Laboration 6: Programbibliotek

Laboration 6 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.

En viktig detalj att påpeka, i det distribuerade länkarscriptet har tyvärr en rad **`*(.bss.*)`** fallit bort. Raden är viktig eftersom den kan innehålla segment som måste initieras till noll innan applikationen startas. För varje ny applikation du skapar ska du därför lägga till denna rad i länkarscriptet enligt:

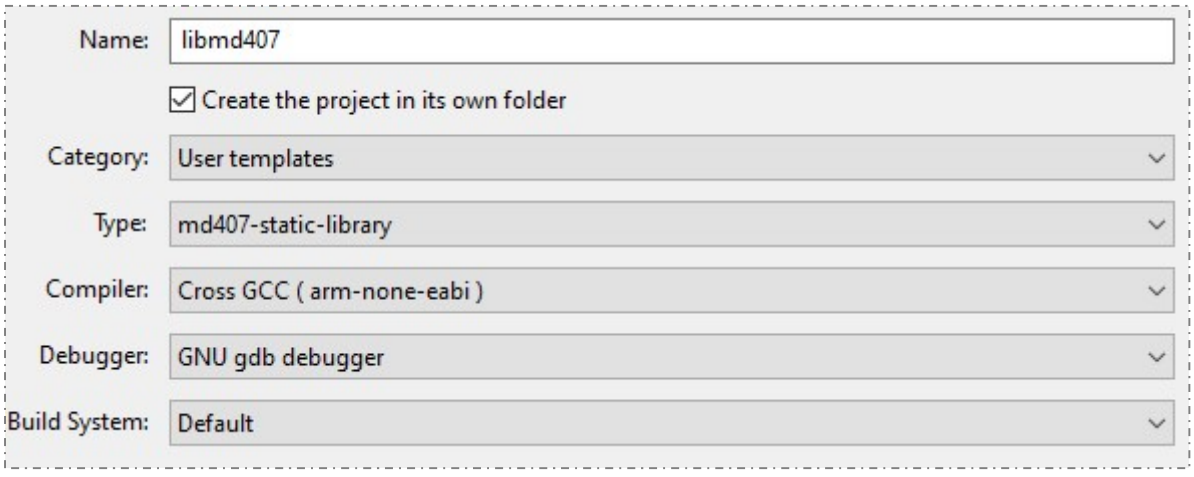
```
__bss_start__ = .;
  *(.bss)
  *(.bss.*)
  *(COMMON)
```

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

Observera att det finns smärre skillnader i filnamn och sökvägar mellan läroboken och detta PM. Använd uppgifterna i denna text för laborationerna.

