CHALMERS

Institutionen för Data och informationsteknik.
Roger Johansson
2021-12-21

Maskinorienterad programmering med MD407

Laborationsdatorn *MD407* är baserad på enchipsdatorn *ST32F407*. Tillsammans med lämplig kringutrustning för laborationer utvecklades konceptet under åren 2013-2016.

All utveckling av hårdvaror gjordes på Chalmers Tekniska Högskola, institutionen för Data och Informationsteknik, och underlag i form av källtexter och produktionsunderlag är fritt tillgängligt.

Laborationsuppgifter med simulator

Detta häfte innehåller anvisningar om hur du förbereder, genomför och redovisar en rad uppgifter som utformats så att de kan utföras enbart med utvecklingsprogram *CodeLite* och simulatorn *SimServer*. Utöver dessa krävs ingen speciell programvara. I första hand anvisar din kursansvarige varifrån du hämtar de versioner som används i kursen.

Senaste distributioner kan hämtas från `www.gbgmv.se` under fliken Studier.

Laborationsuppgifterna förutsätter att du utfört uppgifter som behandlats i läroboken *Maskinorienterad programmering med MD407* och i detta häfte ges tilläggsuppgifter till dessa.

Uppgifterna är oftast omfattande och det är därför viktigt att du börjar arbeta med dom i god tid.

Laboration	Moment
1	Parallell in- och utmatning
2	Tidmätning och synkronisering (alfanumerisk visning)
3	Introduktion till datorgrafik (grafisk visning)
4	Undantagshantering
5	Programbibliotek
6	Applikationsprogrammering

Innehåll

Laboration 1: Parallell in- och utmatning

Under laboration 1 används laborationsmodulen Keypad, för kombinerad inmatning och utmatning samt modulen 7-segmentsdisplay enbart för utmatning.

7-segmentsdisplay (Sju-sifferindikator)

Keypad (Enkelt tangentbord)

Laborationen består av fem uppgifter. Den första uppgiften beskrivs utförligt i läroboken, övriga uppgifter beskrivs här och är baserade på lösningen av den första uppgiften.

Som förberedelse för laborationerna ska du speciellt studerat exempel och gjort uppgifter i läroboken:

- Exempel 2.27 och uppgift 2.10. (Sju-sifferindikator)
- Exempel 4.1 t.o.m 4.4, uppgifter 4.1 och 4.2. (Enkelt tangentbord).
- För att kunna skapa projekt, implementera och testa dina program behöver du dessutom ha arbetat igenom kapitel 3.

Vill du, utöver detta (ej nödvändigt för att genomföra laborationen) skaffa en utförligare förståelse för laborationskortens hårdvara kan du också studera dokumenten:

- Beskrivning av 7-segmentsdisplay
- Beskrivning av Keypad

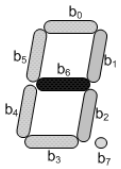
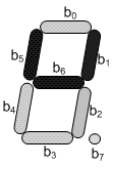
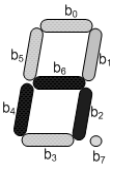
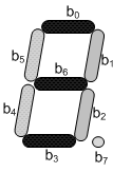
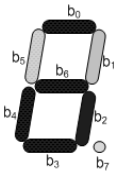
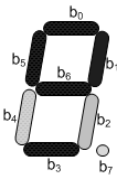
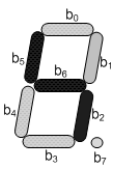
Laborationsuppgift 1.1: (Uppgift 4.2 i läroboken)

Skriv en applikation som kontinuerligt läser av tangentbordet. Om någon tangent är nedtryckt, ska dess hexadecimala tangentkod skrivas till 7-segmentsdisplayen. Om ingen tangent är nedtryckt ska displayen släckas. Applikationen ska vara komplett med `startup`, `main`, initieringsfunktion och portdefinitioner, etc.

- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 1.2:

Genom att modifiera segmentskodstabellen i funktionen `out7seg`, kan vi skapa nya ”teckenuppsättningar". Du ska nu skapa en funktion `void outEmoji( unsigned char c )` som matar ut en av följande figurer baserat på parametern `c`:

Parameter= 0	Parameter= 1	Parameter= 2	Parameter= 3	Parameter= 4	Parameter= 5	Övriga parameter-värden
						
”neutral”	”happy”	”sad”	”angry”	”loud”	”stunned”	”confused”

- Skapa ett nytt projekt `emoji`.
- Implementera den nya funktionen `void outEmoji( unsigned char c )`.
- Använd huvudprogrammet till höger för att testa funktionen:
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

```
void main(void)
{
  unsigned char c;
  init_app();
  while( 1 )
  {
    outEmoji ( keyb() );
  }
}
```

I laborationsuppgifter 1.1 och 1.2 används en enkel tangentbordsrutin som avspeglar ett ”ögonblicksvärde”. Vi kan upptäcka att en tangent är nedtryckt genom att kontinuerligt undersöka tangentbordet. Vi kan dock inte avgöra hur många gånger tangenten tryckts ned. För detta måste vi försäkra oss om att tangenten dessutom släppts upp innan nästa nedtryckning detekteras. I nästa uppgift åstadkommer vi det genom att använda en (global) tillståndsvariabel som ”kommer i håg” att en tangentnedtryckning detekterats.

Laborationsuppgift 1.3:

Utgå från uppgift 1.1. Tangentbordsrutinen ska modifieras så att varje nedtryckning detekteras exakt en gång. Den får dock inte "blockera" anropet i en vänteslinga utan måste alltid omedelbart returnera något värde Följande algoritm kan användas för uppgiften

```
Algoritm: keyb_enhanced:
returnerar kod för nedtryckt tangent en gång.
tillståndsvariabel anger: initialtillstånd eller väntetillstånd

Keyb_state = initialtillstånd;

keyb_enhanced;
        Om initialtillstånd
                Avsök tangentbord, om någon tangent nedtryckt:
                        Övergå till väntetillstånd
                        returnera tangentkod
        Om väntetillstånd
                Aktivera samtliga rader;
                Om någon tangent fortfarande nedtryckt
                        Stanna i väntetillstånd
                annars
                        Tangent har släppts upp. Återgå till initialtillstånd
                returnera 0xFF;
```

Tips: För att aktivera samtliga rader för avsökning kan exempelvis `kbdActivate` kompletteras enligt följande:

```
switch( row )
{
  ...
  case 5: *GPIO_D_ODRHIGH = 0xF0; break;
}
```

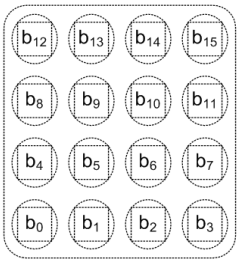
- Skapa ett nytt projekt `keyb_enhanced`.
- Implementera den nya tangentbordsrutinen `unsigned char keyb_enhanced(void)`.
- Använd följande huvudprogram för att testa funktionen:

```
void main(void)
{
  unsigned char c;
  init_app();
  while( 1 )
  {
    c = keyb_enhanced();
    if( c != 0xFF )
      out7seg( c );
  }
}
```

- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 1.4:

I vissa fall räcker det inte med att kunna detektera exakt en nedtryckt tangent. Man vill i bland kunna detektera flera samtidigt nedtryckta tangenter, exempelvis Ctrl- eller Alt- funktioner hos ett vanligt tangentbord. I denna uppgift ska du därför konstruera en tangentbordsrutin som returnerar ett statusord (16 bitar) där varje tangent har en bitposition och värdet 1 indikerar en nedtryckt tangent medan värdet 0 indikerar en uppsläppt tangent. På grund av tangentbordets konstruktion är det lämpligt att välja avbildning enligt figuren till höger mellan bitposition och tangent:



