hz}{((83+1)(999+1))} = \frac{84\ 10^6 Hz}{84\ 10^3} = 10^3 Hz$$

EXEMPEL 6.9 ”BLOCKERANDE” FÖRDRÖJNING

Vi vill skapa en blockerande fördröjning om 1 ms med TIM6. Vi kan använda samma teknik som med SysTick i kapitel 4:

```
*TIM6_CR1 &= ~CEN;        /* Deaktivera räknaren */
*TIM6_PSC= 83;            /* Sätt tidbas till 1 ms */
*TIM6_ARR = 999;
*TIM6_DIER |= UIE;        /* Tillåt UE */
*TIM6_CR1 |= CEN;         /* Aktivera räknaren */
while(( *TIM6_SR & UIF ) == 0 );    /* Vänta tills UE */
*TIM6_CR1 &= ~CEN;        /* Deaktivera räknaren */
```

EXEMPEL 6.10 ”ICKE-BLOCKERANDE” FÖRDRÖJNING

Vi vill skapa en icke-blockerande variabel fördröjning i steg om 1 ms. Jämför med uppgift 6.3 Systick med avbrott".

```
static volatile int   flag;
static volatile int   delay_count;

void timer6_interrupt_handler( void )
{
  *TIM6_SR &= ~UIF; /*Kvittera avbrott*/
  delay_count -- ;
  if( delay_count > 0 )
    return;
  *TIM6_CR1 &= ~CEN; /*Deaktivera TIM6 */
  flag = 1;
}
   ... Huvudprogram...
   delay( antal millisekunder );
   while(1)
   {
     if(flag )
       break;
     /* Här placeras kod som kan
        utföras under väntetiden */
   }
   /* Här finns den kod som ”väntar” på
       time-out /
```

```
void delay( unsigned int count )
{
  if( count == 0)
    return;
  delay_count = count;
  flag = 0;
  *TIM6_CR1 &= ~CEN;

  /* Sätt avbrottsvektorn */
  *TIM6_IRQVEC = timer6_interrupt_handler;
  /* Möjliggör avbrott */
  *NVIC_TIM6_ISER |= NVIC_TIM6_IRQ_BPOS;

  /* Sätt tidbas */
  *TIM6_PSC= 83;
  *TIM6_ARR = 999;

  /* Möjliggör UE */
  *TIM6_DIER |= UIE;

  /* Aktivera TIM6 */
  *TIM6_CR |= CEN | OPM;
}
```

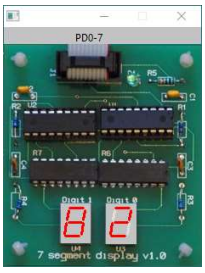
EXEMPEL 6.11 ATT MÄTA VERKLIG TID

Exemplet visar hur TIM6 kan användas för att skapa grunden till en realtidsklocka med 0,1 sekunds upplösning. Uppdateringsfrekvensen är 10 Hz, dvs periodtiden är 100ms. Vid varje klockavbrott ("tick") uppdateras den globala variabeln `ticks`. Efter 10 avbrott har det då gått 1 sekund, och vi håller reda på antalet sekunder i ytterligare en variabel seconds.

```
volatile int ticks;
volatile int seconds;
void timer6_init(void)
{
  ticks = 0;
  seconds = 0;
  *TIM6_CR1 &= ~CEN;
  *TIM6_IRQVEC = timer6_interrupt;
  *NVIC_TIM6_ISER |= NVIC_TIM6_IRQ_BPOS;

  /* 100 ms tidbas*/
 *TIM6_PSC= 839;
 *TIM6_ARR = 9999;
 *TIM6_DIER |= UIE;
 *TIM6_CR1 |= CEN;
}
```

```
void timer6_interrupt( void )
{  /* 100 ms period */
  *TIM6_SR &= ~UIF;
  ticks++;
  if( ticks > 9)
  {
    ticks = 0;
    seconds++;
  }
}
```

UPPGIFT 6.5 JUSTERA REALTIDSKLOCKA FÖR VERKLIG TID



Med denna uppgift skapar du en realtidsklocka som samtidigt justeras för att visa verklig tid även i *MD407*-simulatorn.
Problemet med simulering är att MD407:s systemklocka här blir beroende av det använda utvecklingssystemet och därav behovet av justering.

Utgå från exempel 6.11, anslut en visningsenhet till port D 0-7.
Den högra indikatorn visar sekunder och den vänstra indikatorn visar tiotals sekunder.

Följande huvudprogram kan användas för uppgiften:

```
void main(void)
{
 char nbcd;
 gpio_init();
 timer6_init();

 while( 1 )
 {
    nbcd = (seconds/10) << 4;
    nbcd |= (seconds % 10 ) & 0xF;
    *GPIO_D_ODRLOW = seconds;
 }
}
```

Avbrottsrutinen är utformat så att 10 avbrott motsvarar en sekund. Detta är också fallet då koden exekveras i en MD407. Försök att få simulatorn att komma så nära en sekund som möjligt genom att justera detta värde. Prova koden genom att mäta över ett längre tidsintervall, exempelvis 30 sekunder.