Laborationsuppgift 6.1: 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 talar om för länkaren att funktionen inte får optimeras bort */
__attribute__ ( (used) )
volatile void _crt_init( int not_used) {
  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( a );
       __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 -h –S
$(IntermediateDirectory)/libmd407.a > $(IntermediateDirectory)/$(ProjectName).headers
```

Anm: Observera att OBJDUMP-kommandot skrivs på en rad!

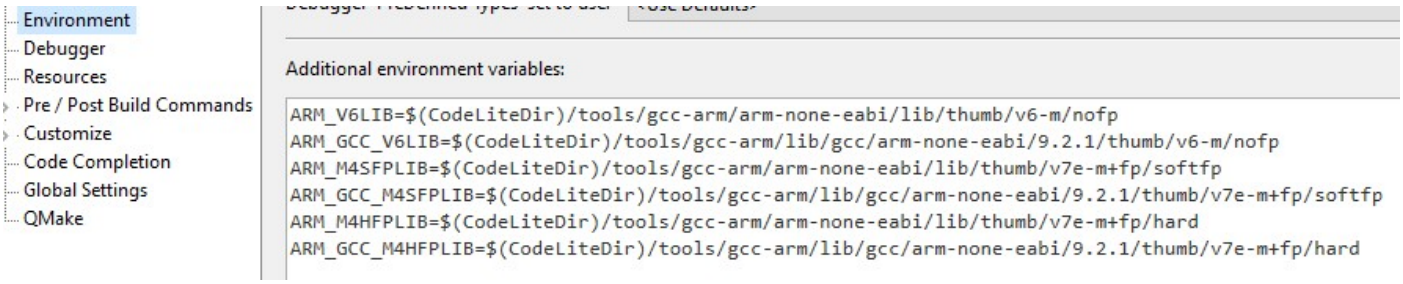
Anm: För Linux krävs rotbehörighet för att kopiera filerna, `cp`-kommandona ska då inte vara med här, se Appendix.

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.

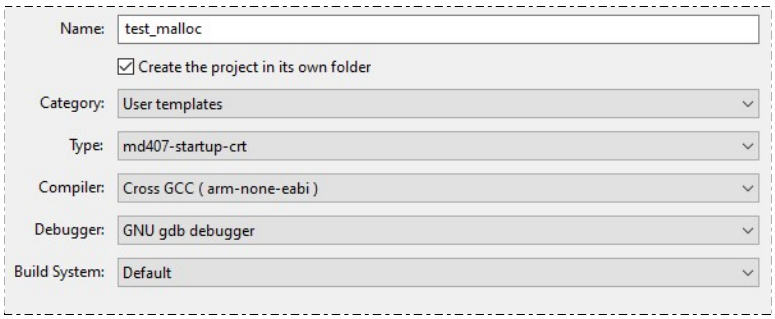
De två sista raderna här, observera att de ska skrivas in som en rad i fönstret, skapar en disassemblering av programbibliotekets kod. Den resulterande filen `libmd407.headers` skapas i projektets underbibliotek Debug.

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.



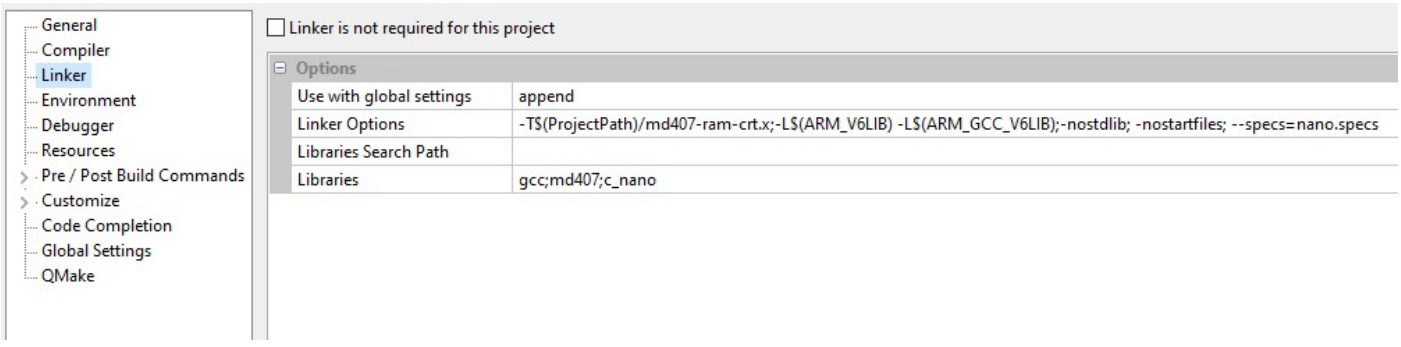
Filen `startup-crt.c` innehåller nu en mängd kod som inte längre ska finnas där, vi ”flyttar” den efter hand till programbiblioteket, ersätt därför innehållet med vårt testprogram enligt följande:

```
__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(".L1: B .L1\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 );
}
```

Lägg till den saknade raden i `md407-ram-crt.x`.

```
__bss_start__ = .;
  *(.bss)
  *(.bss.*)
  *(COMMON)
```

Kontrollera inställningar för länkningen, observera att ordningen som programbiblioteken räknas upp (Libraries) i vissa fall kan ha betydelse.



- 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 6.2: 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.

```
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(".L1: B .L1\n");                       /* never return */
}

#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");
}
```

Glöm inte att lägga till den saknade raden i `md407-ram-crt.x`.

```
__bss_start__ = .;
  *(.bss)
  *(.bss.*)
  *(COMMON)
```

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 6.3: 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 6.4: 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 );
            c = fgetc( stdin );
            if( c == '\n' ) break;
            write( ASCIIDISPLAY_FILENO, &c, 1 );      }
}
```

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

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.