Följande gäller då för avbildning mellan *tangentkod* och bitposition:

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Kod (hex)	0A	03	02	01	0B	06	05	04	0C	09	08	07	0D	0F	00	0E

Även denna uppgift baseras i viss mån på uppgift 1.1. Tangentbordsrutinen ska utformas så att varje nedtryckt tangent detekteras och indikeras med en bit i ett statusord som returneras av rutinen. Följande algoritm ska användas:

```
Algoritm: keyb_alt_ctrl
Keyb_status = 0;
Aktivera rad 4, placera kolumnvärdena i Keyb_status( b0..b3)
Aktivera rad 3, lägg till kolumnvärdena i Keyb_status( b4..b7)
Aktivera rad 2, lägg till kolumnvärdena i Keyb_status( b8..b11)
Aktivera rad 1, lägg till kolumnvärdena i Keyb_status( b12..b14)
returnera Keyb_status
```

- Du ska konstruera `unsigned short keyb_alt_ctrl(void)` baserat på den givna algoritmen.
- Du ska nu konstruera en funktion `unsigned char is_numeric( unsigned short s )` som avgör om någon av tangenterna 0 till 9 är nedtryckt baserat på statusordet i parametern `s`.

— Om någon av de numeriska tangenterna 0..9 indikeras, returneras koden (0..9) för denna. Om flera numeriska tangenter är nedtryckta samtidigt ska den lägsta koden returneras.
— Funktionen ska ignorera icke-numeriska tangenter (tangentkoder 0x0A..0x0F).
— Om ingen numerisk tangent är nedtryckt ska 0xFF returneras.

Det är lämpligt att definiera bitmasker för tangentkod eller grupp av tangentkoder. För att avgöra om exempelvis någon av tangenterna C (”Ctrl”) eller A (”Alt”) är nedtryckta kan vi använda:

```
#define      Ctrl   0x80
#define      Alt    0x8000
```

För att filtrera ut de numeriska tangenterna är det lämpligt att på samma sätt definiera en bitmask som motsvarar denna grupp.

- Skapa ett nytt projekt `keyb_alt_ctrl`.
- Implementera funktionen `unsigned char is_numeric( unsigned short s ).`
- Använd huvudprogrammet till höger för att testa funktionen.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

```
void main(void)
{
  unsigned short s;
  unsigned char c;
  init_app();

  while( 1 )
  {
    s = keyb_alt_ctrl();
    c = is_numeric( s );
    if( c != 0xFF )
      out7seg( c );
  }
}
```

Laborationsuppgift 1.5:

Modifiera funktionerna `out7seg` och `outEmoji` från tidigare uppgifter så att dessa kan skriva ut decimalpunkten på 7-sifferindikatorn:

```
out7seg( unsigned char c, int dp );
void outEmoji( unsigned char c, int dp );
```

Om parametern `dp` är skild från 0 ska även decimalpunkten tändas.

- Skapa ett nytt projekt `decimal_point`.
- Implementera funktionen
  `unsigned char is_numeric( unsigned short s ).`
- Använd huvudprogrammet till höger för att testa funktionen.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

```
void main(void)
{
  unsigned short s;
  unsigned char c;
  int dp;

  init_app();

  while( 1 )
  {
    s = keyb_alt_ctrl();
    dp = s & Alt;
    c = is_numeric( s );
    if( c != 0xFF )
    {
      if( s & Ctrl )
        outEmoji( c, dp );
      else
        out7seg( c, dp );
    }else
      *GPIO_D_ODRLOW = 0;
    }
}
```

Laboration 2: Tidmätning och synkronisering (ASCII-display)

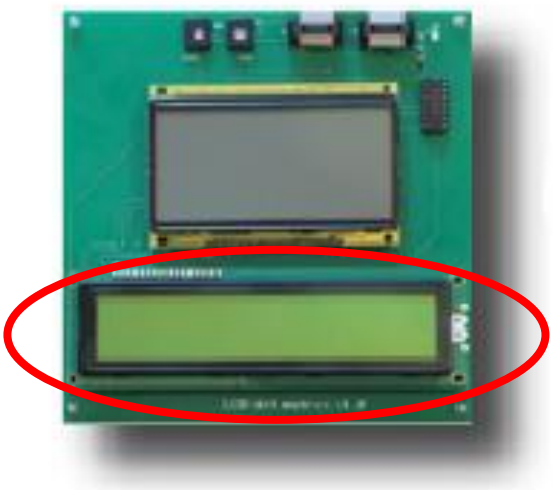
Laboration 2 består av fyra uppgifter. De båda inledande uppgifterna behandlas utförligt i läroboken där de beskrivs i form av deluppgifter som succesivt måste klaras av innan själva laborationsuppgifterna kan lösas.

För att kunna genomföra laborationen måste du ha arbetat igenom avsnitten 5.1 t.o.m 5.3 i läroboken.
Laborationsuppgifterna baseras på följande exempel och uppgifter:

- Exempel 5.1 och 5.2, uppgifter 5.1 t.o.m 5.9.

För en fördjupad förståelse kan du också studera dokumenten:

- Quick Guide: SysTick
- Datablad LCD – ASCII, sidor 12-16, 25, 28,29,33,34



Laborationsuppgift 2.1: Räknarmodul SysTick (Uppgift 5.3 i läroboken)

Uppgiften är att skapa de fördröjningsrutiner (realtid) som senare krävs för den synkroniserade kommunikationen med ACII-displayen.

- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 2.2: LCD – ASCII-display (Uppgift 5.9 i läroboken)

Uppgiften är att skapa funktioner för utmatning av text till ASCII-displayen.

- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 2.3: ASCII-display och Keypad med textredigering

I denna uppgift kombinerar vi inmatningsrutinen `keyb_enhanced` från laboration 1 med ASCII-displayen för att skapa en applikation med enkel redigeringsfunktion.

Du får här ett exempel på hur man kan skapa ett numeriskt (0..9) tangentbord med ytterligare möjligheter som att radera ett inmatat tecken eller att avsluta applikationen.

I exemplet har vi implementerat funktionen på enklast tänkbara sätt och den har också minst en allvarlig svaghet: det görs ingen kontroll om insättningspunkten är giltig, dvs. inom ramen för ASCII-displayens positioner (1..20). Utöver de initieringar som krävs ska sådan kontroll också implementeras. Ett försök till textinmatning utanför de tillåtna positionerna ska ignoreras.

Delar av ett program som hanterar den enkla textredigeringen visas till höger

Vi använder här ASCII-displayens rad 2 för textredigeringsfunktionen. En variabel `cursor_pos` anger insättningspunkten i raden.

För inmatning av tangentkoder 0-9 ska motsvarande ASCII-tecken skrivas i insättningspunkten.

Vid inmatning av tangentkod 0xB ska det sist inmatade tecknet raderas och insättningspunkten justeras.

Vid inmatning av tangentkod 0xD ska programmet avslutas.

```
cursor_pos = 1;
while( 1 )
{
  c = keyb_enhanced();
  switch( c )
  {
    case 0xD: return 0;
    case 0xB:
    cursor_pos --;
    ascii_gotoxy(cursor_pos,2);
    ascii_write_char( ' ' );
    ascii_gotoxy(cursor_pos,2);
    break;

    default:
    if( c < 10 )
    {
      ascii_write_char( c+'0');
      cursor_pos ++;
    }
  }
}
```

- Skapa ett nytt projekt `ascii_textedit`.
- Implementera huvudprogrammet med kontrollfunktion för insättningspunkten
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

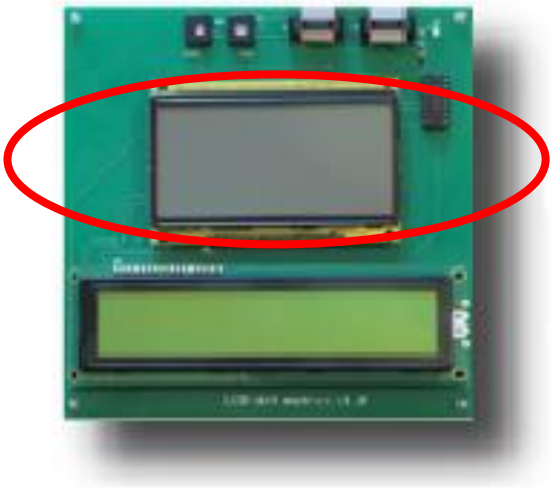
Laboration 3: Grafisk display

Laboration 3 består av tre uppgifter varav två behandlas utförligt utförligt i läroboken där de beskrivs i form av deluppgifter som succesivt måste klaras av innan själva laborationsuppgifterna kan lösas.

För att kunna genomföra laborationen måste du ha arbetat igenom avsnitt 5.4 i läroboken.

Laborationsuppgifterna baseras på följande exempel och uppgifter:

- Exempel 5.3 t.o.m 5.6, uppgifter 5.10 t.o.m 5.17.



Laborationsuppgift 3.1: Enkla geometrier (Uppgift 5.14 i läroboken)

- Uppgiften är att konstruera en funktion som ritar en polygon till grafikdisplayen.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 3.2: Spindeljakt (Uppgift 5.17 i läroboken)

Uppgiften är att skapa en liten spelapplikation (1 användare) där en ”spindel” kan manövreras med hjälp av en keypad för att fånga in en liten boll som rör sig autonomt över skärmen. Som förberedelse för denna uppgift ska du även ha gjort uppgift 5.15 i läroboken.

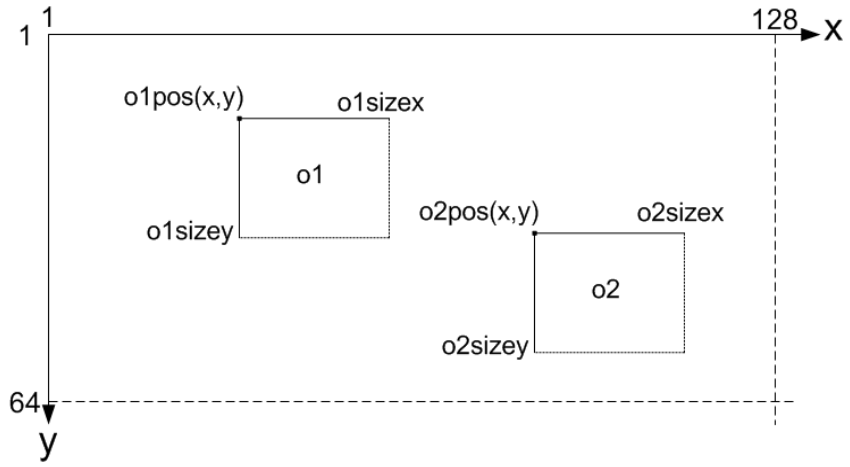
Metoder för att kontrollera överlapp mellan två objekt

Det intiutivt första försöket kan vara att jämföra de pixlar de respektive objekten ockuperar:

```
int  pixel_overlap(POBJECT o1, POBJECT o2)
{
  int offset1x = o1->posx;
  int offset1y = o1->posy;
  int offset2x = o2->posx;
  int offset2y = o2->posy;
  for( int i = 0; i < o1->geo->numpoints; i++ )
  {
    for( int j = 0; j < o2->geo->numpoints; j++ )
      if( ( offset1x+o1->geo->px[i].x == offset2x+o2->geo->px[j].x ) &&
          ( offset1y+o1->geo->px[i].y == offset2y+o2->geo->px[j].y ) )
             return 1;
  }
  return 0;
}
```

Metoden är exakt i den mening att bara ett geometriöverlapp kommer att detekteras. Om objekten överlappar krävs i värsta fall `i*j` iterationer där `i` är antalet pixlar i det första objektet och `j` är antalet pixlar i det andra objektet. För objekt som inte överlappar krävs `i*j` iterationer för att konstatera detta. Det kan bli mycket exekveringstid, speciellt som kontrollen måste utföras en gång för varje förflyttning av något objekt.

Som approximation kan man i stället jämföra objektens största utsträckningar i form av de största rektanglar som kan innesluta respektive objekts pixlar. Betrakta två objekt `o1` och `o2` med omslutande geometrier (ges i datastrukturen):



Av figuren framgår att vi med en approximerande funktion (med enkel komplexitet) snabbt kan undersöka om dessa båda rektanglar överlappar varandra genom att jämföra horisontella och vertikala gränsytor.

Detta ger en betydligt snabbare funktion än den föregående och kan lämpligen användas här. Man kan förfina genom att först använda en approximerande funktion och därefter (om så krävs) använda den exakta jämförelsen.

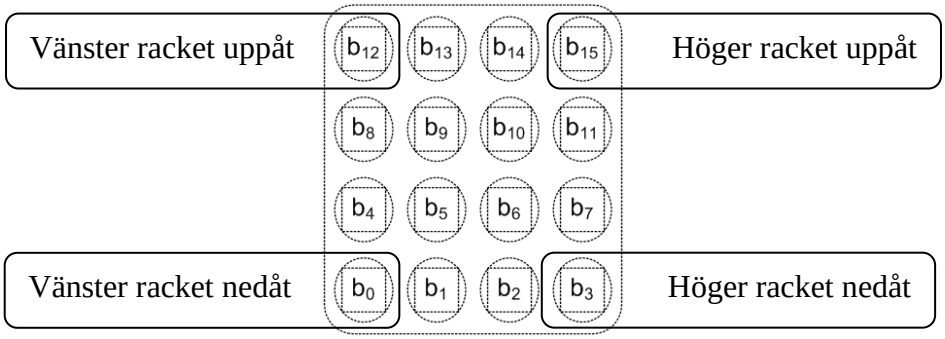
Implementera en funktion `int objects_contact( POBJECT o1, POBJECT o2)` genom att först göra en jämförelse mellan objektens omslutande rektanglar och därefter, om så krävs, en exakt jämförelse med pixel_overlap..

- Använd testprogrammet från lärobokens uppgift 5.17.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 3.3: Klassikern PONG för två spelare

I denna uppgift konstruerar du det klassiska tennisspelet PONG. Här ger vi ett minsta programskelett som löser uppgiften. Som förberedelse för denna uppgift ska du även ha gjort uppgift 5.16 i läroboken.

Det nya i denna uppgift är att vi nu använder två racketar i stället för en. Vi använder därför tangentbordsrutinen `keyb_alt_ctrl` från inlämning 1 där tangenterna nu ges följande betydelse:



Flera tangenter ignoreras alltså här men kan ges funktioner om man så önskar. Det finns möjligheter att utveckla detta senare under laborationen Applikationsprogrammering..

- Skapa ett nytt projekt `pong`.
- Modifiera rörelsefunktionerna för objekten, tänk på att bollen nu bara studsar mot de horisontella väggarna. Huvudprogrammet kontrollerar om racketen hindrar bollen från att röra sig ut över kanten.
- Använd huvudprogrammet till höger för att testa din PONG-applikation.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

```
int main()
{
  unsigned short s;
  POBJECT ball = &ball_object;
  POBJECT paddle1 = &paddle1_object;
  POBJECT paddle2 = &paddle2_object;
  init_app();
  graphic_initalize();
  graphic_clearScreen();
  ball->set_speed( ball, 4, 1);

  while( 1 )
  {
    ball->move( ball );
    paddle1->move( paddle1 );
    paddle2->move( paddle2 );
    s = keyb_alt_ctrl();

    if((s & (LEFT_UP | LEFT_DOWN))==(LEFT_UP|LEFT_DOWN))
      paddle1->set_speed( paddle1, 0, 0);
    else if (s & LEFT_UP )
      paddle1->set_speed( paddle1, 0, -4);
    else if (s & LEFT_DOWN )
      paddle1->set_speed( paddle1, 0, 4);
    else
      paddle1->set_speed( paddle1, 0, 0);

    if((s & (RIGHT_UP|RIGHT_DOWN))==(RIGHT_UP|RIGHT_DOWN))
      paddle2->set_speed( paddle2, 0, 0);
    else if (s & RIGHT_UP )
      paddle2->set_speed( paddle2, 0, -4);
    else if (s & RIGHT_DOWN )
      paddle2->set_speed( paddle2, 0, 4);
    else
      paddle2->set_speed( paddle2, 0, 0);

    if( objects_contact( ball, paddle1 ))
    {
      ball->clear(ball);
      ball->dirx = - ball->dirx;
      ball->draw(ball);
    }else if( ball->posx < (1 + ball->geo->sizex) )
    {
      break;
    }

    if( objects_contact( ball, paddle2 ))
    {
      ball->clear(ball);
      ball->dirx = - ball->dirx;
      ball->draw(ball);
    }else if( ball->posx > (128 - ball->geo->sizex) )
    {
      break;
    }
    delay_milli(40);
  }
}
```

Laboration 4: Undantagshantering

Under denna laboration utförs sex uppgifter varav fyra behandlas utförligt i läroboken där de beskrivs i form av deluppgifter som succesivt måste klaras av innan själva laborationsuppgifterna kan lösas.

För att kunna genomföra hela laborationen måste du ha arbetat igenom hela kapitel 6 i läroboken. Du kan dock dela upp det så här:

- Studera avsnitten t.o.m 6.2, speciellt exemplen och utför uppgifter t.o.m 6.3.
  Utför därefter laborationsuppgifter 4.1 och 4.2.
- Fortsätt därefter med avsnittet 6.3. Studera speciellt exemplen och utför uppgifterna 6.4 och 6.5. Utför därefter laborationsuppgift 4.3.
- Avsnitten 6.4 och 6.5 kan du sedan läsa kursivt. Dom är inte centrala för förståelsen av nästa uppgift.
- Avsnitt 6.6 och 6.7 är grundläggande för laborationsuppgifter 4.4 - 4.6. Försäkra dig om att du förstår avsnitten. Utför därefter dessa laborationsuppgifter.
- I den avslutande laborationsuppgiften 4.6 kombinerar du slutligen lösningar av tidigare uppgifter i denna laboration för att skapa ett *Stoppur*.

Laborationsuppgift 4.1: Meddelandeskickning (Uppgift 6.3 i läroboken)

- Uppgiften är att konstruera en avbrottsdriven fördröjningsrutin med *SysTick*-räknaren.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 4.2: Meddelandeskickning utan blockering

För att belysa *väntetiden*, ska du nu utgå från laborationsuppgift 4.1, och lägga till kod som illustrerar hur mycket ”att göra under tiden” kan vara. Förändringen påverkar huvudsakligen huvudprogrammets utformning:

```
    if( systick_flag )
      break;
/* Här placeras kod som kan utföras under väntetiden
Lägg till en diodramp för utmatning från port D(bit 8..16). Inför en variabel som ökas med 1
varje gång i programslingan och skrivs till den nya diodrampen.

*/
```

Utöver förändringen i huvudprogrammet krävs alltså också att hela porten initierats för utmatning.

- Skapa ett nytt projekt `systick_nonblocking`.
- Exekvera programmet och observera speciellt hur mycket som hinner utföras under ”väntetiden”.

Laborationsuppgift 4.3: Realtidsklocka (Uppgift 6.5 i läroboken)

- Uppgiften är att konstruera en simulerad realtidsklocka och justera denna mot ”verklig tid” i det värddatorsystem du använder.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 4.4: Externavbrott, en avbrottsvektor (Uppgift 6.9 i läroboken)

- Uppgiften är att konstruera en applikation med flera externa enheter som genererar avbrott och där samma avbrottsrutin används för att hantera samtliga avbrott.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 4.5: Externavbrott, flera avbrottsvektorer (Uppgift 6.10 i läroboken)

- I denna uppgift modifieras föregående applikation så att varje avbrott dirigeras till en egen avbrottsrutin.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 4.6: Stoppur

Ett *stoppur* ska konstrueras. För detta använder vi en MD407 och två typer av laborationskort: IRQ Flip Flop med en inbyggd räknarkrets och 2 st visningsenheter 7 segment display, se figur 1.

Stoppuret kontrolleras med två brytare. Överst till vänster finns *start/stopp*-funktionen. Första nedtryckningen startar klockan, nästa stoppar den, ytterligare nedtryckning startar den igen osv.

Nedanför denna brytare finns *återställningsfunktionen*. En tryckning på denna knapp återställer tiden till 00 00.

För tidmätningen används laborationskortets inbyggda räknarkrets med frekvensen 10 Hz. Stoppurets mätperiod blir därför max 9 minuter 59,9 sekunder och upplösningen 1/10-dels sekund. Tiden visas på de fyra sifferindikatorerna från vänster enligt:

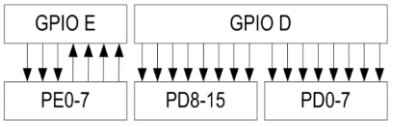
- minuter
- 10-tal sekunder
- 1-tal sekunder
- tiondels sekunder

Figur 1

Om mätningen uppnår maximal tid (95 99) ska den stannas och stå kvar i detta läge ända tills återställningsfunktionen aktiveras.
Vid en återställning nollställs tiden oavsett om stoppuret har startats eller ej.

Figur 2 visar hur laborationskorten ansluts till laborationsdatorn.

Alla *händelser* dvs. knapptryckningar och indikation av periodintervall från räknarkretsen, ska hanteras med processorns *avbrottsmekanismer*.



Figur 2

- Skapa ett nytt projekt `stopwatch`.
- Implementera stoppuret enligt specifikationen.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Anm: Tänk på att IO-simulatorns klockfrekvens är betydligt lägre än hårdvaran, 1/10 sekund motsvarar snarare 1 sekund i simulatorn. Se även klippet med demonstration av uppgiften.

Laboration 5: Programbibliotek

Laboration 5 är en större uppgift av mer admininstrativ karaktär. Du kommer att behöva tillämpa kunskaper inhämtade från många delar av kursens stoff.

Målsättningen är att du ska skapa ett programlager som gör det möjligt för en applikationsprogrammerare att använda *MD407* utan någon ingående kännedom om hårdvaran, bara skriva kod som använder standard C och de ingående programbiblioteken.

Programrutiner för IO-enheter från tidigare laborationsuppgifter ska nu placeras i ett programbibliotek för laborationsdatorn. Dessutom ska demonstrationsapplikationer användas som funktionskontroll av biblioteket.

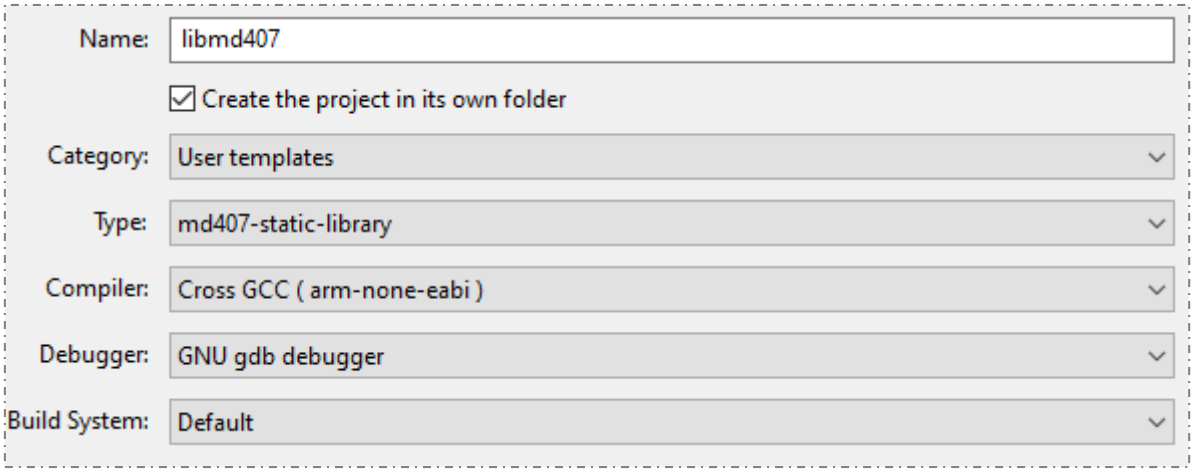
Som förberedelse för att genomföra laborationen behöver du arbeta igenom följande delar av kapitel 8 i läroboken:

- Läs översikligt avsnitten 8.1 och 8.2. Studera exemplen men du behöver inte göra dessa uppgifter (8.1-8.6).
- Studera avsnitten 8.3 och 8.4, utför uppgifterna 8.7-8.10.

För laborationsuppgiften kan du nu utgå från din "lättviktsimplementering", dvs. de maskinberoende delar av standard-C biblioteket du implementerade för MD407.

Laborationsuppgift 5.1: Implementering av `malloc/free` (Uppgift 8.7 i läroboken)

Skapa ett nytt projekt, ett programbibliotek enligt :



Två nya filer skapas i projektet (`libNAME.h` och `libNAME.c`), ändra respektive filnamn till `libmd407.h` och `libmd407`.c.

Lägg till deklarationer i `libmd407.h`:

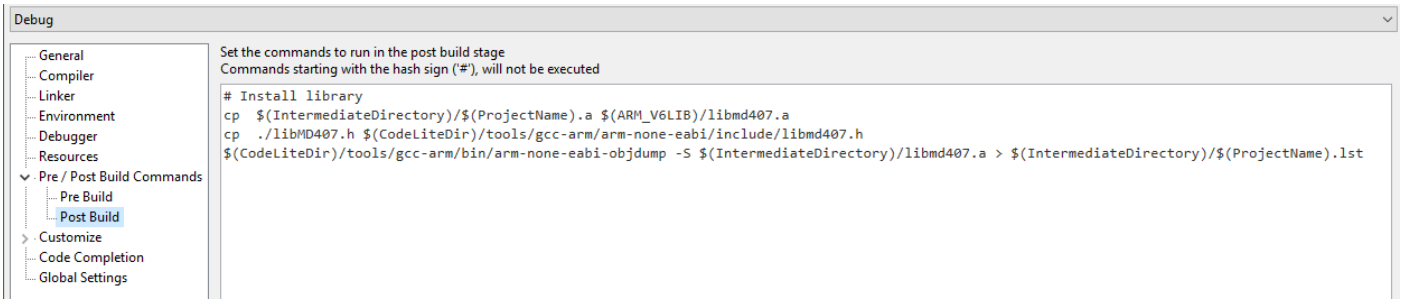
```
#ifndef _LIBMD407_H          /* Vi lägger en "include-guard" här */
#define _LIBMD407_H
/* Använd standard header filer */
#include <stdio.h>
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>
/* Konstanter som definieras av länkaren */
extern char __heap_low;
extern char __heap_top;
extern char __bss_start;
extern char __bss_end;
/* Funktioner i libmd407.c */
void crt_init( void );
void crt_deinit( void );
#endif
```

Implementera de första funktionerna i `libmd407.c`:

```
#include      <libmd407.h>

static char *heap_end;
char * _sbrk(int incr) {
  if (heap_end == 0) {
    heap_end = &__heap_low;
  }
  prev_heap_end = heap_end;
  if (heap_end + incr > &__heap_top) {
    errno = ENOMEM;
    return (char *)-1;
        }
        heap_end += incr;
        return (char *) prev_heap_end;
}

/* Attributet 'used' talar om för länkaren att funktionen inte får optimeras bort */
__attribute__ ( (used) )
volatile void _crt_init() {
  char *s;
  heap_end = 0;
  s = &__bss_start__;
  while( s < &__bss_end__ )
    *s++ = 0;
}

__attribute__ ( (used) )
volatile void _crt_deinit( int not_used ) {
}

void _exit( int a )
{
        _crt_deinit();
        __asm__ volatile(".L1: B .L1\n");
}

int _close(int file) { return -1; }
int _open(const char *name, int flags, int mode) { return -1; }
int _fstat(int file, struct stat *st) { st->st_mode = S_IFCHR; return 0; }
int _lseek(int file, int ptr, int dir) { return 0;}
int _isatty(int file) { return 0; }
int _write(int file, char *ptr, int len) { return 0; }
int _read(int file, char *ptr, int len) { return 0;}
```

Dessa funktioner kommer vi att bygga på efter hand som programbiblioteket växer. I projektets inställningar kan vi nu lägga till direktiv som installerar vårt bibliotek på rätt platser i filsystemet så att vi enkelt kan nå dom senare.

Öppna projektets inställningar och välj fliken Pre/Post Build Commands:



Följande rader läggs till här:

```
# Install library
cp  $(IntermediateDirectory)/$(ProjectName).a $(ARM_V6LIB)/libmd407.a
cp  ./libMD407.h $(CodeLiteDir)/tools/gcc-arm/arm-none-eabi/include/libmd407.h
$(CodeLiteDir)/tools/gcc-arm/bin/arm-none-eabi-objdump –S $(IntermediateDirectory)/libmd407.a >
$(IntermediateDirectory)/$(ProjectName).lst
```

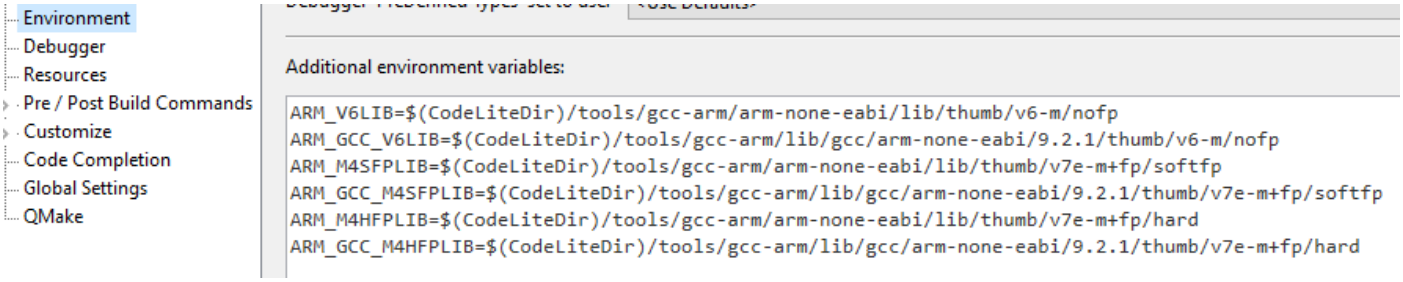
Anm: Observera att OBJDUMP-kommandot skrivs på en rad!
Anm: För Linux krävs rotbehörighet för att kopiera filerna, `cp`-kommandona ska då inte vara med här, se Appendix.

På den första raden kopieras själva programbiblioteket till samma ställe som alla andra programbibliotek för konfigurationen.

På den andra raden kopieras header-filen till platsen för övriga standard headers.

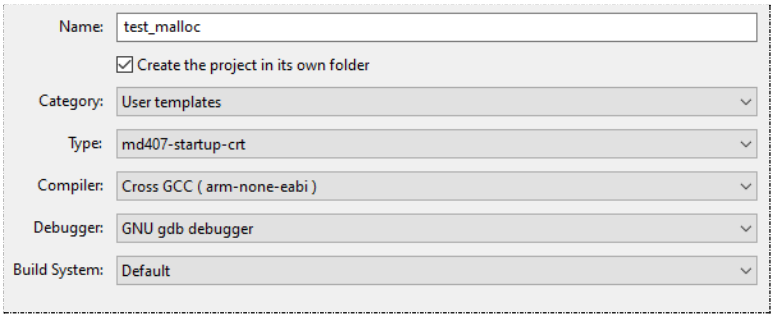
De två sista raderna här, observera att de ska skrivas in som en rad i fönstret, skapar en disassemblering av programbibliotekets kod. Den resulterande filen `libmd407.headers` skapas i projektets underbibliotek Debug.

Kontrollera också att dina miljövariabler innehåller sökvägar till programbiblioteken



- Bygg nu projektet och kontrollera att de skapade filerna hamnat på rätt ställen.

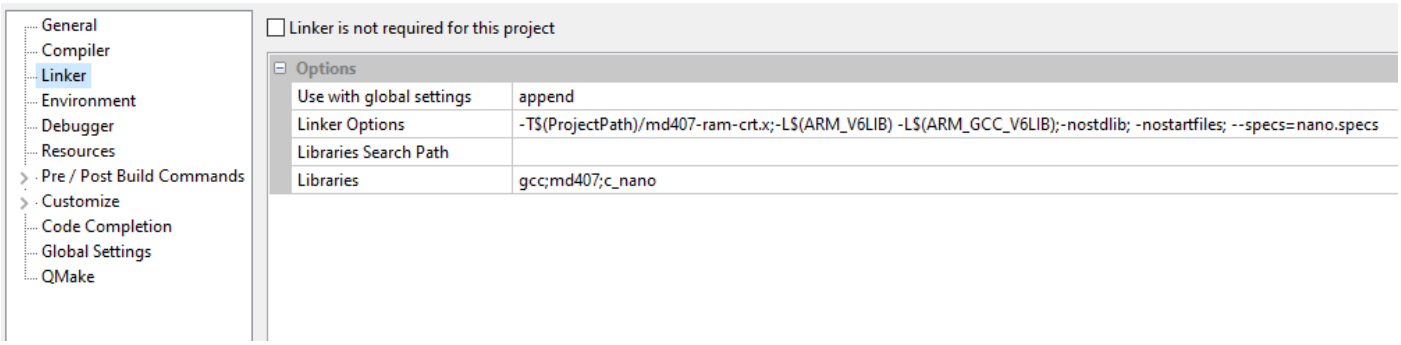
Nu skapar vi en enkel applikation `test_malloc`, för test av `malloc` och `free` funktionerna.



Filen `startup-crt.c` innehåller en modifierad "startup", komplettera med testprogram enligt följande:

```
__attribute__((naked)) __attribute__((section (".start_section")) )
void startup ( void )
{
__asm__ volatile(" LDR R0,=__stack_top\n");
__asm__ volatile(" MOV SP,R0\n");
__asm__ volatile(" BL _crt_init\n");
__asm__ volatile(" BL main\n");
__asm__ volatile(" BL _crt_deinit\n");
__asm__ volatile(" B .\n");
__asm__ volatile(" .LTORG\n");
}

#include <stdlib.h>       /* Deklaration av malloc etc. */

int main(void)
{
  char str1[]="Beginning of string";
  char str2[]="Continuation of string";
  char *s1,*s2;
  s1 = malloc( strlen(str1)+1 );
  if( s1 == 0 ) exit(-1);
  strcpy( s1, str1 );
  s2 = realloc( s1, strlen(str1)+ strlen(str2)+1 );
  if( s2 == 0 ) exit(-1);
  strcat( s2,str2 );
  free( s2 );
}
```

Kontrollera inställningar för länkningen, observera att ordningen som programbiblioteken räknas upp (Libraries) i vissa fall kan ha betydelse. Man kan ange ett bibliotek flera gånger, de genomsöks i den ordning de räknas upp.



- Implementera testprogrammet enligt specifikationen.
- Kontrollera programmets funktion med *CodeLite* och *SimServer* genom att stga genom funktionen rad för rad, kontrollera de lokala variablernas värden.

Laborationsuppgift 5.2: Implementering av drivrutin för USART

Vi skapar nu ett generellt gränssnitt mot standard-C biblioteket, detta innebär att vi måste tillhandahålla de grundläggande operationerna för varje enhet vi stödjer i vårt system (MD407 i detta fall), vi kallar det enhetens *drivrutiner*.

Utgå från kod i bokens uppgift 8,8 där det visas hur en konsollenhet (Terminal) kan läggas till.

Vi börjar med att definiera den datastruktur *device descriptor*, som innehåller all information om enhetens drivrutiner, följande typdefinition läggs till i `libmd407.h`:

```
typedef struct
{
  char name[16];
  int (*init) (int);
  void (*deinit) (int);
  int (*fstat)(struct stat *st);
  int (*isatty)(void);
  int (*open)(const char name,int flags,int mode);
  int (*close)(void);
  int (*lseek)(int ptr, int dir);
  int (*write)(char *ptr, int len);
  int (*read)(char *ptr, int len);
} DEV_DRIVER_DESC, *PDEV_DRIVER_DESC;
```

- Initiering av drivrutinen, `init`, ska göras från funktionen `_crt_init`, följande kod läggs därför till i denna funktion:

```
 ...
PDEV_DRIVER_DESC fd;
...
 for( int i = 0; i < MAX_DEVICE; i++ )
  {
    fd = device_table[i];
    if( fd && fd->init != 0)
      (void) fd->init( 0 );
  }
...
```

Konstanten `MAX_DEVICE` representerar totala antalet implementerade enheter i systemet, vi återkommer till denna.

- För symmetri, inför vi funktionen `deinit`, som utförs *efter* applikationen, det kan exempelvis vara att deaktivera avbrottsmekanismer etc. Följande kod läggs till `_crt_deinit`:

```
PDEV_DRIVER_DESC fd;
...
 for( int i = 0; i < MAX_DEVICE; i++ )
  {
    fd = device_table[i];
    if( fd && fd->deinit != 0)
      (void) fd->deinit( 0 );
  }
...
```

För de resterande funktionerna gäller att dessa ska dirigeras till sin respektive enhets drivrutin. För exempelvis funktionen `_write`, får vi då implementeringen:

```
int _write(int file, char *ptr, int len) {
  PDEV_DRIVER_DESC fd;
  if( ( file < 0) || (file >= MAX_DEVICE) )
      return 0;
  fd = device_table[file];
  if( fd && fd->write != 0)
      return fd->write(ptr,len);
  return 0;
}
```

- Färdigställ funktionen `_write` i filen `libmd407.c`
- Fortsätt därefter och färdigställ implementeringarna av resterande funktioner i gränssnittet, `_fstat`, `_isatty`, `_lseek` och `_read` enligt samma mönster. _close och _open lämnar vi utan implementering, de måste dock definieras. I vår implementering låter vi helt enkelt returnera en felkod (-1).

```
int _close(int file) {
      return -1;
}
int _open(const char *name, int flags, int mode){
      return -1;
}
```

Implementering av USART drivrutiner

Vi övergår nu till de enhetsspecifika delarna för seriekommunikationskretsen. Standarbiblioteket specificerar tre enheter som måste finnas, de tre första enheterna i ett standard C bibliotek hanterar in- och utmatning via en konsoll (Terminal).

Enhet 0 : *standard input*, från denna enhet läses tecken från terminalen.

Enhet 1: *standard output*, till denna enhet skrivs meddelanden till terminalen.

Enhet 2: *standard error output*, applikationsprogram kan utformas så att exempelvis felmeddelanden dirigeras till en separat terminal, för att lättare uppmärksammas från ett program som producerar stora mängder utdata. I vår implementering skiljer vi dock inte på enheterna 1 och 2, de implementeras med samma USART-krets.

Det är inte alltid nödvändigt att implementera *samtliga* funktioner för varje enhet. För vår seriekommunikationskrets är det till exempel ingen mening med att implementera `open`, `close` eller `lseek`, eftersom dessa inte används av systemet. Dessa enheter är alltid öppna för varje applikation.

- Lägg till en ny fil `driver_usart.c`, i projektet `libmd407.`

```
/*
 *  libMD407, driver_usart.c
 *  Drivrutiner för STDIN_FILENO, STDOUT_FILENO och STDERR_FILENO
 */
#include    "libmd407.h"

/* Deklarationer av de funktioner som ingår, krävs för tabellen */
static int usart_init( int initval );
static void usart_deinit( int deinitval);
static int usart_write(char *ptr, int len);
static int usart_read(char *ptr, int len);
static int usart_isatty(void);
```

```
/* Deklarera och initiera deskriptorer för varje enhet (0,1,2) */
```

```
DEV_DRIVER_DESC StdIn =
{
  {"stdin"},
  usart_init,
  usart_deinit,
  0,
  usart_isatty,
  0,
  0,
  0,
  0,
  usart_read
};
```

```
DEV_DRIVER_DESC StdOut =
{
  {"stdout"},
  0,
  0,
  0,
  usart_isatty,
  0,
  0,
  0,
  usart_write,
  0
};
```

```
DEV_DRIVER_DESC StdErr =
{
  {"stderr"},
  0,
  0,
  0,
  usart_isatty,
  0,
  0,
  0,
  usart_write,
  0
};
```

Anm: Ovanstående innebär att `usart_init` och `usart_deinit` bara anropas en gång för `StdIn` vilket därför är en gemensam initiering (deinitiering) för `StdOut och StdErr`.

Vi måste nu koppla dessa deskriptorer till programbiblioteket, denna koppling består av pekare till fälten av descriptorer, `device_table[]` som nu måste definieras. I `libmd407.c` lägger vi därför till följande:

```
extern DEV_DRIVER_DESC StdIn,StdOut,StdErr;
PDEV_DRIVER_DESC device_table[MAX_DEVICE] =
{
      &StdIn,
      &StdOut,
      &StdErr
};
```

Konstanten `MAX_DEVICE` repesenterar antalet enheter i vårt system. Vi har nu skapat tre sådana och därför:

- Definiera konstanten `MAX_DEVICE` som nu ska vara 3, `i libmd407.h`

Vi återgår till `driver_usart.c,` efter deklaration av deskriptorerna ska nu implementeringen slutföras och resten av filen får därför följande innehåll:

```
/* Här definierar du dina USART-funktioner, se kapitel 7 i läroboken.
 Du väljer själv vilken variant (med/utan avbrott, bufferthantering etc.)
 Det kan vara lämpligt att börja med den enklaste lösningen
*/
static int usart_init( int initval ){}
static void usart_deinit( int deinitval) {}
static int usart_write(char *ptr, int len) {}
static int usart_read(char *ptr, int len) {}
static int usart_isatty(void){}
```

- Färdigställ filen `driver_usart.c` med drivrutinerna för enheterna 0,1 och 2.
- Bygg och installera det utökade programbiblioteket.

Funktionstest av USART drivrutiner

Nu skapar vi ett nytt projekt `test_usart`, för test av `USART` funktionerna. Innehållet i filen `startup-crt.c` ersätts med vårt testprogram enligt följande:

```
__attribute__((naked)) __attribute__((section (".start_section")) )
void startup ( void )
{
__asm__ volatile(" LDR R0,=__stack_top\n");          /* set stack */
__asm__ volatile(" MOV SP,R0\n");
__asm__ volatile(" BL _crt_init\n");                 /* init C-runtime library */
__asm__ volatile(" BL main\n");                      /* call main */
__asm__ volatile(" BL _crt_deinit\n");               /* deinit C-runtime library */
__asm__ volatile(" B .\n");                          /* never return */
__asm__ volatile(" .LTORG\n");                       /* dump literals */
}

#include     <stdio.h>
#include     <unistd.h>
void echo_filehandles(void)
{      /* Läs från konsoll och skriv tillbaks, avsluta vid <CR> */
       char c;
       while(1){
              c = fgetc( stdin );
              fputc( c, stdout );
              if( c == '\n')
                     return;
       }
}
void echo_devicehandles(void)
{      /* Läs från konsoll och skriv tillbaks, avsluta vid <CR> */
       char c;
       while(1){
              read ( STDIN_FILENO, &c, 1 );
              write( STDOUT_FILENO, &c, 1 );
              if( c == '\n')
                     return;
       }
}
void output(void)
{
       printf( "Hello World 1\n"  );
       fprintf( stdout, "Hello World 2\n" );
       fwrite("Hello World 3\n", 1, 14,stdout);
       write( STDOUT_FILENO, "Hello World 4\n", strlen("Hello World 4\n" ) );
}
void main(void)
{
       output();
       echo_filehandles();
       echo_devicehandles();
       printf("Exited\n");
}
```

Kontrollera inställningar för länkningen, observera att ordningen som programbiblioteken räknas upp (Libraries) i vissa fall kan ha betydelse. I detta fall låter vi länkaren söka igenom `libc_nano` två gånger för att alla referenser ska komma med:

Anm: Skillnaden mellan testfunktionerna (`echo_`) är att vi använder filoperationer (`fputc` och `fgetc`). För filoperationer tillhandahåller standard-C biblioteket buffring, vilket gör att de inte omedelbart dyker upp på skärmen eller returneras vid anrop. Detta kan vara förvirrande och man kan stänga av denna buffring med funktionen `setvbuf`. Vill du stänga av buffring kan du därför lägga till följande rader sist i `_crt_init`.

```
setvbuf( stdin, NULL, _IONBF, 0 );
setvbuf( stdout, NULL, _IONBF, 0 );
setvbuf( stderr, NULL, _IONBF, 0 );
```

- Implementera testprogrammet enligt specifikationen.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 5.3: Implementering av drivrutin KEYPAD

- Lägg till en ny fil `driver_keypad.c`, i projektet `libmd407.`

Precis som för USART ska filen ha följande struktur:

```
#include    "libMD407.h"            /* Inkludera gemensamma definitioner */
/* Prototypdeklarationer av de ingående funktionerna */
```

Keypad kräver initiering, eventuellt deinitiering och en funktion för att läsa från enheten, deskriptorn kan därför beskrivas av:

```
DEV_DRIVER_DESC KeyPad =
{
  {"Keypad"},
  keypad_init,
  keypad_deinit,
  0,
  0,
  0,
  0,
  0,
  0,
  keypad_read
};
```

Som läsrutin är det lämpligt att använda `keyb_enhanced(void)`, från laboration 1.

- Färdigställ filen `driver_keypad.c` med drivrutinerna för `KeyPad`.
- Slutligen läggs deskriptorn till de övriga i `libmd407.c`:

```
extern DEV_DRIVER_DESC StdIn,StdOut,StdErr,KeyPad;
PDEV_DRIVER_DESC device_table[MAX_DEVICE] =
{
      &StdIn,
      &StdOut,
      &StdErr,
      &KeyPad
};
```

Vi har nu fyra enheter i vår system och därför:

- Definiera konstanten `MAX_DEVICE` som nu ska vara 4, `i libmd407.h.` Här är det också lämpligt att definiera en symbolisk konstant för enheten vilken helt enkelt är den samma som enhetens index i deskriptorn: `enum {KEYPAD_FILENO=3};`
- Bygg och installera det utökade programbiblioteket.

Funktionstest av KeyPad drivrutiner

- Skapa ett nytt projekt `test_keypad`, för test av funktionerna
- Lägg till den saknade raden i `md407-ram-crt.x`
- Kontrollera inställningarna för länkaren, de ska se ut på samma sätt som då du testade derivrutinerna för USART.
- Innehållet i filen `startup-crt.c` ersätts med testprogram enligt följande:

```
static char to_ascii( char c )
{
      if( c < 10 ) return c+'0';
      return (c-10+'A');
}

int main( void )
{
      /* Testapplikationen läser ett tecken från KeyPad
       * översätter detta till motsvarande ASCII-tecken
       * och skriver detta till en Terminal
       */
       char c;
       while(1)
       {
             read( KEYPAD_FILENO, &c, 1);
             c = to_ascii( c );
             fputc( c, stdout );
       }
}
```

- Implementera testprogrammet enligt ovan.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laborationsuppgift 5.4: Implementering av drivrutin `ASCIIDISPLAY`

- Lägg till en ny fil `driver_asciidisplay.c`, i projektet `libmd407.`

Precis som tidigare ska filen ha följande struktur:

```
#include    "libMD407.h"            /* Inkludera gemensamma definitioner */
/* Prototypdeklarationer av de ingående funktionerna */
```

ASCII-displayen kräver initiering, eventuellt deinitiering och en funktion för att skriva till enheten, deskriptorn kan därför beskrivas av:

```
DEV_DRIVER_DESC AsciiDisplay =
{
  {"AsciiDisplay"},
  asciidisplay_init,
  asciidisplay_deinit,
  0,
  0,
  0,
  0,
  0,
  asciidisplay_write,
  0
};
```

Som skrivrutin kan du börja med att implementera direkt utskrift, dvs. laboration 2.2. När detta är klart kan du gå vidare och (frivillig uppgift) även implementera textredigering för `'\b'` (*backspace*) och `'\n'` (*newline*) tecknen.

- Färdigställ filen `driver_asciidisplay.c` med drivrutinerna för ASCII-displayen.
- Slutligen läggs deskriptorn till de övriga i `libmd407.c`:

```
extern DEV_DRIVER_DESC StdIn,StdOut,StdErr,KeyPad,AsciiDisplay;
PDEV_DRIVER_DESC device_table[MAX_DEVICE] =
{
      &StdIn,
      &StdOut,
      &StdErr,
      &KeyPad
      &AsciiDisplay
};
```

Vi har nu fem enheter i vår system och därför:

- Definiera konstanten `MAX_DEVICE` som nu ska vara 5, `i libmd407.h.` Här är det också lämpligt att definiera en symbolisk konstant för enheten vilken är den samma som enhetens index i deskriptorn: `enum {KEYPAD_FILENO=3, ASCIIDISPLAY_FILENO};`
- Bygg och installera det utökade programbiblioteket.

Funktionstest av ASCII-display drivrutiner

- Skapa ett nytt projekt `test_asciidisplay`, för test av funktionerna
- Lägg till den saknade raden i `md407-ram-crt.x`
- Kontrollera inställningarna för länkaren, de ska se ut på samma sätt som då du testade drivrutinerna för USART.
- Innehållet i filen `startup-crt` ersätts med testprogram enligt följande:

```
static char to_ascii( char c )
{
      if( c < 10 ) return c+'0';
      return (c-10+'A');
}

int main( void )
{
      /* Testapplikationen läser ett ASCII tecken från en terminal
       * och skriver detta till ASCII-displayen.
       * Testprogrammet av tecknet ’\n’
       */
       char c;
       while(1)
       {
            (void) read (STDIN_FILENO, &c, 1 );
            if( c == '\n' ) break;
            write( ASCIIDISPLAY_FILENO, &c, 1 );       }
}
```

- Implementera testprogrammet enligt ovan.
- Kontrollera programmets funktion med *CodeLite* och *SimServer*.

Laboration 6: Applikationsprogrammering

Laboration 6 är en lite större uppgift. En komplett applikation ska konstrueras och demonstreras. För att lösa uppgiften behöver du återanvända kod från de tidigare laborationerna. Du kan också ta tillfället i akt att organisera källtexter på ett bra sätt och förfina kod om du tycker det är lämpligt.

Applikationen ska

- Använda räknarkretsar för tidshantering
- Använda såväl grafisk som alfanumerisk display
- Använda någon form av inmatningsenhet, *keypad* och/eller *USART*

Självfallet får fler enheter användas om du tycker det är lämpligt.

Du väljer själv vad du vill göra för applikation men ett förslag kan vara ett enkelt datorspel av ”retro”-karaktär. Du kan söka inspiration till sådana spel på Internet, några klassiska varianter är:

- Breakout
- Space invaders
- Tetris
- Snake

Se exempelvis ”www.coolaspel.se”, där kan också du prova spelen. På kursens hemsida har vi samlat några klipp av spel som konstruerats av tidigare kursdeltagare. Dessa demonstreras då i det verkliga laborationssystemet men du kan förstås hämta inspiration även från dessa!

Kopiering av programbibliotek under Linux

För Linux krävs rotbehörighet för att kopiera filer till filträdet `/usr/share`. Det är inte speciellt lämpligt att starta *CodeLite* som `root`, inte heller bör man ändra rättigheterna i filträdet. Det bästa är att använda `sudo`-kommandot. Eftersom sudo kräver inmatning (root-lösenord) kan det kan vara lämpligt att göra detta *utanför* CodeLite. Här är ett sätt att göra det:

Skapa en scriptfil `copylib` med följande innehåll, placera scriptfilen *i samma bibliotek* som projektfilen (`libmd407.project`)

```
# copylib, shell-script, kopiera programbibliotek
ARM_V6LIB=/usr/share/codelite/tools/gcc-arm/arm-none-eabi/lib/thumb/v6-m/nofp
sudo cp  ./Debug/libmd407.a ”$ARM_V6LIB”/libmd407.a
sudo cp  ./libMD407.h /usr/share/codelite/tools/gcc-arm/arm-none-eabi/include/libmd407.h
```

Observera att sökvägarna förutsätter att du tidigare installerat programmen enligt anvisningarna. Om inte, behöver du modifiera sökvägarna.

Scriptfilen ersätter nu filkopieringen i *PostBuild* steget i *CodeLite*.

För att installera (kopiera) filerna `libmd407.a` och `libmd407.h`:

- starta en terminal och cd till projektbiblioteket.
- ge kommandot:

`source copylib`

för att utföra filkopieringen.